LeetMastery

Study Guide — Organised by Pattern · ranked fewest → most questions

Graphs · Greedy · JavaScript · String · Sliding Window · Sorting · Bit Manipulation · Backtracking · BFS · Trie · Math · Arrays & Hashing · Two Pointers · Heap · Matrix · Linked List · DFS · Binary Search · Dynamic Programming · Stack · Trees & BST

LeetDoocs · SimplyLeet · WalkCC · LeetCode.ca
All languages · 363 Doocs descriptions · 226 embedded images
LeetCode editorials: 56/331 free · Easy → Medium → Hard
331 Questions · 21 Patterns

Table of Contents

#1 **Graphs** (5 questions)
#128 Longest Consecutive Sequence [Medium]
#277 Find the Celebrity [Medium]
#1136 Parallel Courses [Medium]
#305 Number of Islands II [Hard]
#1153 String Transforms Into Another String [Hard]

#2 **Greedy** (6 questions)
#409 Longest Palindrome [Easy]
#134 Gas Station [Medium]
#179 Largest Number [Medium]
#280 Wiggle Sort [Medium]
#954 Array of Doubled Pairs [Medium]
#2131 Longest Palindrome by Concatenating Two Letter Words [Medium]

#3 **JavaScript** (7 questions)
#2627 Debounce [Medium]
#2628 JSON Deep Equal [Medium]
#2630 Memoize [Medium]
#2633 Convert Object to JSON String [Medium]
#2636 Promise Pool [Medium]
#2675 Array of Objects to Matrix [Medium]
#2700 Differences Between Two Objects [Medium]

#4 **String** (8 questions)
#383 Ransom Note [Easy]
#734 Sentence Similarity [Easy]
#1165 Single Row Keyboard [Easy]
#8 String to Integer (atoi) [Medium]
#249 Group Shifted Strings [Medium]
#271 Encode and Decode Strings [Medium]
#635 Design Log Storage System [Medium]
#1138 Alphabet Board Path [Medium]

#5 **Sliding Window** (9 questions)
#3 Longest Substring Without Repeating Characters [Medium]
#159 Longest Substring with At Most Two Distinct Characters [Medium]
#340 Longest Substring with At Most K Distinct Characters [Medium]
#424 Longest Repeating Character Replacement [Medium]
#438 Find All Anagrams in a String [Medium]
#487 Max Consecutive Ones II [Medium]
#1100 Find K-Length Substrings with No Repeated Characters [Medium]
#1248 Count Number of Nice Subarrays [Medium]
#76 Minimum Window Substring [Hard]

#6 **Sorting** (9 questions)
#169 Majority Element [Easy]

#217 Contains Duplicate [Easy]
#242 Valid Anagram [Easy]
#252 Meeting Rooms [Easy]
#1086 Largest Unique Number [Easy]
#1122 Relative Sort Array [Easy]
#49 Group Anagrams [Medium]
#56 Merge Intervals [Medium]
#1152 Analyze User Website Visit Pattern [Medium]

#7 **Bit Manipulation** (10 questions)
#67 Add Binary [Easy]
#136 Single Number [Easy]
#190 Reverse Bits [Easy]
#191 Number of 1 Bits [Easy]
#268 Missing Number [Easy]
#338 Counting Bits [Easy]
#78 Subsets [Medium]
#90 Subsets II [Medium]
#287 Find the Duplicate Number [Medium]
#1506 Find Root of N-Ary Tree [Medium]

#8 **Backtracking** (10 questions)
#17 Letter Combinations of a Phone Number [Medium]
#22 Generate Parentheses [Medium]
#39 Combination Sum [Medium]
#46 Permutations [Medium]
#79 Word Search [Medium]
#113 Path Sum II [Medium]
#37 Sudoku Solver [Hard]
#51 N-Queens [Hard]
#126 Word Ladder II [Hard]
#996 Number of Squareful Arrays [Hard]

#9 **BFS** (10 questions)
#286 Walls and Gates [Medium]
#322 Coin Change [Medium]
#542 01 Matrix [Medium]
#994 Rotting Oranges [Medium]
#1197 Minimum Knight Moves [Medium]
#1730 Shortest Path to Get Food [Medium]
#127 Word Ladder [Hard]
#317 Shortest Distance from All Buildings [Hard]
#773 Sliding Puzzle [Hard]
#815 Bus Routes [Hard]

#10 **Trie** (12 questions)
#14 Longest Common Prefix [Easy]
#139 Word Break [Medium]

#208 Implement Trie (Prefix Tree) [Medium]
#211 Design Add and Search Words Data Structure [Medium]
#616 Add Bold Tag in String [Medium]
#692 Top K Frequent Words [Medium]
#720 Longest Word in Dictionary [Medium]
#1166 Design File System [Medium]
#212 Word Search II [Hard]
#336 Palindrome Pairs [Hard]
#588 Design In-Memory File System [Hard]
#642 Design Search Autocomplete System [Hard]

#11 **Math** (12 questions)
#9 Palindrome Number [Easy]
#13 Roman to Integer [Easy]
#504 Base 7 [Easy]
#1056 Confusing Number [Easy]
#1228 Missing Number in Arithmetic Progression [Easy]
#1427 Perform String Shifts [Easy]
#7 Reverse Integer [Medium]
#50 Pow(x, n) [Medium]
#380 Insert Delete GetRandom O(1) [Medium]
#1017 Convert to Base -2 [Medium]
#3160 Find the Number of Distinct Colors Among the Balls [Medium]
#68 Text Justification [Hard]

#12 **Arrays & Hashing** (18 questions)
#1 Two Sum [Easy]
#163 Missing Ranges [Easy]
#346 Moving Average from Data Stream [Easy]
#760 Find Anagram Mappings [Easy]
#1426 Counting Elements [Easy]
#1534 Count Good Triplets [Easy]
#57 Insert Interval [Medium]
#238 Product of Array Except Self [Medium]
#281 Zigzag Iterator [Medium]
#307 Range Sum Query - Mutable [Medium]
#454 4Sum II [Medium]
#525 Contiguous Array [Medium]
#560 Subarray Sum Equals K [Medium]
#1010 Pairs of Songs With Total Durations Divisible by 60 [Medium]
#1272 Remove Interval [Medium]
#1429 First Unique Number [Medium]
#3159 Find Occurrences of an Element in an Array [Medium]
#41 First Missing Positive [Hard]

#13 **Two Pointers** (19 questions)
#88 Merge Sorted Array [Easy]

#125 Valid Palindrome [Easy]
#283 Move Zeroes [Easy]
#408 Valid Word Abbreviation [Easy]
#977 Squares of a Sorted Array [Easy]
#11 Container With Most Water [Medium]
#15 3Sum [Medium]
#16 3Sum Closest [Medium]
#31 Next Permutation [Medium]
#75 Sort Colors [Medium]
#161 One Edit Distance [Medium]
#167 Two Sum II - Input Array Is Sorted [Medium]
#186 Reverse Words in a String II [Medium]
#189 Rotate Array [Medium]
#532 K-diff Pairs in an Array [Medium]
#923 3Sum With Multiplicity [Medium]
#986 Interval List Intersections [Medium]
#1055 Shortest Way to Form String [Medium]
#2563 Count the Number of Fair Pairs [Medium]

#14 **Heap** (20 questions)
#215 Kth Largest Element in an Array [Medium]
#253 Meeting Rooms II [Medium]
#347 Top K Frequent Elements [Medium]
#505 The Maze II [Medium]
#621 Task Scheduler [Medium]
#658 Find K Closest Elements [Medium]
#787 Cheapest Flights Within K Stops [Medium]
#973 K Closest Points to Origin [Medium]
#1057 Campus Bikes [Medium]
#1167 Minimum Cost to Connect Sticks [Medium]
#1424 Diagonal Traverse II [Medium]
#23 Merge k Sorted Lists [Hard]
#218 The Skyline Problem [Hard]
#272 Closest Binary Search Tree Value II [Hard]
#295 Find Median from Data Stream [Hard]
#358 Rearrange String k Distance Apart [Hard]
#499 The Maze III [Hard]
#632 Smallest Range Covering Elements from K Lists [Hard]
#759 Employee Free Time [Hard]
#1425 Constrained Subsequence Sum [Hard]

#15 **Matrix** (20 questions)
#422 Valid Word Square [Easy]
#566 Reshape the Matrix [Easy]
#766 Toeplitz Matrix [Easy]
#867 Transpose Matrix [Easy]

#2373 Largest Local Values in a Matrix [Easy]
#36 Valid Sudoku [Medium]
#48 Rotate Image [Medium]
#54 Spiral Matrix [Medium]
#59 Spiral Matrix II [Medium]
#64 Minimum Path Sum [Medium]
#73 Set Matrix Zeroes [Medium]
#74 Search a 2D Matrix [Medium]
#221 Maximal Square [Medium]
#311 Sparse Matrix Multiplication [Medium]
#498 Diagonal Traverse [Medium]
#531 Lonely Pixel I [Medium]
#723 Candy Crush [Medium]
#835 Image Overlap [Medium]
#1198 Find Smallest Common Element in All Rows [Medium]
#1074 Number of Submatrices That Sum to Target [Hard]

#16 **Linked List** (23 questions)
#21 Merge Two Sorted Lists [Easy]
#141 Linked List Cycle [Easy]
#206 Reverse Linked List [Easy]
#234 Palindrome Linked List [Easy]
#706 Design HashMap [Easy]
#876 Middle of the Linked List [Easy]
#1474 Delete N Nodes After M Nodes of Linked List [Easy]
#2 Add Two Numbers [Medium]
#19 Remove Nth Node From End of List [Medium]
#24 Swap Nodes in Pairs [Medium]
#61 Rotate List [Medium]
#146 LRU Cache [Medium]
#148 Sort List [Medium]
#328 Odd Even Linked List [Medium]
#369 Plus One Linked List [Medium]
#430 Flatten a Multilevel Doubly Linked List [Medium]
#622 Design Circular Queue [Medium]
#707 Design Linked List [Medium]
#708 Insert into a Sorted Circular Linked List [Medium]
#1171 Remove Zero Sum Consecutive Nodes from Linked List [Medium]
#25 Reverse Nodes in k-Group [Hard]
#432 All O'one Data Structure [Hard]
#460 LFU Cache [Hard]

#17 **DFS** (23 questions)
#463 Island Perimeter [Easy]
#733 Flood Fill [Easy]
#130 Surrounded Regions [Medium]

#133 Clone Graph [Medium]
#200 Number of Islands [Medium]
#207 Course Schedule [Medium]
#210 Course Schedule II [Medium]
#261 Graph Valid Tree [Medium]
#310 Minimum Height Trees [Medium]
#323 Number of Connected Components in an Undirected Graph [Medium]
#417 Pacific Atlantic Water Flow [Medium]
#490 The Maze [Medium]
#694 Number of Distinct Islands [Medium]
#695 Max Area of Island [Medium]
#721 Accounts Merge [Medium]
#785 Is Graph Bipartite? [Medium]
#1236 Web Crawler [Medium]
#1242 Web Crawler Multithreaded [Medium]
#269 Alien Dictionary [Hard]
#329 Longest Increasing Path in a Matrix [Hard]
#685 Redundant Connection II [Hard]
#1036 Escape a Large Maze [Hard]
#1192 Critical Connections in a Network [Hard]

#18 Binary Search (24 questions)
#35 Search Insert Position [Easy]
#69 Sqrt(x) [Easy]
#278 First Bad Version [Easy]
#374 Guess Number Higher or Lower [Easy]
#704 Binary Search [Easy]
#33 Search in Rotated Sorted Array [Medium]
#34 Find First and Last Position of Element in Sorted Array [Medium]
#153 Find Minimum in Rotated Sorted Array [Medium]
#300 Longest Increasing Subsequence [Medium]
#362 Design Hit Counter [Medium]
#528 Random Pick with Weight [Medium]
#852 Peak Index in a Mountain Array [Medium]
#981 Time Based Key-Value Store [Medium]
#1011 Capacity to Ship Packages Within D Days [Medium]
#1060 Missing Element in Sorted Array [Medium]
#1062 Longest Repeating Substring [Medium]
#1533 Find the Index of the Large Integer [Medium]
#4 Median of Two Sorted Arrays [Hard]
#315 Count of Smaller Numbers After Self [Hard]
#410 Split Array Largest Sum [Hard]
#668 Kth Smallest Number in Multiplication Table [Hard]
#710 Random Pick with Blacklist [Hard]
#774 Minimize Max Distance to Gas Station [Hard]

#1235 Maximum Profit in Job Scheduling [Hard]

#19 Dynamic Programming (24 questions)
#70 Climbing Stairs [Easy]
#121 Best Time to Buy and Sell Stock [Easy]
#5 Longest Palindromic Substring [Medium]
#53 Maximum Subarray [Medium]
#55 Jump Game [Medium]
#62 Unique Paths [Medium]
#72 Edit Distance [Medium]
#91 Decode Ways [Medium]
#152 Maximum Product Subarray [Medium]
#198 House Robber [Medium]
#256 Paint House [Medium]
#309 Best Time to Buy and Sell Stock with Cooldown [Medium]
#377 Combination Sum IV [Medium]
#416 Partition Equal Subset Sum [Medium]
#435 Non-overlapping Intervals [Medium]
#516 Longest Palindromic Subsequence [Medium]
#576 Out of Boundary Paths [Medium]
#1143 Longest Common Subsequence [Medium]
#10 Regular Expression Matching [Hard]
#115 Distinct Subsequences [Hard]
#123 Best Time to Buy and Sell Stock III [Hard]
#132 Palindrome Partitioning II [Hard]
#312 Burst Balloons [Hard]
#1000 Minimum Cost to Merge Stones [Hard]

#20 Stack (25 questions)
#20 Valid Parentheses [Easy]
#232 Implement Queue using Stacks [Easy]
#844 Backspace String Compare [Easy]
#143 Reorder List [Medium]
#150 Evaluate Reverse Polish Notation [Medium]
#155 Min Stack [Medium]
#173 Binary Search Tree Iterator [Medium]
#227 Basic Calculator II [Medium]
#255 Verify Preorder Sequence in Binary Search Tree [Medium]
#394 Decode String [Medium]
#439 Ternary Expression Parser [Medium]
#484 Find Permutation [Medium]
#735 Asteroid Collision [Medium]
#739 Daily Temperatures [Medium]
#962 Maximum Width Ramp [Medium]
#1214 Two Sum BSTs [Medium]
#1762 Buildings With an Ocean View [Medium]

#32 Longest Valid Parentheses [Hard]
#42 Trapping Rain Water [Hard]
#84 Largest Rectangle in Histogram [Hard]
#224 Basic Calculator [Hard]
#239 Sliding Window Maximum [Hard]
#716 Max Stack [Hard]
#772 Basic Calculator III [Hard]
#895 Maximum Frequency Stack [Hard]

#21 Trees & BST (37 questions)
#100 Same Tree [Easy]
#101 Symmetric Tree [Easy]
#104 Maximum Depth of Binary Tree [Easy]
#108 Convert Sorted Array to Binary Search Tree [Easy]
#110 Balanced Binary Tree [Easy]
#112 Path Sum [Easy]
#226 Invert Binary Tree [Easy]
#270 Closest Binary Search Tree Value [Easy]
#501 Find Mode in Binary Search Tree [Easy]
#543 Diameter of Binary Tree [Easy]
#572 Subtree of Another Tree [Easy]
#98 Validate Binary Search Tree [Medium]
#102 Binary Tree Level Order Traversal [Medium]
#103 Binary Tree Zigzag Level Order Traversal [Medium]
#105 Construct Binary Tree from Preorder and Inorder Traversal [Medium]
#199 Binary Tree Right Side View [Medium]
#230 Kth Smallest Element in a BST [Medium]
#235 Lowest Common Ancestor of a Binary Search Tree [Medium]
#236 Lowest Common Ancestor of a Binary Tree [Medium]
#250 Count Univalue Subtrees [Medium]
#285 Inorder Successor in BST [Medium]
#298 Binary Tree Longest Consecutive Sequence [Medium]
#314 Binary Tree Vertical Order Traversal [Medium]
#333 Largest BST Subtree [Medium]
#366 Find Leaves of Binary Tree [Medium]
#437 Path Sum III [Medium]
#545 Boundary of Binary Tree [Medium]
#549 Binary Tree Longest Consecutive Sequence II [Medium]
#582 Kill Process [Medium]
#662 Maximum Width of Binary Tree [Medium]
#863 All Nodes Distance K in Binary Tree [Medium]
#1120 Maximum Average Subtree [Medium]
#1490 Clone N-ary Tree [Medium]
#1522 Diameter of N-Ary Tree [Medium]
#1650 Lowest Common Ancestor of a Binary Tree III [Medium]

#124 Binary Tree Maximum Path Sum [Hard]
#297 Serialize and Deserialize Binary Tree [Hard]

#1 Graphs — 5 questions · #1 of 21

#128 MEDIUM

Longest Consecutive Sequence

LeetDocs • SimplyLeet • WalkCC • LeetCode • Editorial

Array · Hash Table · Union Find

Lists: Grind 169 | CodeSignal

Problem:

Given an unsorted array of integers nums, return the length of the longest consecutive elements sequence.

You must write an algorithm that runs in O(n) time.

Example 1:

```
Input: nums = [100,4,200,1,3,2]
Output: 4
Explanation: The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.
```

Example 2:

```
Input: nums = [0,3,7,2,5,8,4,6,0,1]
Output: 9
```

Example 3:

```
Input: nums = [1,0,1,2]
Output: 3
```

Constraints:

- 0 <= nums.length <= 105
- -109 <= nums[i] <= 109

>> Brute Force [Graphs] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def longestConsecutive(self, nums):
        # Brute Force O(n^2) - for each number extend its sequence
        num_set = set(nums)
        best = 0
        for num in nums:
            length = 0
            while num + length in num_set:
                length += 1
            best = max(best, length)
        return best
```

Python

Python

```python
# Python solution for Longest Consecutive Sequence

def longestConsecutive(nums):
    # Create a set from the list for O(1) look-up times
    num_set = set(nums)
    longest_streak = 0

    # Iterate through each number in the set
    for num in num_set:
        # Only start counting if 'num-1' is not present in the set
        if num - 1 not in num_set:
            current_num = num
            current_streak = 1

            # Count consecutive numbers starting from current_num
            while current_num + 1 in num_set:
                current_num += 1
                current_streak += 1

            # Update the longest streak if current one is longer
            longest_streak = max(longest_streak, current_streak)

    return longest_streak

# Example usage:

print(longestConsecutive([100, 4, 200, 1, 3, 2])) # Output: 4
```

Python

```python
class Solution:
  def longestConsecutive(self, nums: list[int]) -> int:
    ans = 0
    seen = set(nums)

    for num in seen:
      # 'num' is the start of a sequence.
      if num - 1 in seen:
        continue
      length = 0
      while num in seen:
        num += 1
        length += 1
      ans = max(ans, length)

    return ans
```

Python


```python
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        s = set(nums)
        ans = 0
        d = defaultdict(int)
        for x in nums:
            y = x
            while y in s:
                s.remove(y)
                y += 1
            d[x] = d[y] + y - x
            ans = max(ans, d[x])
        return ans
```

Python

```python
class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        s = set(nums)
        ans = 0
        for x in nums:
            if x - 1 not in s:
                y = x + 1
                while y in s:
                    y += 1
                ans = max(ans, y - x)
        return ans
```

[*] Graphs — Pattern Solution (stored optimal)

Python

```python
# Python solution for Longest Consecutive Sequence

def longestConsecutive(nums):
    # Create a set from the list for O(1) look-up times
    num_set = set(nums)
    longest_streak = 0

    # Iterate through each number in the set
    for num in num_set:
        # Only start counting if 'num-1' is not present in the set
        if num - 1 not in num_set:
            current_num = num
            current_streak = 1

            # Count consecutive numbers starting from current_num
            while current_num + 1 in num_set:
                current_num += 1
                current_streak += 1

            # Update the longest streak if current one is longer
            longest_streak = max(longest_streak, current_streak)

    return longest_streak

# Example usage:

print(longestConsecutive([100, 4, 200, 1, 3, 2])) # Output: 4
```


#277 MEDIUM

Find the Celebrity

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Two Pointers · Graph · Interactive

Lists: Premium 98

Problem:

Suppose you are at a party with n people labeled from 0 to n − 1 and among them, there may exist one celebrity. The definition of a celebrity is that all the other n − 1 people know the celebrity, but the celebrity does not know any of them.

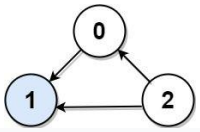
Now you want to find out who the celebrity is or verify that there is not one. You are only allowed to ask questions like: "Hi, A. Do you know B?" to get information about whether A knows B. You need to find out the celebrity (or verify there is not one) by asking as few questions as possible (in the asymptotic sense).

You are given an integer n and a helper function bool knows(a, b) that tells you whether a knows b. Implement a function int findCelebrity(n). There will be exactly one celebrity if they are at the party.

Return the celebrity's label if there is a celebrity at the party. If there is no celebrity, return −1.

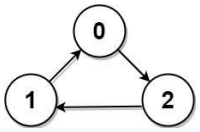
Note that the n x n 2D array graph given as input is not directly available to you, and instead only accessible through the helper function knows. graph[i][j] == 1 represents person i knows person j, whereas graph[i][j] == 0 represents person j does not know person i.

Example 1:




```
Input: graph = [[1,1,0],[0,1,0],[1,1,1]]
Output: 1
Explanation: There are three persons labeled with 0, 1 and 2. graph[i][j] = 1 means person i
knows person j, otherwise graph[i][j] = 0 means person i does not know person j. The
celebrity is the person labeled as 1 because both 0 and 2 know him but 1 does not know
anybody.
```

Example 2:




```
Input: graph = [[1,0,1],[1,1,0],[0,1,1]]
Output: -1
Explanation: There is no celebrity.
```

Constraints:

- n == graph.length == graph[i].length
- 2 <= n <= 100
- graph[i][j] is 0 or 1.
- graph[i][i] == 1

Follow up: If the maximum number of allowed calls to the API knows is 3 * n, could you find a solution without exceeding the maximum number of calls?

>> Brute Force [Graphs] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findCelebrity(self, n):
        # Brute force - try all paths via exhaustive DFS
        visited = set()
        def dfs(node):
            if node == dst: return True
            visited.add(node)
            for nxt in graph.get(node, []):
                if nxt not in visited and dfs(nxt): return True
            visited.remove(node)
            return False
        return dfs(src)
```

Python

Python


```python
# Function definition for findCelebrity using the helper knows(a, b) API.
def findCelebrity(n):
    # Step 1: Candidate Selection.
    candidate = 0
    for i in range(1, n):
        # If candidate knows i, then candidate cannot be the celebrity.
        if knows(candidate, i):
            candidate = i # Select new candidate.

    # Step 2: Verification.
    # Check if candidate does not know any other person.
    for i in range(n):
        if i != candidate:
            # Either candidate knows someone or someone doesn't know candidate.
            if knows(candidate, i) or not knows(i, candidate):
                return -1
    return candidate

# Example helper function (for testing, this is normally provided in the problem context).
def knows(a, b):
    # This function should return True if person a knows person b, otherwise False.
    pass
```

Python

```python
# The knows API is already defined for you.
# Returns a bool, whether a knows b
# Def knows(a: int, b: int) -> bool:


class Solution:
  def findCelebrity(self, n: int) -> int:
    candidate = 0

    # Everyone knows the celebrity.
    for i in range(1, n):
      if knows(candidate, i):
        candidate = i

    # The candidate knows nobody and everyone knows the celebrity.
    for i in range(n):
      if i < candidate and knows(candidate, i) or not knows(i, candidate):
        return -1
      if i > candidate and not knows(i, candidate):
        return -1

    return candidate
```

Python


```python
# The knows API is already defined for you.
# return a bool, whether a knows b
# def knows(a: int, b: int) -> bool:


class Solution:
    def findCelebrity(self, n: int) -> int:
        ans = 0
        for i in range(1, n):
            if knows(ans, i):
                ans = i
        for i in range(n):
            if ans != i:
                if knows(ans, i) or not knows(i, ans):
                    return -1
        return ans
```

Python

```python
# The knows API is already defined for you.
# return a bool, whether a knows b
# def knows(a: int, b: int) -> bool:


class Solution:
    def findCelebrity(self, n: int) -> int:
        ans = 0
        for i in range(1, n):
            if knows(ans, i):
                ans = i
        for i in range(n):
            if ans != i:
                if knows(ans, i) or not knows(i, ans):
                    return -1
        return ans
```

[*] Graphs — Pattern Solution (stored optimal)

Python

```python
# Function definition for findCelebrity using the helper knows(a, b) API.
def findCelebrity(n):
    # Step 1: Candidate Selection.
    candidate = 0
    for i in range(1, n):
        # If candidate knows i, then candidate cannot be the celebrity.
        if knows(candidate, i):
            candidate = i # Select new candidate.

    # Step 2: Verification.
    # Check if candidate does not know any other person.
    for i in range(n):
        if i != candidate:
            # Either candidate knows someone or someone doesn't know candidate.
            if knows(candidate, i) or not knows(i, candidate):
                return -1
    return candidate

# Example helper function (for testing, this is normally provided in the problem context).
def knows(a, b):
    # This function should return True if person a knows person b, otherwise False.
    pass
```

#1136 MEDIUM

Parallel Courses

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Graph · Topological Sort

Lists: Premium 98

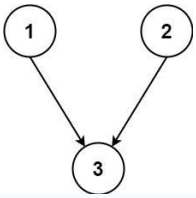
Problem:

You are given an integer n, which indicates that there are n courses labeled from 1 to n. You are also given an array relations where relations[i] = [prevCoursei, nextCoursei], representing a prerequisite relationship between course prevCoursei and course nextCoursei: course prevCoursei has to be taken before course nextCoursei.

In one semester, you can take any number of courses as long as you have taken all the prerequisites in the previous semester for the courses you are taking.

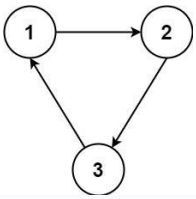
Return the minimum number of semesters needed to take all courses. If there is no way to take all the courses, return -1.

Example 1:




```
Input: n = 3, relations = [[1,3],[2,3]]
Output: 2
Explanation: The figure above represents the given graph.
In the first semester, you can take courses 1 and 2.
In the second semester, you can take course 3.
```

Example 2:




```
Input: n = 3, relations = [[1,2],[2,3],[3,1]]
Output: -1
Explanation: No course can be studied because they are prerequisites of each other.
```

Constraints:

- 1 <= n <= 5000
- 1 <= relations.length <= 5000
- relations[i].length == 2
- 1 <= prevCoursei, nextCoursei <= n

- prevCoursei != nextCoursei
- All the pairs [prevCoursei, nextCoursei] are unique.

>> Brute Force [Graphs] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def minimumSemesters(self, numCourses, relations):
        # Brute force - try all paths via exhaustive DFS
        visited = set()
        def dfs(node):
            if node == dst: return True
            visited.add(node)
            for nxt in graph.get(node, []):
                if nxt not in visited and dfs(nxt): return True
            visited.remove(node)
            return False
        return dfs(src)
```

Python

Python

```python
# Python Implementation

from collections import deque

def minimumSemesters(numCourses, relations):
    # Build graph and in-degree dictionary.
    graph = {i: [] for i in range(1, numCourses + 1)}
    in_degree = {i: 0 for i in range(1, numCourses + 1)}

    # Construct graph from relations.
    for pre, course in relations:
        graph[pre].append(course)
        in_degree[course] += 1

    # Initialize queue with all courses having no prerequisites.
    queue = deque()
    for course in range(1, numCourses + 1):
        if in_degree[course] == 0:
            queue.append(course)

    semesters = 0
    takenCourses = 0

    # Process each level (semester) using BFS.
    while queue:
```



```python
        semesters += 1
        current_level_size = len(queue)
        for _ in range(current_level_size):
            course = queue.popleft()
            takenCourses += 1
            # Decrement in-degree for neighboring courses.
            for neighbor in graph[course]:
                in_degree[neighbor] -= 1
                if in_degree[neighbor] == 0:
                    queue.append(neighbor)

    # If not all courses have been taken, a cycle exists.
    return semesters if takenCourses == numCourses else -1

# Example usage:
print(minimumSemesters(3, [[1, 3], [2, 3]])) # Expected output: 2
```

Python

```python
from enum import Enum


class State(Enum):
  INIT = 0
  VISITING = 1
  VISITED = 2


class Solution:
  def minimumSemesters(self, n: int, relations: list[list[int]]) -> int:
    graph = [[] for _ in range(n)]
    states = [State.INIT] * n
    depth = [1] * n

    for u, v in relations:
      graph[u - 1].append(v - 1)

    def hasCycle(u: int) -> bool:
      if states[u] == State.VISITING:
        return True
      if states[u] == State.VISITED:
        return False
      states[u] = State.VISITING
      for v in graph[u]:
```

```python
        if hasCycle(v):
          return True
        depth[u] = max(depth[u], 1 + depth[v])
      states[u] = State.VISITED
      return False

    if any(hasCycle(i) for i in range(n)):
      return -1
    return max(depth)
```

Python


```python
class Solution:
    def minimumSemesters(self, n: int, relations: list[list[int]]) -> int:
        graph = [[] for _ in range(n)]
        inDegrees = [0] * n

        # Build the graph.
        for u, v in relations:
          graph[u - 1].append(v - 1)
          inDegrees[v - 1] += 1

        # Perform topological sorting.
        q = collections.deque([i for i, d in enumerate(inDegrees) if d == 0])

        step = 0
        while q:
          for _ in range(len(q)):
            u = q.popleft()
            n -= 1
            for v in graph[u]:
              inDegrees[v] -= 1
              if inDegrees[v] == 0:
                q.append(v)
          step += 1

        return step if n == 0 else -1
```

Python

```python
class Solution:
    def minimumSemesters(self, n: int, relations: List[List[int]]) -> int:
        q = defaultdict(list)
        indeg = [0] * n
        for prev, nxt in relations:
            prev, nxt = prev - 1, nxt - 1
            q[prev].append(nxt)
            indeg[nxt] += 1
        q = deque(i for i, v in enumerate(indeg) if v == 0)
        ans = 0
        while q:
            ans += 1
            for _ in range(len(q)):
                i = q.popleft()
                n -= 1
                for j in q[i]:
                    indeg[j] -= 1
                    if indeg[j] == 0:
                        q.append(j)
        return -1 if n else ans
```



```python
class Solution:
    def minimumSemesters(self, n: int, relations: List[List[int]]) -> int:
        q = defaultdict(list)
        indeg = [0] * n
        for prev, nxt in relations:
            prev, nxt = prev - 1, nxt - 1
            q[prev].append(nxt)
            indeg[nxt] += 1
        q = deque(i for i, v in enumerate(indeg) if v == 0)
        ans = 0
        while q:
            ans += 1
            for _ in range(len(q)):
                i = q.popleft()
                n -= 1
                for j in q[i]:
                    indeg[j] -= 1
                    if indeg[j] == 0:
                        q.append(j)
        return -1 if n else ans
```

[*] Graphs — Pattern Solution (matched from community answers)

Python

```python
# Python Implementation

from collections import deque

def minimumSemesters(numCourses, relations):
    # Build graph and in-degree dictionary.
    graph = {i: [] for i in range(1, numCourses + 1)}
    in_degree = {i: 0 for i in range(1, numCourses + 1)}

    # Construct graph from relations.
    for pre, course in relations:
        graph[pre].append(course)
        in_degree[course] += 1

    # Initialize queue with all courses having no prerequisites.
    queue = deque()
    for course in range(1, numCourses + 1):
        if in_degree[course] == 0:
            queue.append(course)

    semesters = 0
    takenCourses = 0

    # Process each level (semester) using BFS.
    while queue:
```

```python
        semesters += 1
        current_level_size = len(queue)
        for _ in range(current_level_size):
            course = queue.popleft()
            takenCourses += 1
            # Dec/count in-degree for neighboring courses.
            for neighbor in graph[course]:
                in_degree[neighbor] -= 1
                if in_degree[neighbor] == 0:
                    queue.append(neighbor)

    # If not all courses have been taken, a cycle exists.
    return semesters if takenCourses == numCourses else -1

# Example usage:
print(minimumSemesters(3, [[1, 3], [2, 3]])) # Expected output: 2
```

#305 HARD

Number of Islands II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Union Find · Matrix

Lists: Premium 98

Problem:

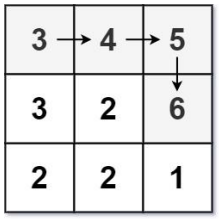
You are given an empty 2D binary grid grid of size m x n. The grid represents a map where 0's represent water and 1's represent land. Initially, all the cells of grid are water cells (i.e., all the cells are 0's).

We may perform an add land operation which turns the water at position into a land. You are given an array positions where positions[i] = [ri, ci] is the position (ri, ci) at which we should operate the ith operation.

Return an array of integers answer where answer[i] is the number of islands after turning the cell (ri, ci) into a land.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:




```
Input: m = 3, n = 3, positions = [[0,0],[0,1],[1,2],[2,1]]
Output: [1,1,2,3]
Explanation:
Initially, the 2d grid is filled with water.
- Operation #1: addLand(0, 0) turns the water at grid[0][0] into a land. We have 1 island.
- Operation #2: addLand(0, 1) turns the water at grid[0][1] into a land. We still have 1
island.
- Operation #3: addLand(1, 2) turns the water at grid[1][2] into a land. We have 2 islands.
- Operation #4: addLand(2, 1) turns the water at grid[2][1] into a land. We have 3 islands.
```

Example 2:

```
Input: m = 1, n = 1, positions = [[0,0]]
Output: [1]
```

Constraints:

- 1 <= m, n, positions.length <= 10^4
- 1 <= m * n <= 10^4
- positions[i].length == 2
- 0 <= ri < m
- 0 <= ci < n

Follow up: Could you solve it in time complexity O(k log(mn)), where k == positions.length?

>> Brute Force [Graphs] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def numIslands2(self, m, n, positions):
        # Brute Force = try all paths via exhaustive DFS
        visited = set()
        def dfs(node):
            if node == dst: return True
            visited.add(node)
            for nxt in graph.get(node, []):
                if nxt not in visited and dfs(nxt): return True
            visited.remove(node)
            return False
        return dfs(src)
```

Python

Python

```python
# Python implementation using Union Find with path compression and union by rank.
class UnionFind:
    def __init__(self, size):
        # parent: stores parent of each node, -1 indicates water.
        self.parent = [-1] * size
        self.rank = [0] * size
        self.count = 0 # number of islands

    def find(self, x):
        # Find with path compression.
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        # Union the sets, if union is successful, reduce island count.
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX == rootY:
            return False
        # union by rank
        if self.rank[rootX] < self.rank[rootY]:
            self.parent[rootX] = rootY
        elif self.rank[rootX] > self.rank[rootY]:
            self.parent[rootY] = rootX
```



```python
        else:
            self.parent[rootY] = rootX
            self.rank[rootX] += 1
        self.count -= 1
        return True

class Solution:
    def numIslands2(self, m, n, positions):
        uf = UnionFind(m * n)
        result = []
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        for r, c in positions:
            idx = r * n + c
            # If the position is already land, skip.
            if uf.parent[idx] != -1:
                result.append(uf.count)
                continue

            # Mark this cell as land forming a new island.
            uf.parent[idx] = idx
            uf.count += 1

            # Union with all 4 neighbors if they are already land.
            for dr, dc in directions:

                nr, nc = r + dr, c + dc
                nidx = nr * n + nc
                if 0 <= nr < m and 0 <= nc < n and uf.parent[nidx] != -1:
                    uf.union(idx, nidx)
            result.append(uf.count)
        return result

# Example run:
# s = Solution()
# print(s.numIslands2(3, 3, [[0,0],[0,1],[1,2],[2,1]]))
```

Python


```python
class UnionFind:
    def __init__(self, n: int):
        self.id = [-1] * n
        self.rank = [0] * n

    def unionByRank(self, u: int, v: int) -> None:
        i = self.find(u)
        j = self.find(v)
        if i == j:
            return
        if self.rank[i] < self.rank[j]:
            self.id[i] = j
        elif self.rank[i] > self.rank[j]:
            self.id[j] = i
        else:
            self.id[i] = j
            self.rank[j] += 1

    def find(self, u: int) -> int:
        if self.id[u] != u:
            self.id[u] = self.find(self.id[u])
        return self.id[u]


class Solution:

    def numIslands2(
        self,
        m: int,
        n: int,
        positions: list[list[int]],
    ) -> list[int]:
        DIRS = ((0, 1), (1, 0), (0, -1), (-1, 0))
        ans = []
        seen = [[False] * n for _ in range(m)]
        uf = UnionFind(m * n)
        count = 0

        def getId(i: int, j: int, n: int) -> int:
            return i * n + j

        for i, j in positions:
            if seen[i][j]:
                ans.append(count)
                continue
            seen[i][j] = True
            id = getId(i, j, n)
            uf.id[id] = id
            count += 1
            for dx, dy in DIRS:
                x = i + dx
```



```python
            y = j + dy
            if x < 0 or x >= m or y < 0 or y >= n:
                continue
            neighborId = getId(x, y, n)
            if uf.id[neighborId] == -1: # water
                continue
            currentParent = uf.find(id)
            neighborParent = uf.find(neighborId)
            if currentParent != neighborParent:
                uf.unionByRank(currentParent, neighborParent)
                count -= 1
        ans.append(count)

        return ans
```

Python

```python
class Solution:
    def numIslands2(self, m: int, n: int, positions: List[List[int]] -> List[int]:
        def check(i, j):
            return 0 <= i < m and 0 <= j < n and grid[i][j] == 1

        def find(x):
            if p[x] != x:
                p[x] = find(p[x])
            return p[x]

        p = list(range(m * n))
        grid = [[0] * n for _ in range(m)]
        res = []
        cur = 0 # current island count
        for i, j in positions:
            if grid[i][j] == 1: # already counted, same as previous island count
                res.append(cur)
                continue
            grid[i][j] = 1
            cur += 1
            for x, y in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                if check(i + x, j + y) and find(i * n + j) != find((i + x) * n + j + y):
                    p[find(i * n + j)] = find((i + x) * n + j + y)
                    cur -= 1
                    # 1 0 1
```

```
                    # 0 1 0
                    # for above, if setting 1st-row 2nd-col 0-to-1
                    # cur will -1 for 3 times
                    # but cur += 1 after grid[i][j] = 1 so cur final result is 1
                res.append(cur)
            return res

###########

class UnionFind(object):
    def __init__(self, m, n):
        self.dad = [i for i in range(m * n)]
        self.rank = [0 for _ in range(m * n)]

    def find(self, x):
        if self.dad[x] != x:
            self.dad[x] = self.find(self.dad[x])
        return self.dad[x]

    def union(self, xy):# xy is a tuple (x,y), with x and y value inside
        x, y = map(self.find, xy)
        if x == y:
            return False
        if self.rank[x] > self.rank[y]:
            self.dad[y] = x

        elif self.rank[x] < self.rank[y]:
            self.dad[x] = y
        else:
            self.dad[y] = x # search to the left, to find parent
            self.rank[x] += 1 # now x a parent, so +1 its rank
        return True


class Solution(object):
    def numIslands2(self, m, n, positions):
        """
        :type m: int
        :type n: int
        :type positions: List[List[int]]
        :rtype: List[int]
        """
        uf = UnionFind(m, n)
        ans = 0
        ret = []
        dirs = [(0, -1), (0, 1), (1, 0), (-1, 0)]
        grid = set()
        for i, j in positions:
            ans += 1
            grid |= {(i, j)}
            for di, dj in dirs:

                ni, nj = i + di, j + dj
                if 0 <= ni < m and 0 <= nj < n and (ni, nj) in grid:
                    if uf.union((ni * n + nj, i * n + j)):
                        ans -= 1
            ret.append(ans)
```


[*] Graphs — Pattern Solution (matched from community answers)

Python

```
# Python implementation using Union Find with path compression and union by rank.
class UnionFind:
    def __init__(self, size):
        # parent: stores parent of each node. -1 indicates water.
        self.parent = [-1] * size
        self.rank = [0] * size
        self.count = 0 # number of islands

    def find(self, x):
        # Find with path compression.
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    def union(self, x, y):
        # Union the sets; if union is successful, reduce island count.
        rootX = self.find(x)
        rootY = self.find(y)
        if rootX == rootY:
            return False
        # union by rank.
        if self.rank[rootX] < self.rank[rootY]:
            self.parent[rootX] = rootY
        elif self.rank[rootX] > self.rank[rootY]:
            self.parent[rootY] = rootX

        else:
            self.parent[rootY] = rootX
            self.rank[rootX] += 1
        self.count -= 1
        return True

class Solution:
    def numIslands2(self, m, n, positions):
        uf = UnionFind(m * n)
        result = []
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]

        for r, c in positions:
            idx = r * n + c
            # If the position is already land, skip,
            if uf.parent[idx] != -1:
                result.append(uf.count)
                continue

            # Mark this cell as land forming a new island.
            uf.parent[idx] = idx
            uf.count += 1

            # Union with all 4 neighbors if they are already land.
            for dr, dc in directions:
```



```
                nr, nc = r + dr, c + dc
                nidx = nr * n + nc
                if 0 <= nr < m and 0 <= nc < n and uf.parent[nidx] != -1:
                    uf.union(idx, nidx)
            result.append(uf.count)
        return result

# Example run:
# s = Solution()
# print(s.numIslands2(3, 3, [(0,0),(0,1),(1,2),(2,1)]))
```

#1153 HARD

String Transforms Into Another String

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Union Find

Lists: Denny Zhang

Problem:

Given two strings str1 and str2 of the same length, determine whether you can transform str1 into str2 by doing zero or more conversions.

In one conversion you can convert all occurrences of one character in str1 to any other lowercase English character.

Return true if and only if you can transform str1 into str2.

Example 1:

```
Input: str1 = "aabcc", str2 = "ccdee"
Output: true
Explanation: Convert 'c' to 'e' then 'b' to 'd' then 'a' to 'c'. Note that the order of
conversions matter.
```

Example 2:

```
Input: str1 = "leetcode", str2 = "codeleet"
Output: false
Explanation: There is no way to transform str1 to str2.
```

Constraints:

- 1 <= str1.length == str2.length <= 104
- str1 and str2 contain only lowercase English letters.

Python

Python


```
# Python solution with line-by-line comments
def canConvert(str1: str, str2: str) -> bool:
    # If both strings are equal, no conversion is needed.
    if str1 == str2:
        return True

    # Dictionary to hold the mapping from characters in str1 to characters in str2
    mapping = {}

    # Iterate through the characters of both strings simultaneously.
    for c1, c2 in zip(str1, str2):
        # If c1 is already in the mapping, check for consistency.
        if c1 in mapping:
            if mapping[c1] != c2:
                # Found an inconsistent mapping.
                return False
        else:
            mapping[c1] = c2

    # Count distinct characters in str2.
    # If str2 uses all 26 letters, then there is no extra character available to break cycles.
    if len(set(str2)) == 26:
        return False

    return True


# Example Usage:
#print(canConvert("aabcc", "ccdee")) # Expected True
#print(canConvert("leetcode", "codeleet")) # Expected False
```

Python

```
class Solution:
    def canConvert(self, str1: str, str2: str) -> bool:
        if str1 == str2:
            return True

        mappings = {}

        for a, b in zip(str1, str2):
            if mappings.get(a, b) != b:
                return False
            mappings[a] = b

        # No letter in the str1 maps to > 1 letter in the str2 and there is at
        # last one temporary letter can break any loops.
        return len(set(str2)) < 26
```

Python


```
class Solution:
    def canConvert(self, str1: str, str2: str) -> bool:
        if str1 == str2:
            return True
        if len(set(str2)) == 26:
            return False
        d = {}
        for a, b in zip(str1, str2):
            if a not in d:
                d[a] = b
            elif d[a] != b:
                return False
        return True
```

Python

```
class Solution:
    def canConvert(self, str1: str, str2: str) -> bool:
        if str1 == str2:
            return True
        if len(set(str2)) == 26:
            return False
        d = {}
        for a, b in zip(str1, str2):
            if a not in d:
                d[a] = b
            elif d[a] != b:
                return False
        return True
```

[*] Graphs — Pattern Solution (stored optimal)

Python

```
# Python solution with line-by-line comments
def canConvert(str1: str, str2: str) -> bool:
    # If both strings are equal, no conversion is needed.
    if str1 == str2:
        return True

    # Dictionary to hold the mapping from characters in str1 to characters in str2
    mapping = {}

    # Iterate through the characters of both strings simultaneously.
    for c1, c2 in zip(str1, str2):
        # If c1 is already in the mapping, check for consistency.
        if c1 in mapping:
            if mapping[c1] != c2:
                # Found an inconsistent mapping.
                return False
        else:
            mapping[c1] = c2

    # Count distinct characters in str2.
    # If str2 uses all 26 letters, then there is no extra character available to break cycles.
    if len(set(str2)) == 26:
        return False

    return True
```



```
# Example Usage:
#print(canConvert("aabcc", "ccdee")) # Expected True
#print(canConvert("leetcode", "codeleet")) # Expected False
```

Quick Review — Graphs 5 questions · insights · complexity · solution

#128 Longest Consecutive Sequence [Medium]

Key Insights

- Utilize a set (or hash table) for constant time look-ups.
- Only start counting a sequence from a number that is the beginning (its previous number is not in the set).
- Each number is visited only once, ensuring an O(n) time complexity.
- Update the maximum sequence length as you discover longer consecutive sequences.

Space & Time

Time Complexity: O(n)

Space Complexity: O(n)

Solution

We first insert all the numbers into a hash set to enable O(1) membership checks. For each number in the set, we check if it is the start of a sequence by ensuring that the number minus one is not in the set. If it is the beginning, we then count all consecutive numbers following it, updating the maximum sequence length when necessary. This method guarantees that we only count each number once, satisfying the O(n) time requirement.

#277 Find the Celebrity [Medium]

Key Insights

- Use pairwise comparisons to eliminate non-celebrities.
- If a candidate knows another person, the candidate cannot be the celebrity.
- Verify the candidate by checking that they do not know anyone and that everyone knows them.
- Achieve the solution in two passes: one for candidate selection and the other for validation.

Space & Time

Time Complexity: O(n) - One pass for candidate selection and one for validation.

Space Complexity: O(1) - Only constant extra space is used.

Solution

The approach consists of two main steps:

- Candidate Selection: Iterate through the people and maintain a candidate. For every person i, if the current candidate knows i, then set candidate = i. This works because if candidate knows i, candidate cannot be the celebrity. - Verification: Check that the candidate does not know any other person and that every other person knows the candidate. If both conditions hold, candidate is the celebrity; otherwise, return -1.

The main data structures used are simple integer variables. The algorithm leverages a two-pass technique to reduce the number of knows API calls while ensuring correctness.

#1136 Parallel Courses [Medium]

Key Insights

- Model the problem as a directed graph where nodes are courses and edges represent prerequisites.
- Use topological sorting (Kahn's algorithm) to process courses with zero prerequisites.
- Each level (or layer) of processing represents one semester.
- Detect cycles by ensuring all courses are eventually processed; if not, a cycle exists.
- The number of BFS levels gives the minimum number of semesters necessary.

Space & Time

Time Complexity: O(n + m) where n is the number of courses and m is the length of relations (prerequisites).

Space Complexity: O(n + m) for storing the graph, in-degree counts, and the processing queue.

Solution

We solve the problem using a topological sort with Kahn's algorithm. First, construct a graph representation using an adjacency list and compute the in-degree for each course. Initialize a queue with courses that have no prerequisites (in-degree == 0). Then, process courses level by level. Each level processed represents one semester. For every course taken in that semester, decrease the in-degree of all its neighbors. Add any neighbor with an in-degree that becomes zero to the queue for the next semester. If, after processing, the total courses taken is less than n, then a cycle exists and return -1. Otherwise, the number of semesters processed is the answer.

#305 Number of Islands II [Hard]

Key Insights

- Use Union Find (Disjoint Set Union) to efficiently manage connected components (islands).
- Map 2D grid coordinates to a 1D index.
- When adding a new land, check its 4 neighbors (up, down, left, right) and perform unions if they are also lands.
- Avoid duplicate processing if the same cell is added multiple times.
- Maintain a counter for islands that is updated during each union operation.

Space & Time

Time Complexity: O(K * α(m

n)) where K is the number of operations and α is the inverse Ackermann function (almost constant time per union/find).

Space Complexity: O(m

n) for storing the union-find structure.

Solution

The solution relies on the Union Find data structure to dynamically merge islands. When a new land cell is added:

- If the cell is already land, record the current island count. - Otherwise, mark the cell as land and increment the island count. - Check the four neighboring cells; if any neighbor is land, attempt to union the current cell with that neighbor. - If a union actually happens (i.e., the two cells were in separate islands), decrement the island counter. - Output the island count after each add-land operation. The key trick is converting 2D indices to a 1D representation (r * n + c) and using path compression along with union by rank in the Union Find implementation for optimal performance.

#1153 String Transforms Into Another String [Hard]

Key Insights

- Each character mapping from str1 to str2 must be consistent; one character in str1 always maps to the same character in str2.
- A bijective mapping is not necessary; multiple characters in str1 can map to the same character in str2.
- The order of conversions matters. When there is a cycle (e.g., mapping 'a'->'b' and 'b'->'a'), an intermediate unused character might be required to break the cycle.
- If str2 uses all 26 letters, no temporary character is available to break mapping cycles. In this case, transformation is possible only if str1 is already equal to str2.

Space & Time

Time Complexity: O(n), where n is the length of the strings, because we traverse the strings to build and check the mapping.

Space Complexity: O(1) or O(26), which is constant space, to store the mapping for the 26 lowercase English letters.

Solution

The approach is as follows:

- If str1 equals str2, return true immediately. - Build a mapping from each character in str1 to the corresponding character in str2. While creating the mapping, check that each character maps consistently. - Count the number of

distinct characters in str2. If this number is 26 and str1 is not already equal to str2, it is impossible to perform the transformation because there is no spare character available as a temporary placeholder. - If the mapping is consistent and there exists at least one character not used in str2 (i.e., distinct characters in str2 is less than 26), the transformation is possible. The important trick is identifying the need for a free character when the target mapping is "full" (i.e., covers all 26 letters). Without a free character, you cannot avoid cycles where a temporary intermediate is necessary.

#2 Greedy — 6 questions · #2 of 21

#409 EASY

Longest Palindrome

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Greedy

Lists: Grind 169

Problem:

Given a string s which consists of lowercase or uppercase letters, return the length of the longest palindrome that can be built with those letters.

Letters are case sensitive, for example, "Aa" is not considered a palindrome.

Example 1:

```
Input: s = "abcccccdd"
Output: 7
Explanation: One longest palindrome that can be built is "dccaccd", whose length is 7.
```

Example 2:

```
Input: s = "a"
Output: 1
Explanation: The longest palindrome that can be built is "a", whose length is 1.
```

Constraints:

- 1 <= s.length <= 2000
- s consists of lowercase and/or uppercase English letters only.

Python

Python


```python
# Python solution for the Longest Palindrome problem

def longestPalindrome(s: str) -> int:
    # Create a dictionary to store the frequency of each character
    char_counts = {}
    for char in s:
        # Increment the count for the character
        char_counts[char] = char_counts.get(char, 0) + 1

    # Variable to store the length of the palindrome
    palindrome_length = 0
    # Flag to indicate whether we can add one more character in the middle for an odd count
    center_added = False

    # Iterate through the character counts
    for count in char_counts.values():
        # Add the largest even number from count to palindrome_length
        palindrome_length += (count // 2) * 2
        # If count is odd and we haven't added the center character yet
        if count % 2 == 1 and not center_added:
            palindrome_length += 1
            center_added = True

    return palindrome_length

# Example usage:
print(longestPalindrome("abcccccdd"))
```

Python

```python
class Solution:
    def longestPalindrome(self, s: str) -> int:
        ans = 0
        count = collections.Counter(s)

        for c in count.values():
          ans += c if c % 2 == 0 else c - 1

        hasOddCount = any(c % 2 == 1 for c in count.values())
        return ans + hasOddCount
```

Python

```python
class Solution:
    def longestPalindrome(self, s: str) -> int:
        cnt = Counter(s)
        ans = sum(v // 2 * 2 for v in cnt.values())
        ans += int(ans < len(s))
        return ans
```

Python

```python
class Solution:
    def longestPalindrome(self, s: str) -> int:
        odd = defaultdict(int)
        cnt = 0
        for c in s:
            odd[c] ^= 1
            cnt += 1 if odd[c] else -1
        return len(s) - cnt + 1 if cnt else len(s)
```


Python

```python
class Solution:
    def longestPalindrome(self, s: str) -> int:
        cnt = Counter(s)
        ans = 0
        for v in cnt.values():
            ans += v - (v & 1)
            ans += (ans & 1 ^ 1) and (v & 1)
        return ans
```

[*] Greedy — Pattern Solution (stored optimal)

Python

```python
# Python solution for the Longest Palindrome problem

def longestPalindrome(s: str) -> int:
    # Create a dictionary to store the frequency of each character
    char_counts = {}
    for char in s:
        # Increment the count for the character
        char_counts[char] = char_counts.get(char, 0) + 1

    # Variable to store the length of the palindrome
    palindrome_length = 0
    # Flag to indicate whether we can add one more character in the middle for an odd count
    center_added = False

    # Iterate through the character counts
    for count in char_counts.values():
        # Add the largest even number from count to palindrome_length
        palindrome_length += (count // 2) * 2
        # If count is odd and we haven't added the center character yet
        if count % 2 == 1 and not center_added:
            palindrome_length += 1
            center_added = True

    return palindrome_length

# Example usage:
print(longestPalindrome("abcccccdd"))
```

#134 MEDIUM

Gas Station

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Greedy

Lists: Grind 169

Problem:

There are n gas stations along a circular route, where the amount of gas at the ith station is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the ith station to its next (i + 1)th station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays gas and cost, return the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return -1. If there exists a solution, it is guaranteed to be unique.

Example 1:

```
Input: gas = [1,2,3,4,5], cost = [3,4,5,1,2]
Output: 3
Explanation:
Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 4. Your tank = 4 - 1 + 5 = 8
Travel to station 0. Your tank = 8 - 2 + 1 = 7
Travel to station 1. Your tank = 7 - 3 + 2 = 6
Travel to station 2. Your tank = 6 - 4 + 3 = 5
Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.
Therefore, return 3 as the starting index.
```

Example 2:

```
Input: gas = [2,3,4], cost = [3,4,3]
Output: -1
Explanation:
You can't start at station 0 or 1, as there is not enough gas to travel to the next station.
Let's start at station 2 and fill up with 4 unit of gas. Your tank = 0 + 4 = 4
Travel to station 0. Your tank = 4 - 3 + 2 = 3
Travel to station 1. Your tank = 3 - 3 + 3 = 3
You cannot travel back to station 2, as it requires 4 unit of gas but you only have 3.
Therefore, you can't travel around the circuit once no matter where you start.
```

Constraints:

- n == gas.length == cost.length
- 1 <= n <= 105
- 0 <= gas[i], cost[i] <= 104
- The input is generated such that the answer is unique.

>> Brute Force [Greedy] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def canCompleteCircuit(self, gas, cost):
        # Brute Force O(n^2) - try starting at every station
        n = len(gas)
        for start in range(n):
            tank = 0
            for step in range(n):
                i = (start + step) % n
                tank += gas[i] - cost[i]
                if tank < 0: break
            else:
                return start
        return -1
```

Python

Python


```python
# Define a function to find starting gas station index
def canCompleteCircuit(gas, cost):
    total_surplus = 0 # Total gas surplus after completing the route
    current_surplus = 0 # Gas surplus while iterating from a candidate start station
    start_index = 0 # Candidate starting station index

    # Loop through each station
    for i in range(len(gas)):
        # Calculate net gas after leaving station i
        surplus = gas[i] - cost[i]
        total_surplus += surplus
        current_surplus += surplus

        # If current surplus becomes negative, station i is not a valid start.
        if current_surplus < 0:
            # Reset starting station to the next station
            start_index = i + 1
            current_surplus = 0 # Reset current surplus
    # Check if overall surplus is non-negative
    return start_index if total_surplus >= 0 else -1

# Example usage:
gas = [1,2,3,4,5]
cost = [3,4,5,1,2]
print(canCompleteCircuit(gas, cost)) # Output: 3
```

Python

```python
class Solution:
    def canCompleteCircuit(self, gas: list[int], cost: list[int]) -> int:
        ans = 0
        net = 0
        summ = 0

        # Try to start from each index.
        for i in range(len(gas)):
          net += gas[i] - cost[i]
          summ += gas[i] - cost[i]
          if summ < 0:
            summ = 0
            ans = i + 1 # Start from the next index.

        return -1 if net < 0 else ans
```

Python

```python
class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        n = len(gas)
        i = j = n - 1
        cnt = s = 0
        while cnt < n:
            s += gas[j] - cost[j]
            cnt += 1
            j = (j + 1) % n
            while s < 0 and cnt < n:
                i -= 1
                s += gas[i] - cost[i]
                cnt += 1
        return -1 if s < 0 else i
```


Python

```python
class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        n = len(gas)
        i = j = n - 1
        cnt = s = 0
        while cnt < n:
            s += gas[j] - cost[j]
            cnt += 1
            j = (j + 1) % n
            while s < 0 and cnt < n:
                i -= 1
                s += gas[i] - cost[i]
                cnt += 1
        return -1 if s < 0 else i
```

[*] Greedy — Pattern Solution (stored optimal)

Python

```python
# Define a function to find starting gas station index
def canCompleteCircuit(gas, cost):
    total_surplus = 0 # Total gas surplus after completing the route
    current_surplus = 0 # Gas surplus while iterating from a candidate start station
    start_index = 0 # Candidate starting station index

    # Loop through each station
    for i in range(len(gas)):
        # Calculate net gas after leaving station i
        surplus = gas[i] - cost[i]
        total_surplus += surplus
        current_surplus += surplus

        # If current surplus becomes negative, station i is not a valid start.
        if current_surplus < 0:
            # Reset starting station to the next station
            start_index = i + 1
            current_surplus = 0 # Reset current surplus
    # Check if overall surplus is non-negative
    return start_index if total_surplus >= 0 else -1

# Example usage:
gas = [1,2,3,4,5]
cost = [3,4,5,1,2]
print(canCompleteCircuit(gas, cost)) # Output: 3
```

#179 MEDIUM

Largest Number

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · String · Greedy · Sorting

Lists: Grind 169

Problem:

Given a list of non-negative integers nums, arrange them such that they form the largest number and return it.

Since the result may be very large, so you need to return a string instead of an integer.

Example 1:

```
Input: nums = [10,2]
Output: "210"
```

Example 2:

```
Input: nums = [3,30,34,5,9]
Output: "9534330"
```

Constraints:

- 1 <= nums.length <= 100
- 0 <= nums[i] <= 109

>> Brute Force [Greedy] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def compare(self, x, y):
        # Brute Force O(n! * n) - try all orderings, pick largest concatenation
        from itertools import permutations
        best = ''
        for perm in permutations(map(str, nums)):
            candidate = ''.join(perm)
            if candidate > best: best = candidate
        return best.lstrip('0') or '0'
```

Python

Python


```python
# Import cmp_to_key to convert a comparator function to a key function for sorting
from functools import cmp_to_key

# Define the comparator function
def compare(x, y):
    # Compare two possible orders: x+y and y+x
    if x + y > y + x:
        return -1 # x should come before y
    elif x + y < y + x:
        return 1 # y should come before x
    else:
        return 0 # both orders are equal

def largestNumber(nums):
    # Convert all integers to strings
    nums_str = list(map(str, nums))
    # Sort the strings using the custom comparator
    nums_str.sort(key=cmp_to_key(compare))
    # If the largest number is '0', the entire number is 0 so return '0'
    if nums_str[0] == "0":
        return "0"
    # Concatenate the sorted numbers and return the result
    return "".join(nums_str)

# Example usage:

print(largestNumber([3, 30, 34, 5, 9])) # Output: "9534330"
```

Python

```python
class LargerStrKey(str):
    def __lt__(x: str, y: str) -> bool:
        return x + y > y + x


class Solution:
    def largestNumber(self, nums: list[int]) -> str:
        return ''.join(sorted(map(str, nums), key=LargerStrKey)).lstrip('0') or '0'
```

Python

```python
class Solution:
    def largestNumber(self, nums: List[int]) -> str:
        nums = [str(v) for v in nums]
        nums.sort(key=cmp_to_key(lambda a, b: 1 if a + b < b + a else -1))
        return "0" if nums[0] == "0" else "".join(nums)
```

Python

```python
class Solution:
    def largestNumber(self, nums: List[int]) -> str:
        nums = [str(v) for v in nums]
        nums.sort(key=cmp_to_key(lambda a, b: 1 if a + b < b + a else -1))
        return "0" if nums[0] == "0" else "".join(nums)
```

[*] Greedy — Pattern Solution (matched from community answers)

Python


```python
class LargerStrKey(str):
    def __lt__(x: str, y: str) -> bool:
        return x + y > y + x


class Solution:
    def largestNumber(self, nums: list[int]) -> str:
        return ''.join(sorted(map(str, nums), key=LargerStrKey)).lstrip('0') or '0'
```

#280 MEDIUM

Wiggle Sort

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Greedy · Sorting

Lists: Premium 98

Problem:

Given an integer array nums, reorder it such that nums[0] <= nums[1] >= nums[2] <= nums[3]....

You may assume the input array always has a valid answer.

Example 1:

```
Input: nums = [3,5,2,1,6,4]
Output: [3,5,1,6,2,4]
Explanation: [1,6,2,5,3,4] is also accepted.
```

Example 2:

```
Input: nums = [6,6,5,6,3,8]
Output: [6,6,5,6,3,8]
```

Constraints:

- 1 <= nums.length <= 5 * 104
- 0 <= nums[i] <= 104
- It is guaranteed that there will be an answer for the given input nums.

Follow up: Could you solve the problem in O(n) time complexity?

>> Brute Force [Greedy] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def wiggleSort(self, nums):
        # Brute Force O(n^2) - check all subsets/orderings exhaustively
        from itertools import permutations
        best = float('inf')
        for perm in permutations(nums):
            cost = sum(perm)
            best = min(best, cost)
        return best
```

Python

Python

```python
# Python solution for Wiggle Sort
def wiggleSort(nums):
    # Iterate over the list starting from index 1
    for i in range(1, len(nums)):
        # Check the condition based on the parity of the index
        if i % 2 == 1:
            # For odd index, if previous element is greater, swap them
            if nums[i] < nums[i-1]:
                nums[i], nums[i-1] = nums[i-1], nums[i]
        else:
            # For even index, if previous element is smaller, swap them
            if nums[i] > nums[i-1]:
                nums[i], nums[i-1] = nums[i-1], nums[i]
    return nums

# Example usage:
# nums = [3,5,2,1,6,4]
# print(wiggleSort(nums))
```

Python

```python
class Solution:
   def wiggleSort(self, nums: list[int]) -> None:
      # 1. If i is even, then nums[i] <= nums[i - 1].
      # 2. If i is odd, then nums[i] >= nums[i - 1].
      for i in range(1, len(nums)):
        if (i % 2 == 0 and nums[i] > nums[i - 1] or
            i % 2 == 1 and nums[i] < nums[i - 1]):
            nums[i], nums[i - 1] = nums[i - 1], nums[i]
```

Python

```python
class Solution:
    def wiggleSort(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        for i in range(1, len(nums)):
            if (i % 2 == 1 and nums[i] < nums[i - 1]) or (
                i % 2 == 0 and nums[i] > nums[i - 1]
            ):
                nums[i], nums[i - 1] = nums[i - 1], nums[i]
```

Python


```python
'''
math prove:

example [3, 5, 2, 1]
when reaching 2, all good
then next, is 1,
so here comes the question, will the number after 2 violating the wiggle rule?
answer is no.

in example, when reaching 2:
+ if [3, 5, 2, 10], so 10 is bigger than 2, then no swap needed according to the if () conditions

+ if [3, 5, 2, 1], so previous round, guaranteed that 2 is smaller than 5.
    And, now a swap needed meaning the number is smaller than 2, i.e. must be smaller than 5, so wiggle
rule is still good

'''
class Solution:
    def wiggleSort(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        for i in range(1, len(nums)):
            if (i % 2 == 1 and nums[i] < nums[i - 1]) or (
                i % 2 == 0 and nums[i] > nums[i - 1]
            ):
                nums[i], nums[i - 1] = nums[i - 1], nums[i]

###########

class Solution: # sort list first
    def wiggleSort(self, nums: List[int]) -> None:
        nums.sort() # Sort the list in ascending order
        n = len(nums)

        for i in range(1, n - 1, 2):
            nums[i], nums[i + 1] = nums[i + 1], nums[i] # Swap adjacent elements

        """
        If the list has an odd length, the last element will be compared with the second-to-last element
        and swapped if necessary to form a wiggle sequence.
        """
        if n % 2 == 0 or nums[-1] < nums[-2]:
            nums[-1], nums[-2] = nums[-2], nums[-1]

###########
import random

class Solution:
    def wiggleSort(self, nums: List[int]) -> None:
        """
```



```python
        Do not return anything, modify nums in-place instead.
        """
        if len(nums) <= 1:
            return

        # Find the median using quickselect
        n = len(nums)
        mid = self.quickselect(nums, 0, n-1, (n-1)//2)

        # Create an empty array with the same length as nums
        ans = [None] * n

        # Assign the median to every other element
        i = 0
        j = n-1 if n % 2 == 1 else n-2
        for k in range(n):
            if nums[k] < mid:
                ans[i] = nums[k]
                i += 2
            elif nums[k] > mid:
                ans[i] = nums[k]
                i += 2
            else:
                ans[j+1] = mid


        # Copy the result back to nums
        nums[:] = ans

    def quickselect(self, nums, left, right, k):
        """
```

[*] Greedy — Pattern Solution (matched from community answers)

Python


```python
'''
math prove:

example [3, 5, 2, 1]
when reaching 2, all good
then next, is 1,
so here comes the question, will the number after 2 violating the wiggle rule?
answer is no.

in example, when reaching 2:
+ if [3, 5, 2, 10], so 10 is bigger than 2, then no swap needed according to the if () conditions

+ if [3, 5, 2, 1], so previous round, guaranteed that 2 is smaller than 5.
    And, now a swap needed meaning the number is smaller than 2, i.e. must be smaller than 5, so wiggle
rule is still good

'''
class Solution:
    def wiggleSort(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        for i in range(1, len(nums)):
            if (i % 2 == 1 and nums[i] < nums[i - 1]) or (
                i % 2 == 0 and nums[i] > nums[i - 1]
            ):
                nums[i], nums[i - 1] = nums[i - 1], nums[i]

###########

class Solution: # sort list first
    def wiggleSort(self, nums: List[int]) -> None:
        nums.sort() # Sort the list in ascending order
        n = len(nums)

        for i in range(1, n - 1, 2):
            nums[i], nums[i + 1] = nums[i + 1], nums[i] # Swap adjacent elements

        """
        If the list has an odd length, the last element will be compared with the second-to-last element
        and swapped if necessary to form a wiggle sequence.
        """
        if n % 2 == 0 or nums[-1] < nums[-2]:
            nums[-1], nums[-2] = nums[-2], nums[-1]

###########
import random

class Solution:
    def wiggleSort(self, nums: List[int]) -> None:
        """
```



```python
        Do not return anything, modify nums in-place instead.
        """
        if len(nums) <= 1:
            return

        # Find the median using quickselect
        n = len(nums)
        mid = self.quickselect(nums, 0, n-1, (n-1)//2)

        # Create an empty array with the same length as nums
        ans = [None] * n

        # Assign the median to every other element
        i = 0
        j = n-1 if n % 2 == 1 else n-2
        for k in range(n):
            if nums[k] < mid:
                ans[i] = nums[k]
                i += 2
            elif nums[k] > mid:
                ans[i] = nums[k]
                i += 2
            else:
                ans[j+1] = mid


        # Copy the result back to nums
        nums[:] = ans

    def quickselect(self, nums, left, right, k):
        """
```

#954 MEDIUM

Array of Doubled Pairs

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Greedy · Sorting

Lists: CodeSignal

Problem:

Given an integer array of even length arr, return true if it is possible to reorder arr such that arr[2 * i + 1] = 2 * arr[2 * i] for every 0 <= i < len(arr) / 2, or false otherwise.

Example 1:

```
Input: arr = [3,1,3,6]
Output: false
```

Example 2:

```
Input: arr = [2,1,2,6]
Output: false
```

Example 3:

```
Input: arr = [4,-2,2,-4]
Output: true
```



```
Explanation: We can take two groups, [-2,-4] and [2,4] to form [-2,-4,2,4] or [2,4,-2,-4].
```

Constraints:

- 2 <= arr.length <= 3 * 104
- arr.length is even.
- -105 <= arr[i] <= 105

>> Brute Force [Greedy] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def canReorderDoubled(self, arr):
        # Brute Force O(n!) - check all subsets/orderings exhaustively
        from itertools import permutations
        best = float('inf')
        for perm in permutations(arr):
            cost = sum(perm)
            best = min(best, cost)
        return best
```

Python

Python

```python
from collections import Counter

def canReorderDoubled(arr):
    # Count frequency for each element in arr
    count = Counter(arr)
    # Sort the array based on the absolute value of elements
    for x in sorted(arr, key=abs):
        # If no, x is left to be paired, skip to the next element
        if count[x] == 0:
            continue
        # Check if double the current number exists in sufficient frequency
        if count[2 * x] < count[x]:
            return False
        # Decrement the frequency for the double of x
        count[2 * x] -= count[x]
    return True

# Example usage:
print(canReorderDoubled([4,-2,2,-4])) # Expected Output: True
```

Python

```python
class Solution:
   def canReorderDoubled(self, arr: list[int]) -> bool:
      count = collections.Counter(arr)

      for key in sorted(count, key=abs):
        if count[key] > count[2 * key]:
          return False
        count[2 * key] -= count[key]

      return True
```

Python

```python
class Solution:
    def canReorderDoubled(self, arr: List[int]) -> bool:
        freq = Counter(arr)
        if freq[0] & 1:
            return False
        for x in sorted(freq, key=abs):
            if freq[x << 1] < freq[x]:
                return False
            freq[x << 1] -= freq[x]
        return True
```

Python

```python
class Solution:
    def canReorderDoubled(self, arr: List[int]) -> bool:
        freq = Counter(arr)
        if freq[0] & 1:
            return False
        for x in sorted(freq, key=abs):
            if freq[x << 1] < freq[x]:
                return False
            freq[x << 1] -= freq[x]
        return True
```

[*] Greedy — Pattern Solution (matched from community answers)

Python

```python
from collections import Counter

def canReorderDoubled(arr):
    # Count frequency for each element in arr
    count = Counter(arr)
    # Sort the array based on the absolute value of elements
    for x in sorted(arr, key=abs):
        # If no x is left to be paired, skip to the next element
        if count[x] == 0:
            continue
        # Check if double the current number exists in sufficient frequency
        if count[2 * x] < count[x]:
            return False
        # Decrement the frequency for the double of x
        count[2 * x] -= count[x]
    return True

# Example usage:
print(canReorderDoubled([4,-2,2,-4])) # Expected Output: True
```

#2131 MEDIUM

Longest Palindrome by Concatenating Two Letter Words

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · String · Greedy · Counting

Lists: CodeSignal

Problem:

You are given an array of strings words. Each element of words consists of two lowercase English letters.

Create the longest possible palindrome by selecting some elements from words and concatenating them in any order. Each element can be selected at most once.

Return the length of the longest palindrome that you can create. If it is impossible to create any palindrome, return 0.

A palindrome is a string that reads the same forward and backward.

Example 1:

```
Input: words = ["lc","cl","gg"]
Output: 6
Explanation: One longest palindrome is "lc" + "gg" + "cl" = "lcggcl", of length 6.
Note that "clgglc" is another longest palindrome that can be created.
```

Example 2:

```
Input: words = ["ab","ty","yt","lc","cl","ab"]
Output: 8
Explanation: One longest palindrome is "ty" + "lc" + "cl" + "yt" = "tylcclyt", of length 8.
Note that "lcyttycl" is another longest palindrome that can be created.
```

Example 3:

```
Input: words = ["cc","ll","xx"]
Output: 2
Explanation: One longest palindrome is "cc", of length 2.
Note that "ll" is another longest palindrome that can be created, and so is "xx".
```

Constraints:

- 1 <= words.length <= 105
- words[i].length == 2
- words[i] consists of lowercase English letters.

>> Brute Force [Greedy] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def longestPalindrome(self, words):
        # Brute-force [Greedy] - check all subsets/orderings exhaustively
        from itertools import permutations
        best = float('inf')
        for perm in permutations(words):
            cost = sum(perm)
            best = min(best, cost)
        return best
```

Python

Python


```python
# Python solution
from collections import Counter

class Solution:
    def longestPalindrome(self, words):
        # Count the frequency of each word
        word_count = Counter(words)
        length = 0 # Total words used count (each adds 2 letters)
        center_used = False

        for word, count in list(word_count.items()):
            reverse = word[::-1] # Get the reverse of the current word

            if word == reverse:
                # For words that are palindromic themselves (e.g. "gg")
                pairs = count // 2
                length += pairs * 2 # Each pair adds 2 words (4 letters)
                # If an extra word exists and center is not used, add one word in the center
                if count % 2 == 1 and not center_used:
                    length += 1
                    center_used = True
            else:
                # For words that are not self-palindromic.
                if reverse in word_count:
                    # We only count each pair once, so use minimum and then set the reverse count to 0.

                    pair_count = min(count, word_count[reverse])
                    length += pair_count * 2 # Each pair contributes two words
                    # Set reverse count to 0 to avoid double counting later.
                    word_count[reverse] = 0
        return length + 2 # Each word is of length 2

# Example usage:
# sol = Solution()
# print(sol.longestPalindrome(["lc", "cl", "gg"])) # Output: 6
```

Python

```python
class Solution:
    def longestPalindrome(self, words: list[str]) -> int:
        ans = 0
        cnt = [[0] * 26 for _ in range(26)]

        for a, b in words:
            i = ord(a) - ord('a')
            j = ord(b) - ord('a')
            if cnt[j][i]:
                ans += 4
                cnt[j][i] -= 1
            else:
                cnt[i][j] += 1

        for i in range(26):
            if cnt[i][i]:
                return ans + 2

        return ans
```

Python


```python
class Solution:
    def longestPalindrome(self, words: List[str]) -> int:
        cnt = Counter(words)
        ans = x = 0
        for k, v in cnt.items():
            if k[0] == k[1]:
                x += v & 1
                ans += v // 2 * 2 * 2
            else:
                ans += min(v, cnt[k[::-1]]) * 2
        ans += 2 if x else 0
        return ans
```

Python

```python
class Solution:
    def longestPalindrome(self, words: List[str]) -> int:
        cnt = Counter(words)
        ans = x = 0
        for k, v in cnt.items():
            if k[0] == k[1]:
                x += v & 1
                ans += v // 2 * 2 * 2
            else:
                ans += min(v, cnt[k[::-1]]) * 2
        ans += 2 if x else 0
        return ans
```

[*] Greedy — Pattern Solution (stored optimal)

Python

```python
# Python solution
from collections import Counter

class Solution:
    def longestPalindrome(self, words):
        # Count the frequency of each word
        word_count = Counter(words)
        length = 0 # Total words used count (each adds 2 letters)
        center_used = False

        for word, count in list(word_count.items()):
            reverse = word[::-1] # Get the reverse of the current word

            if word == reverse:
                # For words that are palindromic themselves (e.g. "gg")
                pairs = count // 2
                length += pairs * 2 # Each pair adds 2 words (4 letters)
                # If an extra word exists and center is not used, add one word in the center
                if count % 2 == 1 and not center_used:
                    length += 1
                    center_used = True
            else:
                # For words that are not self-palindromic.
                if reverse in word_count:
                    # We only count each pair once, so use minimum and then set the reverse count to 0.
```



```python
                    pair_count = min(count, word_count[reverse])
                    length += pair_count * 2 # Each pair contributes two words
                    # Set reverse count to 0 to avoid double counting later.
                    word_count[reverse] = 0
        return length + 2 # Each word is of length 2

# Example usage:
# sol = Solution()
# print(sol.longestPalindrome(["lc", "cl", "gg"])) # Output: 6
```


Quick Review — Greedy 6 questions · insights · complexity · solution

#409 Longest Palindrome [Easy]

Key Insights

- Count the frequency of each character.
- For each character, you can use pairs (even count or odd count minus one) to form the palindrome.
- If any character has an odd frequency, you can add one extra character as the center of the palindrome.

Space & Time

Time Complexity: O(n), where n is the length of the string.

Space Complexity: O(1) since the frequency counter will have at most 52 keys (considering both uppercase and lowercase letters).

Solution

The solution uses a hash table to count the frequency of each character. For every character, the maximum number of characters that can be used in a palindrome is determined by taking the largest even number that is less than or equal to its frequency (which is frequency - (frequency % 2)). If there exists any character with an odd frequency, one of these can be used as the center of the palindrome, adding an extra character to the total length. This greedy approach efficiently computes the length of the largest possible palindrome from the available characters.

#134 Gas Station [Medium]

Key Insights

- If the total amount of gas available is less than the total travel cost, then completing the circuit is impossible.
- A greedy approach can be used by keeping track of the current surplus of gas. If the surplus becomes negative at any station, the journey cannot start from any station before that point.
- Restart the journey from the next station whenever the surplus goes negative.
- The problem guarantees that if a solution exists, it is unique.

Space & Time

Time Complexity: O(n), where n is the number of gas stations, as we traverse the list only once.

Space Complexity: O(1), as no extra space is used other than a few variables.

Solution

The solution involves iterating over the list of stations while maintaining two variables: one for the current gas surplus and one for the total gas surplus. If at any station the current surplus drops below zero, the current starting station is not viable and we set the starting station to the next index and reset the current surplus. After one pass, if the total gas surplus is non-negative, then the identified station is the valid starting point.

#179 Largest Number [Medium]

Key Insights

- The order in which numbers appear drastically affects the concatenated result.
- A custom sorting strategy is required where two numbers are compared by checking which concatenation (first-second vs second-first) yields a larger number.
- Edge case: When the numbers are all zeros, the result should be "0" instead of several leading zeros.

Space & Time

Time Complexity: O(n log n * k), where n is the number of elements in the array and k is the maximum number of digits in a number (due to string comparisons in the custom sort).

Space Complexity: O(n * k) for storing and processing the strings.

Solution

We solve the problem by converting all integers to strings so that we can use string concatenation to compare the results. A custom comparator is defined that, given two number strings, compares the concatenated results in both orders. By sorting the array using this comparator in descending order, we ensure that placing the highest contributing number first yields the largest possible number. After sorting, a check is performed to return "0" if the highest number is "0". Finally, the sorted strings are concatenated to form the answer.

#280 Wiggle Sort [Medium]

Key Insights

- The alternating "wiggle" requirement can be satisfied by ensuring two conditions:
- For every odd index i, nums[i] should be greater than or equal to nums[i-1].
- For every even index i (where i > 0), nums[i] should be less than or equal to nums[i-1].
- A single pass through the array can ensure these conditions by swapping adjacent elements when the condition is violated.
- This greedy approach works because the local adjustments guarantee the overall wiggle pattern without needing to sort the entire array.

Space & Time

Time Complexity: O(n) — Only one pass through the array is needed.

Space Complexity: O(1) — The modifications are done in-place without extra data structures.

Solution

We can achieve a wiggle sort in one pass by iterating through the array and, at each index i (starting from 1), checking if the pair (nums[i-1], nums[i]) satisfies the wiggle condition. If i is odd, nums[i] should be at least nums[i-1]; if not, we swap them. Conversely, if i is even and nums[i] is greater than nums[i-1], then we swap them. These local swaps adjust the order incrementally while ensuring that the overall wiggle pattern is maintained. This greedy approach works because only two adjacent numbers are involved in any violation and fixing them locally does not disturb the rest of the order.

#954 Array of Doubled Pairs [Medium]

Key Insights

- Sort the array based on the absolute value of the elements to handle negative numbers correctly.
- Use a hash table (or frequency counter) to track the occurrence of each element.
- For each number in the sorted list, if the number is still available in the frequency map, try to pair it with its double.
- Decrement the appropriate counts and validate that it is possible to find sufficient doubles for all numbers.
- Special case: when dealing with zeros, pairing works within the same value.

Space & Time

Time Complexity: O(n log n) due to sorting.

Space Complexity: O(n) for storing the frequency map.

Solution

The solution starts by counting the frequency of each element in the array. Sorting the array by absolute value ensures that when we process an element, any potential pair (i.e., its double) comes later in the sequence. For each number, if there are available occurrences, we try to match it with its double. If there are not enough doubles available, the answer is false. If all elements can be paired appropriately by decrementing their counts in the frequency map, the function returns true.

#2131 Longest Palindrome by Concatenating Two Letter Words [Medium]

Key Insights

- Pair words with their reverse (e.g., "ab" with "ba") to form a palindrome.

- Handle words that are palindromic by themselves (like "gg" or "aa") differently; they can be used in pairs and one extra can be placed in the center.
- Use a hash map (or dictionary) to count frequencies of each word.
- For non-palindromic word pairs, consider each pair only once to avoid double counting.
- For self-palindromic words, count how many pairs can be used and whether an extra word can be placed in the middle to maximize length.

Space & Time

Time Complexity: O(n), where n is the number of words, since we iterate through the list and perform constant time operations per word.

Space Complexity: O(n), to store counts of words in the hash map.

Solution

The approach is to count the occurrences of each word using a hash map. For each word:

- If the word is not self-palindromic, check if its reverse exists and form pairs by considering the minimum count between the word and its reverse. - If the word is self-palindromic, form as many pairs as possible (using count // 2) and note if there is any word left; you can use one extra self-palindromic word as the center of the palindrome. Finally, compute the total length by multiplying the number of words used by 2, since each word contributes 2 characters.

#3 JavaScript — 7 questions · #3 of 21

#2627 MEDIUM

Debounce

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

JavaScript

Lists: Premium 98

Problem:

Given a function fn and a time in milliseconds t, return a debounced version of that function.

A debounced function is a function whose execution is delayed by t milliseconds and whose execution is cancelled if it is called again within that window of time. The debounced function should also receive the passed parameters.

For example, let's say t = 50ms, and the function was called at 30ms, 60ms, and 100ms.

The first 2 function calls would be cancelled, and the 3rd function call would be executed at 150ms.

If instead t = 35ms, The 1st call would be cancelled, the 2nd would be executed at 95ms, and the 3rd would be executed at 135ms.

The above diagram shows how debounce will transform events. Each rectangle represents 100ms and the debounce time is 400ms. Each color represents a different set of inputs.

Please solve it without using lodash's _.debounce() function.

Example 1:

```
Input:
t = 50
calls = [
  {"t": 50, inputs: [1]},
  {"t": 75, inputs: [2]}
]
Output: [{"t": 125, inputs: [2]}]
Explanation:
let start = Date.now();
function log(...inputs) {
  console.log([Date.now() - start, inputs ])
}
const dlog = debounce(log, 50);
setTimeout(() => dlog(1), 50);
setTimeout(() => dlog(2), 75);

The 1st call is cancelled by the 2nd call because the 2nd call occurred before 100ms
The 2nd call is delayed by 50ms and executed at 125ms. The inputs were (2).
```

Example 2:

```
Input:
t = 20
calls = [
```



```
  {"t": 50, inputs: [1]},
  {"t": 100, inputs: [2]}
]
Output: [{"t": 70, inputs: [1]}, {"t": 120, inputs: [2]}]
Explanation:
The 1st call is delayed until 70ms. The inputs were (1).
The 2nd call is delayed until 120ms. The inputs were (2).
```

Example 3:

```
Input:
t = 150
calls = [
  {"t": 50, inputs: [1, 2]},
  {"t": 300, inputs: [3, 4]},
  {"t": 300, inputs: [5, 6]}
]
Output: [{"t": 200, inputs: [1,2]}, {"t": 450, inputs: [5, 6]}]
Explanation:
The 1st call is delayed by 150ms and ran at 200ms. The inputs were (1, 2).
The 2nd call is cancelled by the 3rd call
The 3rd call is delayed by 150ms and ran at 450ms. The inputs were (5, 6).
```

Constraints:

- 0 <= t <= 1000
- 1 <= calls.length <= 10
- 0 <= calls[i].t <= 1000
- 0 <= calls[i].inputs.length <= 10

TypeScript

TypeScript

```tsx
type F = (...args: number[]) => void;

function debounce(fn: F, t: number): F {
  let timeout: ReturnType<typeof setTimeout> | undefined;
  return function (...args) {
    clearTimeout(timeout);
    timeout = setTimeout(() => fn(...args), t);
  };
}
```

TypeScript


```tsx
type F = (...p: any[]) => any;

function debounce(fn: F, t: number): F {
    let timeout: ReturnType<typeof setTimeout> | undefined;

    return function (...args) {
        if (timeout !== undefined) {
            clearTimeout(timeout);
        }
        timeout = setTimeout(() => {
            fn.apply(this, args);
        }, t);
    };
}

/**
 * const log = debounce(console.log, 100);
 * log('Hello'); // cancelled
 * log('Hello'); // cancelled
 * log('Hello'); // Logged at t=100ms
 */
```

TypeScript

```tsx
type F = (...p: any[]) => any;

function debounce(fn: F, t: number): F {
    let timeout: ReturnType<typeof setTimeout> | undefined;

    return function (...args) {
        if (timeout !== undefined) {
            clearTimeout(timeout);
        }
        timeout = setTimeout(() => {
            fn.apply(this, args);
        }, t);
    };
}

/**
 * const log = debounce(console.log, 100);
 * log('Hello'); // cancelled
 * log('Hello'); // cancelled
 * log('Hello'); // Logged at t=100ms
 */
```

[*] JavaScript — Pattern Solution (stored optimal)

Python


```python
import threading

def debounce(fn, t):
    # Timer reference to manage the scheduled execution
    timer = None

    def debounced(*args, **kwargs):
        nonlocal timer
        # Cancel previous timer if it exists
        if timer is not None:
            timer.cancel()
        # Start a new timer (convert milliseconds to seconds)
        timer = threading.Timer(t / 1000.0, lambda: fn(*args, **kwargs))
        timer.start()

    return debounced

# Example usage:
if __name__ == "__main__":
    import time
    start = time.time()

    def log(*args):
        print((time.time() - start, args))

    dlog = debounce(log, 50)
    dlog(1)
    time.sleep(0.025) # 25ms delay before the next call
    dlog(2)
    time.sleep(0.1) # Wait to allow the debounced function to execute
```

#2628 MEDIUM

JSON Deep Equal

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

JavaScript

Lists: Premium 98

Problem:

Given two values o1 and o2, return a boolean value indicating whether two values, o1 and o2, are deeply equal.

For two values to be deeply equal, the following conditions must be met:

- If both values are primitive types, they are deeply equal if they pass the === equality check.
- If both values are arrays, they are deeply equal if they have the same elements in the same order, and each element is also deeply equal according to these conditions.
- If both values are objects, they are deeply equal if they have the same keys, and the associated values for each key are also deeply equal according to these conditions.

You may assume both values are the output of JSON.parse. In other words, they are valid JSON.

Please solve it without using lodash's _.isEqual() function

Example 1:

```
Input: o1 = {"x":1,"y":2}, o2 = {"x":1,"y":2}
Output: true
Explanation: The keys and values match exactly.
```

Example 2:

```
Input: o1 = {"y":2,"x":1}, o2 = {"x":1,"y":2}
Output: true
Explanation: Although the keys are in a different order, they still match exactly.
```

Example 3:

```
Input: o1 = {"x":null,"L":[1,2,3]}, o2 = {"x":null,"L":["1","2","3"]}
Output: false
Explanation: The array of numbers is different from the array of strings.
```

Example 4:

```
Input: o1 = true, o2 = false
Output: false
Explanation: true !== false
```

Constraints:

- 1 <= JSON.stringify(o1).length <= 105
- 1 <= JSON.stringify(o2).length <= 105
- maxNestingDepth <= 1000

TypeScript

TypeScript

```typescript
type JSONValue =
  | null
  | boolean
  | number
  | string
  | JSONValue[]
  | { [key: string]: JSONValue };

function areDeeplyEqual(o1: JSONValue, o2: JSONValue): boolean {
  if (o1 === o2) {
    return true;
  }
  if (o1 === null || o1 === undefined || o2 === null || o2 === undefined) {
    return false;
  }
  if (typeof o1 !== 'object' || typeof o2 !== 'object') {
    return false;
  }
  if (Array.isArray(o1) != Array.isArray(o2)) {
    return false;
  }
  if (Object.keys(o1).length != Object.keys(o2).length) {
    return false;
  }
  for (const key in o1) {
```



```
    if (!areDeeplyEqual(o1[key], o2[key])) {
      return false;
    }
  }
  return true;
}
```

[*] JavaScript — Pattern Solution (stored optimal)

Python

```python
def deep_equal(o1, o2):
    # If both are exactly equal, return True
    if o1 == o2:
        return True
    # If types differ, they are not equal
    if type(o1) != type(o2):
        return False
    # If both are lists, check length and recursively compare each element
    if isinstance(o1, list):
        if len(o1) != len(o2):
            return False
        for item1, item2 in zip(o1, o2):
            if not deep_equal(item1, item2):
                return False
        return True
    # If both are dictionaries, check that they have the same keys and values
    if isinstance(o1, dict):
        if set(o1.keys()) != set(o2.keys()):
            return False
        for key in o1:
            if not deep_equal(o1[key], o2[key]):
                return False
        return True
    # Fallback for other types (should not reach here for valid JSON)
    return False


# Example usage:
# print(deep_equal({"a": 1, "y": 2}, {"y": 2, "x": 1}))
```

#2630 MEDIUM

Memoize

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

JavaScript

Lists: Premium 98

Problem:

Given a function fn, return a memoized version of that function.

A memoized function is a function that will never be called twice with the same inputs. Instead it will return a cached value.

fn can be any function and there are no constraints on what type of values it accepts. Inputs are considered identical if they are === to each other.

Example 1:

```
Input:
getInputs = () => [[2,2],[2,2],[1,2]]
fn = function (a, b) { return a + b; }
Output: [{"val":4,"calls":1},{"val":4,"calls":1},{"val":3,"calls":2}]
Explanation:
const inputs = getInputs();
const memoized = memoize(fn);
for (const arr of inputs) {
memoized(...arr);
}

For the inputs of (2, 2): 2 + 2 = 4, and it required a call to fn().
For the inputs of (2, 2): 2 + 2 = 4, but those inputs were seen before so no call to fn() was
required.
For the inputs of (1, 2): 1 + 2 = 3, and it required another call to fn() for a total of 2.
```

Example 2:

```
Input:
getInputs = () => [[{},{}],[{},{}],[{},{}]]
fn = function (a, b) { return ({...a, ...b}); }
Output: [{"val":{},"calls":1},{"val":{},"calls":2},{"val":{},"calls":3}]
Explanation:
Merging two empty objects will always result in an empty object. It may seem like there
should only be 1 call to fn() because of cache-hits, however none of those objects are === to
each other.
```

Example 3:

```
Input:
getInputs = () => { const o = {}; return [[o,o],[o,o],[o,o]]; }
fn = function (a, b) { return ({...a, ...b}); }
Output: [{"val":{},"calls":1},{"val":{},"calls":1},{"val":{},"calls":1}]
Explanation:
Merging two empty objects will always result in an empty object. The 2nd and 3rd third
function calls result in a cache-hit. This is because every object passed in is identical.
```

Constraints:

- 1 <= inputs.length <= 105
- 0 <= inputs.flat().length <= 105
- inputs[i][j] != NaN

TypeScript

TypeScript


```typescript
type Fn = (...params: any) => any;

function memoize(fn: Fn): Fn {
    const idxMap: Map<string, number> = new Map();
    const cache: Map<string, any> = new Map();

    const getIdx = (obj: any): number => {
        if (!idxMap.has(obj)) {
            idxMap.set(obj, idxMap.size);
        }
        return idxMap.get(obj)!;
    };

    return function (...params: any) {
        const key = params.map(getIdx).join(',');
        if (!cache.has(key)) {
            cache.set(key, fn(...params));
        }
        return cache.get(key)!;
    };
}

/**
 * let callCount = 0;
 * const memoizedFn = memoize(function (a, b) {
```

```
 * callCount += 1;
 * return a + b;
 * })
 * memoizedFn(2, 3) // 5
 * memoizedFn(2, 3) // 5
 * console.log(callCount) // 1
 */
```

TypeScript


```typescript
type Fn = (...params: any) => any;

function memoize(fn: Fn): Fn {
    const idxMap: Map<string, number> = new Map();
    const cache: Map<string, any> = new Map();

    const getIdx = (obj: any): number => {
        if (!idxMap.has(obj)) {
            idxMap.set(obj, idxMap.size);
        }
        return idxMap.get(obj)!;
    };

    return function (...params: any) {
        const key = params.map(getIdx).join(',');
        if (!cache.has(key)) {
            cache.set(key, fn(...params));
        }
        return cache.get(key)!;
    };
}

/**
 * let callCount = 0;
 * const memoizedFn = memoize(function (a, b) {
 * callCount += 1;
 * return a + b;
 * })
 * memoizedFn(2, 3) // 5
 * memoizedFn(2, 3) // 5
 * console.log(callCount) // 1
 */
```


[*] JavaScript — Pattern Solution (stored optimal)

Python


```python
# Define a function 'memoize' that takes a function 'fn' and returns a memoized version.
def memoize(fn):
    cache = {} # Dictionary to cache computed values
    call_count = 0 # Counter for real function calls

    # The wrapper function handles any number of arguments.
    def wrapper(*args):
        nonlocal call_count
        # Use the arguments tuple as a key (order sensitive).
        if args not in cache:
            call_count += 1 # Increment call count when computing new result
            cache[args] = fn(*args)
        return cache[args]

    # Expose the getCallCount method on the memoized function.
    wrapper.getCallCount = lambda: call_count
    return wrapper

# Example usage:
# def sum_func(a, b): return a + b
# memoizedSum = memoize(sum_func)
# print(memoizedSum(2, 2))
# print(memoizedSum(2, 2))
# print(memoizedSum.getCallCount())
```

#2633 MEDIUM

Convert Object to JSON String

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

JavaScript

Lists: Premium 98

Problem:

Given a value, return a valid JSON string of that value. The value can be a string, number, array, object, boolean, or null. The returned string should not include extra spaces. The order of keys should be the same as the order returned by Object.keys().

Please solve it without using the built-in JSON.stringify method.

Example 1:

```
Input: object = {"y":1,"x":2}
Output: {"y":1,"x":2}
Explanation:
Return the JSON representation.
Note that the order of keys should be the same as the order returned by Object.keys().
```

Example 2:

```
Input: object = {"a":"str","b":-12,"c":true,"d":null}
Output: {"a":"str","b":-12,"c":true,"d":null}
Explanation:
The primitives of JSON are strings, numbers, booleans, and null.
```

Example 3:

```
Input: object = {"key":{"a":1,"b":[{},null,"Hello"]}}
Output: {"key":{"a":1,"b":[{},null,"Hello"]}}
Explanation:
Objects and arrays can include other objects and arrays.
```

Example 4:

```
Input: object = true
Output: true
Explanation:
Primitive types are valid inputs.
```

Constraints:

- value is a valid JSON value
- 1 <= JSON.stringify(object).length <= 105
- maxNestingLevel <= 1000
- all strings contain only alphanumeric characters

TypeScript

TypeScript

```typescript
type JSONValue =
  | null
  | boolean
  | number
  | string
  | JSONValue[]
  | { [key: string]: JSONValue };

function jsonStringify(object: JSONValue): string {
  if (object === null) {
    return 'null';
  }
  if (typeof object === 'boolean' || typeof object === 'number') {
    return String(object);
  }
  if (typeof object === 'string') {
    return `"${object}"`;
  }
  if (Array.isArray(object)) {
    const elems = object.map((elem) => jsonStringify(elem));
    return `[${elems.join(',')}]`;
  }
  const pairs = Object.keys(object).map(
    (key) => `"${key}":${jsonStringify(object[key])}`
  );

  return `{${pairs.join(',')}}`;
}
```

[*] JavaScript — Pattern Solution (stored optimal)

Python


```
def myStringify(value):
    # If value is None, return "null"
    if value is None:
        return "null"
    # If value is a boolean, convert to "true" or "false"
    if isinstance(value, bool):
        return "true" if value else "false"
    # If value is a number, return its string representation
    if isinstance(value, (int, float)):
        return str(value)
    # If value is a string, enclose it in double quotes
    if isinstance(value, str):
        return '"' + value + '"'
    # If value is a list, recursively process its elements
    if isinstance(value, list):
        return '[' + ','.join(myStringify(item) for item in value) + ']'
    # If value is a dictionary, iterate keys in the order of insertion
    if isinstance(value, dict):
        items = []
        for key in value:
            items.append('"' + key + '":' + myStringify(value[key]))
        return '{' + ','.join(items) + '}'
    # Fallback (should not be reached with valid JSON types)
    return ""

# Example usage:
test_obj = {"y": 1, "x": 2}
print(myStringify(test_obj))
```

#2636 MEDIUM

Promise Pool

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

JavaScript · Concurrency

Lists: Premium 98

Problem:

Given an array of asynchronous functions functions and a pool limit n, return an asynchronous function promisePool. It should return a promise that resolves when all the input functions resolve.

Pool limit is defined as the maximum number promises that can be pending at once. promisePool should begin execution of as many functions as possible and continue executing new functions when old promises resolve. promisePool should execute functions[i] then functions[i + 1] then functions[i + 2], etc. When the last promise resolves, promisePool should also resolve.

For example, if n = 1, promisePool will execute one function at a time in series. However, if n = 2, it first executes two functions. When either of the two functions resolve, a 3rd function should be executed (if available), and so on until there are no functions left to execute.

You can assume all functions never reject. It is acceptable for promisePool to return a promise that resolves any value.

Example 1:


```
Input:
functions = [
  () => new Promise(res => setTimeout(res, 300)),
  () => new Promise(res => setTimeout(res, 400)),
  () => new Promise(res => setTimeout(res, 200))
]
n = 2
Output: [[300,400,500],500]
Explanation:
Three functions are passed in. They sleep for 300ms, 400ms, and 200ms respectively.
They resolve at 300ms, 400ms, and 500ms respectively. The returned promise resolves at 500ms.
At t=0, the first 2 functions are executed. The pool size limit of 2 is reached.
At t=300, the 1st function resolves, and the 3rd function is executed. Pool size is 2.
At t=400, the 2nd function resolves. There is nothing left to execute. Pool size is 1.
At t=500, the 3rd function resolves. Pool size is zero so the returned promise also resolves.
```

Example 2:

```
Input:
functions = [
  () => new Promise(res => setTimeout(res, 300)),
  () => new Promise(res => setTimeout(res, 400)),
  () => new Promise(res => setTimeout(res, 200))
]
n = 5
Output: [[300,400,200],400]
Explanation:
The three input promises resolve at 300ms, 400ms, and 200ms respectively.
The returned promise resolves at 400ms.
At t=0, all 3 functions are executed. The pool limit of 5 is never met.
At t=200, the 3rd function resolves. Pool size is 2.
At t=300, the 1st function resolves. Pool size is 1.
At t=400, the 2nd function resolves. Pool size is 0, so the returned promise also resolves.
```

Example 3:

```
Input:
functions = [
  () => new Promise(res => setTimeout(res, 300)),
  () => new Promise(res => setTimeout(res, 400)),
  () => new Promise(res => setTimeout(res, 200))
]
n = 1
Output: [[300,700,900],900]
Explanation:
The three input promises resolve at 300ms, 700ms, and 900ms respectively.
The returned promise resolves at 900ms.
At t=0, the 1st function is executed. Pool size is 1.
At t=300, the 1st function resolves and the 2nd function is executed. Pool size is 1.
At t=700, the 2nd function resolves and the 3rd function is executed. Pool size is 1.
At t=900, the 3rd function resolves. Pool size is 0 so the returned promise resolves.
```

Constraints:

- 0 <= functions.length <= 10

- 1 <= n <= 10

TypeScript

TypeScript

```typescript
type F = () => Promise<any>;

function promisePool(functions: F[], n: number): Promise<any> {
  const next = (i =>functions[i++?i:0].then(next);
  return Promise.all(functions.slice(0, n).map(f) => f().then(next)));
}
```

TypeScript

```typescript
type F = () => Promise<any>;

function promisePool(functions: F[], n: number): Promise<any> {
    const wrappers = functions.map(fn => async () => {
        await fn();
        const nxt = waiting.shift();
        nxt && (await nxt());
    });

    const running = wrappers.slice(0, n);
    const waiting = wrappers.slice(n);
    return Promise.all(running.map(fn => fn()));
}
```

TypeScript

```typescript
type F = () => Promise<any>;

function promisePool(functions: F[], n: number): Promise<any> {
    const wrappers = functions.map(fn => async () => {
        await fn();
        const nxt = waiting.shift();
        nxt && (await nxt());
    });

    const running = wrappers.slice(0, n);
    const waiting = wrappers.slice(n);
    return Promise.all(running.map(fn => fn()));
}
```

[*] JavaScript — Pattern Solution (matched from community answers)

Python


```python
import asyncio

async def promise_pool(functions, n):
    # Shared index among all tasks.
    i = 0

    async def worker():
        nonlocal i
        while i < len(functions):
            # Grab the current function and increment the index for next worker.
            fn = functions[i]
            i += 1
            # Await the async function.
            await fn()

    # Create n worker tasks.
    tasks = [asyncio.create_task(worker()) for _ in range(n)]
    # Wait until all worker tasks finish.
    await asyncio.gather(*tasks)

# Example usage:
if __name__ == "__main__":
    async def async_task(delay):
        await asyncio.sleep(delay)

    functions = [
        lambda: async_task(0.3),
        lambda: async_task(0.4),
        lambda: async_task(0.2)
    ]
    asyncio.run(promise_pool(functions, 2))
```

#2675 MEDIUM

Array of Objects to Matrix

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

JavaScript · Array

Lists: Premium 98

Problem:

Write a function that converts an array of objects arr into a matrix m.

arr is an array of objects or arrays. Each item in the array can be deeply nested with child arrays and child objects. It can also contain numbers, strings, booleans, and null values.

The first row m should be the column names. If there is no nesting, the column names are the unique keys within the objects. If there is nesting, the column names are the respective paths in the object separated by ".".

Each of the remaining rows corresponds to an object in arr. Each value in the matrix corresponds to a value in an object. If a given object doesn't contain a value for a given column, the cell should contain an empty string "".

The columns in the matrix should be in lexographically ascending order.

Example 1:


```
Input:
arr = [
  {"b": 1, "a": 2},
  {"b": 3, "a": 4}
]
Output:
[
  ["a", "b"],
  [2, 1],
  [4, 3]
]

Explanation:
There are two unique column names in the two objects: "a" and "b".
"a" corresponds with [2, 4].
"b" corresponds with [1, 3].
```

Example 2:

```
Input:
arr = [
  {"a": 1, "b": 2},
  {"c": 3, "d": 4},
  {}
]
Output:
[
  ["a", "b", "c", "d"],
  [1, 2, "", ""],
  ["", "", 3, 4],
  ["", "", "", ""]
]

Explanation:
There are 4 unique column names: "a", "b", "c", "d".
The first object has values associated with "a" and "b".
The second object has values associated with "c" and "d".
The third object has no keys, so it is just a row of empty strings.
```

Example 3:

```
Input:
arr = [
  {"a": {"b": 1, "c": 2}},
  {"a": {"b": 3, "d": 4}}
]
Output:
[
  ["a.b", "a.c", "a.d"],
  [1, 2, ""],
  [3, "", 4]
]

Explanation:
```

```
In this example, the objects are nested. The keys represent the full path to each value
separated by periods.
There are three paths: "a.b", "a.c", "a.d".
```

Example 4:

```
Input:
arr = [
  [{"a": null}],
  [{"b": true}],
  [{"c": "x"}]
]
Output:
[
  ["0.a", "0.b", "0.c"],
  [null, "", ""],
  ["", true, ""],
  ["", "", "x"]
]

Explanation:
Arrays are also considered objects with their keys being their indices.
Each array has one element so the keys are "0.a", "0.b", and "0.c".
```

Example 5:

```
Input:
arr = [
{},
  {},
  {},
]
Output:
[
  [],
  [],
  [],
  []
]

Explanation:
There are no keys so every row is an empty array.
```

Constraints:

- arr is a valid JSON array
- 1 <= arr.length <= 1000
- unique keys <= 1000

TypeScript

TypeScript


```typescript
function jsonToMatrix(arr: any[]): (string | number | boolean | null)[][] {
  const isObject = (o: any) => o !== null && typeof o === 'object';

  // Returns the keys of a JSON-like object by recursively unwrapping the nests.
  const getKeys = (json: any): string[] => {
    if (!isObject(json)) {
      return [''];
    }
    return Object.keys(json).reduce((acc: string[], currKey: string) => {
      return (
        acc.push(
          ...getKeys(json[currKey]).map((nextKey: string) =>
            nextKey === '' ? currKey : `${currKey}.${nextKey}`
          )
        ),
        acc
      );
    }, []);
  };

  const sortedKeys: string[] = [
    ...arr.reduce((acc: Set<string>, curr: any) => {
      getKeys(curr).forEach((key: string) => acc.add(key));
      return acc;
    }, new Set<string>()),

  ].sort();

  // Returns the value of `obj` keyed by `nestedKey`.
  const getValue = (
    obj: any,
    nestedKey: string
  ): string | number | boolean | null => {
    let value: any = obj;
    for (const key of nestedKey.split('.')) {
      if (!isObject(value) || !(key in value)) {
        return '';
      }
      value = value[key];
    }
    return isObject(value) ? '' : value;
  };

  const matrix: (string | number | boolean | null)[][] = [sortedKeys];
  arr.forEach((obj: any) => {
    matrix.push(
      sortedKeys.map((nestedKey: string) => getValue(obj, nestedKey))
    );
  });

  return matrix;

}
```

TypeScript


```typescript
function jsonToMatrix(arr: any[]): (string | number | boolean | null)[] {
    const dfs = (key: string, obj: any) => {
        if (
            typeof obj === 'number' ||
            typeof obj === 'string' ||
            typeof obj === 'boolean' ||
            obj === null
        ) {
            return { [key]: obj };
        }
        const res: any[] = [];
        for (const [k, v] of Object.entries(obj)) {
            const newKey = key ? `${key}.${k}` : `${k}`;
            res.push(dfs(newKey, v));
        }
        return res.flat();
    };

    const kv = arr.map(obj => dfs('', obj));
    const keys = [
        ...new Set(
            kv
                .flat()
                .map(obj => Object.keys(obj))
                .flat(),

        ),
    ].sort();
    const ans: any[] = [keys];
    for (const row of kv) {
        const newRow: any[] = [];
        for (const key of keys) {
            const v = row.find(r => r.hasOwnProperty(key))?.[key];
            newRow.push(v === undefined ? '' : v);
        }
        ans.push(newRow);
    }
    return ans;
}
```

[*] JavaScript — Pattern Solution (stored optimal)

Python


```python
from typing import List

class Solution:
    def jsonToMatrix(self, arr: List[dict]) -> List[List]:
        # Collect all unique keys across all objects
        keys = set()
        for obj in arr:
            keys.update(obj.keys())
        keys = sorted(keys) # sort alphabetically for consistent column order

        # First row = column headers
        result = [keys]

        # Each subsequent row = object values (empty string if key missing)
        for obj in arr:
            row = [obj.get(key, "") for key in keys]
            result.append(row)

        return result
```

#2700 MEDIUM

Differences Between Two Objects

LeetDoocs • SimplyLeet • WalkCC • LeetCode • Editorial

JavaScript

Lists: Premium 98

Problem:

Write a function that accepts two deeply nested objects or arrays obj1 and obj2 and returns a new object representing their differences.

The function should compare the properties of the two objects and identify any changes. The returned object should only contains keys where the value is different from obj1 to obj2.

For each changed key, the value should be represented as an array [obj1 value, obj2 value]. Keys that exist in one object but not in the other should not be included in the returned object. The end result should be a deeply nested object where each leaf value is a difference array.

When comparing two arrays, the indices of the arrays are considered to be their keys.

You may assume that both objects are the output of JSON.parse.

Example 1:

```
Input:
obj1 = {}
obj2 = {
  "a": 1,
"b": 2
}
Output: {}
Explanation: There were no modifications made to obj1. New keys "a" and "b" appear in obj2,
but keys that are added or removed should be ignored.
```

Example 2:


```
Input:
obj1 = {
  "a": 1,
  "v": 3,
  "x": [],
  "z": {
    "a": null
  }
}
obj2 = {
  "a": 2,
  "v": 4,
  "x": [],
  "z": {
    "a": 2
  }
}
Output:
{
  "a": [1, 2],
"v": [3, 4],
  "z": {
  "a": [null, 2]
  }
}
Explanation: The keys "a", "v", and "z" all had changes applied. "a" was changed from 1 to 2.
"v" was changed from 3 to 4. "z" had a change applied to a child object. "z.a" was changed
from null to 2.
```

Example 3:

```
Input:
obj1 = {
  "a": 5,
  "v": 6,
  "z": [1, 2, 4, [2, 5, 7]]
}
obj2 = {
  "a": 5,
  "v": 7,
  "z": [1, 2, 3, [1]]
}
Output:
{
  "v": [6, 7],
  "z": {
  "2": [4, 3],
  "3": {
  "0": [2, 1]
  }
  }
}
```



```
Explanation: In obj1 and obj2, the keys "v" and "z" have different assigned values. "a" is
ignored because the value is unchanged. In the key "z", there is a nested array. Arrays are
treated like objects where the indices are keys. There were two alterations to the the array:
z[2] and z[3][0]. z[0] and z[1] were unchanged and thus not included. z[3][1] and z[3][2]
were removed and thus not included.
```

Example 4:

```
Input:
obj1 = {
  "a": {"b": 1},
}
obj2 = {
  "a": [5],
}
Output:
{
"a": [{"b": 1}, [5]]
}
Explanation: The key "a" exists in both objects. Since the two associated values have
different types, they are placed in the difference array.
```

Example 5:

```
Input:
obj1 = {
  "a": [1, 2, {}],
  "b": false
}
obj2 = {
  "b": false,
  "a": [1, 2, {}]
}
Output:
{}
Explanation: Apart from a different ordering of keys, the two objects are identical so an
empty object is returned.
```

Constraints:

- obj1 and obj2 are valid JSON objects or arrays
- 2 <= JSON.stringify(obj1).length <= 10^4
- 2 <= JSON.stringify(obj2).length <= 10^4

TypeScript

TypeScript

```
type JSONValue =
    | null
    | boolean
    | number
    | string
    | JSONValue[]
    | { [key: string]: JSONValue };
type Obj = Record<string, JSONValue> | Array<JSONValue>;

function objDiff(obj1: Obj, obj2: Obj): Obj {
    if (obj1 === obj2) {
      return {};
    }
    if (obj1 === null || obj2 === null) {
      return [obj1, obj2];
    }
    if (typeof obj1 !== 'object' || typeof obj2 !== 'object') {
      return [obj1, obj2];
    }
    if (Array.isArray(obj1) !== Array.isArray(obj2)) {
      return [obj1, obj2];
    }
    const ans = {};
    for (const key in obj1) {
      if (key in obj2) {

        const subDiff = objDiff(obj1[key], obj2[key]);
        if (Object.keys(subDiff).length > 0) {
          ans[key] = subDiff;
        }
      }
    }
    return ans;
}
```

[*] JavaScript — Pattern Solution (stored optimal)

Python


```
def diff_objects(obj1, obj2):
    # If both values are dictionaries
    if isinstance(obj1, dict) and isinstance(obj2, dict):
        diff = {}
        # Only iterate on the keys present in both dictionaries
        for key in obj1.keys() & obj2.keys():
            sub_diff = diff_objects(obj1[key], obj2[key])
            # Only add if there is an actual difference
            if sub_diff is not None:
                diff[key] = sub_diff
        return diff if diff else None
    # If both values are lists, treat indices as keys
    elif isinstance(obj1, list) and isinstance(obj2, list):
        diff = {}
        # Iterate over common indices only
        for i in range(min(len(obj1), len(obj2))):
            sub_diff = diff_objects(obj1[i], obj2[i])
            if sub_diff is not None:
                # Use string keys for consistency with object keys
                diff[str(i)] = sub_diff
        return diff if diff else None
    else:
        # If the two values are equal, there's no difference
        if obj1 == obj2:
            return None

    else:
        # If types differ or values are different, return the difference array.
        return [obj1, obj2]

# A wrapper function to ensure we always return an object (or empty dict)
def differences_between_two_objects(obj1, obj2):
    result = diff_objects(obj1, obj2)
    return result if result is not None else {}

# Example usage:
if __name__ == "__main__":
    obj1 = {"a": 5, "v": 6, "z": [1, 2, 4, [2, 5, 7]]}
    obj2 = {"a": 5, "v": 7, "z": [1, 2, 3, [1]]}
    print(differences_between_two_objects(obj1, obj2))
```


Quick Review — JavaScript 7 questions · insights · complexity · solution

#2627 Debounce [Medium]

Key Insights

- Each call to the debounced function resets the timer.
- Only the most recent function call (that isn't subsequently cancelled) will eventually execute.
- Passing of parameters is handled by storing the arguments of the latest call.
- The solution leverages timers (or equivalent scheduling mechanisms) and cancellation techniques.

Space & Time

Time Complexity: O(1) per call, as each invocation only clears and sets a timer.

Space Complexity: O(1), since only a single timer reference is maintained.

Solution

The solution creates a wrapper function that maintains an internal timer reference. On each call to the debounced function, if a timer exists, it is cancelled. A new timer is then set to execute the original function after t milliseconds with the provided arguments. This ensures that only the last invocation runs if multiple calls occur within the delay period.

#2628 JSON Deep Equal [Medium]

Key Insights

- Use recursion to compare nested structures.
- For primitive types, a simple equality check (=== in JavaScript, == in Python, etc.) suffices.
- For arrays, verify both have the same length and compare each element recursively.
- For objects, ensure both have the same keys (order does not matter) then recursively compare the corresponding values.
- Handle cases where one operand is null or differs by type early to avoid unnecessary recursion.

Space & Time

Time Complexity: O(N) in the worst-case, where N is the total number of elements/values in the JSON structure.

Space Complexity: O(N) due to the recursion call stack in the worst-case scenario of deeply nested objects or arrays.

Solution

We solve this problem using a recursive approach. The algorithm first checks if both values are strictly equal, which handles primitives immediately. For arrays and objects, it verifies that the sizes (length for arrays and number of keys for objects) match. Then it recurses into each element (for arrays) or checks each key-value pair (for objects) to ensure that they are also deeply equal. For objects, the order of keys is irrelevant, so we compare key sets. This method naturally handles nested structures by delegating equality checks to the same function.

#2630 Memoize [Medium]

Key Insights

- Cache previously computed results using a data structure keyed by the function arguments.
- Ensure the key uniquely represents the argument order; simple tuple (or equivalent) suffices.
- Use closures (or object state) to maintain both the cache and a call count tracking actual function invocations.
- Expose a method (getCallCount) on the memoized function to report the number of times the original function was actually called.
- The memoized version works for functions that might take one or more parameters.

Space & Time

Time Complexity: O(1) average for each call assuming efficient hashing of arguments.

Space Complexity: O(n) for caching n distinct input combinations.

Solution

We use a hash-based cache (a dictionary or map) to store results for every unique set of arguments. Upon each call, we check if the key (constructed from the input arguments) exists in the cache. If not, we increment a counter, call the original function, store the result in the cache, and return it. Otherwise, we return the cached result. The memoized function is augmented with a getCallCount method to report the number of actual calls made to the original function.

#2633 Convert Object to JSON String [Medium]

Key Insights

- The solution must support all JSON primitives: strings, numbers, booleans, and null.
- Composite types such as arrays and objects require recursive traversal for proper serializing.
- For strings, wrap them in double quotes.
- For arrays, recursively process every element and join them with commas between square brackets.
- For objects, iterate keys in insertion order and combine key-value pairs (key must be a string in double quotes) with commas between curly braces.
- Use recursion carefully to handle nested structures without adding unnecessary spaces.

Space & Time

Time Complexity: O(n), where n is the total number of elements/characters in the JSON structure.

Space Complexity: O(n) due to the recursion stack and the constructed output string.

Solution

We use a recursive approach that inspects the type of the value and processes it accordingly:

- If the value is null, output "null". - If the value is a boolean or number, convert it directly to its string representation. - For a string, ensure it is enclosed in double quotes. - For an array, iterate over each element, serialize each recursively, join them with commas, and enclose in square brackets. - For an object, iterate over its keys in insertion order (or using Object.keys in JS / LinkedHashMap in Java), serialize both key (wrapped in double quotes) and its corresponding value, join with commas, and enclose in curly braces. This approach leverages recursion to handle arbitrarily nested objects and arrays while ensuring there are no extra spaces in the final string.

#2636 Promise Pool [Medium]

Key Insights

- Use a concurrency control mechanism to limit the number of actively running promises.
- Maintain an index or pointer to track which function from the input array should be executed next.
- Upon the resolution of any promise, start a new one if available until all functions have been processed.
- The overall task is complete when there are no functions left and all running promises have resolved.

Space & Time

Time Complexity: O(m), where m is the number of functions to execute.

Space Complexity: O(n) for the active pool of concurrent promises.

Solution

We can solve the problem by maintaining an index that tracks which asynchronous function to execute next. We define a helper routine (or executor) that:

- Checks if there is a function left to run. - If yes, increments the index and awaits the result of running that function. - Once the function completes, the routine recursively calls itself to pick up the next function. We start n such routines (to respect the pool limit) and use Promise.all (or the equivalent in other languages) to wait until all concurrent routines have finished executing. This structure ensures that we never run more than n promises concurrently and that a new promise starts as soon as one finishes.

#2675 Array of Objects to Matrix [Medium]

Key Insights

- Flatten each object or array into dot-separated paths such as `a.b` or `a.0.c`.
- Use DFS so nested objects and nested arrays are handled uniformly.
- For each row, store a map from flattened path to value.
- Collect every path seen across all rows into a set of column names.
- Sort the column names lexicographically to build the header row.
- For each flattened row map, output values in header order and use an empty string when a column is missing.

Space & Time

Time Complexity: O(T + R * C log C), where T is the total number of nested nodes visited, R is the number of rows, and C is the number of unique flattened columns.

Space Complexity: O(R * C + C) in the worst case for the flattened row maps and the set of all column names.

Solution

We solve the problem by flattening every input row first. A DFS walks through objects and arrays, extending the current path with either the property name or the array index. When the DFS reaches a primitive value or `null`, it stores that value under the built path. While flattening each row, we also collect every path into a global set of column names. After all rows are processed, we sort the columns lexicographically, place them in the first row of the matrix, and then build each remaining row by reading values from that row's flattened map, using `""` for missing paths.

#2700 Differences Between Two Objects [Medium]

Key Insights

- Use recursion to traverse nested objects and arrays.
- Compare only keys that exist in both objects.
- When encountering two values of different types or unequal primitives, record the difference as [value from obj1, value from obj2].
- When both values are objects or arrays, recursively compute their differences and include them only if the resulting nested diff is non-empty.
- Arrays are treated like objects with indices as keys.
- The solution requires careful type checking and handling of recursive structures.

Space & Time

Time Complexity: O(n) in average, where n is the number of keys/elements in the common parts of the structures.

Space Complexity: O(n) due to recursion and storage needed for the output differences.

Solution

We solve the problem using a recursive function that compares keys present in both input objects. For each common key:

- If both values are objects (or both are arrays), recursively compute their differences. - If the recursive call returns a non-empty object, include that key in the result. - If the two values are of different types or are primitives that differ, add the key with a difference array [value from obj1, value from obj2]. - Ignore keys that exist only in one object. This recursive approach naturally handles the deeply nested structure while ensuring only common keys with different values are captured.

#4 String — 8 questions · #4 of 21

#383 EASY

Ransom Note

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Counting

Lists: Grind 169

Problem:

Given two strings ransomNote and magazine, return true if ransomNote can be constructed by using the letters from magazine and false otherwise.

Each letter in magazine can only be used once in ransomNote.

Example 1:

```
Input: ransomNote = "a", magazine = "b"
Output: false
```

Example 2:

```
Input: ransomNote = "aa", magazine = "ab"
Output: false
```

Example 3:

```
Input: ransomNote = "aa", magazine = "aab"
Output: true
```

Constraints:

- 1 <= ransomNote.length, magazine.length <= 105
- ransomNote and magazine consist of lowercase English letters.

Python

Python

```
# Function to check if ransom note can be constructed from magazine
def canConstruct(ransomNote: str, magazine: str) -> bool:
    # Create a dictionary to hold the frequency of each character in magazine
    char_frequency = {}
    for char in magazine:
        char_frequency[char] = char_frequency.get(char, 0) + 1

    # Check each character in ransomNote against the frequency map
    for char in ransomNote:
        if char_frequency.get(char, 0) == 0:
            return False # Character not available or used up
        char_frequency[char] -= 1 # Use the character

    return True

# Example usage:
if __name__ == "__main__":
    print(canConstruct("aa", "aab")) # Expected output: True
```

Python

```python
class Solution:
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:
        count1 = collections.Counter(ransomNote)
        count2 = collections.Counter(magazine)
        return all(count1[c] <= count2[c] for c in string.ascii_lowercase)
```

Python

```python
class Solution:
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:
        cnt = Counter(magazine)
        for c in ransomNote:
            cnt[c] -= 1
            if cnt[c] < 0:
                return False
        return True
```

Python

```python
class Solution:
    def canConstruct(self, ransomNote: str, magazine: str) -> bool:
        cnt = Counter(magazine)
        for c in ransomNote:
            cnt[c] -= 1
            if cnt[c] < 0:
                return False
        return True
```

[*] String — Pattern Solution (stored optimal)

Python

```python
# Function to check if ransom note can be constructed from magazine
def canConstruct(ransomNote: str, magazine: str) -> bool:
    # Create a dictionary to hold the frequency of each character in magazine
    char_frequency = {}
    for char in magazine:
        char_frequency[char] = char_frequency.get(char, 0) + 1

    # Check each character in ransomNote against the frequency map
    for char in ransomNote:
        if char_frequency.get(char, 0) == 0:
            return False # Character not available or used up
        char_frequency[char] -= 1 # Use the character

    return True

# Example usage:
if __name__ == "__main__":
    print(canConstruct("aa", "aab")) # Expected output: True
```

#734 EASY

Sentence Similarity

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · String

Lists: Premium 98

Problem:

We can represent a sentence as an array of words, for example, the sentence "I am happy with leetcode" can be represented as arr = ["I","am","happy","with","leetcode"].

Given two sentences sentence1 and sentence2 each represented as a string array and given an array of string pairs similarPairs where similarPairs[i] = [xi, yi] indicates that the two words xi and yi are similar.

Return true if sentence1 and sentence2 are similar, or false if they are not similar.

Two sentences are similar if:

- They have the same length (i.e., the same number of words)
- sentence1[i] and sentence2[i] are similar.

Notice that a word is always similar to itself, also notice that the similarity relation is not transitive. For example, if the words a and b are similar, and the words b and c are similar, a and c are not necessarily similar.

Example 1:

```
Input: sentence1 = ["great","acting","skills"], sentence2 = ["fine","drama","talent"],
similarPairs = [["great","fine"],["drama","acting"],["skills","talent"]]
Output: true
Explanation: The two sentences have the same length and each word i of sentence1 is also
similar to the corresponding word in sentence2.
```

Example 2:

```
Input: sentence1 = ["great"], sentence2 = ["great"], similarPairs = []
Output: true
Explanation: A word is similar to itself.
```

Example 3:

```
Input: sentence1 = ["great"], sentence2 = ["doubleplus","good"], similarPairs =
[["great","doubleplus"]]
Output: false
Explanation: As they don't have the same length, we return false.
```

Constraints:

- 1 <= sentence1.length, sentence2.length <= 1000
- 1 <= sentence1[i].length, sentence2[i].length <= 20
- sentence1[i] and sentence2[i] consist of English letters.
- 0 <= similarPairs.length <= 1000
- similarPairs[i].length == 2
- 1 <= xi.length, yi.length <= 20
- xi and yi consist of lower-case and upper-case English letters.
- All the pairs (xi, yi) are distinct.

>> Brute Force [String] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def are_sentences_similar(self, sentence1, sentence2, similarPairs):
        # Brute Force O(n^2) - check all substrings
        n = len(sentence1)
        best = ''
        for i in range(n):
            for j in range(i + 1, n + 1):
                sub = sentence1[i:j]
                if len(sub) > len(best): best = sub
        return best
```

Python

Python

```python
# Function to determine if two sentences are similar
def are_sentences_similar(sentence1, sentence2, similarPairs):
    # Check if sentences are of the same length
    if len(sentence1) != len(sentence2):
        return False

    # Build dictionary to store similar word pairs in both directions
    similar_dict = {}
    for word1, word2 in similarPairs:
        if word1 not in similar_dict:
            similar_dict[word1] = set()
        similar_dict[word1].add(word2)
        if word2 not in similar_dict:
            similar_dict[word2] = set()
        similar_dict[word2].add(word1)

    # Check each pair of words in the sentences
    for w1, w2 in zip(sentence1, sentence2):
        # If words are the same, continue
        if w1 == w2:
            continue
        # Check if w1 is similar to w2 explicitly
        if (w1 not in similar_dict or w2 not in similar_dict[w1]) and (w2 not in similar_dict or w1 not in
similar_dict[w2]):
            return False

    return True

# Example usage:
# sentence1 = ["great", "acting", "skills"]
# sentence2 = ["fine", "drama", "talent"]
# similarPairs = [["great", "fine"], ["drama", "acting"], ["skills", "talent"]]
# print(are_sentences_similar(sentence1, sentence2, similarPairs)) # Output: True
```

Python


```python
class Solution:
    def areSentencesSimilar(
        self,
        sentence1: list[str],
        sentence2: list[str],
        similarPairs: list[list[str]],
    ) -> bool:
        if len(sentence1) != len(sentence2):
            return False

        # map[key] := all the similar words of key
        map = collections.defaultdict(set)

        for a, b in similarPairs:
            map[a].add(b)
            map[b].add(a)

        for word1, word2 in zip(sentence1, sentence2):
            if word1 == word2:
                continue
            if word1 not in map:
                return False
            if word2 not in map[word1]:
                return False

        return True
```

Python

```python
class Solution:
    def areSentencesSimilar(
        self, sentence1: List[str], sentence2: List[str], similarPairs: List[List[str]]
    ) -> bool:
        if len(sentence1) != len(sentence2):
            return False
        s = {(x, y) for x, y in similarPairs}
        for x, y in zip(sentence1, sentence2):
            if x != y and (x, y) not in s and (y, x) not in s:
                return False
        return True
```

Python

```python
class Solution:
    def areSentencesSimilar(
        self, sentence1: List[str], sentence2: List[str], similarPairs: List[List[str]]
    ) -> bool:
        if len(sentence1) != len(sentence2):
            return False
        s = {(a, b) for a, b in similarPairs}
        return all(
            a == b or (a, b) in s or (b, a) in s for a, b in zip(sentence1, sentence2)
        )
```

[*] String — Pattern Solution (stored optimal)

Python


```python
# Function to determine if two sentences are similar
def are_sentences_similar(sentence1, sentence2, similarPairs):
    # Check if sentences are of the same length
    if len(sentence1) != len(sentence2):
        return False

    # Build dictionary to store similar word pairs in both directions
    similar_dict = {}
    for word1, word2 in similarPairs:
        if word1 not in similar_dict:
            similar_dict[word1] = set()
        similar_dict[word1].add(word2)
        if word2 not in similar_dict:
            similar_dict[word2] = set()
        similar_dict[word2].add(word1)

    # Check each pair of words in the sentences
    for w1, w2 in zip(sentence1, sentence2):
        # If words are the same, continue
        if w1 == w2:
            continue
        # Check if w1 is similar to w2 explicitly
        if (w1 not in similar_dict or w2 not in similar_dict[w1]) and (w2 not in similar_dict or w1 not in
similar_dict[w2]):
            return False

    return True

# Example usage:
# sentence1 = ["great", "acting", "skills"]
# sentence2 = ["fine", "drama", "talent"]
# similarPairs = [["great", "fine"], ["drama", "acting"], ["skills", "talent"]]
# print(are_sentences_similar(sentence1, sentence2, similarPairs)) # Output: True
```

#1165 EASY

Single Row Keyboard

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String

Lists: Premium 98

Problem:

There is a special keyboard with all keys in a single row.

Given a string keyboard of length 26 indicating the layout of the keyboard (indexed from 0 to 25). Initially, your finger is at index 0. To type a character, you have to move your finger to the index of the desired character. The time taken to move your finger from index i to index j is |i − j|.

You want to type a string word. Write a function to calculate how much time it takes to type it with one finger.

Example 1:

```
Input: keyboard = "abcdefghijklmnopqrstuvwxyz", word = "cba"
Output: 4
Explanation: The index moves from 0 to 2 to write 'c' then to 1 to write 'b' then to 0 again
```



```
to write 'a'.
Total time = 2 + 1 + 1 = 4.
```

Example 2:

```
Input: keyboard = "pqrstuvwxyzabcdefghijklmno", word = "leetcode"
Output: 73
```

Constraints:

- keyboard.length == 26
- keyboard contains each English lowercase letter exactly once in some order.
- 1 <= word.length <= 104
- word[i] is an English lowercase letter.

Python

Python

```python
# Build a dictionary to map each character to its index in the keyboard string.
def calculateTime(keyboard: str, word: str) -> int:
    # Create a mapping for character to index
    char_index = {char: idx for idx, char in enumerate(keyboard)}

    # Initialize the total time and starting position
    total_time = 0
    current_position = 0

    # Iterate over each character in the word
    for char in word:
        # Find the target index for the character
        target_position = char_index[char]
        # Calculate the time to move to the target position
        total_time += abs(target_position - current_position)
        # Update current position to the target position
        current_position = target_position

    return total_time

# Example usage:
# print(calculateTime("abcdefghijklmnopqrstuvwxyz", "cba"))
```

Python

```python
class Solution:
    def calculateTime(self, keyboard: str, word: str) -> int:
        letterToIndex = {c: i for i, c in enumerate(keyboard)}
        return (letterToIndex[word[0]] +
                sum(abs(letterToIndex[a] - letterToIndex[b])
                    for a, b in itertools.pairwise(word)))
```

Python

```python
class Solution:
    def calculateTime(self, keyboard: str, word: str) -> int:
        pos = {c: i for i, c in enumerate(keyboard)}
        ans = i = 0
        for c in word:
            ans += abs(pos[c] - i)
            i = pos[c]
        return ans
```

Python

```python
class Solution:
    def calculateTime(self, keyboard: str, word: str) -> int:
        pos = {c: i for i, c in enumerate(keyboard)}
        ans = i = 0
        for c in word:
            ans += abs(pos[c] - i)
            i = pos[c]
        return ans
```

[*] String — Pattern Solution (stored optimal)

Python

```python
# Build a dictionary to map each character to its index in the keyboard string.
def calculateTime(keyboard: str, word: str) -> int:
    # Create a mapping for character to index
    char_index = {char: idx for idx, char in enumerate(keyboard)}

    # Initialize the total time and starting position
    total_time = 0
    current_position = 0

    # Iterate over each character in the word
    for char in word:
        # Find the target index for the character
        target_position = char_index[char]
        # Calculate the time to move to the target position
        total_time += abs(target_position - current_position)
        # Update current position to the target position
        current_position = target_position

    return total_time

# Example usage:
# print(calculateTime("abcdefghijklmnopqrstuvwxyz", "cba"))
```

#8 MEDIUM

String to Integer (atoi)

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String

Lists: Grind 169

Problem:

Implement the myAtoi(string s) function, which converts a string to a 32-bit signed integer.

The algorithm for myAtoi(string s) is as follows:

1. Whitespace: Ignore any leading whitespace (" ").
2. Signedness: Determine the sign by checking if the next character is '-' or '+', assuming positivity if neither present.
3. Conversion: Read the integer by skipping leading zeros until a non-digit character is encountered or the end of the string is reached. If no digits were read, then the result is 0.

4. Rounding: If the integer is out of the 32-bit signed integer range [-231, 231 - 1], then round the integer to remain in the range. Specifically, integers less than -231 should be rounded to -231, and integers greater than 231 - 1 should be rounded to 231 - 1.

Return the integer as the final result.

Example 1:

Input: s = "42"

Output: 42

Explanation:

```
The underlined characters are what is read in and the caret is the current reader position.
Step 1: "42" (no characters read because there is no leading whitespace)
^
Step 2: "42" (no characters read because there is neither a '-' nor '+')
^
Step 3: "42" ("42" is read in)
^
```

Example 2:

Input: s = " -042"

Output: -42

Explanation:

```
Step 1: " -042" (leading whitespace is read and ignored)
^
Step 2: " -042" ('-' is read, so the result should be negative)
^
Step 3: " -042" ("042" is read in, leading zeros ignored in the result)
^
```

Example 3:

Input: s = "1337c0d3"

Output: 1337

Explanation:

```
Step 1: "1337c0d3" (no characters read because there is no leading whitespace)
^
Step 2: "1337c0d3" (no characters read because there is neither a '-' nor '+')
^
Step 3: "1337c0d3" ("1337" is read in; reading stops because the next character is a
non-digit)
^
```

Example 4:

Input: s = "0-1"

Output: 0

Explanation:

```
Step 1: "0-1" (no characters read because there is no leading whitespace)
^
```



```
Step 2: "0-1" (no characters read because there is neither a '-' nor '+')
^
Step 3: "0-1" ("0" is read in; reading stops because the next character is a non-digit)
^
```

Example 5:

Input: s = "words and 987"

Output: 0

Explanation:

Reading stops at the first non-digit character 'w'.

Constraints:

- 0 <= s.length <= 200
- s consists of English letters (lower-case and upper-case), digits (0-9), ' ', '+', '-', and '.'.

Python

Python

```python
# Python solution for the myAtoi problem.
def myAtoi(s: str) -> int:
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31

    i, n = 0, len(s)
    # Skip leading whitespace.
    while i < n and s[i] == ' ':
        i += 1

    # Handle sign.
    sign = 1
    if i < n and (s[i] == '+' or s[i] == '-'):
        if s[i] == '-':
            sign = -1
        i += 1

    result = 0
    # Convert digits and build the integer.
    while i < n and s[i].isdigit():
        digit = int(s[i])
        # Check for overflow or underflow.
        if result > (INT_MAX - digit) // 10:
            return INT_MAX if sign == 1 else INT_MIN
        result = result * 10 + digit

        i += 1

    return sign * result

# Example usage:
print(myAtoi("42"))
print(myAtoi(" -042"))
print(myAtoi("1337c0d3"))
print(myAtoi("0-1"))
print(myAtoi("words and 987"))
```


Python

```python
class Solution:
    def myAtoi(self, s: str) -> int:
        s = s.strip()
        if not s:
            return 0

        sign = -1 if s[0] == '-' else 1
        if s[0] in ('-', '+'):
            s = s[1:]

        num = 0

        for c in s:
            if not c.isdigit():
                break
            num = num * 10 + int(c)
            if sign * num <= -2**31:
                return -2**31
            if sign * num >= 2**31 - 1:
                return 2**31 - 1

        return sign * num
```

Python

```python
class Solution:
    def myAtoi(self, s: str) -> int:
        if not s:
            return 0
        n = len(s)
        if n == 0:
            return 0
        i = 0
        while s[i] == ' ':
            i += 1
            if i == n:
                return 0
        sign = -1 if s[i] == '-' else 1
        if s[i] in ['-', '+']:
            i += 1
        res, flag = 0, (2**31 - 1) // 10
        while i < n:
            if not s[i].isdigit():
                break
            c = int(s[i])
            if res > flag or (res == flag and c > 7):
                return 2**31 - 1 if sign > 0 else -(2**31)
            res = res * 10 + c
            i += 1
        return sign * res
```

Python


```python
class Solution:
    def myAtoi(self, s: str) -> int:
        if not s:
            return 0
        n = len(s)
        if n == 0:
            return 0
        i = 0
        while s[i] == ' ':
            i += 1
            # only contains blank space
            if i == n:
                return 0
        sign = -1 if s[i] == '-' else 1
        if s[i] in ['-', '+']:
            i += 1

        res, flag = 0, (2**31 - 1) // 10
        while i < n:
            # not a number, exit the loop
            if not s[i].isdigit():
                break
            c = int(s[i])
            # if overflow
            if res > flag or (res == flag and c > 7):
                return 2**31 - 1 if sign > 0 else -(2**31)

            res = res * 10 + c
            i += 1
        return sign * res
```

[*] String — Pattern Solution (stored optimal)

Python

```python
# Python solution for the myAtoi problem.
def myAtoi(s: str) -> int:
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31

    i, n = 0, len(s)
    # Skip leading whitespace.
    while i < n and s[i] == ' ':
        i += 1

    # Handle sign.
    sign = 1
    if i < n and (s[i] == '+' or s[i] == '-'):
        if s[i] == '-':
            sign = -1
        i += 1

    result = 0
    # Convert digits and build the integer.
    while i < n and s[i].isdigit():
        digit = int(s[i])
        # Check for overflow or underflow.
        if result > (INT_MAX - digit) // 10:
            return INT_MAX if sign == 1 else INT_MIN
        result = result * 10 + digit
```



```python
        i += 1

    return sign * result

# Example usage:
print(myAtoi("42"))
print(myAtoi(" -042"))
print(myAtoi("1337c0d3"))
print(myAtoi("0-1"))
print(myAtoi("words and 987"))
```

#249 MEDIUM

Group Shifted Strings

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · String

Lists: Premium 98

Problem:

Perform the following shift operations on a string:

- Right shift: Replace every letter with the successive letter of the English alphabet, where 'z' is replaced by 'a'. For example, "abc" can be right-shifted to "bcd" or "xyz" can be right-shifted to "yza".
- Left shift: Replace every letter with the preceding letter of the English alphabet, where 'a' is replaced by 'z'. For example, "bcd" can be left-shifted to "abc" or "yza" can be left-shifted to "xyz".

We can keep shifting the string in both directions to form an endless shifting sequence.

- For example, shift "abc" to form the sequence: ... <-> "abc" <-> "bcd" <-> ... <-> "xyz" <-> "yza" <-> .... <-> "zab" <-> "abc" <-> ...

You are given an array of strings strings, group together all strings[i] that belong to the same shifting sequence. You may return the answer in any order.

Example 1:

Input: strings = ["abc","bcd","acef","xyz","az","ba","a","z"]

Output: [["acef"],["a","z"],["abc","bcd","xyz"],["az","ba"]]

Example 2:

Input: strings = ["a"]

Output: [["a"]]

Constraints:

- 1 <= strings.length <= 200
- 1 <= strings[i].length <= 50
- strings[i] consists of lowercase English letters.

>> Brute Force [String] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def group_shifted_strings(self, strings):
        # Brute Force O(n^2) - check all substrings
        n = len(strings)
        best = ''
        for i in range(n):
            for j in range(i + 1, n + 1):
                sub = strings[i:j]
                if len(sub) > len(best): best = sub
        return best
```

Python

Python

```python
# Python solution for grouping shifted strings

def group_shifted_strings(strings):
    # Helper function to generate a unique key for each string based on differences
    def get_key(s):
        # Single character strings return a fixed key
        if len(s) == 1:
            return "0"
        # Compute differences modulo 26 for each adjacent characters
        key = []
        for i in range(1, len(s)):
            diff = (ord(s[i]) - ord(s[i-1]) + 26) % 26
            key.append(str(diff))
        # Use tuple interleaving or join to form a string key
        return ',',.join(key)

    groups = {}
    # Group strings by their computed key
    for s in strings:
        key = get_key(s)
        if key not in groups:
            groups[key] = []
        groups[key].append(s)
    # Return grouped strings as a list of lists
    return list(groups.values())


# Example usage:
print(group_shifted_strings(["abc","bcd","acef","xyz","az","ba","a","z"]))
```

Python


```python
class Solution:
  def groupStrings(self, strings: list[str]) -> list[list[str]]:
    keyToStrings = collections.defaultdict(list)

    def getKey(s: str) -> str:
      """
      Returns the key of 's' by pairwise calculation of differences.
      e.g. getKey("abc") -> "1,1" because diff(a, b) = 1 and diff(b, c) = 1.
      """
      diffs = []

      for i in range(1, len(s)):
        diff = (ord(s[i]) - ord(s[i - 1]) + 26) % 26
        diffs.append(str(diff))

      return ',',.join(diffs)

    for s in strings:
      keyToStrings[getKey(s)].append(s)

    return keyToStrings.values()
```

Python

```python
class Solution:
    def groupStrings(self, strings: List[str]) -> List[List[str]]:
        g = defaultdict(list)
        for s in strings:
            diff = ord(s[0]) - ord("a")
            t = []
            for c in s:
                c = ord(c) - diff
                if c < ord("a"):
                    c += 26
                t.append(chr(c))
            g["".join(t)].append(s)
        return list(g.values())
```

Python


```
'''
>>> a = defaultdict(list)
>>> b = defaultdict(()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: first argument must be callable or None


>>> ord('c')
99
>>> ord('c') - ord('a')
2
>>> ord('d') - 2
98
>>> chr(98)
'b'


>>> a = {"s":1, "t":2, "x":3}
>>> a
{'s': 1, 't': 2, 'x': 3}
>>> a.items()
dict_items([('s', 1), ('t', 2), ('x', 3)])
>>> a.values()
dict_values([1, 2, 3])

>>> list(a.values())
[1, 2, 3]
'''

from collections import defaultdict

class Solution:
    def groupStrings(self, strings: List[str]) -> List[List[str]]:
        mp = defaultdict(list)
        for s in strings:
            t = []
            diff = ord(s[0]) - ord('a')
            for c in s:
                d = ord(c) - diff
                if d < ord('a'):
                    d += 26
                t.append(chr(d))
            k = ''.join(t)
            mp[k].append(s)
        return list(mp.values())
```

[*] String — Pattern Solution (matched from community answers)

Python


```python
class Solution:
    def groupStrings(self, strings: List[str]) -> List[List[str]]:
        g = defaultdict(list)
        for s in strings:
            diff = ord(s[0]) - ord("a")
            t = []
            for c in s:
                c = ord(c) - diff
                if c < ord("a"):
                    c += 26
                t.append(chr(c))
            g["".join(t)].append(s)
        return list(g.values())
```

#271 MEDIUM

Encode and Decode Strings

LeetDoocs · SimplyLaet · WalkCC · LeetCode · Editorial

Array · String · Design

Lists: Grind 169 | Premium 98

Problem:

Design an algorithm to encode a list of strings to a string. The encoded string is then sent over the network and is decoded back to the original list of strings.

Machine 1 (sender) has the function:

```
string encode(vector<string> strs) {
// ... your code
return encoded_string;
}
```

Machine 2 (receiver) has the function:

```
vector<string> decode(string s) {
//... your code
return strs;
}
```

So Machine 1 does:

```
string encoded_string = encode(strs);
```

and Machine 2 does:

```
vector<string> strs2 = decode(encoded_string);
```

strs2 in Machine 2 should be the same as strs in Machine 1.

Implement the encode and decode methods.

You are not allowed to solve the problem using any serialize methods (such as eval).

Example 1:

```
Input: dummy_input = ["Hello","World"]
Output: ["Hello","World"]
Explanation:
Machine 1:
```



```
Codec encoder = new Codec();
String msg = encoder.encode(strs);
Machine 1 ---msg---> Machine 2

Machine 2:
Codec decoder = new Codec();
String[] strs = decoder.decode(msg);
```

Example 2:

```
Input: dummy_input = [""]
Output: [""]
```

Constraints:

- 1 <= strs.length <= 200
- 0 <= strs[i].length <= 200
- strs[i] contains any possible characters out of 256 valid ASCII characters.

Follow up: Could you write a generalized algorithm to work on any possible set of characters?

Python

Python

```python
class Codec:
    # Encodes a list of strings to a single string.
    def encode(self, strs):
        encoded = []
        # Iterate through each string in the list.
        for s in strs:
            # Append the length of the string, a '#' as delimiter, and the string.
            encoded.append(str(len(s)) + '#' + s)
        # Join all parts into one encoded string.
        return ''.join(encoded)

    # Decodes a single string back to a list of strings.
    def decode(self, s):
        decoded = []
        i = 0 # Pointer to traverse the encoded string.
        while i < len(s):
            j = i
            # Find the '#' delimiter to determine the length.
            while s[j] != '#':
                j += 1
            # Convert the substring from i to j into an integer length.
            length = int(s[i:j])
            # Extract the actual string using the obtained length.
            string = s[j+1:j+1+length]
            decoded.append(string)

            # Move the pointer to the start of the next encoded segment.
            i = j + 1 + length
        return decoded

# Example usage:
# codec = Codec()
# encoded_str = codec.encode(["Hello", "World"])
# decoded_list = codec.decode(encoded_str)
```


Python

```python
class Codec:
  def encode(self, strs: list[str]) -> str:
    """Encodes a list of strings to a single string."""
    return ''.join(str(len(s)) + '/' + s for s in strs)

  def decode(self, s: str) -> list[str]:
    """Decodes a single string to a list of strings."""
    decoded = []

    i = 0
    while i < len(s):
      slash = s.find('/', i)
      length = int(s[i:slash])
      i = slash + length + 1
      decoded.append(s[slash + 1:i])

    return decoded
```

Python

```python
class Codec:
    def encode(self, strs: List[str]) -> str:
        """Encodes a list of strings to a single string."""
        ans = []
        for s in strs:
            ans.append('{:4}'.format(len(s)) + s)
        return ''.join(ans)

    def decode(self, s: str) -> List[str]:
        """Decodes a single string to a list of strings."""
        ans = []
        i, n = 0, len(s)
        while i < n:
            size = int(s[i : i + 4])
            i += 4
            ans.append(s[i : i + size])
            i += size
        return ans


# Your Codec object will be instantiated and called as such:
# codec = Codec()
# codec.decode(codec.encode(strs))
```

Python

```
'''
>>> s = 'abcde'
>>> '{:4}'.format(len(s)) + s
' 5abcde'
>>>
>>> '{:4}'.format(333)
' 333'
>>>
>>> '{:4}'.format(1234)
'1234'
>>> '{:4}'.format(12345)
'12345'

>>> '{0:4}'.format(12345)
'12345'
>>> '{1:4}'.format(12345)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: Replacement index 1 out of range for positional args tuple
>>> '{1:4}'.format(12345, 678910)
'678910'
>>> '{1:4}'.format(12345, 678910)
'12345'
>>> '{0:4}'.format(12345, 678910)
'12345'

'''

class Codec: # no splitter
    def encode(self, strs: List[str]) -> str:
        """Encodes a list of strings to a single string."""
        ans = []
        for s in strs:
            ans.append('{:4}'.format(len(s)) + s)
        return ''.join(ans)

    def decode(self, s: str) -> List[str]:
        """Decodes a single string to a list of strings."""
        ans = []
        i, n = 0, len(s)
        while i < n:
            size = int(s[i : i + 4]) # so here potential overflow, if larger than int-max, need to clarify
assumption with interviewer
            i += 4
            ans.append(s[i : i + size]) # note: exclusive for right index
            i += size
        return ans


# Your Codec object will be instantiated and called as such:
# codec = Codec()
# codec.decode(codec.encode(strs))
```



```
########

class Codec:
    def encode(self, strs: List[str]) -> str:
        encoded = ""
        for s in strs:
            # codec fix input: ["abcd", "####"]
            encoded += str(len(s)) + "#" + s.replace("#", "ss") + "#"

        return encoded

    def decode(self, s: str) -> List[str]:
        decoded = []
        i = 0
        while i < len(s):
            j = s.find("#", i)
            length = int(s[i:j])
            decoded.append(s[j+1:j+1+length].replace("ss", "#"))
            i = j + 1 + length + 1

        return decoded

###########

class Codec: # wrong compared with above, for input: ["abcd", "####"]
    def encode(self, strs):
        encoded = ""
        for s in strs:
            encoded += str(len(s)) + "#" + s
```

[*] String — Pattern Solution (stored optimal)

Python


```
class Codec:
    # Encodes a list of strings to a single string.
    def encode(self, strs):
        encoded = []
        # Iterate through each string in the list.
        for s in strs:
            # Append the length of the string, a '#' as delimiter, and the string.
            encoded.append(str(len(s)) + "#" + s)
        # Join all parts into one encoded string.
        return ''.join(encoded)

    # Decodes a single string back to a list of strings.
    def decode(self, s):
        decoded = []
        i = 0 # Pointer to traverse the encoded string.
        while i < len(s):
            j = i
            # Find the '#' delimiter to determine the length.
            while s[j] != "#":
                j += 1
            # Convert the substring from i to j into an integer length.
            length = int(s[i:j])
            # Extract the actual string using the obtained length.
            string = s[j+1:j+1+length]
            decoded.append(string)

            # Move the pointer to the start of the next encoded segment.
            i = j + 1 + length
        return decoded

# Example usage:
# codec = Codec()
# encoded_str = codec.encode(["Hello", "World"])
# decoded_list = codec.decode(encoded_str)
```

#635 MEDIUM

Design Log Storage System

LeetCodex · SimpleLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Design · Ordered Set

Lists: Denny Zhang

Problem:

You are given several logs, where each log contains a unique ID and timestamp. Timestamp is a string that has the following format: Year:Month:Day:Hour:Minute:Second, for example, 2017:01:01:23:59:59. All domains are zero-padded decimal numbers.

Implement the LogSystem class:

- LogSystem() Initializes the LogSystem object.
- void put(int id, string timestamp) Stores the given log (id, timestamp) in your storage system.
- int[] retrieve(string start, string end, string granularity) Returns the IDs of the logs whose timestamps are within the range from start to end inclusive. start and end all have the same format as timestamp, and granularity means how precise the range should be (i.e. to the exact Day, Minute, etc.). For

example, start = "2017:01:01:23:59:59", end = "2017:01:02:23:59:59", and granularity = "Day" means that we need to find the logs within the inclusive range from Jan. 1st 2017 to Jan. 2nd 2017, and the Hour, Minute, and Second for each log entry can be ignored.

Example 1:

```
Input
["LogSystem", "put", "put", "put", "retrieve", "retrieve"]
[[], [1, "2017:01:01:23:59:59"], [2, "2017:01:01:22:59:59"], [3, "2016:01:01:00:00:00"],
["2016:01:01:01:01:01", "2017:01:01:23:00:00", "Year"], ["2016:01:01:01:01:01",
"2017:01:01:23:00:00", "Hour"]]
Output
[null, null, null, null, [3, 2, 1], [2, 1]]

Explanation
LogSystem logSystem = new LogSystem();
logSystem.put(1, "2017:01:01:23:59:59");
logSystem.put(2, "2017:01:01:22:59:59");
logSystem.put(3, "2016:01:01:00:00:00");

// return [3,2,1], because you need to return all logs between 2016 and 2017.
logSystem.retrieve("2016:01:01:01:01:01", "2017:01:01:23:00:00", "Year");

// return [2,1], because you need to return all logs between Jan. 1, 2016 01:XX:XX and Jan.
1, 2017 23:XX:XX.
// Log 3 is not returned because Jan. 1, 2016 00:00:00 comes before the start of the range.
logSystem.retrieve("2016:01:01:01:01:01", "2017:01:01:23:00:00", "Hour");
```

Constraints:

- 1 <= id <= 500
- 2000 <= Year <= 2017
- 1 <= Month <= 12
- 1 <= Day <= 31
- 0 <= Hour <= 23
- 0 <= Minute, Second <= 59
- granularity is one of the values ["Year", "Month", "Day", "Hour", "Minute", "Second"].
- At most 500 calls will be made to put and retrieve.

Python

Python


```
class LogSystem:
    def __init__(self):
        # List to store logs as (timestamp, id) tuples
        self.logs = []
        # Mapping granularity to index for the substring (fixed format: Year:Month:Day:Hour:Minute:Second)
        self.granularity_to_index = {
            "Year": 4,
            "Month": 7,
            "Day": 10,
            "Hour": 13,
            "Minute": 16,
            "Second": 19
        }

    def put(self, id, timestamp):
        # Append the log as a tuple
        self.logs.append((timestamp, id))

    def retrieve(self, start, end, granularity):
        # Determine the slicing index based on granularity
        idx = self.granularity_to_index[granularity]
        # Slice the given start and end timestamps
        start_slice = start[:idx]
        end_slice = end[:idx]
        result = []

        # Iterate over all logs and check if the timestamp slice is within the range
        for ts, log_id in self.logs:
            ts_slice = ts[:idx]
            if start_slice <= ts_slice <= end_slice:
                result.append(log_id)
        return result

# Example usage:
# logSystem = LogSystem()
# logSystem.put(1, "2017:01:01:23:59:59")
# logSystem.put(2, "2017:01:01:22:59:59")
# logSystem.put(3, "2016:01:01:00:00:00")
# print(logSystem.retrieve("2016:01:01:01:01:01", "2017:01:01:23:00:00", "Year")) # Output: [3, 2, 1]
# print(logSystem.retrieve("2016:01:01:01:01:01", "2017:01:01:23:00:00", "Hour")) # Output: [2, 1]
```

Python

```
class LogSystem:
    def __init__(self):
        self.granularityToIndices = {'Year': 4, 'Month': 7, 'Day': 10,
                                     'Hour': 13, 'Minute': 16, 'Second': 19}

        self.idAndTimestamps = []

    def put(self, id: int, timestamp: str) -> None:
        self.idAndTimestamps.append((id, timestamp))

    def retrieve(self, start: str, end: str, granularity: str) -> list[int]:
        index = self.granularityToIndices[granularity]
        s = start[:index]
        e = end[:index]
        return [id for id, timestamp in self.idAndTimestamps
                if s <= timestamp[:index] <= e]
```

Python


```
class LogSystem:
    def __init__(self):
        self.logs = []
        self.d = {
            "Year": 4,
            "Month": 7,
            "Day": 10,
            "Hour": 13,
            "Minute": 16,
            "Second": 19,
        }

    def put(self, id: int, timestamp: str) -> None:
        self.logs.append((id, timestamp))

    def retrieve(self, start: str, end: str, granularity: str) -> List[int]:
        i = self.d[granularity]
        return [id for id, ts in self.logs if start[:i] <= ts[:i] <= end[:i]]


# Your LogSystem object will be instantiated and called as such:
# obj = LogSystem()
# obj.put(id,timestamp)
# param_2 = obj.retrieve(start,end,granularity)
```

[*] String — Pattern Solution (stored optimal)

Python

```
class LogSystem:
    def __init__(self):
        # List to store logs as (timestamp, id) tuples
        self.logs = []
        # Mapping granularity to index for the substring (fixed format: Year:Month:Day:Hour:Minute:Second)
        self.granularity_to_index = {
            "Year": 4,
            "Month": 7,
            "Day": 10,
            "Hour": 13,
            "Minute": 16,
            "Second": 19
        }

    def put(self, id, timestamp):
        # Append the log as a tuple
        self.logs.append((timestamp, id))

    def retrieve(self, start, end, granularity):
        # Determine the slicing index based on granularity
        idx = self.granularity_to_index[granularity]
        # Slice the given start and end timestamps
        start_slice = start[:idx]
        end_slice = end[:idx]
        result = []
```

```python
        # Iterate over all logs and check if the timestamp slice is within the range
        for ts, log_id in self.logs:
            ts_slice = ts[:idx]
            if start_slice <= ts_slice <= end_slice:
                result.append(log_id)
        return result

# Example usage:
# logSystem = LogSystem()
# logSystem.put(1, "2017:01:01:23:59:59")
# logSystem.put(2, "2017:01:01:22:59:59")
# logSystem.put(3, "2016:01:01:00:00:00")
# print(logSystem.retrieve("2016:01:01:01:01:01", "2017:01:01:23:00:00", "Year")) # Output: [3, 2, 1]
# print(logSystem.retrieve("2016:01:01:01:01:01", "2017:01:01:23:00:00", "Hour")) # Output: [2, 1]
```

#1138 MEDIUM

Alphabet Board Path

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String

Lists: Denny Zhang

Problem:

On an alphabet board, we start at position (0, 0), corresponding to character board[0][0].

Here, board = ["abcde", "fghij", "klmno", "pqrst", "uvwxy", "z"], as shown in the diagram below.

a	b	c	d	e
f	g	h	i	j
k	l	m	n	o
p	q	r	s	t
u	v	w	x	y
z				

We may make the following moves:

- 'U' moves our position up one row, if the position exists on the board;

- 'D' moves our position down one row, if the position exists on the board;
- 'L' moves our position left one column, if the position exists on the board;
- 'R' moves our position right one column, if the position exists on the board;
- '!' adds the character board[r][c] at our current position (r, c) to the answer.

(Here, the only positions that exist on the board are positions with letters on them.)

Return a sequence of moves that makes our answer equal to target in the minimum number of moves. You may return any path that does so.

Example 1:

```
Input: target = "leet"

Output: "DDR!UURRR!!DDD!"
```

Example 2:

```
Input: target = "code"

Output: "RR!DDRR!UUL!R!"
```

Constraints:

- 1 <= target.length <= 100
- target consists only of English lowercase letters.

Python

Python

```python
# Python solution with clear variable names and inline comments.
class Solution:
    def alphabetBoardPath(self, target: str) -> str:
        # Starting position at the top-left corner (0,0)
        current_row, current_col = 0, 0
        result = []

        # Helper function to convert letter to its (row, col) coordinates
        def getPosition(letter):
            index = ord(letter) - ord('a')
            return index // 5, index % 5

        for char in target:
            target_row, target_col = getPosition(char)
            # If target character is 'z', move horizontally first then vertically
            if char == 'z':
                # Move left if needed
                while current_col > target_col:
                    result.append("L")
                    current_col -= 1
                # Move right if needed
                while current_col < target_col:
                    result.append("R")
                    current_col += 1
                # Then move down if needed
```



```python
                while current_row < target_row:
                    result.append("D")
                    current_row += 1
                # Then move up if needed
                while current_row > target_row:
                    result.append("U")
                    current_row -= 1
            else:
                # For all other characters, move vertically first
                while current_row > target_row:
                    result.append("U")
                    current_row -= 1
                while current_row < target_row:
                    result.append("D")
                    current_row += 1
                # Then move horizontally
                while current_col > target_col:
                    result.append("L")
                    current_col -= 1
                while current_col < target_col:
                    result.append("R")
                    current_col += 1
            # Append pick-up action for the letter
            result.append("!")
        return "".join(result)
```

Python

```python
class Solution:
    def alphabetBoardPath(self, target: str) -> str:
        i = j = 0
        ans = []
        for c in target:
            v = ord(c) - ord("a")
            x, y = v // 5, v % 5
            while j > y:
                j -= 1
                ans.append("L")
            while i > x:
                i -= 1
                ans.append("U")
            while j < y:
                j += 1
                ans.append("R")
            while i < x:
                i += 1
                ans.append("D")
            ans.append("!")
        return "".join(ans)
```

Python


```python
class Solution:
    def alphabetBoardPath(self, target: str) -> str:
        i = j = 0
        ans = []
        for c in target:
            v = ord(c) - ord("a")
            x, y = v // 5, v % 5
            while j > y:
                j -= 1
                ans.append("L")
            while i > x:
                i -= 1
                ans.append("U")
            while j < y:
                j += 1
                ans.append("R")
            while i < x:
                i += 1
                ans.append("D")
            ans.append("!")
        return "".join(ans)
```

[*] String — Pattern Solution (matched from community answers)

Python

```python
# Python solution with clear variable names and inline comments.
class Solution:
    def alphabetBoardPath(self, target: str) -> str:
        # Starting position at the top-left corner (0,0)
        current_row, current_col = 0, 0
        result = []

        # Helper function to convert letter to its (row, col) coordinates
        def getPosition(letter):
            index = ord(letter) - ord('a')
            return index // 5, index % 5

        for char in target:
            target_row, target_col = getPosition(char)
            # If target character is 'z', move horizontally first then vertically
            if char == 'z':
                # Move left if needed
                while current_col > target_col:
                    result.append("L")
                    current_col -= 1
                # Move right if needed
                while current_col < target_col:
                    result.append("R")
                    current_col += 1
                # Then move down if needed
```



```python
                while current_row < target_row:
                    result.append("D")
                    current_row += 1
                # Then move up if needed
                while current_row > target_row:
                    result.append("U")
                    current_row -= 1
            else:
                # For all other characters, move vertically first
                while current_row > target_row:
                    result.append("U")
                    current_row -= 1
                while current_row < target_row:
                    result.append("D")
                    current_row += 1
                # Then move horizontally
                while current_col > target_col:
                    result.append("L")
                    current_col -= 1
                while current_col < target_col:
                    result.append("R")
                    current_col += 1
            # Append pick-up action for the letter
            result.append("!")
        return "".join(result)
```


Quick Review — String 8 questions · insights · complexity · solution

#383 **Ransom Note** [Easy]

Key Insights

- Use a hash table (or dictionary) to store the frequency of each character present in magazine.
- Iterate over every character in ransomNote and check if the corresponding count in the hash table is available and sufficient.
- Decrease the character count for every match; if any character count falls below zero, return false.
- If all characters in ransomNote can be accounted for, return true.

Space & Time

Time Complexity: O(n + m), where n is the length of magazine and m is the length of ransomNote.

Space Complexity: O(1), since the extra space required for the hash table is limited to at most 26 entries for lowercase English letters.

Solution

We construct a frequency map for the magazine string using a hash table. This frequency map records how many times each letter appears in magazine. Then we iterate through the ransomNote string, and for every letter, we check the corresponding frequency in the map. If a letter is not found or its frequency is zero, we immediately return false as it means the letter cannot be used to build the note. Otherwise, we decrement the frequency count by one. If we successfully process all characters in ransomNote, then it can be constructed from magazine, and we return true.

#734 **Sentence Similarity** [Easy]

Key Insights

- First, check if both sentences have the same length.
- A word is always similar to itself.
- Similarity is defined only by the given pairs; it is not transitive.
- Use a hash table (dictionary) to store bi-directional similar pairs for quick look-ups.

Space & Time

Time Complexity: O(n + m), where n is the number of words in the sentences and m is the number of similar pairs.

Space Complexity: O(m), for storing the similar pairs in a hash table.

Solution

- Check if the two sentences have the same length; if not, return false. - Build a hash table where each word maps to a set of words it is similar to. For each pair [x, y] in similarPairs, add y to the set for x and x to the set for y. - Iterate over the words of both sentences simultaneously. For each corresponding pair, if the words are identical, continue; otherwise, check if one word appears in the similar set of the other. - If all pairs meet the similarity condition, return true; otherwise, return false.

#1165 **Single Row Keyboard** [Easy]

Key Insights

- Create a mapping (hash table/dictionary) from each character to its index on the keyboard for O(1) lookups.
- The cost to type each character is the absolute difference between the current position and the target character's position on the keyboard.
- Start from index 0 and sum up the distances moved for each successive character in the word.

Space & Time

Time Complexity: O(n), where n is the length of the word.

Space Complexity: O(1) (constant space for a mapping of 26 characters).

Solution

We solve the problem by first building a dictionary that maps each character in the keyboard layout to its corresponding index. We then initialize a variable to track the current finger position (starting at 0). For each character in the word, we look up its index, calculate the absolute difference from the current position, add that distance to the total time, and update the current position. This approach uses a single pass over the word with constant time lookups, ensuring an efficient solution.

#8 String to Integer (atoi) [Medium]

Key Insights

- Skip leading spaces in the string.
- Detect and handle the optional sign indicator ('+' or '-').
- Read and convert digit characters until a non-digit is encountered.
- Handle cases with no digits or leading zeros.
- Clamp the result to fit within the 32-bit signed integer range.

Space & Time

Time Complexity: O(n) where n is the length of the string.

Space Complexity: O(1) as only a constant amount of extra space is used.

Solution

We iterate through the string to first eliminate any leading whitespace. Next, we check for a sign character to determine if the resulting integer should be negative. We then convert each consecutive digit into an integer, all the while updating the result and checking for potential overflow. If an overflow is detected, we immediately return the appropriate maximum or minimum 32-bit signed integer boundary value. This approach ensures that we strictly follow the conversion rules and efficiently handle edge cases.

#249 Group Shifted Strings [Medium]

Key Insights

- All strings in the same group have the same relative differences between adjacent characters.
- Normalize each string's pattern by converting it to a key representation based on the differences between consecutive letters.
- A single character string forms its own group as there are no differences to compute.

Space & Time

Time Complexity: O(n * m) where n is the number of strings and m is the average length of the strings (to compute each string's key).

Space Complexity: O(n * m) to store the keys and grouping in the hash table.

Solution

The solution involves iterating over each string and generating a unique key that represents the relative differences between adjacent characters. This key is calculated by computing the difference (mod 26) between every consecutive pair in the string. Strings that share the same key can be grouped together since they can be shifted into one another. A hash table (or dictionary) is used to store and group the strings by their computed key. This approach handles edge cases like single-character strings separately.

#271 Encode and Decode Strings [Medium]

Key Insights

- Use length-prefix encoding for each string to avoid ambiguity.
- Append a delimiter (e.g., "#") after the length to separate it from the string content.
- This method ensures that even if the original strings contain special characters, including the delimiter, the decoding remains unambiguous.
- The approach handles edge cases, such as empty strings.

Space & Time

Time Complexity: O(n), where n is the total number of characters across all strings.

Space Complexity: O(n), required for the encoded string and the resulting decoded list.

Solution

We encode by iterating over each string and building a new string that contains the length of the string, a delimiter (e.g., "#"), and then the string itself. This way, during decoding, we can safely determine how many characters to extract by first reading until the delimiter to get the length, and then reading that number of characters. This length-prefix approach avoids conflicts with potential characters in the strings.

#635 Design Log Storage System [Medium]

Key Insights

- Store logs in a simple data structure such as a list or array.
- Convert the timestamps into strings; due to fixed-length and zero padding, lexicographical string comparisons are valid.
- Use a mapping to determine the slice length for each granularity, e.g. "Year" corresponds to the first 4 characters, "Month" corresponds to the first 7 characters, etc.
- During retrieval, compare the relevant substring of each stored timestamp against the start and end query timestamp slices.
- The inclusive nature of the query simplifies checking conditions.

Space & Time

Time Complexity: O(N) per retrieval query, where N is the number of logs stored (since each log is checked). In the worst case, all log entries are examined.

Space Complexity: O(N) where N is the number of logs stored.

Solution

We implement a LogSystem class that supports putting log entries and retrieving them using a specified granularity.

- Data Structures: Use a list/array to store log entries as tuples (or objects) containing the timestamp and log ID. A dictionary/map is used to store the mapping from granularity string to the substring index (e.g., "Year" => 4, "Month" => 7, etc.). - For the put() method, simply append the new log entry into the list. - For the retrieve() method, compute the substring length based on the provided granularity. Then iterate over all stored logs and compare the sliced timestamp against the sliced start and end query values. If it falls within the bounds, include the log ID in the result. - Since the timestamps have fixed width with leading zeros, slicing and lexicographical comparisons are both valid and simple.

#1138 Alphabet Board Path [Medium]

Key Insights

- Determine the row and column for each letter (using integer division and modulus).
- Because "z" is in a special row that only contains one column, the move order is critical to avoid invalid positions.
- For target letter "z", perform horizontal moves before vertical moves; for all other letters, do vertical moves before horizontal moves.
- Build the answer step-by-step by moving from your current position to each target letter.

Space & Time

Time Complexity: O(N) where N is the length of target (each move is processed in constant time)

Space Complexity: O(N) for the resulting string of moves

Solution

The approach calculates the destination coordinates for each character. For each letter in the target:

- Convert the letter to its board coordinates (row and col). For example, for letter c, row = (ord(c) - ord('a')) // 5 and col = (ord(c) - ord('a')) % 5. Note that "z" (the 26th letter) always maps to (5, 0). - To move from the current position (curRow, curCol) to the target's (targetRow, targetCol), decide on the order of moves: If the target letter is "z", first

move horizontally (left or right) then vertically (up or down) to avoid stepping into invalid regions from the extra row. For any other letter, move vertically (up or down) first, then horizontally (left or right). - Append "!" to the moves after reaching each letter. - Update the current position to the letter's coordinates.

This method ensures you only traverse valid cells on the board and minimizes moves by taking the optimal coordinate difference each step.

#5 Sliding Window — 9 questions · #5 of 21

#3 MEDIUM

Longest Substring Without Repeating Characters

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Sliding Window

Lists: Grind 169

Problem:

Given a string s, find the length of the longest substring without duplicate characters.

Example 1:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3. Note that "bca" and "cab" are also
correct answers.
```

Example 2:

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.
```

Constraints:

- 0 <= s.length <= 5 * 104
- s consists of English letters, digits, symbols and spaces.

>> Brute Force [Sliding Window] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def length_of_longest_substring(self, s):
        # Brute Force O(n^2) - enumerate every starting index; expand until repeat
        res = 0
        for i in range(len(s)):
            seen = set()
            for j in range(i, len(s)):
                if s[j] in seen:
                    break
                seen.add(s[j])
                res = max(res, j - i + 1)
        return res
```

Python

Python

```python
# Function to find the length of the longest substring without repeating characters
def length_of_longest_substring(s):
    # Dictionary to store the most recent index of each character
    char_index = {}
    # Initialize starting index of current window and max length
    left = 0
    max_length = 0

    # Iterate over the string with the right pointer
    for right, char in enumerate(s):
        # If char is in dictionary and its index is within the current window
        if char in char_index and char_index[char] >= left:
            # Move the left pointer to one position after the last occurrence of char
            left = char_index[char] + 1
        # Update the latest index of the char
        char_index[char] = right
        # Update max_length with the current window size if larger
        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
# print(length_of_longest_substring("abcabcbb")) # Expected output: 3
```

Python

```python
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        ans = 0
        count = collections.Counter()

        l = 0
        for r, c in enumerate(s):
            count[c] += 1
            while count[c] > 1:
                count[s[l]] -= 1
                l += 1
            ans = max(ans, r - l + 1)

        return ans
```

Python

```python
class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        cnt = Counter()
        ans = l = 0
        for r, c in enumerate(s):
            cnt[c] += 1
            while cnt[c] > 1:
                cnt[s[l]] -= 1
                l += 1
            ans = max(ans, r - l + 1)
        return ans
```

Python


```python
class Solution:
    def lengthOfLongestSubstring(self, s): # map for index
        d = {} # value => its index
        i = 0
        ans = 0
        for j, c in enumerate(s):
            if c in d:
                # must max check i, example "abba"
                i = max(i, d[c] + 1)
            d[c] = j
            ans = max(ans, j - i + 1)
        return ans


class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:

        if not s:
            return 0

        l = 0 # left
        r = 0 # right

        result = 0
        isFoundInWindow = [False] * 256


        while r < len(s):

            while isFoundInWindow[ord(s[r])]:
                isFoundInWindow[ord(s[l])] = False
                l += 1

            isFoundInWindow[ord(s[r])] = True

            # check before shrink
            result = max(result, r - l + 1)

            r += 1

        return result

class Solution: # extra space for set!)
    def lengthOfLongestSubstring(self, s: str) -> int:

        d = collections.defaultdict(int)
        start = 0
        ans = 0
        for i,c in enumerate(s):
            # iterate
            while d[c] > 0:
```

```
                d[s[start]] -= 1 # not s[start], not start
                start += 1
            ans = max(ans, i - start + 1)
            d[c] = 1
        return ans

###########

class Solution(object):
    def _lengthOfLongestSubstring(self, s): # no extra data structure
        """
        :type s: str
        :rtype: int
        """
        d = collections.defaultdict(int)
        l = ans = 0
        for i, c in enumerate(s):
            while l > 0 and d[c] > 0:
                d[s[i - l]] -= 1
                l -= 1
            d[c] += 1
            l += 1
            ans = max(ans, l)
        return ans
```

```
###########

class Solution:
    def lengthOfLongestSubstring(self, s: str) -> int:
        ss = set()
```

[*] Sliding Window — Pattern Solution (stored optimal)

Python

```
# Function to find the length of the longest substring without repeating characters
def length_of_longest_substring(s):
    # Dictionary to store the most recent index of each character
    char_index = {}
    # Initialize starting index of current window and max length
    left = 0
    max_length = 0

    # Iterate over the string with the right pointer
    for right, char in enumerate(s):
        # If char is in dictionary and its index is within the current window
        if char in char_index and char_index[char] >= left:
            # Move the left pointer to one position after the last occurrence of char
            left = char_index[char] + 1
        # Update the latest index of the char
        char_index[char] = right
        # Update max_length with the current window size if larger
        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
# print(length_of_longest_substring("abcabcbb")) # Expected output: 3
```


#159 MEDIUM

Longest Substring with At Most Two Distinct Characters

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Sliding Window

Lists: Premium 98

Problem:

Given a string s, return the length of the longest substring that contains at most two distinct characters.

Example 1:

```
Input: s = "eceba"
Output: 3
Explanation: The substring is "ece" which its length is 3.
```

Example 2:

```
Input: s = "ccaabbb"
Output: 5
Explanation: The substring is "aabbb" which its length is 5.
```

Constraints:

- 1 <= s.length <= 105
- s consists of English letters.

Python

Python

```
# Python solution for Longest Substring with At Most Two Distinct Characters
class Solution:
    def lengthOfLongestSubstringTwoDistinct(self, s: str) -> int:
        left = 0 # Initialize left pointer of the window
        max_length = 0 # Variable to store the maximum window size found
        char_count = {} # Dictionary to count characters in the current window

        for right, char in enumerate(s):
            # Add the current character to the window (increase count)
            char_count[char] = char_count.get(char, 0) + 1

            # If there are more than two distinct characters, shrink the window from the left
            while len(char_count) > 2:
                left_char = s[left]
                char_count[left_char] -= 1
                if char_count[left_char] == 0:
                    del char_count[left_char]
                left += 1

            # Update max_length if current window is larger
            max_length = max(max_length, right - left + 1)

        return max_length
```

Python


```
class Solution:
    def lengthOfLongestSubstringTwoDistinct(self, s: str) -> int:
        ans = 0
        distinct = 0
        count = [0] * 128

        l = 0
        for r, c in enumerate(s):
          count[ord(c)] += 1
          if count[ord(c)] == 1:
            distinct += 1
          while distinct == 3:
            count[ord(s[l])] -= 1
            if count[ord(s[l])] == 0:
              distinct -= 1
            l += 1
          ans = max(ans, r - l + 1)

        return ans
```

Python

```
class Solution:
    def lengthOfLongestSubstringTwoDistinct(self, s: str) -> int:
        cnt = Counter()
        ans = j = 0
        for i, c in enumerate(s):
            cnt[c] += 1
            while len(cnt) > 2:
                cnt[s[j]] -= 1
                if cnt[s[j]] == 0:
                    cnt.pop(s[j])
                j += 1
            ans = max(ans, i - j + 1)
        return ans
```

Python


```
'''
>>> from collections import Counter
>>> c = Counter()
>>> c
Counter()
>>> c['a'] += 1
>>> c
Counter({'a': 1})


>>> c
Counter({'a': 1, 'c': 1, 'b': 1})
>>> c.pop(2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 2
>>>
>>> c.pop('c')
1
>>> c
Counter({'a': 1, 'b': 1})
'''

from collections import Counter
```

```
class Solution:
    def lengthOfLongestSubstringTwoDistinct(self, s: str) -> int:
        cnt = Counter() # or, cnt = defaultdict(int)
        ans = i = 0
        for j, c in enumerate(s):
            cnt[c] += 1
            while len(cnt) > 2: # shrink
                cnt[s[i]] -= 1
                if cnt[s[i]] == 0:
                    cnt.pop(s[i])
                i += 1
            ans = max(ans, j - i + 1)
        return ans
```

[*] Sliding Window — Pattern Solution (matched from community answers)

Python


```
# Python solution for Longest Substring with At Most Two Distinct Characters
class Solution:
    def lengthOfLongestSubstringTwoDistinct(self, s: str) -> int:
        left = 0 # Initialize left pointer of the window
        max_length = 0 # Variable to store the maximum window size found
        char_count = {} # Dictionary to count characters in the current window

        for right, char in enumerate(s):
            # Add the current character to the window (increase count)
            char_count[char] = char_count.get(char, 0) + 1

            # If there are more than two distinct characters, shrink the window from the left
            while len(char_count) > 2:
                left_char = s[left]
                char_count[left_char] -= 1
                if char_count[left_char] == 0:
                    del char_count[left_char]
                left += 1

            # Update max_length if current window is larger
            max_length = max(max_length, right - left + 1)

        return max_length
```

#340 MEDIUM

Longest Substring with At Most K Distinct Characters

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Sliding Window

Lists: Premium 98

Problem:

Given a string s and an integer k, return the length of the longest substring of s that contains at most k distinct characters.

Example 1:

```
Input: s = "eceba", k = 2
Output: 3
Explanation: The substring is "ece" with length 3.
```

Example 2:

```
Input: s = "aa", k = 1
Output: 2
Explanation: The substring is "aa" with length 2.
```

Constraints:

- 1 <= s.length <= 5 * 104
- 0 <= k <= 50

Python

Python


```
# Python solution for Longest Substring with At Most K Distinct Characters

def longest_substring_with_k_distinct(s: str, k: int) -> int:
    # Handle edge case: if k is 0, no valid substring exists
    if k == 0:
        return 0

    # Dictionary to store frequency of characters in the current window
    char_count = {}
    left = 0 # Left pointer for the window
    max_length = 0 # Variable to track the maximum length found

    # Iterate through the string using the right pointer
    for right in range(len(s)):
        # Add/update the count of the current character in the dictionary
        current_char = s[right]
        char_count[current_char] = char_count.get(current_char, 0) + 1

        # If window contains more than k distinct characters, shrink from the left
        while len(char_count) > k:
            left_char = s[left]
            char_count[left_char] -= 1
            # Remove the character count if it becomes 0
            if char_count[left_char] == 0:
                del char_count[left_char]

            left += 1

        # Update the maximum length of the valid window found so far
        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
# s = "eceba", k = 2
# print(longest_substring_with_k_distinct(s, 2)) # Output: 3
```

Python

```
class Solution:
  def lengthOfLongestSubstringKDistinct(self, s: str, k: int) -> int:
    ans = 0
    distinct = 0
    count = collections.Counter()

    l = 0
    for r, c in enumerate(s):
      count[c] += 1
      if count[c] == 1:
        distinct += 1
      while distinct == k + 1:
        count[s[l]] -= 1
        if count[s[l]] == 0:
          distinct -= 1
        l += 1
      ans = max(ans, r - l + 1)

    return ans
```

Python

```python
class Solution:
    def lengthOfLongestSubstringKDistinct(self, s: str, k: int) -> int:
        l = 0
        cnt = Counter()
        for c in s:
            cnt[c] += 1
            if len(cnt) > k:
                cnt[s[l]] -= 1
                if cnt[s[l]] == 0:
                    del cnt[s[l]]
                l += 1
        return len(s) - l
```

Python

```python
class Solution:
    def lengthOfLongestSubstringTwoDistinct(self, s: str) -> int:
        cnt = Counter() # or, cnt = defaultdict(int)
        ans = i = 0
        for j, c in enumerate(s):
            cnt[c] += 1
            while len(cnt) > k:
                cnt[s[i]] -= 1
                if cnt[s[i]] == 0:
                    cnt.pop(s[i])
                i += 1
            ans = max(ans, j - i + 1)
        return ans
```

[*] Sliding Window — Pattern Solution (stored optimal)

Python

```python
# Python solution for Longest Substring with At Most K Distinct Characters

def longest_substring_with_k_distinct(s: str, k: int) -> int:
    # Handle edge case: if k is 0, no valid substring exists
    if k == 0:
        return 0

    # Dictionary to store frequency of characters in the current window
    char_count = {}
    left = 0 # Left pointer for the window
    max_length = 0 # Variable to track the maximum length found

    # Iterate through the string using the right pointer
    for right in range(len(s)):
        # Add/update the count of the current character in the dictionary
        current_char = s[right]
        char_count[current_char] = char_count.get(current_char, 0) + 1

        # If window contains more than k distinct characters, shrink from the left
        while len(char_count) > k:
            left_char = s[left]
            char_count[left_char] -= 1
            # Remove the character count if it becomes 0
            if char_count[left_char] == 0:
                del char_count[left_char]
```



```python
            left += 1

        # Update the maximum length of the valid window found so far
        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
# s = "ecebaa", k = 2
# print(longest_substring_with_k_distinct(s, 2)) # Output: 3
```

#424 MEDIUM

Longest Repeating Character Replacement

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Sliding Window

Lists: Grind 169

Problem:

You are given a string s and an integer k. You can choose any character of the string and change it to any other uppercase English character. You can perform this operation at most k times.

Return the length of the longest substring containing the same letter you can get after performing the above operations.

Example 1:

```
Input: s = "ABAB", k = 2
Output: 4
Explanation: Replace the two 'A's with two 'B's or vice versa.
```

Example 2:

```
Input: s = "AABABBA", k = 1
Output: 4
Explanation: Replace the one 'A' in the middle with 'B' and form "AABBBBA".
The substring "BBBB" has the longest repeating letters, which is 4.
There may exists other ways to achieve this answer too.
```

Constraints:

- 1 <= s.length <= 105
- s consists of only uppercase English letters.
- 0 <= k <= s.length

Python

Python


```python
# Python solution using the sliding window technique
def characterReplacement(s: str, k: int) -> int:
    # Dictionary to store frequency of characters in the current window
    char_counts = {}
    max_freq = 0 # Track the maximum frequency of a single character in the window
    max_length = 0
    left = 0 # Left pointer of the sliding window

    # Expand the sliding window with the right pointer
    for right in range(len(s)):
        # Count the current character
        char_counts[s[right]] = char_counts.get(s[right], 0) + 1
        # Update the max frequency in the current window
        max_freq = max(max_freq, char_counts[s[right]])

        # Check if the number of replacements needed exceeds k
        # (current window size) - (count of most frequent character) > k
        if (right - left + 1) - max_freq > k:
            # If condition fails, shrink the window by moving left pointer
            char_counts[s[left]] -= 1
            left += 1

        # Update the maximum length found so far
        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
# print(characterReplacement("AABABBA", 1)) # Expected output: 4
```

Python

```python
class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        ans = 0
        maxCount = 0
        count = collections.Counter()

        l = 0
        for r, c in enumerate(s):
          count[c] += 1
          maxCount = max(maxCount, count[c])
          while maxCount + k < r - l + 1:
            count[s[l]] -= 1
            l += 1
          ans = max(ans, r - l + 1)

        return ans
```

Python


```python
class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        maxCount = 0
        count = collections.Counter()

        # l and r track the maximum window instead of the valid window.
        l = 0
        for r, c in enumerate(s):
          count[c] += 1
          maxCount = max(maxCount, count[c])
          while maxCount + k < r - l + 1:
            count[s[l]] -= 1
            l += 1

        return r - l + 1
```

Python

```python
class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        cnt = Counter()
        l = mx = 0
        for r, c in enumerate(s):
            cnt[c] += 1
            mx = max(mx, cnt[c])
            if r - l + 1 - mx > k:
                cnt[s[l]] -= 1
                l += 1
        return len(s) - l
```

Python

```python
class Solution:
    def characterReplacement(self, s: str, k: int) -> int:
        counter = [0] * 26
        i = j = maxCnt = 0
        while i < len(s):
            counter[ord(s[i]) - ord('A')] += 1
            maxCnt = max(maxCnt, counter[ord(s[i]) - ord('A')])
            if i - j + 1 > maxCnt + k:
                counter[ord(s[j]) - ord('A')] -= 1
                j += 1
            i += 1
        return i - j
```

[*] Sliding Window — Pattern Solution (matched from community answers)

Python


```python
# Python solution using the sliding window technique
def characterReplacement(s: str, k: int) -> int:
    # Dictionary to store frequency of characters in the current window
    char_counts = {}
    max_freq = 0 # Track the maximum frequency of a single character in the window
    max_length = 0
    left = 0 # Left pointer of the sliding window

    # Expand the sliding window with the right pointer
    for right in range(len(s)):
        # Count the current character
        char_counts[s[right]] = char_counts.get(s[right], 0) + 1
        # Update the max frequency in the current window
        max_freq = max(max_freq, char_counts[s[right]])

        # Check if the number of replacements needed exceeds k
        # (current window size) - (count of most frequent character) > k
        if (right - left + 1) - max_freq > k:
            # If condition fails, shrink the window by moving left pointer
            char_counts[s[left]] -= 1
            left += 1

        # Update the maximum length found so far
        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
# print(characterReplacement("AABABBA", 1)) # Expected output: 4
```

#438 MEDIUM

Find All Anagrams in a String

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Sliding Window

Lists: Grind 169

Problem:

Given two strings s and p, return an array of all the start indices of p's anagrams in s. You may return the answer in any order.

Example 1:

```
Input: s = "cbaebabacd", p = "abc"
Output: [0,6]
Explanation:
The substring with start index = 0 is "cba", which is an anagram of "abc".
The substring with start index = 6 is "bac", which is an anagram of "abc".
```

Example 2:

```
Input: s = "abab", p = "ab"
Output: [0,1,2]
Explanation:
The substring with start index = 0 is "ab", which is an anagram of "ab".
```



```
The substring with start index = 1 is "ba", which is an anagram of "ab".
The substring with start index = 2 is "ab", which is an anagram of "ab".
```

Constraints:

- 1 <= s.length, p.length <= 3 * 104
- s and p consist of lowercase English letters.

>> Brute Force [Sliding Window] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findAnagrams(self, s: str, p: str):
        # Brute Force O(n^2) - enumerate all windows
        n = len(s)
        best = 0
        for l in range(n):
            window_state = 0
            for r in range(l, n):
                window_state += s[r]
                best = max(best, r - l + 1)
        return best
```

Python

Python

```python
# Define the function to find all anagram indices in the string
def findAnagrams(s: str, p: str):
    # Import Counter for frequency counting
    from collections import Counter
    # List to store the starting indices of all anagrams
    result = []
    # Length of string p
    p_length = len(p)
    # Frequency count for characters in p
    p_count = Counter(p)
    # Frequency count for the current window in s
    s_count = Counter()

    # Iterate over the string s
    for i in range(len(s)):
        # Add current character to the window count
        s_count[s[i]] += 1

        # Remove the leftmost character from window if window size exceeds p_length
        if i >= p_length:
            if s_count[s[i - p_length]] == 1:
                del s_count[s[i - p_length]] # Remove key completely if count becomes 0
            else:
                s_count[s[i - p_length]] -= 1
```

```
        # Compare window count with p_count when window size is equal to p_length
        if s_count == p_count:
            result.append(i - p_length + 1)

    return result

# Example usage:
print(findAnagrams("cbaebabacd", "abc")) # Expected output: [0, 6]
```

Python

```
class Solution:
  def findAnagrams(self, s: str, p: str) -> list[int]:
    ans = []
    count = collections.Counter(p)
    required = len(p)

    for r, c in enumerate(s):
      count[c] -= 1
      if count[c] == 0:
        required -= 1
      if r >= len(p):
        count[s[r - len(p)]] += 1
        if count[s[r - len(p)]] > 0:
          required += 1
      if required == 0:
        ans.append(r - len(p) + 1)

    return ans
```

Python

```
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        m, n = len(s), len(p)
        ans = []
        if m < n:
            return ans
        cnt1 = Counter(p)
        cnt2 = Counter(s[: n - 1])
        for i in range(n - 1, m):
            cnt2[s[i]] += 1
            if cnt1 == cnt2:
                ans.append(i - n + 1)
            cnt2[s[i - n + 1]] -= 1
        return ans
```

Python


```
class Solution:
    def findAnagrams(self, s: str, p: str) -> List[int]:
        m, n = len(s), len(p)
        ans = []
        if m < n:
            return ans
        cnt1 = Counter(p)
        cnt2 = Counter(s[: n - 1])
        for i in range(n - 1, m):
            cnt2[s[i]] += 1
            if cnt1 == cnt2:
                ans.append(i - n + 1)
            cnt2[s[i - n + 1]] -= 1
        return ans
```

[*] Sliding Window — Pattern Solution (stored optimal)

Python

```
# Define the function to find all anagram indices in the string
def findAnagrams(s: str, p: str):
    # Import Counter for frequency counting
    from collections import Counter
    # List to store the starting indices of all anagrams
    result = []
    # Length of string p
    p_length = len(p)
    # Frequency count for characters in p
    p_count = Counter(p)
    # Frequency count for the current window in s
    s_count = Counter()

    # Iterate over the string s
    for i in range(len(s)):
        # Add current character to the window count
        s_count[s[i]] += 1

        # Remove the leftmost character from window if window size exceeds p_length
        if i >= p_length:
            if s_count[s[i - p_length]] == 1:
                del s_count[s[i - p_length]] # Remove key completely if count becomes 0
            else:
                s_count[s[i - p_length]] -= 1


        # Compare window count with p_count when window size is equal to p_length
        if s_count == p_count:
            result.append(i - p_length + 1)

    return result

# Example usage:
print(findAnagrams("cbaebabacd", "abc")) # Expected output: [0, 6]
```

#487 MEDIUM

Max Consecutive Ones II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming · Sliding Window

Lists: Premium 98

Problem:

Given a binary array nums, return the maximum number of consecutive 1's in the array if you can flip at most one 0.

Example 1:

```
Input: nums = [1,0,1,1,0]
Output: 4
Explanation:
- If we flip the first zero, nums becomes [1,1,1,1,0] and we have 4 consecutive ones.
- If we flip the second zero, nums becomes [1,0,1,1,1] and we have 3 consecutive ones.
The max number of consecutive ones is 4.
```

Example 2:

```
Input: nums = [1,0,1,1,0,1]
Output: 4
Explanation:
- If we flip the first zero, nums becomes [1,1,1,1,0,1] and we have 4 consecutive ones.
- If we flip the second zero, nums becomes [1,0,1,1,1,1] and we have 4 consecutive ones.
The max number of consecutive ones is 4.
```

Constraints:

- 1 <= nums.length <= 105
- nums[i] is either 0 or 1.

Follow up: What if the input numbers come in one by one as an infinite stream? In other words, you can't store all numbers coming from the stream as it's too large to hold in memory. Could you solve it efficiently?

>> Brute Force [Sliding Window] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def findMaxConsecutiveOnes(self, nums):
        # Brute Force O(n^2) - enumerate all windows
        n = len(nums)
        best = 0
        for l in range(n):
            window_state = 0
            for r in range(l, n):
                window_state += nums[r]
                best = max(best, r - l + 1)
        return best
```

Python

Python


```
# Function to find the maximum consecutive ones with at most one zero flip
def findMaxConsecutiveOnes(nums):
    max_length = 0 # Tracks the maximum length of the window
    left = 0 # Left pointer for the sliding window
    zero_count = 0 # Counter for zeros in the current window

    # Iterate with right pointer over the entire array
    for right in range(len(nums)):
        # If current element is zero, increase the zero_count
        if nums[right] == 0:
            zero_count += 1

        # If window invalid (more than one zero), shrink from left
        while zero_count > 1:
            if nums[left] == 0:
                zero_count -= 1
            left += 1

        # Update max_length if current window is larger
        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
# nums = [1,0,1,1,0]

# print(findMaxConsecutiveOnes(nums)) # Expected output: 4
```

Python

```
class Solution:
  def findMaxConsecutiveOnes(self, nums: list[int]) -> int:
    ans = 0
    zeros = 0

    l = 0
    for r, num in enumerate(nums):
      if num == 0:
        zeros += 1
      while zeros == 2:
        if nums[l] == 0:
          zeros -= 1
        l += 1
      ans = max(ans, r - l + 1)

    return ans
```

Python

```
class Solution:
    def findMaxConsecutiveOnes(self, nums: List[int]) -> int:
        l = cnt = 0
        for x in nums:
            cnt += x ^ 1
            if cnt > 1:
                cnt -= nums[l] ^ 1
                l += 1
        return len(nums) - l
```

Python


```
from collections import deque

class Solution:
    def findMaxConsecutiveOnes(self, nums: List[int]) -> int:

        res, zero, left, k = 0, 0, 0, 1

        for right in range(len(nums)):
            if nums[right] == 0:
                zero += 1
            while zero > k:
                if nums[left] == 0:
                    zero -= 1
                left += 1
            res = max(res, right - left + 1) # max-check every for iteration

        return res

class Solution: # followup, extra space
    def findMaxConsecutiveOnes(self, nums: List[int]) -> int:

        res, left, k = 0, 0, 1
        q = deque()

        for right in range(len(nums)):


            if nums[right] == 0:
                q.append(right)
            if len(q) > k:
                left = q.popleft() + 1
            res = max(res, right - left + 1) # max-check every for iteration

        return res

###########

# maintain a sliding window of size 2, that keeps track of the counts.
class Solution: # follow up, optimized
    def findMaxConsecutiveOnes(self, nums: List[int]) -> int:
        count = 0 # Count of consecutive ones
        prev_count = 0 # Count of consecutive ones before a zero
        max_count = 0 # Maximum count of consecutive ones

        for num in nums:
            if num == 1:
                count += 1
                prev_count += 1
            else:
                count = prev_count + 1
                prev_count = 0
```



```
            max_count = max(max_count, count)

            # Reset the count and prev_count if we encounter two zeros in a row
            if num == 0 and prev_count == 0:
                count = 0
                prev_count = 0

        return max_count

###########

class Solution:
    def findMaxConsecutiveOnes(self, nums: List[int]) -> int:
        l = r = 0
        k = 1
        while r < len(nums):
            if nums[r] == 0:
                k -= 1
            if k < 0:
                if nums[l] == 0:
                    k += 1
                l += 1
            r += 1
        return r - l

###########

class Solution(object):
  def __init__(self):
    self.ans = 0
```

[*] Sliding Window — Pattern Solution (stored optimal)

Python

```python
# Function to find the maximum consecutive ones with at most one zero flip
def findMaxConsecutiveOnes(nums):
    max_length = 0 # Tracks the maximum length of the window
    left = 0 # Left pointer for the sliding window
    zero_count = 0 # Counter for zeros in the current window

    # Iterate with right pointer over the entire array
    for right in range(len(nums)):
        # If current element is zero, increase the zero_count
        if nums[right] == 0:
            zero_count += 1

        # If window invalid (more than one zero), shrink from left
        while zero_count > 1:
            if nums[left] == 0:
                zero_count -= 1
            left += 1

        # Update max_length if current window is larger
        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
# nums = [1,0,1,1,0]

# print(findMaxConsecutiveOnes(nums)) # Expected output: 4
```

#1100 MEDIUM

Find K-Length Substrings with No Repeated Characters

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Sliding Window

Lists: Premium 98

Problem:

Given a string s and an integer k, return the number of substrings in s of length k with no repeated characters.

Example 1:

```
Input: s = "havefunonleetcode", k = 5
Output: 6
Explanation: There are 6 substrings they are:
'havef','avefu','vefun','efuno','etcod','tcode'.
```

Example 2:

```
Input: s = "home", k = 5
Output: 0
Explanation: Notice k can be larger than the length of s. In this case, it is not possible to
find any substring.
```

Constraints:

- 1 <= s.length <= 104

- s consists of lowercase English letters.
- 1 <= k <= 104

Python

Python

```python
def numKLenSubstrNoRepeats(s: str, k: int) -> int:
    # If k is larger than the string length, no valid substring exists.
    if k > len(s):
        return 0

    count = 0
    left = 0
    # Dictionary to store the frequency of characters in the current window.
    char_count = {}

    # Iterate over the string using right pointer.
    for right in range(len(s)):
        # Update frequency count for the current character.
        char_count[s[right]] = char_count.get(s[right], 0) + 1

        # If current character frequency > 1, slide window from the left to remove duplicates.
        while char_count[s[right]] > 1:
            char_count[s[left]] -= 1
            left += 1

        # When window size equals k, we have a valid substring.
        if right - left + 1 == k:
            count += 1
            # Slide the window by removing the leftmost character.
            char_count[s[left]] -= 1

            left += 1
    return count

# Example usage:
# print(numKLenSubstrNoRepeats("havefunonleetcode", 5)) -> 6
```

Python

```python
class Solution:
  def numKLenSubstrNoRepeats(self, s: str, k: int) -> int:
    ans = 0
    unique = 0
    count = collections.Counter()

    for i, c in enumerate(s):
      count[c] += 1
      if count[c] == 1:
        unique += 1
      if i >= k:
        count[s[i - k]] -= 1
        if count[s[i - k]] == 0:
          unique -= 1
        if unique == k:
          ans += 1

    return ans
```


Python

```python
class Solution:
    def numKLenSubstrNoRepeats(self, s: str, k: int) -> int:
        cnt = Counter(s[:k])
        ans = int(len(cnt) == k)
        for i in range(k, len(s)):
            cnt[s[i]] += 1
            cnt[s[i - k]] -= 1
            if cnt[s[i - k]] == 0:
                cnt.pop(s[i - k])
            ans += int(len(cnt) == k)
        return ans
```

Python

```python
class Solution:
    def numKLenSubstrNoRepeats(self, s: str, k: int) -> int:
        n = len(s)
        if k > n or k > 26:
            return 0
        ans = j = 0
        cnt = Counter()
        for i, c in enumerate(s):
            cnt[c] += 1
            while cnt[c] > 1 or i - j + 1 > k:
                cnt[s[j]] -= 1
                j += 1
            ans += i - j + 1 == k
        return ans
```

[*] Sliding Window — Pattern Solution (stored optimal)

Python

```python
def numKLenSubstrNoRepeats(s: str, k: int) -> int:
    # If k is larger than the string length, no valid substring exists.
    if k > len(s):
        return 0

    count = 0
    left = 0
    # Dictionary to store the frequency of characters in the current window.
    char_count = {}

    # Iterate over the string using right pointer.
    for right in range(len(s)):
        # Update frequency count for the current character.
        char_count[s[right]] = char_count.get(s[right], 0) + 1

        # If current character frequency > 1, slide window from the left to remove duplicates.
        while char_count[s[right]] > 1:
            char_count[s[left]] -= 1
            left += 1

        # When window size equals k, we have a valid substring.
        if right - left + 1 == k:
            count += 1
            # Slide the window by removing the leftmost character.
            char_count[s[left]] -= 1
```



```
            left += 1
    return count

# Example usage:
# print(numKLenSubstrNoRepeats("havefunonleetcode", 5)) -> 6
```

#1248 MEDIUM

Count Number of Nice Subarrays

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Math · Sliding Window · Prefix Sum

Lists: CodeSignal

Problem:

Given an array of integers nums and an integer k. A continuous subarray is called nice if there are k odd numbers on it.

Return the number of nice sub-arrays.

Example 1:

```
Input: nums = [1,1,2,1,1], k = 3
Output: 2
Explanation: The only sub-arrays with 3 odd numbers are [1,1,2,1] and [1,2,1,1].
```

Example 2:

```
Input: nums = [2,4,6], k = 1
Output: 0
Explanation: There are no odd numbers in the array.
```

Example 3:

```
Input: nums = [2,2,2,1,2,2,1,2,2,2], k = 2
Output: 16
```

Constraints:

- 1 <= nums.length <= 50000
- 1 <= nums[i] <= 10^5
- 1 <= k <= nums.length

>> Brute Force [Sliding Window] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def numberOfSubarrays(self, nums, k):
        # Brute force O(n^2) - enumerate all windows
        n = len(nums)
        best = 0
        for l in range(n):
            window_state = 0
            for r in range(l, n):
                window_state += nums[r]
                best = max(best, r - l + 1)
        return best
```


Python

Python

```python
# Python solution with line-by-line comments

def numberOfSubarrays(nums, k):
    # Initialize dictionary to store frequency of prefix sums
    prefix_count = {0: 1} # There is one way to have zero odd numbers initially.
    count = 0 # This will store the final count of nice subarrays
    odd_sum = 0 # Prefix sum representing the count of odd numbers so far

    # Iterate through each number in the array
    for num in nums:
        # If the number is odd, increment the odd_count prefix sum
        if num % 2 == 1:
            odd_sum += 1

        # If there is a prefix such that (current odd_sum - prefix) equals k,
        # then we found subarrays ending here with k odd numbers
        count += prefix_count.get(odd_sum - k, 0)

        # Update the frequency for the current odd_sum in the hash table
        prefix_count[odd_sum] = prefix_count.get(odd_sum, 0) + 1

    # Return the total number of nice subarrays
    return count

# Example usage:

# nums = [1,1,2,1,1]
# k = 3
# print(numberOfSubarrays(nums, k)) # Expected output is 2
```

Python

```python
class Solution:
   def numberOfSubarrays(self, nums: list[int], k: int) -> int:
     def numberOfSubarraysAtMost(k: int) -> int:
       ans = 0
       l = 0
       r = 0

       while r <= len(nums):
         if k >= 0:
           ans += r - l
           if r == len(nums):
             break
           if nums[r] & 1:
             k -= 1
           r += 1
         else:
           if nums[l] & 1:
             k += 1
           l += 1
       return ans

     return numberOfSubarraysAtMost(k) - numberOfSubarraysAtMost(k - 1)
```

Python


```python
class Solution:
    def numberOfSubarrays(self, nums: List[int], k: int) -> int:
        cnt = Counter({0: 1})
        ans = t = 0
        for v in nums:
            t += v & 1
            ans += cnt[t - k]
            cnt[t] += 1
        return ans
```

Python

```python
class Solution:
    def numberOfSubarrays(self, nums: List[int], k: int) -> int:
        cnt = Counter({0: 1})
        ans = t = 0
        for v in nums:
            t += v & 1
            ans += cnt[t - k]
            cnt[t] += 1
        return ans
```

[*] Sliding Window — Pattern Solution (stored optimal)

Python

```python
# Python solution with line-by-line comments

def numberOfSubarrays(nums, k):
    # Initialize dictionary to store frequency of prefix sums
    prefix_count = {0: 1} # There is one way to have zero odd numbers initially.
    count = 0 # This will store the final count of nice subarrays
    odd_sum = 0 # Prefix sum representing the count of odd numbers so far

    # Iterate through each number in the array
    for num in nums:
        # If the number is odd, increment the odd_count prefix sum
        if num % 2 == 1:
            odd_sum += 1

        # If there is a prefix such that (current odd_sum - prefix) equals k,
        # then we found subarrays ending here with k odd numbers
        count += prefix_count.get(odd_sum - k, 0)

        # Update the frequency for the current odd_sum in the hash table
        prefix_count[odd_sum] = prefix_count.get(odd_sum, 0) + 1

    # Return the total number of nice subarrays
    return count

# Example usage:

# nums = [1,1,2,1,1]
# k = 3
# print(numberOfSubarrays(nums, k)) # Expected output is 2
```

#76 HARD

Minimum Window Substring

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Sliding Window

Lists: Grind 169

Problem:

Given two strings s and t of lengths m and n respectively, return the minimum window substring of s such that every character in t (including duplicates) is included in the window. If there is no such substring, return the empty string "".

The testcases will be generated such that the answer is unique.

Example 1:

```
Input: s = "ADOBECODEBANC", t = "ABC"
Output: "BANC"
Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.
```

Example 2:

```
Input: s = "a", t = "a"
Output: "a"
Explanation: The entire string s is the minimum window.
```

Example 3:

```
Input: s = "a", t = "aa"
Output: ""
Explanation: Both 'a's from t must be included in the window.
Since the largest window of s only has one 'a', return empty string.
```

Constraints:

- m == s.length
- n == t.length
- 1 <= m, n <= 105
- s and t consist of uppercase and lowercase English letters.

Follow up: Could you find an algorithm that runs in O(m + n) time?

Python

Python


```python
# Python implementation of Minimum Window Substring solution

def minWindow(s: str, t: str) -> str:
    # If t is longer than s, no solution exists.
    if len(t) > len(s):
        return ""

    # Dictionary to store the frequency of each character in t.
    t_freq = {}
    for char in t:
        t_freq[char] = t_freq.get(char, 0) + 1

    # Dictionary to keep track of the characters in the current window.
    window_freq = {}

    required = len(t_freq) # Unique characters in t to be matched
    formed = 0 # Number of unique characters in the current window that meet required frequency

    # (window_length, left, right)
    answer = (float('inf'), None, None)

    l = 0 # Left pointer
    # Start sliding the window with the right pointer.
    for r, char in enumerate(s):
        # Add the current character to the window counter.

        window_freq[char] = window_freq.get(char, 0) + 1

        # Check if the current character's frequency in the window matches that in t.
        if char in t_freq and window_freq[char] == t_freq[char]:
            formed += 1

        # Try and contract the window if all characters are matched.
        while l <= r and formed == required:
            character = s[l]

            # Update answer if the current window is smaller than the previous best.
            if r - l + 1 < answer[0]:
                answer = (r - l + 1, l, r)

            # The character at the left is going to be removed from the window.
            window_freq[character] -= 1
            if character in t_freq and window_freq[character] < t_freq[character]:
                formed -= 1

            # Move the left pointer ahead.
            l += 1

    # Return the smallest window, if found.
    return "" if answer[0] == float('inf') else s[answer[1]:answer[2]+1]

# Example usage:
if __name__ == "__main__":
    s = "ADOBECODEBANC"
    t = "ABC"
    print(minWindow(s, t)) # Expected output: "BANC"
```

Python


```python
class Solution:
    def minWindow(self, s: str, t: str) -> str:
        count = collections.Counter(t)
        required = len(t)
        bestLeft = -1
        minLength = len(s) + 1

        l = 0
        for r, c in enumerate(s):
            count[c] -= 1
            if count[c] >= 0:
                required -= 1
            while required == 0:
                if r - l + 1 < minLength:
                    bestLeft = l
                    minLength = r - l + 1
                count[s[l]] += 1
                if count[s[l]] > 0:
                    required += 1
                l += 1

        return '' if bestLeft == -1 else s[bestLeft: bestLeft + minLength]
```

Python

```python
class Solution:
    def minWindow(self, s: str, t: str) -> str:
        need = Counter(t)
        window = Counter()
        cnt = l = 0
        k, mi = -1, inf
        for r, c in enumerate(s):
            window[c] += 1
            if need[c] >= window[c]:
                cnt += 1
            while cnt == len(t):
                if r - l + 1 < mi:
                    mi = r - l + 1
                    k = l
                if need[s[l]] >= window[s[l]]:
                    cnt -= 1
                window[s[l]] -= 1
                l += 1
        return "" if k < 0 else s[k : k + mi]
```

Python


```
'''
>>> deq = collections.deque([])
>>> deq.append(11)
>>> deq.append(22)
>>> deq.append(33)
>>>
>>> deq[0]
11
'''

import collections

class Solution:
    def minWindow(self, s: str, t: str) -> str:
        cnt = 0
        need = collections.Counter(t)
        start, end = len(s), 3 + len(s) # Arbitrary 3, just make sure end-start is larger than input s
length, for later min-check
        d = {}
        # Using queue to store indexes, good for a large amount of API calls
        deq = collections.deque([])
        for i, c in enumerate(s):
            if c in need:
                deq.append(i)
                d[c] = d.get(c, 0) + 1
                # '=' also +1, because it's increased already one line above :)

                if d[c] <= need[c]:
                    cnt += 1
                while deq and d[s[deq[0]]] > need[s[deq[0]]]:
                    d[s[deq.popleft()]] -= 1
                if cnt == len(t) and deq[-1] - deq[0] < end - start:
                    start, end = deq[0], deq[-1]
        return s[start:end + 1]

###########

from collections import Counter

class Solution:
    def minWindow(self, s: str, t: str) -> str:
        ans = ''
        m, n = len(s), len(t)
        if m < n:
            return ans
        need = Counter(t)
        window = Counter()
        i, cnt, mi = 0, 0, inf
        for j, c in enumerate(s):
            window[c] += 1
            if need[c] >= window[c]: # >= , because 1 line above already +=1
                cnt += 1
```



```
            while cnt == m:
                if j - i + 1 < mi: # in while
                    mi = j - i + 1
                    ans = s[i : j + 1]
            c = s[i]
            if need[c] >= window[c]: # char in window but not in need, need[c]=0, window[c]=1..2..
                cnt -= 1
            window[c] -= 1
            i += 1
        return ans
```

[*] Sliding Window — Pattern Solution (stored optimal)

Python

```python
# Python implementation of Minimum Window Substring solution

def minWindow(s: str, t: str) -> str:
    # If t is longer than s, no solution exists.
    if len(t) > len(s):
        return ""

    # Dictionary to store the frequency of each character in t.
    t_freq = {}
    for char in t:
        t_freq[char] = t_freq.get(char, 0) + 1

    # Dictionary to keep track of the characters in the current window.
    window_freq = {}

    required = len(t_freq) # Unique characters in t to be matched
    formed = 0 # Number of unique characters in the current window that meet required frequency

    # (window_length, left, right)
    answer = (float('inf'), None, None)

    l = 0 # Left pointer
    # Start sliding the window with the right pointer.
    for r, char in enumerate(s):
        # Add the current character to the window counter.
```



```
        window_freq[char] = window_freq.get(char, 0) + 1

        # Check if the current character's frequency in the window matches that in t.
        if char in t_freq and window_freq[char] == t_freq[char]:
            formed += 1

        # Try and contract the window if all characters are matched.
        while l <= r and formed == required:
            character = s[l]

            # Update answer if the current window is smaller than the previous best.
            if r - l + 1 < answer[0]:
                answer = (r - l + 1, l, r)

            # The character at the left is going to be removed from the window.
            window_freq[character] -= 1
            if character in t_freq and window_freq[character] < t_freq[character]:
                formed -= 1

            # Move the left pointer ahead.
            l += 1

    # Return the smallest window, if found.
    return "" if answer[0] == float('inf') else s[answer[1]:answer[2]+1]

# Example usage:
if __name__ == "__main__":
    s = "ADOBECODEBANC"
    t = "ABC"
    print(minWindow(s, t)) # Expected output: "BANC"
```

Quick Review — Sliding Window 9 questions · insights · complexity · solution

#3 Longest Substring Without Repeating Characters [Medium]

Key Insights

- Use the sliding window technique to maintain a window of unique characters.
- Use a hash table (or dictionary) to quickly check for duplicate characters and update window boundaries.
- The window is expanded by moving a pointer (right) and contracted by moving another pointer (left) when duplicates are encountered.
- Time complexity can be maintained at O(n) by ensuring each character is processed a limited number of times.

Space & Time

Time Complexity: O(n), where n is the length of the input string.

Space Complexity: O(min(n, m)), where m is the size of the character set (constant for fixed character sets).

Solution

The solution uses a sliding window technique with two pointers, left and right, to keep track of the current substring without repeating characters. A hash table (dictionary or map) is used to store the most recent index of each character encountered. As the right pointer moves forward, if a duplicate character is found within the current window, the left pointer is advanced to the position right after where the duplicate was last seen. This ensures that the window always contains unique characters. The maximum length is updated throughout the process.

#159 Longest Substring with At Most Two Distinct Characters [Medium]

Key Insights

- Use a sliding window to maintain a contiguous substring.
- Keep track of character frequencies within the current window using a hash map or dictionary.
- When the window contains more than two distinct characters, shrink the window from the left until only two remain.
- Update the maximum length each time the window satisfies the condition.

Space & Time

Time Complexity: O(n) where n is the length of the string, since each character is processed at most twice.

Space Complexity: O(1) because the hash map holds at most 3 entries (usually up to 2 distinct characters).

Solution

We use the sliding window technique with two pointers (left and right) to represent the current window. A hash table/dictionary records the count of each character inside the window. As we extend the right pointer, we update the count. If the hash table contains more than two distinct characters, we start moving the left pointer to reduce the number of distinct characters until we have at most two distinct keys. The length of the current valid window is then used to update our maximum length answer.

#340 Longest Substring with At Most K Distinct Characters [Medium]

Key Insights

- Use the sliding window technique to efficiently scan through the string.
- Maintain a hash table (or dictionary) that keeps track of the frequency of characters in the current window.
- When the number of distinct characters exceeds k, increment the left pointer to shrink the window until the condition is met.
- Update the maximum length of valid substrings throughout the process.

Space & Time

Time Complexity: O(n), where n is the length of the string, as each character is processed at most twice (once when expanding the window and once when contracting it).

Space Complexity: O(min(m, k)), where m is the size of the charset (or number of unique characters in s), since at most k+1 keys are stored.

Solution

This problem is solved using the sliding window technique. The algorithm keeps track of a window defined by two indices (left and right pointers). A hash map is used to count the frequency of each character in the current window. As the right pointer extends the window, the count of the character is incremented. When the count of distinct keys in the map exceeds k, the left pointer is moved forward while decrementing the count of the leftmost character, until the condition is restored. The maximum window length encountered is recorded and returned as the result.

#424 Longest Repeating Character Replacement [Medium]

Key Insights

- Use a Sliding Window algorithm to keep track of a contiguous substring.
- Maintain a frequency count of letters in the current window.
- Keep track of the count of the most frequent character in the window.
- If the current window size minus the maximum frequency is greater than k, shrink the window.
- The maximum window size seen during the process is the answer.

Space & Time

Time Complexity: O(n) where n is the length of the string, since the window traverses the string in linear time.

Space Complexity: O(1) because the frequency count for letters (A-Z) has a constant size.

Solution

The problem is solved using the sliding window technique. We initialize two pointers representing the left and right bounds of the current window. As we expand the window by moving the right pointer, we update the frequency count of the letter at the right pointer and track the letter that appears most frequently. The key observation is that the number of characters that need to be replaced in the current window is the window's size minus the count of the most frequently occurring letter. If this number exceeds k, we increment the left pointer to shrink the window until the condition is satisfied again. The maximum size of the window during the process gives the answer.

#438 Find All Anagrams in a String [Medium]

Key Insights

- Use a sliding window of length equal to p over s.
- Maintain frequency counts of characters in p and within the current window in s.
- Compare frequency counts to determine if the current window is an anagram.
- Update the window efficiently by adding one character and removing the character that goes out of the window.

Space & Time

Time Complexity: O(n) where n is the length of s.

Space Complexity: O(1) since we use fixed-size frequency arrays or hash maps.

Solution

We use the sliding window technique to iterate over s with a window size equal to the length of p. First, count the frequencies of characters in p. Then, iterate over s and update a frequency count for the current window. When the window size equals the length of p, compare the two frequency counts. If they match, record the start index. To slide the window efficiently, increment the count for the incoming character and decrement the count for the outgoing character. This way, we achieve an O(n) solution that is optimal for the problem constraints.

#487 Max Consecutive Ones II [Medium]

Key Insights

- Utilize a sliding window to maintain the current segment.

- The window can contain at most one 0.
- Expand the window with the right pointer and contract it with the left pointer when the constraint is violated.
- The maximum window size observed during this process is the desired result.

Space & Time

Time Complexity: O(n) where n is the number of elements in the array.

Space Complexity: O(1).

Solution

We use a sliding window approach to efficiently find the longest contiguous subarray that contains at most one 0. Two pointers (left and right) define the window. As we iterate with the right pointer, counting zeros in the window, we adjust the left pointer when the count of zeros exceeds one. The length of the valid window gives us the number of consecutive 1's (considering one flipped 0), and we update the maximum length accordingly.

#1100 Find K-Length Substrings with No Repeated Characters [Medium]

Key Insights

- Use a sliding window of fixed size k.
- Track character counts within the current window using a hash table (or frequency array).
- When a duplicate is found in the window, slide the window from the left to remove duplicates.
- Count a valid window when its size equals k and all characters are unique.
- Optimize by adjusting the window immediately after counting a valid substring.

Space & Time

Time Complexity: O(n) where n is the length of s, as each character is visited at most twice.

Space Complexity: O(1) since the frequency tracking is over a fixed set (26 lowercase letters).

Solution

We use a sliding window approach with two pointers: left and right. A hash table (or frequency array) tracks the occurrences of characters in the window. As we move the right pointer, we update the frequency and check for duplicates. If a duplicate is encountered, we increment the left pointer until the duplicate is removed. Once the window size exactly equals k, we have a valid substring, so we increment our count and then slide the window forward. This ensures that every possible k-length substring is examined once.

#1248 Count Number of Nice Subarrays [Medium]

Key Insights

- Convert the problem to counting subarrays with a specific "prefix sum" value where the sum represents the count of odd numbers.
- Use a hash table (or dictionary) to store the frequency of prefix sums encountered.
- The number of nice subarrays ending at the current index equals the frequency of (current prefix sum - k) seen so far.
- This approach allows you to find the answer in one pass over the array.

Space & Time

Time Complexity: O(n), where n is the number of elements in nums, because we traverse the array once.

Space Complexity: O(n), in the worst case where each prefix sum is unique and stored in the hash table.

Solution

We solve the problem using a prefix sum approach. First, we iterate through the array and convert it into a prefix sum representing the cumulative count of odd numbers encountered. As we compute the prefix sums, we use a hash table to record how many times each prefix sum appears. For every current prefix sum value, we check how many times we have seen a prefix sum equal to (current prefix sum - k). The idea is that if prefix[j] - prefix[i] equals k, then between indices i+1 and j, there are exactly k odd numbers. This gives us the count of valid subarrays ending at the current index.

This method works efficiently as it requires only a single pass through the array and maintains a running frequency count using the hash table.

#76 Minimum Window Substring [Hard]

Key Insights

- Use a sliding window approach to dynamically adjust the window boundaries.
- Utilize a hash table (or dictionary) to keep track of the frequency of characters required (from t) and the frequency within the current window.
- Expand the window until it satisfies the requirements, then contract to minimize the window.
- Efficiently update counts and check valid window with two pointers (left and right).

Space & Time

Time Complexity: O(m + n), where m is the length of s and n is the length of t.

Space Complexity: O(m + n) in the worst case due to the storage for the hash maps.

Solution

We use a sliding window technique with two pointers representing the window's boundaries. First, create a frequency map of characters from t. Then, expand the right pointer to include characters in the window until the window contains all required characters (meeting or exceeding the counts in t). Once the requirement is met, try contracting the window by moving the left pointer to reduce its size while still maintaining a valid window. Throughout the process, if a valid window is found that is smaller than the previously recorded minimum window, update the minimum. The solution leans heavily on hash tables to track counts and compares them against the required counts from t.

#6 Sorting — 9 questions · #6 of 21

#169 EASY

Majority Element

LeetCodeCC · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Divide and Conquer · Sorting · Counting

Lists: Grind 169

Problem:

Given an array nums of size n, return the majority element.

The majority element is the element that appears more than n / 2 times. You may assume that the majority element always exists in the array.

Example 1:

```
Input: nums = [3,2,3]
Output: 3
```

Example 2:

```
Input: nums = [2,2,1,1,1,2,2]
Output: 2
```

Constraints:

- n == nums.length
- 1 <= n <= 5 * 104
- -109 <= nums[i] <= 109
- The input is generated such that a majority element will exist in the array.

Follow-up: Could you solve the problem in linear time and in O(1) space?

>> Brute Force [Sorting] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def majorityElement(self, nums):
        # Brute Force O(n^2) - for each element count its frequency
        n = len(nums)
        for num in nums:
            if nums.count(num) > n // 2:
                return num
```

Python

Python


```python
# Python implementation of the Boyer-Moore Voting Algorithm

def majorityElement(nums):
    # Initialize candidate and counter
    candidate = None
    count = 0

    # Iterate over each number in the input list
    for num in nums:
        # If count is 0, choose the current number as the candidate
        if count == 0:
            candidate = num
        # Increment count if the current number equals the candidate, otherwise decrement
        count += 1 if num == candidate else -1
    # Return the candidate, which is the majority element
    return candidate

# Example usage
print(majorityElement([2,2,1,1,1,2,2])) # Output: 2
```

Python

```python
class Solution:
  def majorityElement(self, nums: list[int]) -> int:
    ans = None
    count = 0

    for num in nums:
      if count == 0:
        ans = num
      count += (1 if num == ans else -1)

    return ans
```

Python

```python
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        cnt = m = 0
        for x in nums:
            if cnt == 0:
                m, cnt = x, 1
            else:
                cnt += 1 if m == x else -1
        return m
```

Python

```python
class Solution:
    def majorityElement(self, nums: List[int]) -> int:
        cnt = m = 0
        for x in nums:
            if cnt == 0:
                m, cnt = x, 1
            else:
                cnt += 1 if m == x else -1
        return m
```

[*] Sorting — Pattern Solution (stored optimal)

Python

```python
# Python implementation of the Boyer-Moore Voting Algorithm

def majorityElement(nums):
    # Initialize candidate and counter
    candidate = None
    count = 0

    # Iterate over each number in the input list
    for num in nums:
        # If count is 0, choose the current number as the candidate
        if count == 0:
            candidate = num
        # Increment count if the current number equals the candidate, otherwise decrement
        count += 1 if num == candidate else -1
    # Return the candidate, which is the majority element
    return candidate

# Example usage
print(majorityElement([2,2,1,1,1,2,2])) # Output: 2
```

#217 EASY

Contains Duplicate

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Sorting

Lists: Grind 169

Problem:

Given an integer array nums, return true if any value appears at least twice in the array, and return false if every element is distinct.

Example 1:

Input: nums = [1,2,3,1]

Output: true

Explanation:

The element 1 occurs at the indices 0 and 3.

Example 2:

Input: nums = [1,2,3,4]

Output: false

Explanation:

All elements are distinct.

Example 3:

Input: nums = [1,1,1,3,3,4,3,2,4,2]

Output: true

Constraints:

- 1 <= nums.length <= 105
- −109 <= nums[i] <= 109

>> Brute Force [Sorting] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def contains_duplicate(self, nums):
        # Brute Force O(n^2) - compare every pair
        for i in range(len(nums)):
            for j in range(i + 1, len(nums)):
                if nums[i] == nums[j]:
                    return True
        return False
```

Python

Python

```python
# Function to determine if there are any duplicates in the list.
def contains_duplicate(nums):
    # Create a set to keep track of numbers seen so far.
    seen = set()
    # Iterate over each number in the list.
    for num in nums:
        # If num is already in seen, duplicate found.
        if num in seen:
            return True
        # Add num to the set.
        seen.add(num)
    # No duplicates found.
    return False

# Example usage:
if __name__ == "__main__":
    nums = [1, 2, 3, 1]
    print(contains_duplicate(nums)) # Expected output: True
```

Python

```python
class Solution:
  def containsDuplicate(self, nums: list[int]) -> bool:
    return len(nums) != len(set(nums))
```

Python

```python
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        return any(a == b for a, b in pairwise(sorted(nums)))
```

Python


```python
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        return len(set(nums)) < len(nums)

###########

class Solution(object):
  def containsDuplicate(self, nums):
    """
    :type nums: List[int]
    :rtype: bool
    """
    nums.sort()
    for i in range(0, len(nums) - 1):
      if nums[i] == nums[i + 1]:
        return True
    return False
```

[*] Sorting — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        return any(a == b for a, b in pairwise(sorted(nums)))
```

#242 EASY

Valid Anagram

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Sorting

Lists: Grind 169

Problem:

Given two strings s and t, return true if t is an anagram of s, and false otherwise.

Example 1:

Input: s = "anagram", t = "nagaram"

Output: true

Example 2:

Input: s = "rat", t = "car"

Output: false

Constraints:

- 1 <= s.length, t.length <= 5 * 104
- s and t consist of lowercase English letters.

Follow up: What if the inputs contain Unicode characters? How would you adapt your solution to such a case?

Python

Python

```python
# Python solution for Valid Anagram

def isAnagram(s: str, t: str) -> bool:
    # Check if lengths differ; if so, they can't be anagrams
    if len(s) != len(t):
        return False

    # Create a dictionary to count frequency of each character in s
    char_count = {}
    for char in s:
        # Increase the count of the character
        char_count[char] = char_count.get(char, 0) + 1

    # Decrease the count for each character in t
    for char in t:
        if char not in char_count or char_count[char] == 0:
            # Character not found or count mismatch, not an anagram
            return False
        char_count[char] -= 1

    # If all counts are zero, it's a valid anagram
    return True

# Example usage:
print(isAnagram("anagram", "nagaram")) # Output: True

print(isAnagram("rat", "car")) # Output: False
```

Python

```python
class Solution:
  def isAnagram(self, s: str, t: str) -> bool:
    if len(s) != len(t):
      return False

    count = collections.Counter(s)
    count.subtract(collections.Counter(t))
    return all(freq == 0 for freq in count.values())
```

Python

```python
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        if len(s) != len(t):
            return False
        cnt = Counter(s)
        for c in t:
            cnt[c] -= 1
            if cnt[c] < 0:
                return False
        return True
```

Python

```python
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        return Counter(s) == Counter(t)
```

Python


```python
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        if len(s) != len(t):
            return False
        cnt = Counter(s)
        for c in t:
            cnt[c] -= 1
            if cnt[c] < 0:
                return False
        return True
```

[*] Sorting — Pattern Solution (stored optimal)

Python

```python
# Python solution for Valid Anagram

def isAnagram(s: str, t: str) -> bool:
    # Check if lengths differ; if so, they can't be anagrams
    if len(s) != len(t):
        return False

    # Create a dictionary to count frequency of each character in s
    char_count = {}
    for char in s:
        # Increase the count of the character
        char_count[char] = char_count.get(char, 0) + 1

    # Decrease the count for each character in t
    for char in t:
        if char not in char_count or char_count[char] == 0:
            # Character not found or count mismatch, not an anagram
            return False
        char_count[char] -= 1

    # If all counts are zero, it's a valid anagram
    return True

# Example usage:
print(isAnagram("anagram", "nagaram")) # Output: True

print(isAnagram("rat", "car")) # Output: False
```

#252 EASY

Meeting Rooms

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Sorting

Lists: Grind 169 | Premium 98

Problem:

Given an array of meeting time intervals where intervals[i] = [starti, endi], determine if a person could attend all meetings.

Example 1:


```
Input: intervals = [[0,30],[5,10],[15,20]]
Output: false
```

Example 2:

```
Input: intervals = [[7,10],[2,4]]
Output: true
```

Constraints:

- 0 <= intervals.length <= 104
- intervals[i].length == 2
- 0 <= starti < endi <= 106

>> Brute Force [Sorting] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def canAttendMeetings(self, intervals):
        # Brute Force O(n^2) - check every pair of intervals for overlap
        for i in range(len(intervals)):
            for j in range(i + 1, len(intervals)):
                a, b = intervals[i], intervals[j]
                if a[0] < b[1] and b[0] < a[1]: # overlap
                    return False
        return True
```

Python

Python

```python
# Python solution for checking if one can attend all meetings

def canAttendMeetings(intervals):
    # Sort intervals based on start time
    intervals.sort(key=lambda x: x[0])
    # Iterate through intervals to find any overlap
    for i in range(1, len(intervals)):
        # Check if the previous meeting ends after the current meeting starts
        if intervals[i-1][1] > intervals[i][0]:
            return False
    return True

# Example usage:
print(canAttendMeetings([[0, 30], [5, 10], [15, 20]])) # Expected output: False
print(canAttendMeetings([[7, 10], [2, 4]])) # Expected output: True
```

Python

```python
class Solution:
  def canAttendMeetings(self, intervals: list[list[int]]) -> bool:
    intervals.sort()

    for i in range(1, len(intervals)):
      if intervals[i - 1][1] > intervals[i][0]:
        return False

    return True
```

Python

```python
class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        intervals.sort()
        return all(a[1] <= b[0] for a, b in pairwise(intervals))
```

Python

```python
class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        intervals.sort()
        return all(a[1] <= b[0] for a, b in pairwise(intervals))

###########

from itertools import pairwise
class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        intervals.sort(key=lambda x: x[0])
        return not any(a[1] > b[0] for a,b in pairwise(intervals))
        # any is True, then conflict found

###########

class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        intervals.sort(key=lambda x: x[0])
        for i in range(len(intervals) - 1):
            if intervals[i][1] > intervals[i + 1][0]:
                return False
        return True

###########

# Definition for an interval.
# class Interval(object):
# def __init__(self, s=0, e=0):
# self.start = s
# self.end = e

class Solution(object):
  def canAttendMeetings(self, intervals):
    """
    :type intervals: List[Interval]
    :rtype: bool
    """
    intervals = sorted(intervals, key=lambda x: x.start)
    for i in range(1, len(intervals)):
      if intervals[i].start < intervals[i - 1].end:
        return False
    return True
```

[*] Sorting — Pattern Solution (matched from community answers)

Python


```python
class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        intervals.sort()
        return all(a[1] <= b[0] for a, b in pairwise(intervals))

###########

from itertools import pairwise
class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        intervals.sort(key=lambda x: x[0])
        return not any(a[1] > b[0] for a,b in pairwise(intervals))
        # any is True, then conflict found

###########

class Solution:
    def canAttendMeetings(self, intervals: List[List[int]]) -> bool:
        intervals.sort(key=lambda x: x[0])
        for i in range(len(intervals) - 1):
            if intervals[i][1] > intervals[i + 1][0]:
                return False
        return True

###########

# Definition for an interval.
# class Interval(object):
# def __init__(self, s=0, e=0):
# self.start = s
# self.end = e

class Solution(object):
  def canAttendMeetings(self, intervals):
    """
    :type intervals: List[Interval]
    :rtype: bool
    """
    intervals = sorted(intervals, key=lambda x: x.start)
    for i in range(1, len(intervals)):
      if intervals[i].start < intervals[i - 1].end:
        return False
    return True
```

#1086 EASY

Largest Unique Number

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Sorting

Lists: Premium 98

Problem:

Given a list of the scores of different students, items, where items[i] = [IDi, scorei] represents one score from a student with IDi, calculate each student's top five average.

Return the answer as an array of pairs result, where result[j] = [IDj, topFiveAveragej] represents the student with IDj and their top five average. Sort result by IDj in increasing order.

A student's top five average is calculated by taking the sum of their top five scores and dividing it by 5 using integer division.

Example 1:

```
Input: items =
[[1,91],[1,92],[2,93],[2,97],[1,60],[2,77],[1,65],[1,87],[1,100],[2,100],[2,76]]
Output: [[1,87],[2,88]]
Explanation:
The student with ID = 1 got scores 91, 92, 60, 65, 87, and 100. Their top five average is (100
+ 92 + 91 + 87 + 65) / 5 = 87.
The student with ID = 2 got scores 93, 97, 77, 100, and 76. Their top five average is (100 +
97 + 93 + 77 + 76) / 5 = 88.6, but with integer division their average converts to 88.
```

Example 2:

```
Input: items =
[[1,100],[7,100],[1,100],[7,100],[1,100],[7,100],[1,100],[7,100],[1,100],[7,100]]
Output: [[1,100],[7,100]]
```

Constraints:

- 1 <= items.length <= 1000
- items[i].length == 2
- 1 <= IDi <= 1000
- 0 <= scorei <= 100
- For each IDi, there will be at least five scores.

>> Brute Force [Sorting] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def largestUniqueNumber(self, nums):
        # Brute Force O(n^2) - bubble sort
        arr = list(nums)
        n = len(arr)
        for i in range(n):
            for j in range(n - i - 1):
                if arr[j] > arr[j + 1]:
                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
        return arr
```

Python

Python


```python
class Solution:
  def highFive(self, items: list[list[int]]) -> list[list[int]]:
    idToScores = collections.defaultdict(list)

    for id, score in items:
      heapq.heappush(idToScores[id], score)
      if len(idToScores[id]) > 5:
        heapq.heappop(idToScores[id])

    return [[id, sum(scores) // 5] for id, scores in sorted(idToScores.items())]
```

Python

```python
class Solution:
    def highFive(self, items: List[List[int]]) -> List[List[int]]:
        d = defaultdict(list)
        m = 0
        for i, x in items:
            d[i].append(x)
            m = max(m, i)
        ans = []
        for i in range(1, m + 1):
            if xs := d[i]:
                avg = sum(nlargest(5, xs)) // 5
                ans.append([i, avg])
        return ans
```

Python

```python
class Solution:
    def highFive(self, items: List[List[int]]) -> List[List[int]]:
        d = defaultdict(list)
        m = 0
        for i, x in items:
            d[i].append(x)
            m = max(m, i)
        ans = []
        for i in range(1, m + 1):
            if xs := d[i]:
                avg = sum(nlargest(5, xs)) // 5
                ans.append([i, avg])
        return ans
```

[*] Sorting — Pattern Solution (matched from community answers)

Python

```python
class Solution:
  def highFive(self, items: list[list[int]]) -> list[list[int]]:
    idToScores = collections.defaultdict(list)

    for id, score in items:
      heapq.heappush(idToScores[id], score)
      if len(idToScores[id]) > 5:
        heapq.heappop(idToScores[id])

    return [[id, sum(scores) // 5] for id, scores in sorted(idToScores.items())]
```


#1122 EASY

Relative Sort Array

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Sorting · Counting Sort

Lists: Denny Zhang

Problem:

Given two arrays arr1 and arr2, the elements of arr2 are distinct, and all elements in arr2 are also in arr1.

Sort the elements of arr1 such that the relative ordering of items in arr1 are the same as in arr2. Elements that do not appear in arr2 should be placed at the end of arr1 in ascending order.

Example 1:

```
Input: arr1 = [2,3,1,3,2,4,6,7,9,2,19], arr2 = [2,1,4,3,9,6]
Output: [2,2,2,1,4,3,3,9,6,7,19]
```

Example 2:

```
Input: arr1 = [28,6,22,8,44,17], arr2 = [22,28,8,6]
Output: [22,28,8,6,17,44]
```

Constraints:

- 1 <= arr1.length, arr2.length <= 1000
- 0 <= arr1[i], arr2[i] <= 1000
- All the elements of arr2 are distinct.
- Each arr2[i] is in arr1.

>> Brute Force [Sorting] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def relativeSortArray(self, arr1, arr2):
        # Brute Force O(n^2) - bubble sort
        arr = list(arr1)
        n = len(arr)
        for i in range(n):
            for j in range(n - i - 1):
                if arr[j] > arr[j + 1]:
                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
        return arr
```

Python

Python


```python
# Python solution using a frequency dictionary

def relativeSortArray(arr1, arr2):
    # Count frequency of each element in arr1
    count = {}
    for num in arr1:
        count[num] = count.get(num, 0) + 1

    result = []
    # Process elements defined in arr2: append each element count times
    for num in arr2:
        if num in count:
            result.extend([num] * count[num])
            # Remove the element from the count dictionary
            del count[num]

    # Process remaining numbers; sort them in ascending order and append
    remaining = []
    for num, freq in count.items():
        remaining.extend([num] * freq)
    result.extend(sorted(remaining))

    return result

# Example usage:

print(relativeSortArray([2,3,1,3,2,4,6,7,9,2,19], [2,1,4,3,9,6]))
```

Python

```python
class Solution:
  def relativeSortArray(self, arr1: list[int], arr2: list[int]) -> list[int]:
    ans = []
    count = [0] * 1001

    for a in arr1:
      count[a] += 1

    for a in arr2:
      while count[a] > 0:
        ans.append(a)
        count[a] -= 1

    for num in range(1001):
      for _ in range(count[num]):
        ans.append(num)

    return ans
```

Python

```python
class Solution:
    def relativeSortArray(self, arr1: List[int], arr2: List[int]) -> List[int]:
        pos = {x: i for i, x in enumerate(arr2)}
        return sorted(arr1, key=lambda x: pos.get(x, 1000 + x))
```

Python

```
class Solution:
    def relativeSortArray(self, arr1: List[int], arr2: List[int]) -> List[int]:
        pos = {x: i for i, x in enumerate(arr2)}
        return sorted(arr1, key=lambda x: pos.get(x, 1000 + x))
```

[*] Sorting — Pattern Solution (matched from community answers)

Python

```
# Python solution using a frequency dictionary

def relativeSortArray(arr1, arr2):
    # Count frequency of each element in arr1
    count = {}
    for num in arr1:
        count[num] = count.get(num, 0) + 1

    result = []
    # Process elements defined in arr2: append each element count times
    for num in arr2:
        if num in count:
            result.extend([num] * count[num])
            # Remove the element from the count dictionary
            del count[num]

    # Process remaining numbers: sort them in ascending order and append
    remaining = []
    for num, freq in count.items():
        remaining.extend([num] * freq)
    result.extend(sorted(remaining))

    return result

# Example usage:

print(relativeSortArray([2,3,1,3,2,4,6,7,9,2,19], [2,1,4,3,9,6]))
```

#49 MEDIUM

Group Anagrams

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · String · Sorting

Lists: Grind 169

Problem:

Given an array of strings strs, group the anagrams together. You can return the answer in any order.

Example 1:

Input: strs = ["eat","tea","tan","ate","nat","bat"]

Output: [["bat"],["nat","tan"],["ate","eat","tea"]]

Explanation:

- There is no string in strs that can be rearranged to form "bat".
- The strings "nat" and "tan" are anagrams as they can be rearranged to form each other.

- The strings "ate", "eat", and "tea" are anagrams as they can be rearranged to form each other.

Example 2:

Input: strs = [""]

Output: [[""]]

Example 3:

Input: strs = ["a"]

Output: [["a"]]

Constraints:

- 1 <= strs.length <= 104
- 0 <= strs[i].length <= 100
- strs[i] consists of lowercase English letters.

>> Brute Force [Sorting] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def groupAnagrams(self, strs):
        # Brute Force O(n^2 · k log k) - compare each pair by sorting
        n = len(strs)
        used = [False] * n
        result = []
        for i in range(n):
            if used[i]: continue
            group = [strs[i]]
            used[i] = True
            for j in range(i + 1, n):
                if not used[j] and sorted(strs[i]) == sorted(strs[j]):
                    group.append(strs[j])
                    used[j] = True
            result.append(group)
        return result
```

Python

Python


```
# Python solution for grouping anagrams
from collections import defaultdict

def groupAnagrams(strs):
    # Create a dictionary to hold lists of anagrams; default to an empty list for new keys
    anagrams = defaultdict(list)
    # Iterate over each string in the input list
    for s in strs:
        # Sort the string to use as a key; sorted returns a list so join back to string
        key = "".join(sorted(s))
        # Append the original string under the sorted key
        anagrams[key].append(s)
    # Return the grouped anagrams as a list of lists
    return list(anagrams.values())

# Example usage:
if __name__ == "__main__":
    strs = ["eat","tea","tan","ate","nat","bat"]
    result = groupAnagrams(strs)
    print(result)
```

Python

```
class Solution:
    def groupAnagrams(self, strs: list[str]) -> list[list[str]]:
        dict = collections.defaultdict(list)

        for str in strs:
          key = ''.join(sorted(str))
          dict[key].append(str)

        return dict.values()
```

Python

```
class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        d = defaultdict(list)
        for s in strs:
            k = ''.join(sorted(s))
            d[k].append(s)
        return list(d.values())
```

Python


```
'''
>>> from collections import defaultdict
>>> d = defaultdict(list)
>>> d["a"]=1
>>> d["b"]=2
>>> d
defaultdict(<class 'list'>, {'a': 1, 'b': 2})
>>> d.values()
dict_values([1, 2])
>>> list(d.values())
[1, 2]

### sorted(str) will return a list of chars
>>> a = "ddfddxyz"
>>> sorted(a)
['d', 'd', 'f', 'x', 'x', 'y', 'z']
>>> "".join(sorted(a))
'dddfxxyz'
'''

class Solution:
    def groupAnagrams(self, strs: List[str]) -> List[List[str]]:
        d = defaultdict(list)
        for s in strs:
            k = "".join(sorted(s))

            d[k].append(s)
        return list(d.values())
```

[*] Sorting — Pattern Solution (matched from community answers)

Python

```
# Python solution for grouping anagrams
from collections import defaultdict

def groupAnagrams(strs):
    # Create a dictionary to hold lists of anagrams; default to an empty list for new keys
    anagrams = defaultdict(list)
    # Iterate over each string in the input list
    for s in strs:
        # Sort the string to use as a key; sorted returns a list so join back to string
        key = "".join(sorted(s))
        # Append the original string under the sorted key
        anagrams[key].append(s)
    # Return the grouped anagrams as a list of lists
    return list(anagrams.values())

# Example usage:
if __name__ == "__main__":
    strs = ["eat","tea","tan","ate","nat","bat"]
    result = groupAnagrams(strs)
    print(result)
```

#56 MEDIUM

Merge Intervals

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Sorting

Lists: Grind 169

Problem:

Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

Example 1:

```
Input: intervals = [[1,3],[2,6],[8,10],[15,18]]
Output: [[1,6],[8,10],[15,18]]
Explanation: Since intervals [1,3] and [2,6] overlap, merge them into [1,6].
```

Example 2:

```
Input: intervals = [[1,4],[4,5]]
Output: [[1,5]]
Explanation: Intervals [1,4] and [4,5] are considered overlapping.
```

Example 3:

```
Input: intervals = [[4,7],[1,4]]
Output: [[1,7]]
Explanation: Intervals [1,4] and [4,7] are considered overlapping.
```

Constraints:

- 1 <= intervals.length <= 104
- intervals[i].length == 2
- 0 <= starti <= endi <= 104

>> Brute Force [Sorting] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def merge(self, intervals):
        # Brute Force O(n^2 log n) - repeatedly merge any two overlapping intervals
        intervals.sort(key=lambda x: x[0])
        merged = True
        while merged:
            merged = False
            result = [intervals[0]]
            for iv in intervals[1:]:
                if iv[0] <= result[-1][1]:
                    result[-1][1] = max(result[-1][1], iv[1])
                    merged = True
                else:
                    result.append(iv)
            intervals = result
        return intervals
```

Python

Python


```
def merge(intervals):
    # Edge case: if the list is empty, return an empty list
    if not intervals:
        return []
    # Sort intervals based on the starting time
    intervals.sort(key=lambda x: x[0])
    merged = [] # List to store merged intervals

    # Iterate over each interval in the sorted list
    for interval in intervals:
        # If merged list is not empty and the current interval
        # overlaps with the last one in merged, merge them
        if merged and interval[0] <= merged[-1][1]:
            # Update the end of the last interval in merged if needed
            merged[-1][1] = max(merged[-1][1], interval[1])
        else:
            # No overlap, so add the current interval to merged
            merged.append(interval)
    return merged

# Example usage:
print(merge([[1,3],[2,6],[8,10],[15,18]]))
```

Python

```
class Solution:
   def merge(self, intervals: list[list[int]]) -> list[list[int]]:
      ans = []

      for interval in sorted(intervals):
        if not ans or ans[-1][1] < interval[0]:
          ans.append(interval)
        else:
          ans[-1][1] = max(ans[-1][1], interval[1])

      return ans
```

Python

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort()
        ans = []
        st, ed = intervals[0]
        for s, e in intervals[1:]:
            if ed < s:
                ans.append([st, ed])
                st, ed = s, e
            else:
                ed = max(ed, e)
        ans.append([st, ed])
        return ans
```

Python

```python
# Definition for an interval.
# class Interval(object):
# def __init__(self, s=0, e=0):
# self.start = s
# self.end = e

class Solution(object):
    def merge(self, intervals):
        """
        :type intervals: List[Interval]
        :rtype: List[Interval]
        """
        ans = []
        for intv in sorted(intervals, key=lambda x: x.start):
            if ans and ans[-1].end >= intv.start:
                ans[-1].end = max(ans[-1].end, intv.end)
            else:
                ans.append(intv)
        return ans

######

'''
>>> intervals = [[111,222],[1,3],[2,6],[8,10],[15,18]]
>>> intervals.sort()
```

```
>>> intervals
[[1, 3], [2, 6], [8, 10], [15, 18], [111, 222]]
'''
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        intervals.sort() # default sort also ok
        ans = [intervals[0]]
        for s, e in intervals[1:]:
            if ans[-1][1] >= s:
                ans.append([s, e])
            else:
                ans[-1][1] = max(ans[-1][1], e)
        return ans
```

[*] Sorting — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def merge(self, intervals: list[list[int]]) -> list[list[int]]:
        ans = []

        for interval in sorted(intervals):
            if not ans or ans[-1][1] < interval[0]:
                ans.append(interval)
            else:
                ans[-1][1] = max(ans[-1][1], interval[1])

        return ans
```

#1152 MEDIUM

Analyze User Website Visit Pattern

LeetCode · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Sorting

Lists: Premium 98

Problem:

You are given two string arrays username and website and an integer array timestamp. All the given arrays are of the same length and the tuple [username[i], website[i], timestamp[i]] indicates that the user username[i] visited the website website[i] at time timestamp[i].

A pattern is a list of three websites (not necessarily distinct).

- For example, ["home", "away", "love"], ["leetcode", "love", "leetcode"], and ["luffy", "luffy", "luffy"] are all patterns.

The score of a pattern is the number of users that visited all the websites in the pattern in the same order they appeared in the pattern.

- For example, if the pattern is ["home", "away", "love"], the score is the number of users x such that x visited "home" then visited "away" and visited "love" after that.
- Similarly, if the pattern is ["leetcode", "love", "leetcode"], the score is the number of users x such that x visited "leetcode" then visited "love" and visited "leetcode" one more time after that.
- Also, if the pattern is ["luffy", "luffy", "luffy"], the score is the number of users x such that x visited "luffy" three different times at different timestamps.

Return the pattern with the largest score. If there is more than one pattern with the same largest score, return the lexicographically smallest such pattern.

Note that the websites in a pattern do not need to be visited contiguously, they only need to be visited in the order they appeared in the pattern.

Example 1:

```
Input: username = ["joe","joe","joe","james","james","james","james","mary","mary","mary"],
timestamp = [1,2,3,4,5,6,7,8,9,10], website =
["home","about","career","home","cart","maps","home","home","about","career"]
Output: ["home","about","career"]
Explanation: The tuples in this example are:
["joe","home",1],["joe","about",2],["joe","career",3],["james","home",4],["james","cart",5],[
"james","maps",6],["james","home",7],["mary","home",8],["mary","about",9], and
["mary","career",10].
The pattern ("home", "about", "career") has score 2 (joe and mary).
The pattern ("home", "cart", "maps") has score 1 (james).
The pattern ("home", "cart", "home") has score 1 (james).
The pattern ("home", "maps", "home") has score 1 (james).
The pattern ("cart", "maps", "home") has score 1 (james).
The pattern ("home", "home", "home") has score 0 (no user visited home 3 times).
```

Example 2:

```
Input: username = ["ua","ua","ua","ub","ub","ub"], timestamp = [1,2,3,4,5,6], website =
["a","b","a","a","b","c"]
Output: ["a","b","a"]
```

Constraints:

- 3 <= username.length <= 50

- 1 <= username[i].length <= 10
- timestamp.length == username.length
- 1 <= timestamp[i] <= 109
- website.length == username.length
- 1 <= website[i].length <= 10
- username[i] and website[i] consist of lowercase English letters.
- It is guaranteed that there is at least one user who visited at least three websites.
- All the tuples [username[i], timestamp[i], website[i]] are unique.

>> Brute Force [Sorting] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def mostVisitedPattern(self, username, timestamp, website):
        # Brute Force O(n^2) - bubble sort
        arr = list(username)
        n = len(arr)
        for i in range(n):
            for j in range(n - i - 1):
                if arr[j] > arr[j + 1]:
                    arr[j], arr[j + 1] = arr[j + 1], arr[j]
        return arr
```

Python

Python

```python
# Python code solution for Analyze User Website Visit Pattern

from collections import defaultdict
from itertools import combinations

def mostVisitedPattern(username, timestamp, website):
    # Combine the logs into a list of tuples and sort by timestamp.
    logs = sorted(zip(timestamp, username, website))

    # Dictionary to hold each user's list of website visits
    user_visits = defaultdict(list)
    for time, user, site in logs:
        user_visits[user].append(site)

    # Dictionary to map a pattern (tuple of 3 websites) to set of users who visited that pattern
    pattern_users = defaultdict(set)

    # For each user, generate all unique 3-sequences from their visit list and update pattern_users
    for user, sites in user_visits.items():
        # Use set to avoid counting duplicate patterns from the same user multiple times
        seen_patterns = set()
        if len(sites) < 3:
            continue
        # Generate all 3-sequence combinations (order matters since visits are in time order)
        for pattern in combinations(sites, 3):
```



```
            if pattern not in seen_patterns:
                seen_patterns.add(pattern)
                pattern_users[pattern].add(user)

    # Find the pattern with the highest score (i.e., most users)
    # In case of a tie, lexicographically smallest pattern wins.
    max_count = 0
    result = None
    for pattern, users in pattern_users.items():
        count = len(users)
        if count > max_count or (count == max_count and (result is None or pattern < result)):
            max_count = count
            result = pattern

    return list(result)

# Example usage:
print(mostVisitedPattern(
    ["joe","joe","joe","james","james","james","james","mary","mary","mary"],
    [1,2,3,4,5,6,7,8,9,10],
    ["home","about","career","home","cart","maps","home","home","about","career"]
))
```

Python

```python
class Solution:
    def mostVisitedPattern(
        self,
        username: list[str],
        timestamp: list[int],
        website: list[str],
    ) -> list[str]:
        userToSites = collections.defaultdict(list)

        # Sort websites of each user by timestamp.
        for user, _, site in sorted(
                zip(username, timestamp, website),
                key=lambda x: x[1]):
            userToSites[user].append(site)

        # For each of three websites, count its frequency.
        patternCount = collections.Counter()

        for user, sites in userToSites.items():
            patternCount.update(Counter(set(itertools.combinations(sites, 3))))

        return max(sorted(patternCount), key=patternCount.get)
```

Python


```python
class Solution:
    def mostVisitedPattern(
        self, username: List[str], timestamp: List[int], website: List[str]
    ) -> List[str]:
        d = defaultdict(list)
        for user, _, site in sorted(
            zip(username, timestamp, website), key=lambda x: x[1]
        ):
            d[user].append(site)

        cnt = Counter()
        for sites in d.values():
            m = len(sites)
            s = set()
            if m > 2:
                for i in range(m - 2):
                    for j in range(i + 1, m - 1):
                        for k in range(j + 1, m):
                            s.add((sites[i], sites[j], sites[k]))
            for t in s:
                cnt[t] += 1
        return sorted(cnt.items(), key=lambda x: (-x[1], x[0]))[0][0]
```

Python

```python
class Solution:
    def mostVisitedPattern(
        self, username: List[str], timestamp: List[int], website: List[str]
    ) -> List[str]:
        d = defaultdict(list)
        for user, _, site in sorted(
            zip(username, timestamp, website), key=lambda x: x[1]
        ):
            d[user].append(site)

        cnt = Counter()
        for sites in d.values():
            m = len(sites)
            s = set()
            if m > 2:
                for i in range(m - 2):
                    for j in range(i + 1, m - 1):
                        for k in range(j + 1, m):
                            s.add((sites[i], sites[j], sites[k]))
            for t in s:
                cnt[t] += 1
        return sorted(cnt.items(), key=lambda x: (-x[1], x[0]))[0][0]
```

[*] Sorting — Pattern Solution (matched from community answers)

Python


```python
# Python code solution for Analyze User Website Visit Pattern

from collections import defaultdict
from itertools import combinations

def mostVisitedPattern(username, timestamp, website):
    # Combine the logs into a list of tuples and sort by timestamp.
    logs = sorted(zip(timestamp, username, website))

    # Dictionary to hold each user's list of website visits
    user_visits = defaultdict(list)
    for time, user, site in logs:
        user_visits[user].append(site)

    # Dictionary to map a pattern (tuple of 3 websites) to set of users who visited that pattern
    pattern_users = defaultdict(set)

    # For each user, generate all unique 3-sequences from their visit list and update pattern_users
    for user, sites in user_visits.items():
        # Use set to avoid counting duplicate patterns from the same user multiple times
        seen_patterns = set()
        if len(sites) < 3:
            continue
        # Generate all 3-sequence combinations (order matters since visits are in time order)
        for pattern in combinations(sites, 3):
```

```
            if pattern not in seen_patterns:
                seen_patterns.add(pattern)
                pattern_users[pattern].add(user)

    # Find the pattern with the highest score (i.e., most users)
    # In case of a tie, lexicographically smallest pattern wins.
    max_count = 0
    result = None
    for pattern, users in pattern_users.items():
        count = len(users)
        if count > max_count or (count == max_count and (result is None or pattern < result)):
            max_count = count
            result = pattern

    return list(result)

# Example usage:
print(mostVisitedPattern(
    ["joe","joe","joe","james","james","james","james","mary","mary","mary"],
    [1,2,3,4,5,6,7,8,9,10],
    ["home","about","career","home","cart","maps","home","home","about","career"]
))
```

Quick Review — Sorting 9 questions · insights · complexity · solution

#169 Majority Element [Easy]

Key Insights

- The majority element appears more than half the length of the array.
- A simple hash table approach can count occurrences, but a more efficient solution exists.
- The Boyer-Moore Voting Algorithm finds the majority in O(n) time and O(1) space.
- The algorithm maintains a candidate and a counter, adjusting the candidate when the count drops to zero.

Space & Time

Time Complexity: O(n)

Space Complexity: O(1)

Solution

We use the Boyer-Moore Voting Algorithm to identify the majority element. The algorithm iterates through the array using two variables: one for the candidate and one for the count. Initially, we set the candidate to an arbitrary value and count to 0. For each element, if the count is 0, we update the candidate to the current element. Then, if the current element is the same as the candidate, we increment the count; otherwise, we decrement it. Since the majority element appears more than n/2 times, it will remain as the candidate by the end of the iteration.

#217 Contains Duplicate [Easy]

Key Insights

- A hash set can be used to keep track of seen numbers.
- Iterating once through the array provides an efficient solution.
- If a number is already in the set, a duplicate is found.
- Alternative approaches like sorting have a higher time complexity (O(n log n)).

Space & Time

Time Complexity: O(n)

Space Complexity: O(n)

Solution

The solution uses a hash set to track the numbers seen while iterating through the array. For each number in the array:

- Check if it exists in the hash set. - If it does, return true immediately as a duplicate is found. - Otherwise, add the number to the hash set. If the iteration completes without finding any duplicates, return false. This approach efficiently determines if any duplicates exist in a single pass.

#242 Valid Anagram [Easy]

Key Insights

- If s and t have different lengths, they cannot be anagrams.
- An anagram requires exactly the same character frequency for both strings.
- Using a hash table (or frequency array when limited to 26 lowercase letters) provides an efficient solution.
- An alternative approach is to sort both strings and compare them.
- For Unicode inputs, a hash table is more adaptable than a fixed-size frequency array.

Space & Time

Time Complexity: O(n), where n is the length of the strings (using a hash table for counting).

Space Complexity: O(1) if we use a fixed-size array (for 26 letters), otherwise O(n) for a hash table with a larger character set.

Solution

We use a hash table (dictionary) to count the frequency of each character in the first string s. Then, we iterate over string t and decrement the count for each character. If at any point a character count goes negative or a character is missing in the hash table, t cannot be an anagram of s. Finally, if all counts return to zero, the strings are anagrams. This approach is efficient and handles the problem in linear time.

#252 Meeting Rooms [Easy]

Key Insights

- Sort the intervals based on their starting times.
- Compare each interval's end time with the next interval's start time.
- Detect overlap if the end time of one meeting exceeds the start time of the following meeting.
- If no overlaps are detected, all meetings can be attended.

Space & Time

Time Complexity: O(n log n) due to sorting the intervals.

Space Complexity: O(1) if sorting is done in-place; otherwise O(n).

Solution

The solution starts by sorting the intervals in ascending order by their start times. After sorting, iterate through the intervals and for each consecutive pair, check if the previous meeting's end time exceeds the next meeting's start time. An overlap indicates that the person cannot attend all meetings. The primary data structure used is an array (or list) for holding the intervals. Sorting makes it easier to compare adjacent intervals for potential overlaps.

#1086 Largest Unique Number [Easy]

Key Insights

- Use a hash table (or dictionary) to count the frequency of each number.
- Iterate over the hash table to find the numbers that appear exactly once.
- Identify the maximum number among the unique numbers.
- Return -1 if no unique number exists.

Space & Time

Time Complexity: O(n), where n is the number of elements in nums.

Space Complexity: O(n), for the hash table storing frequency counts.

Solution

The approach starts by initializing a hash table to keep track of the frequency of each element in the array. Once the frequency count is complete, we traverse the hash table to collect numbers that appear exactly once. Finally, we determine the maximum number among these unique elements. If there is no unique element, we return -1. This method is efficient because the frequency count and the search for the maximum unique element both run in linear time relative to the input size.

#1122 Relative Sort Array [Easy]

Key Insights

- Use a hash table (or array for counting given the value constraints) to count the occurrences of each element in arr1.
- Iterate through arr2 and for each element, append it to the result based on its count from the hash table.
- Remove the used elements from the hash table.
- Sort the remaining elements (those not in arr2) in ascending order and append them to the result.

Space & Time

Time Complexity: O(n + m log m) where n is the length of arr1 and m is the number of leftover elements not in arr2. Given that element values are bounded (0 to 1000), a counting sort can make this nearly O(n).

Space Complexity: O(n) for storing the frequency counts and the result array.

Solution

The solution leverages a frequency counter (hash table) to count how many times each element appears in arr1. We then process arr2 to add each number in its given order according to its frequency. Finally, we sort the remaining numbers that are not in arr2 and append them to the result array. This approach ensures that the relative order specified by arr2 is maintained and that the other elements are sorted in ascending order.

#49 Group Anagrams [Medium]

Key Insights

- Anagrams have the same characters when sorted alphabetically.
- Sorting each string provides a unique key for grouping.
- Use a hash table (dictionary) to map each sorted key to a list of its original strings.
- Finally, gather all the lists from the dictionary to form the result.

Space & Time

Time Complexity: O(n * k log k), where n is the number of strings and k is the maximum length of a string (due to sorting each string).

Space Complexity: O(n * k) for storing the grouped strings in the hash table.

Solution

The primary idea is to iterate through each string, sort it to obtain a key that represents its character composition, and then group the original string under this key in a hash table. After processing all strings, the values of the hash table will be the groups of anagrams. This approach efficiently groups anagrams by leveraging sorting and a dictionary for constant-time lookups.

#56 Merge Intervals [Medium]

Key Insights

- Sort the intervals based on their start values.
- Use a pointer or iteration to compare the current interval with the last merged interval.
- If the current interval overlaps with the previous one, merge them by updating the end time.
- Otherwise, add the current interval to the results as a new merged interval.

Space & Time

Time Complexity: O(n log n) due to sorting the intervals.

Space Complexity: O(n) to store the merged intervals.

Solution

First, sort the array of intervals based on the starting time. Then, iterate through the sorted list and check if the current interval overlaps with the last interval in the merged list. If an overlap is detected (i.e., the start of the current interval is less than or equal to the end of the last merged interval), update the last merged interval's end to be the maximum of both ends. If there is no overlap, simply add the current interval to the merged list. This approach efficiently merges overlapping intervals and ensures that all intervals are processed in a single pass after sorting.

#1152 Analyze User Website Visit Pattern [Medium]

Key Insights

- Combine the inputs into a single list of events and sort them by timestamp.
- Group the visits by each user, preserving the sorted order.
- Generate all unique combinations of 3-sequences for each user (order matters and non-contiguous visits are allowed).

- Use a dictionary (or hash map) to map each 3-sequence pattern to the set of users who visited that pattern.
- The problem constraints are small, so generating combinations per user (even using triple nested loops) is acceptable.
- When multiple patterns have the same frequency, choose the lexicographically smallest one.

Space & Time

Time Complexity: O(U * L^3) where U is the number of users and L is the maximum length of visits per user (since we generate combinations of 3 visits per user). Sorting all events costs O(N log N) for N total events.

Space Complexity: O(P) where P is the number of unique 3-sequence patterns stored in the dictionary, plus O(N) for grouping the events.

Solution

- Aggregate the events by user while sorting them based on timestamp. - For each user, generate all sets of 3-website patterns (to avoid duplicate patterns per user). - Use a hash map to count the number of unique users for each pattern. - Traverse the map to find the pattern with the highest count. In case of a tie, compare patterns lexicographically. - Return the pattern that meets the criteria.

#7 Bit Manipulation — 10 questions · #7 of 21

#67 EASY

Add Binary

LeetDoocs · SimplyLuet · WalkCC · LeetCode · Editorial

Math · String · Bit Manipulation · Simulation

Lists: Grind 169

Problem:

Given two binary strings a and b, return their sum as a binary string.

Example 1:

```
Input: a = "11", b = "1"
Output: "100"
```

Example 2:

```
Input: a = "1010", b = "1011"
Output: "10101"
```

Constraints:

- 1 <= a.length, b.length <= 104
- a and b consist only of '0' or '1' characters.
- Each string does not contain leading zeros except for the zero itself.

Python

Python

```python
# Function to add two binary strings
def addBinary(a: str, b: str) -> str:
    i = len(a) - 1 # Pointer for string a
    j = len(b) - 1 # Pointer for string b
    carry = 0 # Initialize carry to 0
    result = [] # List to store binary digits

    # Loop until both strings are processed and no carry remains
    while i >= 0 or j >= 0 or carry:
        total_sum = carry # Start with the carry from previous step

        if i >= 0: # If there is a digit left in a
            total_sum += int(a[i]) # Convert character to integer and add
            i -= 1 # Move to the next digit
        if j >= 0: # If there is a digit left in b
            total_sum += int(b[j]) # Convert character to integer and add
            j -= 1 # Move to the next digit

        # Append the remainder of total_sum divided by 2 (0 or 1)
        result.append(str(total_sum % 2))
        carry = total_sum // 2 # Update the carry

    # Reverse the result since we built it backwards and join to form a string
    return ''.join(result[::-1])
```



```python
# Sample test cases
print(addBinary("11", "1")) # Expected output: "100"
print(addBinary("1010", "1011")) # Expected output: "10101"
```

Python

```python
class Solution:
    def addBinary(self, a: str, b: str) -> str:
        ans = []
        carry = 0
        i = len(a) - 1
        j = len(b) - 1

        while i >= 0 or j >= 0 or carry:
            if i >= 0:
                carry += int(a[i])
                i -= 1
            if j >= 0:
                carry += int(b[j])
                j -= 1
            ans.append(str(carry % 2))
            carry //= 2

        return ''.join(reversed(ans))
```

Python

```python
class Solution:
    def addBinary(self, a: str, b: str) -> str:
        ans = []
        i, j, carry = len(a) - 1, len(b) - 1, 0
        while i >= 0 or j >= 0 or carry:
            carry += (0 if i < 0 else int(a[i])) + (0 if j < 0 else int(b[j]))
            carry, v = divmod(carry, 2)
            ans.append(str(v))
            i, j = i - 1, j - 1
        return "".join(ans[::-1])
```

Python

```python
class Solution:
    def addBinary(self, a: str, b: str) -> str:
        ans = []
        i, j, carry = len(a) - 1, len(b) - 1, 0
        while i >= 0 or j >= 0 or carry:
            carry += (0 if i < 0 else int(a[i])) + (0 if j < 0 else int(b[j]))
            carry, v = divmod(carry, 2)
            ans.append(str(v))
            i, j = i - 1, j - 1
        return "".join(ans[::-1])
```

[*] Bit Manipulation — Pattern Solution (stored optimal)

Python

```python
# Function to add two binary strings
def addBinary(a: str, b: str) -> str:
    i = len(a) - 1 # Pointer for string a
    j = len(b) - 1 # Pointer for string b
    carry = 0 # Initialize carry to 0
    result = [] # List to store binary digits

    # Loop until both strings are processed and no carry remains
    while i >= 0 or j >= 0 or carry:
        total_sum = carry # Start with the carry from previous step

        if i >= 0: # If there is a digit left in a
            total_sum += int(a[i]) # Convert character to integer and add
            i -= 1 # Move to the next digit
        if j >= 0: # If there is a digit left in b
            total_sum += int(b[j]) # Convert character to integer and add
            j -= 1 # Move to the next digit

        # Append the remainder of total_sum divided by 2 (0 or 1)
        result.append(str(total_sum % 2))
        carry = total_sum // 2 # Update the carry

    # Reverse the result since we built it backwards and join to form a string
    return ''.join(result[::-1])
```

```python
# Sample test cases
print(addBinary("11", "1")) # Expected output: "100"
print(addBinary("1010", "1011")) # Expected output: "10101"
```

#136 EASY

Single Number

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Bit Manipulation

Lists: Grind 169

Problem:

Given a non-empty array of integers nums, every element appears twice except for one. Find that single one.

You must implement a solution with a linear runtime complexity and use only constant extra space.

Example 1:

Input: nums = [2,2,1]

Output: 1

Example 2:

Input: nums = [4,1,2,1,2]

Output: 4

Example 3:

Input: nums = [1]

Output: 1

Constraints:

- 1 <= nums.length <= 3 * 104
- -3 * 104 <= nums[i] <= 3 * 104
- Each element in the array appears twice except for one element which appears only once.

>> Brute Force [Bit Manipulation] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def singleNumber(self, nums):
        # Brute Force O(n^2) - for each element, count its occurrences
        for num in nums:
            if nums.count(num) == 1:
                return num
```

Python

Python

```python
# Function to find the single number in the list
def singleNumber(nums):
    # Initialize result to 0; any number XOR 0 is the number itself
    result = 0
    # Iterate through every number in the list
    for num in nums:
        # XOR the current result with the current number
        result ^= num # Using XOR to cancel duplicates
    # Return the unique number that did not cancel out
    return result

# Example usage
nums = [4, 1, 2, 1, 2]
print(singleNumber(nums)) # Output should be 4
```

Python

```python
class Solution:
    def singleNumber(self, nums: list[int]) -> int:
        return functools.reduce(operator.xor, nums, 0)
```

Python

```python
class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        return reduce(xor, nums)
```

Python


```python
'''
>>> from functools import reduce
>>> reduce(lambda x, y: x ^ y, [3, 5, 3])
5
'''

from functools import reduce

class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        return reduce(lambda x, y: x ^ y, nums)

###########

class Solution(object):
    def singleNumber(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        for i in range(1, len(nums)):
            nums[0] ^= nums[i]
        return nums[0]
```

[*] Bit Manipulation — Pattern Solution (matched from community answers)

Python

```python
'''
>>> from functools import reduce
>>> reduce(lambda x, y: x ^ y, [3, 5, 3])
5
'''

from functools import reduce

class Solution:
    def singleNumber(self, nums: List[int]) -> int:
        return reduce(lambda x, y: x ^ y, nums)

###########

class Solution(object):
    def singleNumber(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        for i in range(1, len(nums)):
            nums[0] ^= nums[i]
        return nums[0]
```

#190 EASY

Reverse Bits

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Divide and Conquer · Bit Manipulation

Lists: Grind 169

Problem:

Reverse bits of a given 32 bits signed integer.

Example 1:

Input: n = 43261596

Output: 964176192

Explanation:

Example 2:

Input: n = 2147483644

Output: 1073741822

Explanation:

Constraints:

- 0 <= n <= 231 - 2
- n is even.

Follow up: If this function is called many times, how would you optimize it?

Python

Python

```python
# Function to reverse bits of a 32-bit binary string.
def reverse_bits(n: str) -> int:
    # Reverse the binary string using slicing.
    reversed_binary = n[::-1]
    # Convert the reversed binary string to an integer.
    result = int(reversed_binary, 2)
    return result

# Example usage:
binary_input = "00000010100101000001111010011100"
print(reverse_bits(binary_input)) # Expected output: 964176192
```

Python

```python
class Solution:
    def reverseBits(self, n: int) -> int:
        ans = 0

        for i in range(32):
            if n >> i & 1:
                ans |= 1 << 31 - i

        return ans
```

Python


```python
class Solution:
    def reverseBits(self, n: int) -> int:
        ans = 0
        for i in range(32):
            ans |= (n & 1) << (31 - i)
            n >>= 1
        return ans
```

Python

```python
class Solution:
    def reverseBits(self, n: int) -> int:
        res = 0
        for i in range(32):
            res |= (n & 1) << (31 - i)
            n >>= 1
        return res
```

[*] Bit Manipulation — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def reverseBits(self, n: int) -> int:
        ans = 0

        for i in range(32):
            if n >> i & 1:
                ans |= 1 << 31 - i

        return ans
```

#191 EASY

Number of 1 Bits

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Divide and Conquer · Bit Manipulation

Lists: Grind 169

Problem:

Given a positive integer n, write a function that returns the number of set bits in its binary representation (also known as the Hamming weight).

Example 1:

Input: n = 11

Output: 3

Explanation:

The input binary string 1011 has a total of three set bits.

Example 2:

Input: n = 128

Output: 1

Explanation:

The input binary string 10000000 has a total of one set bit.

Example 3:

Input: n = 2147483645

Output: 30

Explanation:

The input binary string 11111111111111111111111111111101 has a total of thirty set bits.

Constraints:

- 1 <= n <= 231 - 1

Follow up: If this function is called many times, how would you optimize it?

Python

Python

```python
# Function to count the number of set bits in a given integer n
def hammingWeight(n: int) -> int:
    count = 0 # Initialize count of set bits
    while n:
        # Remove the lowest set bit from n
        n &= n - 1
        count += 1 # Increment count for each set bit removed
    return count

# Example usage:
print(hammingWeight(11)) # Output: 3
```

Python

```python
class Solution:
    def hammingWeight(self, n: int) -> int:
        ans = 0

        for i in range(32):
            if (n >> i) & 1:
                ans += 1

        return ans
```

Python

```python
class Solution:
    def hammingWeight(self, n: int) -> int:
        return n.bit_count()
```

Python

```python
class Solution:
    def hammingWeight(self, n: int) -> int:
        ans = 0
        while n:
            n &= n - 1
            ans += 1
        return ans
```

Python

```python
class Solution:
    def hammingWeight(self, n: int) -> int:
        ans = 0
        while n:
            n &= n - 1
            ans += 1
        return ans
```

[*] Bit Manipulation — Pattern Solution (matched from community answers)

Python

```python
class Solution:
   def hammingWeight(self, n: int) -> int:
     ans = 0

     for i in range(32):
       if (n >> i) & 1:
         ans += 1

     return ans
```

#268 EASY

Missing Number

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Math · Binary Search · Bit Manipulation

Lists: Grind 169

Problem:

Given an array nums containing n distinct numbers in the range [0, n], return the only number in the range that is missing from the array.

Example 1:

Input: nums = [3,0,1]

Output: 2

Explanation:

n = 3 since there are 3 numbers, so all numbers are in the range [0,3]. 2 is the missing number in the range since it does not appear in nums.

Example 2:

Input: nums = [0,1]

Output: 2

Explanation:

n = 2 since there are 2 numbers, so all numbers are in the range [0,2]. 2 is the missing number in the range since it does not appear in nums.

Example 3:

Input: nums = [9,6,4,2,3,5,7,0,1]

Output: 8

Explanation:

n = 9 since there are 9 numbers, so all numbers are in the range [0,9]. 8 is the missing number in the range since it does not appear in nums.

Constraints:

- n == nums.length
- 1 <= n <= 104
- 0 <= nums[i] <= n
- All the numbers of nums are unique.

Follow up: Could you implement a solution using only O(1) extra space complexity and O(n) runtime complexity?

>> Brute Force [Bit Manipulation] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def missingNumber(self, nums):
        # Brute Force O(n^2) - for each 0..n check if it's in nums
        for i in range(len(nums) + 1):
            if i not in nums:
                return i
```

Python

Python

```python
# Function to find the missing number in the array
def missingNumber(nums):
    # Calculate n, where n is the length of the nums array
    n = len(nums)
    # Compute the expected total sum using the arithmetic sum formula for numbers 0 to n
    expected_sum = n * (n + 1) // 2
    # Compute the actual sum of the numbers in the array
    actual_sum = sum(nums)
    # The missing number is the difference between expected and actual sums
    return expected_sum - actual_sum

# Example Usage
print(missingNumber([3, 0, 1])) # Output: 2
```

Python

```python
class Solution:
   def missingNumber(self, nums: list[int]) -> int:
     ans = len(nums)

     for i, num in enumerate(nums):
       ans ^= i ^ num

     return ans
```

Python


```python
class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        return reduce(xor, (i ^ v for i, v in enumerate(nums, 1)))
```

Python

```python
class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        n = len(nums)
        return (1 + n) + n // 2 - sum(nums)
```

Python

```python
class Solution:
    def missingNumber(self, nums: List[int]) -> int:
        return reduce(xor, (i ^ v for i, v in enumerate(nums, 1)))
```

[*] Bit Manipulation — Pattern Solution (matched from community answers)

Python

```python
class Solution:
   def missingNumber(self, nums: list[int]) -> int:
     ans = len(nums)

     for i, num in enumerate(nums):
       ans ^= i ^ num

     return ans
```

#338 EASY

Counting Bits

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Dynamic Programming · Bit Manipulation

Lists: Grind 169

Problem:

Given an integer n, return an array ans of length n + 1 such that for each i (0 <= i <= n), ans[i] is the number of 1's in the binary representation of i.

Example 1:

```
Input: n = 2
Output: [0,1,1]
Explanation:
0 --> 0
1 --> 1
2 --> 10
```

Example 2:

```
Input: n = 5
Output: [0,1,1,2,1,2]
Explanation:
0 --> 0
```



```
1 --> 1
2 --> 10
3 --> 11
4 --> 100
5 --> 101
```

Constraints:

- 0 <= n <= 105

Follow up:

- It is very easy to come up with a solution with a runtime of O(n log n). Can you do it in linear time O(n) and possibly in a single pass?
- Can you do it without using any built-in function (i.e., like __builtin_popcount in C++)?

>> Brute Force [Bit Manipulation] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def countBits(self, n):
        # Brute Force O(n · 32) - count 1-bits for each number 0..n individually
        def count_ones(x):
            c = 0
            while x:
                c += x & 1
                x >>= 1
            return c
        return [count_ones(i) for i in range(n + 1)]
```

Python

Python

```python
def countBits(n):
    # Create a list to store the number of 1's for each number from 0 to n
    dp = [0] * (n + 1)
    # Loop over each number starting from 1, since dp[0] is already 0
    for i in range(1, n + 1):
        # dp[i] is computed as the count for i//2 plus 1 if i is odd (i & 1)
        dp[i] = dp[i >> 1] + (i & 1)
    return dp

# Example usage:
# print(countBits(5)) => [0, 1, 1, 2, 1, 2]
```

Python

```python
class Solution:
   def countBits(self, n: int) -> list[int]:
     # f(i) := i's number of 1s in bitmask
     # f(i) = f(i // 2) + i % 2
     ans = [0] * (n + 1)

     for i in range(1, n + 1):
       ans[i] = ans[i // 2] + (i & 1)

     return ans
```


Python

```python
class Solution:
    def countBits(self, n: int) -> List[int]:
        return [i.bit_count() for i in range(n + 1)]
```

Python

```python
class Solution:
    def countBits(self, n: int) -> List[int]:
        ans = [0] * (n + 1)
        for i in range(1, n + 1):
            ans[i] = ans[i & (i - 1)] + 1
        return ans
```

Python

```python
class Solution:
    def countBits(self, n: int) -> List[int]:
        ans = [0] * (n + 1)
        for i in range(1, n + 1):
            ans[i] = ans[i & (i - 1)] + 1
        return ans
```

[*] Bit Manipulation — Pattern Solution (matched from community answers)

Python

```python
def countBits(n):
    # Create a list to store the number of 1's for each number from 0 to n
    dp = [0] * (n + 1)
    # Loop over each number starting from 1, since dp[0] is already 0
    for i in range(1, n + 1):
        # dp[i] is computed as the count for i//2 plus 1 if i is odd (i & 1)
        dp[i] = dp[i >> 1] + (i & 1)
    return dp

# Example usage:
# print(countBits(5)) => [0, 1, 1, 2, 1, 2]
```

#78 MEDIUM

Subsets

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Backtracking · Bit Manipulation

Lists: Grind 169

Problem:

Given an integer array nums of unique elements, return all possible subsets (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

```
Input: nums = [1,2,3]
Output: [[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]
```

Example 2:


```
Input: nums = [0]
Output: [[],[0]]
```

Constraints:

- 1 <= nums.length <= 10
- -10 <= nums[i] <= 10
- All the numbers of nums are unique.

>> Brute Force [Bit Manipulation] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def subsets(self, nums):
        # Brute Force O(2^n · n) - bitmask over all 2^n subsets
        result = []
        for mask in range(1 << len(nums)):
            sub = [nums[i] for i in range(len(nums)) if mask & (1 << i)]
            result.append(sub)
        return result
```

Python

Python

```python
# Define the function to generate all subsets using backtracking.
def subsets(nums):
    # This list will store all the subsets
    result = []

    # Recursive helper function to generate subsets.
    def backtrack(start, current):
        # Append a copy of the current subset to the final result.
        result.append(current.copy())
        # Iterate over the possible next elements starting from index 'start'
        for i in range(start, len(nums)):
            # Include nums[i] in the current subset.
            current.append(nums[i])
            # Recurse with the next starting index.
            backtrack(i + 1, current)
            # Backtrack: remove the last element to explore other possibilities.
            current.pop()

    # Start the recursion with an empty path.
    backtrack(0, [])
    return result

# Example usage:
print(subsets([1, 2, 3]))
```

Python

```python
class Solution:
   def subsets(self, nums: list[int]) -> list[list[int]]:
      ans = []

      def dfs(s: int, path: list[int]) -> None:
        ans.append(path)

        for i in range(s, len(nums)):
          dfs(i + 1, path + [nums[i]])

      dfs(0, [])
      return ans
```

Python

```python
class Solution:
    def subsets(self, nums: List[int]) -> List[List[int]]:
        def dfs(i: int):
            if i == len(nums):
                ans.append(t[:])
                return
            dfs(i + 1)
            t.append(nums[i])
            dfs(i + 1)
            t.pop()

        ans = []
        t = []
        dfs(0)
        return ans
```

Python

```python
from typing import List

class Solution: # bfs
    def subsets(self, nums: List[int]) -> List[List[int]]:
        if not nums:
            return [[]]

        nums.sort() # sort() not necessary if no duplicates
        result = [[]]

        for num in nums:
            result += [subset + [num] for subset in result]

        return result


class Solution: # dfs
    def subsets(self, nums: List[int]) -> List[List[int]]:
        def dfs(u, t):
            ans.append(t[:]) # or, t.copy()
            for i in range(u, len(nums)):
                t.append(nums[i])
                dfs(i + 1, t)
                t.pop()
```



```python
        ans = []
        nums.sort() # sort() not necessary if no duplicates
        dfs(0, [])
        return ans


class Solution: # dfs, just pass down the final path
    def subsets(self, nums: List[int]) -> List[List[int]]:
        def dfs(nums, index, path, ans):
            ans.append(path)
            [dfs(nums, i + 1, path + [nums[i]], ans) for i in range(index, len(nums))]

        ans = []
        dfs(nums, 0, [], ans)
        return ans


class Solution: # dfs, but running slower, since need to reference from parent method for 'nums' and 'ans'
    def subsets(self, nums: List[int]) -> List[List[int]]:
        def dfs(index, path):
            ans.append(path)
            [dfs(i + 1, path + [nums[i]]) for i in range(index, len(nums))]

        ans = []
        dfs(0, [])

        return ans
```

[*] Bit Manipulation — Pattern Solution (stored optimal)

Python

```python
# Define the function to generate all subsets using backtracking.
def subsets(nums):
    # This list will store all the subsets
    result = []

    # Recursive helper function to generate subsets.
    def backtrack(start, current):
        # Append a copy of the current subset to the final result.
        result.append(current.copy())
        # Iterate over the possible next elements starting from index 'start'
        for i in range(start, len(nums)):
            # Include nums[i] in the current subset.
            current.append(nums[i])
            # Recurse with the next starting index.
            backtrack(i + 1, current)
            # Backtrack: remove the last element to explore other possibilities.
            current.pop()

    # Start the recursion with an empty path.
    backtrack(0, [])
    return result

# Example usage:
print(subsets([1, 2, 3]))
```


#90 MEDIUM

Subsets II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Backtracking · Bit Manipulation

Lists: Denny Zhang

Problem:

Given an integer array nums that may contain duplicates, return all possible subsets (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

```
Input: nums = [1,2,2]
Output: [[],[1],[1,2],[1,2,2],[2],[2,2]]
```

Example 2:

```
Input: nums = [0]
Output: [[],[0]]
```

Constraints:

- 1 <= nums.length <= 10
- -10 <= nums[i] <= 10

>> Brute Force [Bit Manipulation] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def subsetsWithDup(self, nums):
        # Brute Force O(2^n · n log n) — bitmask with deduplication via set
        result = set()
        nums.sort()
        for mask in range(1 << len(nums)):
            sub = tuple(nums[i] for i in range(len(nums)) if mask & (1 << i))
            result.add(sub)
        return [list(s) for s in result]
```

Python

Python


```python
def subsetsWithDup(nums):
    # Sort the numbers to handle duplicates
    nums.sort()
    result = [] # To store all subsets
    subset = [] # Current subset being built

    def backtrack(start):
        # Append a copy of the current subset to the result
        result.append(list(subset))
        # Iterate over possible elements to include
        for i in range(start, len(nums)):
            # If the current element is a duplicate and not at the start of this loop, skip it
            if i > start and nums[i] == nums[i - 1]:
                continue
            # Include current element into the subset
            subset.append(nums[i])
            # Recurse with next start index
            backtrack(i + 1)
            # Backtrack and remove the last element added
            subset.pop()

    backtrack(0)
    return result

# Example usage:

print(subsetsWithDup([1, 2, 2]))
```

Python

```python
class Solution:
   def subsetsWithDup(self, nums: list[int]) -> list[list[int]]:
      ans = []

      def dfs(s: int, path: list[int]) -> None:
        ans.append(path)
        if s == len(nums):
          return

        for i in range(s, len(nums)):
          if i > s and nums[i] == nums[i - 1]:
            continue
          dfs(i + 1, path + [nums[i]])

      nums.sort()
      dfs(0, [])
      return ans
```

Python


```python
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        def dfs(i: int):
            if i == len(nums):
                ans.append(t[:])
                return
            t.append(nums[i])
            dfs(i + 1)
            x = t.pop()
            while i + 1 < len(nums) and nums[i + 1] == x:
                i += 1
            dfs(i + 1)

        nums.sort()
        ans = []
        t = []
        dfs(0)
        return ans
```

Python

```python
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        def dfs(u, t):
            ans.append(t[:]) # or, t.copy()
            for i in range(u, len(nums)):
                # also good for [1,3,5], second '5' will be skipped here if
                # but second '5' is always covered, because i==u when ans=[1,5] and i=2
                if i != u and nums[i] == nums[i - 1]:
                    continue
                t.append(nums[i])
                dfs(i + 1, t)
                t.pop()

        ans = []
        nums.sort()
        dfs(0, [])
        return ans


# iteration
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        if not nums:
            return [[]]
```



```python
        nums.sort() # Sorting is necessary to handle duplicates.
        result = [[]]
        start_idx = 0 # Start index for the new subsets
        new_subsets_size = 0

        for i in range(len(nums)):
            size = len(result)
            # If the current number is the same as the previous one,
            # we only want to extend the subsets added in the previous iteration.
            if i > 0 and nums[i] == nums[i - 1]:
                start_idx = size - new_subsets_size
            else:
                start_idx = 0

            new_subsets_size = 0 # Reset the size of newly created subsets
            for j in range(start_idx, size):
                current_subset = result[j] + [nums[i]]
                result.append(current_subset)
                new_subsets_size += 1

        return result


###########

class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        n = len(nums)
        ans = []
        for mask in range(1 << n):
            ok = True
            t = []
            for i in range(n):
                if mask >> i & 1:
                    if i and (mask >> (i - 1) & 1) == 0 and nums[i] == nums[i - 1]:
                        ok = False
                        break
                    t.append(nums[i])
            if ok:
                ans.append(t)
        return ans
```

[*] Bit Manipulation — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        def dfs(u, t):
            ans.append(t[:]) # or, t.copy()
            for i in range(u, len(nums)):
                # also good for [1,3,3], second '3' will be skipped here if
                # but second '3' is always covered, because i==u when ans=[1,3] and i=2
                if i != u and nums[i] == nums[i - 1]:
                    continue
                t.append(nums[i])
                dfs(i + 1, t)
                t.pop()

        ans = []
        nums.sort()
        dfs(0, [])
        return ans


# Iterative
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        if not nums:
            return [[]]

        nums.sort() # Sorting is necessary to handle duplicates.
        result = [[]]
        start_idx = 0 # Start index for the new subsets
        new_subsets_size = 0

        for i in range(len(nums)):
            size = len(result)
            # If the current number is the same as the previous one,
            # we only want to extend the subsets added in the previous iteration.
            if i > 0 and nums[i] == nums[i - 1]:
                start_idx = size - new_subsets_size
            else:
                start_idx = 0

            new_subsets_size = 0 # Reset the size of newly created subsets
            for j in range(start_idx, size):
                current_subset = result[j] + [nums[i]]
                result.append(current_subset)
                new_subsets_size += 1

        return result


###########
```



```python
class Solution:
    def subsetsWithDup(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        n = len(nums)
        ans = []
        for mask in range(1 << n):
            ok = True
            t = []
            for i in range(n):
                if mask >> i & 1:
                    if i and (mask >> (i - 1) & 1) == 0 and nums[i] == nums[i - 1]:
                        ok = False
                        break
                    t.append(nums[i])
            if ok:
                ans.append(t)
        return ans
```

#287 MEDIUM

Find the Duplicate Number

LeetCodeX · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers · Binary Search · Bit Manipulation

Lists: Grind 169

Problem:

Given an array of integers nums containing n + 1 integers where each integer is in the range [1, n] inclusive.

There is only one repeated number in nums, return this repeated number.

You must solve the problem without modifying the array nums and using only constant extra space.

Example 1:

```
Input: nums = [1,3,4,2,2]
Output: 2
```

Example 2:

```
Input: nums = [3,1,3,4,2]
Output: 3
```

Example 3:

```
Input: nums = [3,3,3,3,3]
Output: 3
```

Constraints:

- 1 <= n <= 105
- nums.length == n + 1
- 1 <= nums[i] <= n
- All the integers in nums appear only once except for precisely one integer which appears two or more times.

Follow up:

- How can we prove that at least one duplicate number must exist in nums?
- Can you solve the problem in linear runtime complexity?

>> Brute Force [Bit Manipulation] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findDuplicate(self, nums):
        # Brute Force O(n^2) - for each number count its occurrences
        for i in range(len(nums)):
            count = 0
            for j in range(len(nums)):
                if nums[j] == nums[i]:
                    count += 1
            if count > 1:
                return nums[i]
```

Python

Python

```python
# Function to find the duplicate number using Floyd's Tortoise and Hare algorithm.
def findDuplicate(nums):
    # Phase 1: Find the intersection point inside the cycle.
    tortoise = nums[0] # Initialize tortoise pointer.
    hare = nums[0] # Initialize hare pointer.
    while True:
        tortoise = nums[tortoise] # Move tortoise one step.
        hare = nums[nums[hare]] # Move hare two steps.
        if tortoise == hare: # If pointers meet, cycle is detected.
            break

    # Phase 2: Find the entrance to the cycle.
    pointer = nums[0] # Initialize a new pointer from the start.
    while pointer != tortoise:
        pointer = nums[pointer] # Move pointer one step.
        tortoise = nums[tortoise] # Move tortoise one step.
    return pointer # The duplicate number.

# Example usage:
if __name__ == "__main__":
    sample = [1, 3, 4, 2, 2]
    print(findDuplicate(sample)) # Expected output: 2
```

Python


```python
class Solution:
  def findDuplicate(self, nums: list[int]) -> int:
    slow = nums[nums[0]]
    fast = nums[nums[nums[0]]]

    while slow != fast:
      slow = nums[slow]
      fast = nums[nums[fast]]

    slow = nums[0]

    while slow != fast:
      slow = nums[slow]
      fast = nums[fast]

    return slow
```

Python

```python
class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        def f(x: int) -> bool:
            return sum(v <= x for v in nums) > x

        return bisect_left(range(len(nums)), True, key=f)
```

Python

```python
class Solution:
    def findDuplicate(self, nums: List[int]) -> int:
        def f(x: int) -> bool:
            return sum(v <= x for v in nums) > x

        return bisect_left(range(len(nums)), True, key=f)
```

[*] Bit Manipulation — Pattern Solution (stored optimal)

Python


```python
# Function to find the duplicate number using Floyd's Tortoise and Hare algorithm.
def findDuplicate(nums):
    # Phase 1: Find the intersection point inside the cycle.
    tortoise = nums[0] # Initialize tortoise pointer.
    hare = nums[0] # Initialize hare pointer.
    while True:
        tortoise = nums[tortoise] # Move tortoise one step.
        hare = nums[nums[hare]] # Move hare two steps.
        if tortoise == hare: # If pointers meet, cycle is detected.
            break

    # Phase 2: Find the entrance to the cycle.
    pointer = nums[0] # Initialize a new pointer from the start.
    while pointer != tortoise:
        pointer = nums[pointer] # Move pointer one step.
        tortoise = nums[tortoise] # Move tortoise one step.
    return pointer # The duplicate number.

# Example usage:
if __name__ == "__main__":
    sample = [1, 3, 4, 2, 2]
    print(findDuplicate(sample)) # Expected output: 2
```


#1506 MEDIUM

Find Root of N-Ary Tree

LeetCodeX · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · Bit Manipulation · Tree · DFS

Lists: Premium 98

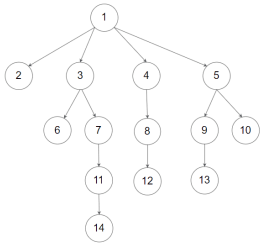
Problem:

You are given all the nodes of an N-ary tree as an array of Node objects, where each node has a unique value.

Return the root of the N-ary tree.

Custom testing:

An N-ary tree can be serialized as represented in its level order traversal where each group of children is separated by the null value (see examples).

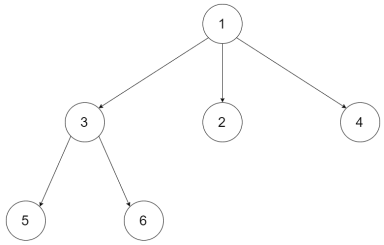


For example, the above tree is serialized as [1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14].

The testing will be done in the following way:

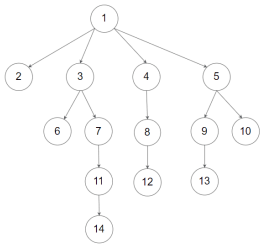
1. The input data should be provided as a serialization of the tree.
2. The driver code will construct the tree from the serialized input data and put each Node object into an array in an arbitrary order.
3. The driver code will pass the array to findRoot, and your function should find and return the root Node object in the array.
4. The driver code will take the returned Node object and serialize it. If the serialized value and the input data are the same, the test passes.

Example 1:




```
Input: tree = [1,null,3,2,4,null,5,6]
Output: [1,null,3,2,4,null,5,6]
Explanation: The tree from the input data is shown above.
The driver code creates the tree and gives findRoot the Node objects in an arbitrary order.
For example, the passed array could be [Node(5),Node(4),Node(3),Node(6),Node(2),Node(1)] or
[Node(2),Node(6),Node(1),Node(3),Node(5),Node(4)].
The findRoot function should return the root Node(1), and the driver code will serialize it
and compare with the input data.
The input data and serialized Node(1) are the same, so the test passes.
```

Example 2:




```
Input: tree =
[1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]
Output:
[1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]
```

Constraints:

- The total number of nodes is between [1, 5 * 104].
- Each node has a unique value.

Follow up:

- Could you solve this problem in constant space complexity with a linear time algorithm?

>> Brute Force [Bit Manipulation] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findRoot(self, tree):
        # Brute Force - check each bit with a loop
        result = 0
        for i in range(32):
            bit = (n >> i) & 1
            result |= (bit << i)
        return result
```

Python

Python


```python
# Python implementation with detailed comments
def findRoot(tree):
    # Initialize variables to store the sum of all node values and children values
    total_sum = 0
    children_sum = 0

    # Iterate over each node in the tree array
    for node in tree:
        total_sum += node.val # Add current node's value to total_sum
        # Iterate over each child of the current node and add their values to children_sum
        for child in node.children:
            children_sum += child.val

    # The root's value is the difference between total_sum and children_sum
    root_val = total_sum - children_sum

    # Find and return the node with the value equal to root_val
    for node in tree:
        if node.val == root_val:
            return node
    return None # This should never happen if the input tree is valid
```

Python

```python
class Solution:
   def findRoot(self, tree: list['Node']) -> 'Node':
     sum = 0

     for node in tree:
       sum ^= node.val
       for child in node.children:
         sum ^= child.val

     for node in tree:
       if node.val == sum:
         return node
```

Python

```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []
"""


class Solution:
    def findRoot(self, tree: List['Node']) -> 'Node':
        x = 0
        for node in tree:
            x ^= node.val
            for child in node.children:
                x ^= child.val
        return next(node for node in tree if node.val == x)
```

Python


```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []
"""


class Solution:
    def findRoot(self, tree: List['Node']) -> 'Node':
        x = 0
        for node in tree:
            x ^= node.val
            for child in node.children:
                x ^= child.val
        return next(node for node in tree if node.val == x)
```


[*] Bit Manipulation — Pattern Solution (matched from community answers)

Python

```python
class Solution:
   def findRoot(self, tree: list['Node']) -> 'Node':
     sum = 0

     for node in tree:
       sum ^= node.val
       for child in node.children:
         sum ^= child.val

     for node in tree:
       if node.val == sum:
         return node
```


Quick Review — Bit Manipulation 10 questions · insights · complexity · solution

#67 Add Binary [Easy]

Key Insights

- Simulate binary addition just like you add numbers digit by digit.
- Process the strings from right to left, handling the carry at each step.
- Use modulo and division operations to extract the current digit and update the carry.
- Append any remaining carry at the end.

Space & Time

Time Complexity: O(max(n, m)), where n and m are the lengths of the two binary strings.

Space Complexity: O(max(n, m)) for storing the resulting binary string.

Solution

The strategy is to iterate over the input strings from the end to the beginning. At each step, calculate the sum of corresponding bits and the existing carry. Use the modulo operation (% 2) to determine the current digit and integer division (// 2) to update the carry. Once all digits have been processed, reverse the result to obtain the final binary string.

#136 Single Number [Easy]

Key Insights

- Using bitwise XOR: XOR-ing a number with itself cancels out to 0.
- XOR is both commutative and associative, so the order of operations does not matter.
- A single traversal (linear time) and constant extra space is sufficient for solving the problem.

Space & Time

Time Complexity: O(n) - We only traverse the array once.

Space Complexity: O(1) - Only a single extra variable is used regardless of input size.

Solution

We can solve this problem by initializing a variable to 0 and then traversing the array while XOR-ing each element with the variable. Due to the properties of XOR, pairs of identical numbers will cancel each other out (x XOR x = 0) and the unique number will be left as the final result. This approach uses constant extra space and performs in linear time.

#190 Reverse Bits [Easy]

Key Insights

- The input is a binary string of fixed length (32 bits).
- Reversing the bits is equivalent to reversing the string.
- The reversed binary string can be converted back into its integer representation.
- Since the number of bits is constant (32), the solution can be achieved in constant time and constant space.
- For performance optimization when this function is called many times, caching may be applied for common intermediate results.

Space & Time

Time Complexity: O(1) - We always process 32 bits regardless of the input.

Space Complexity: O(1) - Only constant extra space is used.

Solution

We solve the problem by reversing the 32-character binary string representing the unsigned integer. Once reversed, the new binary string is converted into its integer value. An alternative solution involves using bitwise operations

where you iterate over each bit, shift the result to the left, and add the current bit. This approach does the reversal one bit at a time. The chosen approach leverages string reversal which is straightforward to implement and understand.

#191 Number of 1 Bits [Easy]

Key Insights

- The problem is essentially counting the number of 1s in the binary representation of a number.
- A common efficient approach uses bit manipulation: repeatedly clear the lowest set bit until n becomes zero.
- The expression n & (n - 1) clears the lowest set bit of n.
- This method only iterates as many times as there are set bits, making it efficient.

Space & Time

Time Complexity: O(k), where k is the number of set bits, worst-case O(log n) (or O(1) considering 32-bit integers).

Space Complexity: O(1).

Solution

To solve the problem, we use the bit manipulation trick n & (n - 1) which removes the lowest set bit from n. The algorithm is as follows:

- Initialize a counter to 0. - While n is not zero, update n by n = n & (n - 1), and increment the counter. - The counter will hold the number of set bits by the end of the loop. This approach leverages efficient bit-level operations and works well even when the function is called frequently.

#268 Missing Number [Easy]

Key Insights

- The range of numbers is continuous from 0 to n, but one number is missing.
- The sum of numbers from 0 to n can be computed using the arithmetic sum formula.
- Subtracting the sum of the given array from the expected sum gives the missing number.
- There is a follow-up challenge to solve the problem in O(n) time and O(1) extra space.

Space & Time

Time Complexity: O(n)

Space Complexity: O(1)

Solution

We compute the expected sum of all numbers from 0 to n using the formula sum = n*(n+1)/2. Then, we compute the actual sum of the numbers in the array. The missing number is obtained by subtracting the actual sum from the expected sum. This approach uses arithmetic operations (O(1) extra space) and loops over the array once (O(n) time).

#338 Counting Bits [Easy]

Key Insights

- We can use dynamic programming to build the solution by utilizing the count of bits for smaller numbers.
- The relationship dp[i] = dp[i >> 1] + (i & 1) holds, where i >> 1 gives the count for i divided by 2 (dropping the least significant bit) and (i & 1) checks if the least significant bit is 1.
- This approach computes the result for each number in constant time, resulting in a linear time solution.

Space & Time

Time Complexity: O(n)

Space Complexity: O(n)

Solution

We use a dynamic programming approach where an array (or list) dp is initialized with size n+1. The key idea is to iterate from 1 to n and for each number, use the formula dp[i] = dp[i >> 1] + (i & 1).

– The expression (i >> 1) effectively divides i by 2, giving the count of 1's for that half number. – The (i & 1) operation checks if i is odd, adding 1 if true. This method ensures each dp[i] is computed in constant time, leading to an overall O(n) time complexity.

#78 Subsets [Medium]

Key Insights

- The problem is equivalent to generating the power set of the given set.
- A recursive backtracking approach can be used to explore both including and excluding each element.
- Iterative (bit manipulation) solutions are also possible since there are 2^n subsets in total.
- The maximum array length is small (up to 10), making exponential time solutions feasible.

Space & Time

Time Complexity: O(2^n * n) – There are 2^n subsets and creating each subset may take up to O(n) time.

Space Complexity: O(2^n * n) – Space for storing all subsets. The recursion stack has a maximum depth of O(n).

Solution

We solve the problem using backtracking, where we decide for each element whether to include it in the current subset or not. We recursively build all subsets starting at an empty list and at each step appending the current subset to our result list. We can also use iterative bit manipulation but here we focus on the backtracking approach. This algorithm leverages recursion and a list to maintain the current state, leading to a clear and concise implementation.

#90 Subsets II [Medium]

Key Insights

- Sort the array first so that duplicate elements are adjacent.
- Use backtracking to generate all possible subsets.
- Skip duplicate elements in the recursion to avoid forming duplicate subsets.
- The use of sorted order and conditionally skipping elements is the trick to handle duplicates.

Space & Time

Time Complexity: O(2^n * n) since each subset creation may take O(n) in the worst case.

Space Complexity: O(2^n * n) for storing the subsets, plus O(n) for recursion stack.

Solution

The solution uses a backtracking technique that builds subsets incrementally. The input array is first sorted to bring duplicates together. During recursion, if the current element is the same as the previous one and it has not been used in the current recursion path, it is skipped to prevent duplicate subsets. This ensures that each subset is unique. The algorithm stores each valid subset by copying the current subset state into the result list.

#287 Find the Duplicate Number [Medium]

Key Insights

- Since there are n+1 elements with values only from 1 to n, by the pigeonhole principle at least one number must be repeated.
- The problem can be approached by mapping the array values to indices, forming a cycle structure.
- Floyd's Tortoise and Hare algorithm (cycle detection) can be applied to find the duplicate number in linear time and constant space.
- There is no need for extra data structures like hash sets or alterations to the original array, satisfying the space and array-modification constraints.

Space & Time

Time Complexity: O(n)

Space Complexity: O(1)

Solution

The solution leverages Floyd's Tortoise and Hare cycle detection algorithm. Think of each array element as a pointer to another index, forming a linked list with a cycle (the duplicate number creates the cycle). The algorithm works in two phases:

– Detect a meeting point within the cycle by having two pointers (tortoise and hare) move at different speeds. – Find the cycle entrance by resetting one pointer to the start and moving both pointers at the same speed. The meeting point reveals the duplicate number.

This approach provides an elegant solution that satisfies both linear runtime and constant space requirements.

#1506 Find Root of N-Ary Tree [Medium]

Key Insights

- Every node except the root appears as a child of another node.
- By summing all node values and then subtracting the sum of all children node values, the remaining value corresponds to the root.
- This subtraction trick allows us to identify the root in a single pass of the data structure without extra memory for tracking child nodes.

Space & Time

Time Complexity: O(n) where n is the number of nodes, since we iterate through all nodes and their children.

Space Complexity: O(1) extra space using arithmetic accumulators.

Solution

The solution calculates two sums: one for all nodes' values and one for all children nodes' values. The difference between these gives the value of the root node since every node's value (except the root's) is added once as a child. Finally, we iterate over the nodes to find the node with the computed root value and return it. This approach meets the requirement of constant space complexity and linear time.

#8 Backtracking — 10 questions · #8 of 21

#17 MEDIUM

Letter Combinations of a Phone Number

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Backtracking

Lists: Grind 169

Problem:

Given a string containing digits from 2–9 inclusive, return all possible letter combinations that the number could represent. Return the answer in any order.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.

Example 1:

```
Input: digits = "23"
Output: ["ad","ae","af","bd","be","bf","cd","ce","cf"]
```

Example 2:

```
Input: digits = "2"
Output: ["a","b","c"]
```

Constraints:

- 1 <= digits.length <= 4

- digits[i] is a digit in the range ['2', '9'].

>> Brute Force [Backtracking] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def letterCombinations(self, digits):
        # Brute Force O(4^n * n) - itertools.product over mapped letters
        if not digits: return []
        from itertools import product
        mapping = {'2':'abc','3':'def','4':'ghi','5':'jkl',
                   '6':'mno','7':'pqrs','8':'tuv','9':'wxyz'}
        return [''.join(c) for c in product(*[mapping[d] for d in digits])]
```

Python

Python

```python
# Python solution for Letter Combinations of a Phone Number

def letterCombinations(digits):
    # Mapping of digits to corresponding letters
    digit_to_letters = {
        "2": "abc",
        "3": "def",
        "4": "ghi",
        "5": "jkl",
        "6": "mno",
        "7": "pqrs",
        "8": "tuv",
        "9": "wxyz"
    }

    # If input is empty, return an empty list
    if not digits:
        return []

    result = []

    # Backtracking function to generate combinations
    def backtrack(index, current_combination):
        # If the current combination length equals the digits length,
        # add the combination to the result list.
```



```python
        if index == len(digits):
            result.append(current_combination)
            return

        # Get the corresponding letters for the current digit
        letters = digit_to_letters[digits[index]]
        # Explore all possible letter routes for the current digit.
        for letter in letters:
            backtrack(index + 1, current_combination + letter)

    # Initiate backtracking starting from index 0 and empty combination.
    backtrack(0, "")
    return result

# Example usage:
#print(letterCombinations("23"))
```

Python

```python
class Solution:
    def letterCombinations(self, digits: str) -> list[str]:
        if not digits:
            return []

        digitToLetters = ['', '', 'abc', 'def', 'ghi',
                          'jkl', 'mno', 'pqrs', 'tuv', 'wxyz']

        ans = []

        def dfs(i: int, path: list[str]) -> None:
            if i == len(digits):
                ans.append(''.join(path))
                return

            for letter in digitToLetters[int(digits[i])]:
                path.append(letter)
                dfs(i + 1, path)
                path.pop()

        dfs(0, [])
        return ans
```

Python

```python
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        if not digits:
            return []

        d = ["abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"]
        ans = [""]
        for i in digits:
            s = d[int(i) - 2]
            ans = [a + b for a in ans for b in s]
        return ans
```

Python


```python
class Solution:
    def letterCombinations(self, digits: str) -> List[str]:
        if not digits:
            return []

        d = ["abc", "def", "ghi", "jkl", "mno", "pqrs", "tuv", "wxyz"]
        ans = [""]
        for i in digits:
            s = d[int(i) - 2]
            ans = [a + b for a in ans for b in s]
        return ans
```

[*] Backtracking — Pattern Solution (matched from community answers)

Python

```python
# Python solution for Letter Combinations of a Phone Number

def letterCombinations(digits):
    # Mapping of digits to corresponding letters
    digit_to_letters = {
        "2": "abc",
        "3": "def",
        "4": "ghi",
        "5": "jkl",
        "6": "mno",
        "7": "pqrs",
        "8": "tuv",
        "9": "wxyz"
    }

    # If input is empty, return an empty list
    if not digits:
        return []

    result = []

    # Backtracking function to generate combinations
    def backtrack(index, current_combination):
        # If the current combination length equals the digits length,
        # add the combination to the result list.
```



```python
        if index == len(digits):
            result.append(current_combination)
            return

        # Get the corresponding letters for the current digit
        letters = digit_to_letters[digits[index]]
        # Explore all possible letter routes for the current digit.
        for letter in letters:
            backtrack(index + 1, current_combination + letter)

    # Initiate backtracking starting from index 0 and empty combination.
    backtrack(0, "")
    return result

# Example usage:
#print(letterCombinations("23"))
```

#22 MEDIUM

Generate Parentheses

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Dynamic Programming · Backtracking

Lists: Grind 169

Problem:

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

Example 1:

```
Input: n = 3
Output: ["((()))","(()())","(())()","()(())","()()()"]
```

Example 2:

```
Input: n = 1
Output: ["()"]
```

Constraints:

- 1 <= n <= 8

>> Brute Force [Backtracking] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def generateParenthesis(self, n):
        # Brute Force O(2^2n) - n) - generate all 2n-length binary strings, filter valid
        def is_valid(comb):
            bal = 0
            for c in comb:
                bal += 1 if c == '(' else -1
                if bal < 0: return False
            return bal == 0
        from itertools import product
        return [''.join(c) for c in product('()', repeat=2*n) if is_valid(c)]
```

Python

Python


```python
def generateParenthesis(n):
    # List to store valid combinations
    result = []

    # Helper function for backtracking
    def backtrack(current, open_count, close_count):
        # If current string reaches the maximum length, it's a valid combination
        if len(current) == 2 * n:
            result.append(current)
            return
        # If we can add an opening parenthesis, do it
        if open_count < n:
            backtrack(current + '(', open_count + 1, close_count)
        # If adding a closing parenthesis maintains the validity, do it
        if close_count < open_count:
            backtrack(current + ')', open_count, close_count + 1)

    # Start the recursion with an empty string and zero counts for both
    backtrack("", 0, 0)
    return result

# Example usage:
print(generateParenthesis(3))
```

Python

```python
class Solution:
  def generateParenthesis(self, n):
    ans = []

    def dfs(l: int, r: int, s: list[str]) -> None:
      if l == 0 and r == 0:
        ans.append(''.join(s))
      if l > 0:
        s.append('(')
        dfs(l - 1, r, s)
        s.pop()
      if l < r:
        s.append(')')
        dfs(l, r - 1, s)
        s.pop()

    dfs(n, n, [])
    return ans
```

Python


```python
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        def dfs(l: int, r: int, t: str):
            if l > n or r > n or l < r:
                return
            if l == n and r == n:
                ans.append(t)
                return
            dfs(l + 1, r, t + "(")
            dfs(l, r + 1, t + ")")

        ans = []
        dfs(0, 0, "")
        return ans
```

Python

```python
class Solution:
    def generateParenthesis(self, n: int) -> List[str]:
        def dfs(l, r, t):
            if l > n or r > n or l < r:
                return
            if l == n and r == n:
                ans.append(t)
                return
            dfs(l + 1, r, t + '(')
            dfs(l, r + 1, t + ')')

        ans = []
        dfs(0, 0, '')
        return ans

###########

class Solution(object):
  def generateParenthesis(self, n):
    """
    :type n: int
    :rtype: List[str]
    """

    def dfs(left, path, res, n):
```



```python
        if len(path) == 2 * n:
          if left == 0:
            res.append("".join(path))
          return

        if left > n:
          path.append("(")
          dfs(left + 1, path, res, n)
          path.pop()
        if left > 0:
          path.append(")")
          dfs(left - 1, path, res, n)
          path.pop()

    res = []
    dfs(0, [], res, n)
    return res
```

[*] Backtracking — Pattern Solution (matched from community answers)

Python

```python
def generateParenthesis(n):
    # List to store valid combinations
    result = []

    # Helper function for backtracking
    def backtrack(current, open_count, close_count):
        # If current string reaches the maximum length, it's a valid combination
        if len(current) == 2 * n:
            result.append(current)
            return
        # If we can add an opening parenthesis, do it
        if open_count < n:
            backtrack(current + '(', open_count + 1, close_count)
        # If adding a closing parenthesis maintains the validity, do it
        if close_count < open_count:
            backtrack(current + ')', open_count, close_count + 1)

    # Start the recursion with an empty string and zero counts for both
    backtrack("", 0, 0)
    return result

# Example usage:
print(generateParenthesis(3))
```

#39 MEDIUM

Combination Sum

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Backtracking

Lists: Grind 169

Problem:

Given an array of distinct integers candidates and a target integer target, return a list of all unique combinations of candidates where the chosen numbers sum to target. You may return the combinations in

any order.

The same number may be chosen from candidates an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

The test cases are generated such that the number of unique combinations that sum up to target is less than 150 combinations for the given input.

Example 1:

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation:
2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.
7 is a candidate, and 7 = 7.
These are the only two combinations.
```

Example 2:

```
Input: candidates = [2,3,5], target = 8
Output: [[2,2,2,2],[2,3,3],[3,5]]
```

Example 3:

```
Input: candidates = [2], target = 1
Output: []
```

Constraints:

- 1 <= candidates.length <= 30
- 2 <= candidates[i] <= 40
- All elements of candidates are distinct.
- 1 <= target <= 40

>> Brute Force [Backtracking] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def combinationSum(self, candidates, target):
        # Brute Force O(n^(target/min)) - recursion without pruning
        result = []
        def backtrack(start, path, remain):
            if remain == 0:
                result.append(list(path)); return
            if remain < 0: return
            for i in range(start, len(candidates)):
                path.append(candidates[i])
                backtrack(i, path, remain - candidates[i])
                path.pop()
        backtrack(0, [], target)
        return result
```

Python

Python


```python
# Function to find all unique combinations that sum to the target
def combinationSum(candidates, target):
    # List to store all valid combinations
    result = []
    # Sort candidates to facilitate pruning of the search
    candidates.sort()

    # Recursive helper function for backtracking
    def backtrack(remaining, start, path):
        # If remaining is 0, path is a valid combination so add a copy of it to result
        if remaining == 0:
            result.append(list(path))
            return
        # Loop through candidates starting from the current index
        for i in range(start, len(candidates)):
            # If candidate is greater than remaining, break out for efficiency
            if candidates[i] > remaining:
                break
            # Include candidate in the current combination path
            path.append(candidates[i])
            # Recursively call backtrack with updated remaining and same index (allowing reuse)
            backtrack(remaining - candidates[i], i, path)
            # Backtrack by removing the candidate added in this iteration
            path.pop()

    # Start backtracking from index 0 with an empty path
    backtrack(target, 0, [])
    return result

# Example usage
print(combinationSum([2,3,6,7], 7))
```

Python

```python
class Solution:
  def combinationSum(self, candidates: list[int],
                     target: int) -> list[list[int]]:

    ans = []

    def dfs(s: int, target: int, path: list[int]) -> None:
      if target < 0:
        return
      if target == 0:
        ans.append(path.clone())
        return

      for i in range(s, len(candidates)):
        path.append(candidates[i])
        dfs(i, target - candidates[i], path)
        path.pop()

    candidates.sort()
    dfs(0, target, [])
    return ans
```

Python

```python
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        def dfs(i: int, s: int):
            if s == 0:
                ans.append(t[:])
                return
            if s < candidates[i]:
                return
            for j in range(i, len(candidates)):
                t.append(candidates[j])
                dfs(j, s - candidates[j])
                t.pop()

        candidates.sort()
        t = []
        ans = []
        dfs(0, target)
        return ans
```

Python

```python
class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        # i: start index for this recursion
        # s: sum
        def dfs(i, s):
            if s == target:
                ans.append(t.copy())
                return
            if s > target:
                return
            for j in range(i, len(candidates)):
                c = candidates[j]
                t.append(c)
                dfs(j, s + c)
                t.pop()

        ans = []
        t = []
        # candidates.sort() # diff from combinationSum-II, no need sorting it
        dfs(0, 0)
        return ans

###########

# dp version
```



```python
class Solution_dp:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        # for each-target (from 1 to target), its dp[i][j]
        # => so 3-D array dp[][][]
        dp = []
        candidates.sort()

        for i in range(1, target+1):
            cur = []
            for j in range(len(candidates)):
                if candidates[j] > i:
                    break
                if candidates[j] == i:
                    one = [candidates[j]]
                    cur.append(one)
                    break
                for a in dp[i - candidates[j] - 1]:
                    if candidates[j] > a[0]:
                        continue
                    deepCopied = a.copy()
                    deepCopied.insert(0, candidates[j])
                    cur.append(deepCopied)
            dp.append(cur)

        return dp[-1]

###########

class Solution(object):
  def combinationSum(self, candidates, target):
    """
    :type candidates: List[int]
    :type target: int
    :rtype: List[List[int]]
    """

    def dfs(candidates, start, target, path, res):
      if target == 0:
        return res.append(path + [])

      for i in range(start, len(candidates)):
        if target - candidates[i] >= 0:
          path.append(candidates[i])
          dfs(candidates, i, target - candidates[i], path, res)
          path.pop()

    res = []
    dfs(candidates, 0, target, [], res)
    return res

#########

class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
```


[*] Backtracking — Pattern Solution (matched from community answers)

Python

```python
# Function to find all unique combinations that sum to the target
def combinationSum(candidates, target):
    # List to store all valid combinations
    result = []
    # Sort candidates to facilitate pruning of the search
    candidates.sort()

    # Recursive helper function for backtracking
    def backtrack(remaining, start, path):
        # If remaining is 0, path is a valid combination so add a copy of it to result
        if remaining == 0:
            result.append(list(path))
            return
        # Loop through candidates starting from the current index
        for i in range(start, len(candidates)):
            # If candidate is greater than remaining, break out for efficiency
            if candidates[i] > remaining:
                break
            # Include candidate in the current combination path
            path.append(candidates[i])
            # Recursively call backtrack with updated remaining and same index (allowing reuse)
            backtrack(remaining - candidates[i], i, path)
            # Backtrack by removing the candidate added in this iteration
            path.pop()

    # Start backtracking from index 0 with an empty path
    backtrack(target, 0, [])
    return result

# Example usage
print(combinationSum([2,3,6,7], 7))
```

#46 MEDIUM

Permutations

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Backtracking

Lists: Grind 169

Problem:

Given an array nums of distinct integers, return all the possible permutations. You can return the answer in any order.

Example 1:

```
Input: nums = [1,2,3]
Output: [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]
```

Example 2:

```
Input: nums = [0,1]
Output: [[0,1],[1,0]]
```


Example 3:

```
Input: nums = [1]
Output: [[1]]
```

Constraints:

- 1 <= nums.length <= 6
- -10 <= nums[i] <= 10
- All the integers of nums are unique.

>> Brute Force [Backtracking] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def permute(self, nums):
        # Brute Force Step: O(n! * n) - itertools.permutations (or full recursion)
        from itertools import permutations as perms
        return [list(p) for p in perms(nums)]
```

Python

Python

```python
def permute(nums):
    result = []

    def backtrack(current, remaining):
        # If no remaining elements, current permutation is complete.
        if not remaining:
            result.append(current.copy())
            return
        # Explore each candidate number.
        for i in range(len(remaining)):
            # Add the chosen number to the current permutation.
            current.append(remaining[i])
            # Recursively generate permutations with the remaining numbers.
            backtrack(current, remaining[:i] + remaining[i+1:])
            # Backtrack by removing the chosen number.
            current.pop()

    backtrack([], nums)
    return result

# Example usage:
print(permute([1,2,3]))
```

Python


```python
class Solution:
    def permute(self, nums: list[int]) -> list[list[int]]:
        ans = []
        used = [False] * len(nums)

        def dfs(path: list[int]) -> None:
            if len(path) == len(nums):
                ans.append(path.copy())
                return

            for i, num in enumerate(nums):
                if used[i]:
                    continue
                used[i] = True
                path.append(num)
                dfs(path)
                path.pop()
                used[i] = False

        dfs([])
        return ans
```

Python

```python
class Solution:
    def permute(self, nums: List[int]) -> List[List[int]]:
        def dfs(i: int):
            if i >= n:
                ans.append(t[:])
                return
            for j, x in enumerate(nums):
                if not vis[j]:
                    vis[j] = True
                    t[i] = x
                    dfs(i + 1)
                    vis[j] = False

        n = len(nums)
        vis = [False] * n
        t = [0] * n
        ans = []
        dfs(0)
        return ans
```

Python


```python
'''
remove an element by its value from a set

>>> my_set = {1, 2, 3, 4, 5}
>>> my_set.remove(3)
>>> print(my_set)
{1, 2, 4, 5}

-------

cannot remove an element by index from a set in Python3
need to convert the set to a list

>>> my_set = {1, 2, 3, 4, 5}
>>> my_list = list(my_set)
>>> del my_list[2] # remove the element at index 2
>>> my_set = set(my_list)
>>> print(my_set)
{1, 2, 4, 5}
'''
class Solution: # iterative
    def permute(self, nums: List[int]) -> List[List[int]]:
        res = [[]]

        if nums is None or len(nums) == 0:
            return ans

        for num in nums:
            new_res = []
            for perm in res:
                for i in range(len(perm) + 1):
                    new_perm = perm[:i] + [num] + perm[i:]
                    new_res.append(new_perm)

            res = new_res

        return res

#############

class Solution: # iterative, single_perm.insert(index, num)
    def permute(self, nums: List[int]) -> List[List[int]]:
        ans = [[]]

        if nums is None or len(nums) == 0:
            return ans

        for num in nums:
            tmp_list = []
```

```
            for single_perm in ans:
                for index in range(len(single_perm) + 1):
                    single_perm.insert(index, num)
                    tmp_list.append(single_perm.copy())
                    single_perm.pop(index)

            ans = tmp_list

        return ans

#############

'''
In Python 3, both the discard() and remove() methods of a set object are used to remove an element from the
set, but there is one key difference:

* discard() removes the specified element from the set if it is present, but does nothing if the element is
not present.
* remove() removes the specified element from the set if it is present, but raises a KeyError exception if
the element is not present.


>>> a
{33, 66, 11, 44, 22, 55}
>>> a.discard(22)
>>> a
{33, 66, 11, 44, 55}
```

```
>>> a.discard(200)
>>>
>>> a.remove(200)
Traceback (most recent call last):
```

[*] Backtracking — Pattern Solution (matched from community answers)

Python


```
def permute(nums):
    result = []

    def backtrack(current, remaining):
        # If no remaining elements, current permutation is complete.
        if not remaining:
            result.append(current.copy())
            return
        # Explore each candidate number.
        for i in range(len(remaining)):
            # Add the chosen number to the current permutation.
            current.append(remaining[i])
            # Recursively generate permutations with the remaining numbers.
            backtrack(current, remaining[:i] + remaining[i+1:])
            # Backtrack by removing the chosen number.
            current.pop()

    backtrack([], nums)
    return result

# Example usage:
print(permute([1,2,3]))
```

#79 MEDIUM

Word Search

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · String · Backtracking · Matrix

Lists: Grind 169

Problem:

Given an m x n grid of characters board and a string word, return true if word exists in the grid.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:

A	B	C	E
S	F	C	S
A	D	E	E

```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"
Output: true
```

Example 2:

A	B	C	E
S	F	C	S
A	D	E	E

```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"
Output: true
```

Example 3:

A	B	C	E
S	F	C	S
A	D	E	E

```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"
Output: false
```

Constraints:

- m == board.length
- n = board[i].length
- 1 <= m, n <= 6
- 1 <= word.length <= 15
- board and word consists of only lowercase and uppercase English letters.

Follow up: Could you use search pruning to make your solution faster with a larger board?

>> Brute Force [Backtracking] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def exist(self, board, word):
        # Brute Force Dfs, m*n1 -> DFS from every cell, backtrack on visited
        m, n = len(board), len(board[0])
        def dfs(r, c, i, visited):
            if i == len(word): return True
            if r<0 or r>=m or c<0 or c>=n: return False
            if (r,c) in visited or board[r][c] != word[i]: return False
            visited.add((r,c))
            found = (dfs(r+1,c,i+1,visited) or dfs(r-1,c,i+1,visited) or
                     dfs(r,c+1,i+1,visited) or dfs(r,c-1,i+1,visited))
            visited.remove((r,c))
            return found
        for r in range(m):
            for c in range(n):
                if dfs(r, c, 0, set()): return True
        return False
```


Python

Python

```
# Define the function to solve the Word Search problem.
def exist(board, word):
    # Get dimensions of the board.
    rows, cols = len(board), len(board[0])

    # Helper function for DFS search.
    def dfs(r, c, index):
        # If the entire word is found, return True.
        if index == len(word):
            return True

        # Check boundaries and if current cell already used or not matching.
        if r < 0 or c < 0 or r >= rows or c >= cols or board[r][c] != word[index]:
            return False

        # Temporarily mark the cell as visited.
        temp = board[r][c]
        board[r][c] = "#"

        # Explore all four directions.
        found = (dfs(r + 1, c, index + 1) or
                 dfs(r - 1, c, index + 1) or
                 dfs(r, c + 1, index + 1) or
                 dfs(r, c - 1, index + 1))


        # Backtrack: restore the original cell value.
        board[r][c] = temp
        return found

    # Initiate DFS from each cell in the grid.
    for i in range(rows):
        for j in range(cols):
            if dfs(i, j, 0):
                return True
    return False

# Example test case
if __name__ == "__main__":
    board = [
        ["A", "B", "C", "E"],
        ["S", "F", "C", "S"],
        ["A", "D", "E", "E"]
    ]
    word = "ABCCED"
    print(exist(board, word)) # Expected output: True
```

Python


```
class Solution:
    def exist(self, board: list[list[str]], word: str) -> bool:
        m = len(board)
        n = len(board[0])

        def dfs(i: int, j: int, s: int) -> bool:
            if i < 0 or i == m or j < 0 or j == n:
                return False
            if board[i][j] != word[s] or board[i][j] == '*':
                return False
            if s == len(word) - 1:
                return True

            cache = board[i][j]
            board[i][j] = '*'
            isExist = (dfs(i + 1, j, s + 1) or
                       dfs(i - 1, j, s + 1) or
                       dfs(i, j + 1, s + 1) or
                       dfs(i, j - 1, s + 1))
            board[i][j] = cache

            return isExist

        return any(dfs(i, j, 0)
                   for i in range(m)

                   for j in range(n))
```

Python

```
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        def dfs(i: int, j: int, k: int) -> bool:
            if k == len(word) - 1:
                return board[i][j] == word[k]
            if board[i][j] != word[k]:
                return False
            c = board[i][j]
            board[i][j] = "@"
            for a, b in pairwise((-1, 0, 1, 0, -1)):
                x, y = i + a, j + b
                ok = 0 <= x < m and 0 <= y < n and board[x][y] != "@"
                if ok and dfs(x, y, k + 1):
                    return True
            board[i][j] = c
            return False

        m, n = len(board), len(board[0])
        return any(dfs(i, j, 0) for i in range(m) for j in range(n))
```

Python

```python
class Solution:
    def exist(self, board: List[List[str]], word: str) -> bool:
        def dfs(i, j, cur):# cur: current char count
            if cur == len(word):
                return True
            if (
                i < 0
                or i >= m
                or j < 0
                or j >= n
                or board[i][j] == '0'
                or word[cur] != board[i][j]
            ):
                return False
            t = board[i][j]
            board[i][j] = '0' # mark as visited
            for a, b in [(0, 1), (0, -1), (-1, 0), (1, 0)]:
                x, y = i + a, j + b
                if dfs(x, y, cur + 1):
                    return True
            board[i][j] = t
            return False

        m, n = len(board), len(board[0])
        return any(dfs(i, j, 0) for i in range(m) for j in range(n))
```

[*] Backtracking — Pattern Solution (stored optimal)

Python

```python
# Define the function to solve the Word Search problem.
def exist(board, word):
    # Get dimensions of the board.
    rows, cols = len(board), len(board[0])

    # Helper function for DFS search.
    def dfs(r, c, index):
        # If the entire word is found, return True.
        if index == len(word):
            return True

        # Check boundaries and if current cell already used or not matching.
        if r < 0 or c < 0 or r >= rows or c >= cols or board[r][c] != word[index]:
            return False

        # Temporarily mark the cell as visited.
        temp = board[r][c]
        board[r][c] = "#"

        # Explore all four directions.
        found = (dfs(r + 1, c, index + 1) or
                 dfs(r - 1, c, index + 1) or
                 dfs(r, c + 1, index + 1) or
                 dfs(r, c - 1, index + 1))
```



```python
        # Backtrack: restore the original cell value.
        board[r][c] = temp
        return found

    # Initiate DFS from each cell in the grid.
    for i in range(rows):
        for j in range(cols):
            if dfs(i, j, 0):
                return True
    return False

# Example test case
if __name__ == "__main__":
    board = [
        ["A", "B", "C", "E"],
        ["S", "F", "C", "S"],
        ["A", "D", "E", "E"]
    ]
    word = "ABCCED"
    print(exist(board, word)) # Expected output: True
```

#113 MEDIUM

Path Sum II

LeetCode · SimplyLeet · WalkCC · LeetCode · Editorial

Backtracking · Tree · DFS

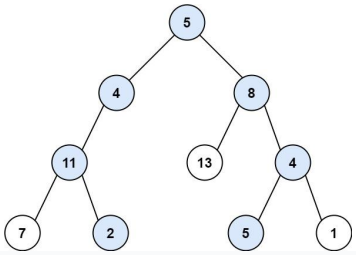
Lists: Grind 169

Problem:

Given the root of a binary tree and an integer targetSum, return all root-to-leaf paths where the sum of the node values in the path equals targetSum. Each path should be returned as a list of the node values, not node references.

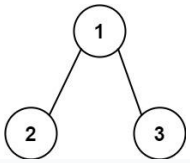
A root-to-leaf path is a path starting from the root and ending at any leaf node. A leaf is a node with no children.

Example 1:




```
Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
Output: [[5,4,11,2],[5,8,4,5]]
Explanation: There are two paths whose sum equals targetSum:
5 + 4 + 11 + 2 = 22
5 + 8 + 4 + 5 = 22
```

Example 2:




```
Input: root = [1,2,3], targetSum = 5
Output: []
```

Example 3:

```
Input: root = [1,2], targetSum = 0
Output: []
```

Constraints:

- The number of nodes in the tree is in the range [0, 5000].
- -1000 <= Node.val <= 1000
- -1000 <= targetSum <= 1000

>> Brute Force [Backtracking] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def pathSum(self, root, targetSum):
        # Brute Force - exhaustive backtracking without pruning
        results = []
        def backtrack(start, path):
            results.append(list(path))
            for i in range(start, len(root)):
                path.append(root[i])
                backtrack(i + 1, path)
                path.pop()
        backtrack(0, [])
        return results
```

Python

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

def pathSum(root, targetSum):
    result = [] # This will store all valid paths

    def dfs(node, current_path, remaining_sum):
        if not node:
            return # Base case: reached the end of a path

        # Add the current node's value to the path
        current_path.append(node.val)

        # Check if it's a leaf and if the path sum equals targetSum
        if not node.left and not node.right and node.val == remaining_sum:
            result.append(list(current_path)) # Add a copy of the current path
        else:
            # Continue the search in the left and right subtrees
            dfs(node.left, current_path, remaining_sum - node.val)
            dfs(node.right, current_path, remaining_sum - node.val)

        # Backtrack: remove the last element before returning to the caller
        current_path.pop()

    dfs(root, [], targetSum)
    return result
```

Python


```python
class Solution:
    def pathSum(self, root: TreeNode, summ: int) -> list[list[int]]:
        ans = []

        def dfs(root: TreeNode, summ: int, path: list[int]) -> None:
          if not root:
            return
          if root.val == summ and not root.left and not root.right:
            ans.append(path + [root.val])
            return

          dfs(root.left, summ - root.val, path + [root.val])
          dfs(root.right, summ - root.val, path + [root.val])

        dfs(root, summ, [])
        return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> List[List[int]]:
        def dfs(root, s):
            if root is None:
                return
            s += root.val
            t.append(root.val)
            if root.left is None and root.right is None and s == targetSum:
                ans.append(t[:])
            dfs(root.left, s)
            dfs(root.right, s)
            t.pop()

        ans = []
        t = []
        dfs(root, 0)
        return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> List[List[int]]:
        def dfs(root, s):
            if root is None:
                return
            s += root.val
            t.append(root.val)
            if root.left is None and root.right is None and s == targetSum:
                ans.append(t[:])
            dfs(root.left, s)
            dfs(root.right, s)
            t.pop()

        ans = []
        t = []
        dfs(root, 0)
        return ans
```

[*] Backtracking — Pattern Solution (matched from community answers)

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

def pathSum(root, targetSum):
    result = [] # This will store all valid paths

    def dfs(node, current_path, remaining_sum):
        if not node:
            return # Base case: reached the end of a path

        # Add the current node's value to the path
        current_path.append(node.val)

        # Check if it's a leaf and if the path sum equals targetSum
        if not node.left and not node.right and node.val == remaining_sum:
            result.append(list(current_path)) # Add a copy of the current path
        else:
            # Continue the search in the left and right subtrees
            dfs(node.left, current_path, remaining_sum - node.val)
            dfs(node.right, current_path, remaining_sum - node.val)

        # Backtrack: remove the last element before returning to the caller
        current_path.pop()

    dfs(root, [], targetSum)
    return result
```

#37 HARD

Sudoku Solver

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Backtracking · Matrix

Lists: Grind 169

Problem:

Write a program to solve a Sudoku puzzle by filling the empty cells.

A sudoku solution must satisfy all of the following rules:

1. Each of the digits 1–9 must occur exactly once in each row.
2. Each of the digits 1–9 must occur exactly once in each column.
3. Each of the digits 1–9 must occur exactly once in each of the 9 3x3 sub-boxes of the grid.

The '.' character indicates empty cells.

Example 1:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

```
Input: board = [["5","3",".",".","7",".",".",".",".],["6",".",".","1","9","5",".",".",".],[
".","9","8",".",".",".",".","6",".],["8",".",".",".","6",".",".",".","3"],["4",".",".","8",".
",".","3",".",".","1"],["7",".",".",".","2",".",".",".","6"],[".","6",".",".",".",".","2","8",
".],[".",".",".","4","1","9",".",".","5"],[".",".",".",".","8",".",".","7","9"]]
Output: [["5","3","4","6","7","8","9","1","2"],["6","7","2","1","9","5","3","4","8"],["1","9"
,"8","3","4","2","5","6","7"],["8","5","9","7","6","1","4","2","3"],["4","2","6","8","5","3",
"7","9","1"],["7","1","3","9","2","4","8","5","6"],["9","6","1","5","3","7","2","8","4"],["2"
,"8","7","4","1","9","6","3","5"],["3","4","5","2","8","6","1","7","9"]]
Explanation: The input board is shown above and the only valid solution is shown below:
```

Constraints:

- board.length == 9
- board[i].length == 9

- board[i][j] is a digit or '.'.
- It is guaranteed that the input board has only one solution.

>> Brute Force [Backtracking] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def solveSudoku(self, board):
        # Brute Force - exhaustive backtracking without pruning
        results = []
        def backtrack(start, path):
            results.append(list(path))
            for i in range(start, len(board)):
                path.append(board[i])
                backtrack(i + 1, path)
                path.pop()
        backtrack(0, [])
        return results
```

Python

Python

```python
def solveSudoku(board):
    # Check if placing the digit 'char' at board[row][col] is valid
    def is_valid(row, col, char):
        for i in range(9):
            # Check the row
            if board[row][i] == char:
                return False
            # Check the column
            if board[i][col] == char:
                return False
            # Check the 3x3 sub-box
            sub_row = 3 * (row // 3) + i // 3
            sub_col = 3 * (col // 3) + i % 3
            if board[sub_row][sub_col] == char:
                return False
        return True

    # Backtracking function to fill the board
    def backtrack():
        for row in range(9):
            for col in range(9):
                if board[row][col] == '.':
                    for char in map(str, range(1, 10)):
                        if is_valid(row, col, char):
                            board[row][col] = char # Place the digit
```



```
                        if backtrack():
                            return True # Recurse; if solution found, bubble up True
                        board[row][col] = '.' # Backtrack if not valid further down
                return False # No valid digit found; trigger backtracking
        return True # Board completely filled

    backtrack()

# Example usage:
board = [["5","3",".",".","7",".",".",".",".],
         ["6",".",".","1","9","5",".",".",".],
         [".","9","8",".",".",".",".","6",".],
         ["8",".",".",".","6",".",".",".","3],
         ["4",".",".","8",".","3",".",".","1],
         ["7",".",".",".","2",".",".",".","6],
         [".","6",".",".",".",".","2","8",".],
         [".",".",".","4","1","9",".",".","5],
         [".",".",".",".","8",".",".","7","9]]
solveSudoku(board)
print(board)
```

Python

```python
class Solution:
    def solveSudoku(self, board: list[list[str]]) -> None:
        def isValid(row: int, col: int, c: str) -> bool:
            for i in range(9):
                if board[i][col] == c or
                    board[row][i] == c or
                    board[3 * (row // 3) + i // 3][3 * (col // 3) + i % 3] == c):
                return False
            return True

        def solve(s: int) -> bool:
            if s == 81:
                return True

            i = s // 9
            j = s % 9

            if board[i][j] != '.':
                return solve(s + 1)

            for c in string.digits[1:]:
                if isValid(i, j, c):
                    board[i][j] = c
                    if solve(s + 1):
                        return True

            board[i][j] = '.'

            return False

        solve(0)
```

Python


```python
class Solution:
    def solveSudoku(self, board: List[List[str]]) -> None:
        def dfs(k):
            nonlocal ok
            if k == len(t):
                ok = True
                return
            i, j = t[k]
            for v in range(9):
                if row[i][v] == col[j][v] == block[i // 3][j // 3][v] == False:
                    row[i][v] = col[j][v] = block[i // 3][j // 3][v] = True
                    board[i][j] = str(v + 1)
                    dfs(k + 1)
                    row[i][v] = col[j][v] = block[i // 3][j // 3][v] = False
            if ok:
                return

        row = [[False] * 9 for _ in range(9)]
        col = [[False] * 9 for _ in range(9)]
        block = [[[False] * 9 for _ in range(3)] for _ in range(3)]
        t = []
        ok = False
        for i in range(9):
            for j in range(9):
                if board[i][j] == '.':

                    t.append((i, j))
                else:
                    v = int(board[i][j]) - 1
                    row[i][v] = col[j][v] = block[i // 3][j // 3][v] = True
        dfs(0)
```

[*] Backtracking — Pattern Solution (matched from community answers)

Python


```python
def solveSudoku(board):
    # Check if placing the digit 'char' at board[row][col] is valid
    def is_valid(row, col, char):
        for i in range(9):
            # Check the row
            if board[row][i] == char:
                return False
            # Check the column
            if board[i][col] == char:
                return False
            # Check the 3x3 sub-box
            sub_row = 3 * (row // 3) + i // 3
            sub_col = 3 * (col // 3) + i % 3
            if board[sub_row][sub_col] == char:
                return False
        return True

    # Backtracking function to fill the board
    def backtrack():
        for row in range(9):
            for col in range(9):
                if board[row][col] == '.':
                    for char in map(str, range(1, 10)):
                        if is_valid(row, col, char):
                            board[row][col] = char # Place the digit

                        if backtrack():
                            return True # Recurse; if solution found, bubble up True
                        board[row][col] = '.' # Backtrack if not valid further down
                return False # No valid digit found; trigger backtracking
        return True # Board completely filled

    backtrack()

# Example usage:
board = [["5","3",".",".","7",".",".",".",".],
         ["6",".",".","1","9","5",".",".",".],
         [".","9","8",".",".",".",".","6",".],
         ["8",".",".",".","6",".",".",".","3],
         ["4",".",".","8",".","3",".",".","1],
         ["7",".",".",".","2",".",".",".","6],
         [".","6",".",".",".",".","2","8",".],
         [".",".",".","4","1","9",".",".","5],
         [".",".",".",".","8",".",".","7","9]]
solveSudoku(board)
print(board)
```

#51 HARD

N-Queens

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Backtracking

Lists: Grind 169

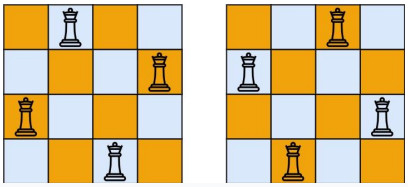
Problem:

The n-queens puzzle is the problem of placing n queens on an n x n chessboard such that no two queens attack each other.

Given an integer n, return all distinct solutions to the n-queens puzzle. You may return the answer in any order.

Each solution contains a distinct board configuration of the n-queens' placement, where 'Q' and '.' both indicate a queen and an empty space, respectively.

Example 1:



```
Input: n = 4
Output: [[".Q..","...Q","Q...","..Q."],["..Q.","Q...","...Q",".Q.."]]
Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above
```

Example 2:

```
Input: n = 1
Output: [["Q"]]
```

Constraints:

- 1 <= n <= 9

>> Brute Force [Backtracking] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def solveNQueens(self, n):
        # Brute Force O(n^n · n) - place a queen in every column each row, filter valid
        from itertools import product
        def is_valid(placement):
            for i in range(len(placement)):
                for j in range(i+1, len(placement)):
                    if placement[i]==placement[j]: return False # same col
                    if abs(placement[i]-placement[j])==abs(i-j): return False # diagonal
            return True
        result = []
        for cols in product(range(n), repeat=n):
            if is_valid(cols):
                board = []
                for c in cols:
                    row = '.'*c + 'Q' + '.'*(n-c-1)
                    board.append(row)
                result.append(board)
        return result
```

Python

Python

```python
def solveNQueens(n):
    result = []

    # Helper function to backtrack
    def backtrack(row, columns, diag1, diag2, board):
        # Base case: if all rows are filled, add a copy of board to result
        if row == n:
            result.append(list(board))
            return

        # Try to place a queen in each column in the current row
        for col in range(n):
            # Check if the column or diagonals are not under attack
            if col in columns or (row - col) in diag1 or (row + col) in diag2:
                continue

            # Place the queen
            board[row] = '.' * col + 'Q' + '.' * (n - col - 1)
            # Mark the column and diagonals as occupied
            columns.add(col)
            diag1.add(row - col)
            diag2.add(row + col)

            # Move to the next row
            backtrack(row + 1, columns, diag1, diag2, board)
```



```python
            # Backtrack: remove the queen and unmark the sets
            columns.remove(col)
            diag1.remove(row - col)
            diag2.remove(row + col)

    # Initialize board with empty rows
    board = ["." * n for _ in range(n)]
    backtrack(0, set(), set(), set(), board)
    return result

# Example usage:
print(solveNQueens(4))
```

Python

```python
class Solution:
    def solveNQueens(self, n: int) -> list[list[str]]:
        ans = []
        cols = [False] * n
        diag1 = [False] * (2 * n - 1)
        diag2 = [False] * (2 * n - 1)

        def dfs(i: int, board: list[int]) -> None:
          if i == n:
            ans.append(board)
            return

          for j in range(n):
            if cols[j] or diag1[i + j] or diag2[j - i + n - 1]:
              continue
            cols[j] = diag1[i + j] = diag2[j - i + n - 1] = True
            dfs(i + 1, board + ['.' * j + 'Q' + '.' * (n - j - 1)])
            cols[j] = diag1[i + j] = diag2[j - i + n - 1] = False

        dfs(0, [])
        return ans
```

Python


```python
class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        def dfs(i: int):
            if i == n:
                ans.append(["".join(row) for row in g])
                return
            for j in range(n):
                if col[j] + dg[i + j] + udg[n - i + j] == 0:
                    g[i][j] = "Q"
                    col[j] = dg[i + j] = udg[n - i + j] = 1
                    dfs(i + 1)
                    col[j] = dg[i + j] = udg[n - i + j] = 0
                    g[i][j] = "."

        ans = []
        g = [["."] * n for _ in range(n)]
        col = [0] * n
        dg = [0] * (n << 1)
        udg = [0] * (n << 1)
        dfs(0)
        return ans
```

Python

```python
class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        def dfs(i: int):
            if i == n:
                ans.append(["".join(row) for row in g])
                return
            for j in range(n):
                if col[j] + dg[i + j] + udg[n - i + j] == 0:
                    g[i][j] = "Q"
                    col[j] = dg[i + j] = udg[n - i + j] = 1
                    dfs(i + 1)
                    col[j] = dg[i + j] = udg[n - i + j] = 0
                    g[i][j] = "."

        ans = []
        g = [["."] * n for _ in range(n)]
        col = [0] * n
        dg = [0] * (n << 1)
        udg = [0] * (n << 1)
        dfs(0)
        return ans
```

[*] Backtracking — Pattern Solution (matched from community answers)

Python


```python
def solveNQueens(n):
    result = []

    # Helper function to backtrack
    def backtrack(row, columns, diag1, diag2, board):
        # Base case: if all rows are filled, add a copy of board to result
        if row == n:
            result.append(list(board))
            return

        # Try to place a queen in each column in the current row
        for col in range(n):
            # Check if the column or diagonals are not under attack
            if col in columns or (row - col) in diag1 or (row + col) in diag2:
                continue

            # Place the queen
            board[row] = '.' * col + 'Q' + '.' * (n - col - 1)
            # Mark the column and diagonals as occupied
            columns.add(col)
            diag1.add(row - col)
            diag2.add(row + col)

            # Move to the next row
            backtrack(row + 1, columns, diag1, diag2, board)
```

```python
            # Backtrack: remove the queen and unmark the sets
            columns.remove(col)
            diag1.remove(row - col)
            diag2.remove(row + col)

    # Initialize board with empty rows
    board = ["." * n for _ in range(n)]
    backtrack(0, set(), set(), set(), board)
    return result

# Example usage:
print(solveNQueens(4))
```

#126 HARD

Word Ladder II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Backtracking · BFS

Lists: Denny Zhang

Problem:

A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

- Every adjacent pair of words differs by a single letter.
- Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
- sk == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return all the shortest transformation sequences from beginWord to endWord, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words [beginWord, s1, s2, ..., sk].

Example 1:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]
Explanation: There are 2 shortest transformation sequences:
"hit" -> "hot" -> "dot" -> "dog" -> "cog"
"hit" -> "hot" -> "lot" -> "log" -> "cog"
```

Example 2:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: []
Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation
sequence.
```

Constraints:

- 1 <= beginWord.length <= 5
- endWord.length == beginWord.length
- 1 <= wordList.length <= 500
- wordList[i].length == beginWord.length
- beginWord, endWord, and wordList[i] consist of lowercase English letters.
- beginWord != endWord
- All the words in wordList are unique.
- The sum of all shortest transformation sequences does not exceed 105.

>> Brute Force [Backtracking] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findLadders(self, beginWord, endWord, wordList):
        # Brute Force - exhaustive backtracking without pruning
        results = []
        def backtrack(start, path):
            results.append(list(path))
            for i in range(start, len(beginWord)):
                path.append(beginWord[i])
                backtrack(i + 1, path)
                path.pop()
        backtrack(0, [])
        return results
```

Python

Python


```python
# Python solution for Word Ladder II

from collections import defaultdict, deque

def findLadders(beginWord, endWord, wordList):
    # Convert list to set for fast lookup
    wordSet = set(wordList)
    if endWord not in wordSet:
        return []

    # Dictionary to hold parent pointers for backtracking: child -> list of parents
    parents = defaultdict(list)
    # Set for words visited at current BFS level
    level = {beginWord}
    # Flag to stop BFS when endWord is reached
    found = False

    while level and not found:
        next_level = set()
        # Remove words of current level from wordSet to prevent cycles
        wordSet -= level
        for word in level:
            # Try all possible one-letter transformations
            for i in range(len(word)):
                for char in 'abcdefghijklmnopqrstuvwxyz':
```

```python
                    candidate = word[:i] + char + word[i+1:]
                    if candidate in wordSet:
                        # If candidate is the endWord, note that we have found a path
                        if candidate == endWord:
                            found = True
                        # Record the current word as a parent of candidate word
                        parents[candidate].append(word)
                        next_level.add(candidate)
        # Move to the next level of the BFS
        level = next_level

    # The result list to hold all valid transformation sequences
    res = []

    # Helper function to backtrack from endWord to beginWord using the parents dictionary
    def backtrack(word, path):
        if word == beginWord:
            # Append reversed path to have the sequence from beginWord to endWord
            res.append(path[::-1])
            return
        # Recurse over all parents of the current word
        for parent in parents[word]:
            backtrack(parent, path + [parent])

    if found:
```

```python
        backtrack(endWord, [endWord])
    return res

# Example usage:
# print(findLadders("hit", "cog", ["hot","dot","dog","lot","log","cog"]))
```

Python

```python
class Solution:
    def findLadders(self, beginWord: str, endWord: str, wordList: list[str]) -> list[list[str]]:
        wordSet = set(wordList)
        if endWord not in wordList:
            return []

        # {"hit": ["hot"], "hot": ["dot", "lot"], ...}
        graph: dict[str, list[str]] = collections.defaultdict(list)

        # Build the graph from the beginWord to the endWord.
        if not self._bfs(beginWord, endWord, wordSet, graph):
            return []

        ans = []
        self._dfs(graph, beginWord, endWord, [beginWord], ans)
        return ans

    def _bfs(
        self,
        beginWord: str,
        endWord: str,
        wordSet: set[str],
        graph: dict[str, list[str]],
    ) -> bool:
        currentLevelWords = {beginWord}

        while currentLevelWords:
            for word in currentLevelWords:
                wordSet.discard(word)
            nextLevelWords = set()
            reachEndWord = False
            for parent in currentLevelWords:
                for child in self._getChildren(parent, wordSet):
                    if child in wordSet:
                        nextLevelWords.add(child)
                        graph[parent].append(child)
                    if child == endWord:
                        reachEndWord = True
            if reachEndWord:
                return True
            currentLevelWords = nextLevelWords

        return False

    def _getChildren(self, parent: str, wordSet: set[str]) -> list[str]:
        children = []
        s = list(parent)

        for i, cache in enumerate(s):
            for c in string.ascii_lowercase:
```



```python
                if c == cache:
                    continue
                s[i] = c
                child = ''.join(s)
                if child in wordSet:
                    children.append(child)
            s[i] = cache

        return children

    def _dfs(
        self,
        graph: dict[str, list[str]],
        word: str,
        endWord: str,
        path: list[str],
        ans: list[list[str]],
    ) -> None:
        if word == endWord:
            ans.append(path.copy())
            return

        for child in graph.get(word, []):
            path.append(child)
            self._dfs(graph, child, endWord, path, ans)

            path.pop()
```

Python

```
...
remove() same as discard()

>>> a = set({11,22,33})
>>> a.remove(22)
>>> a
{33, 11}

>>> a = set({11,22,33})
>>> a.discard(22)
>>> a
{33, 11}


>>> a=set({1,2,3})
>>> a.discard(555)
>>> a.remove(555)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 555
...

class Solution:
    def findLadders(
        self, beginWord: str, endWord: str, wordList: List[str]
```



```
    ) -> List[List[str]]:
        # endWord to beginWord
        # better than begin to end, too many paths not in final result
        def dfs(path, cur):
            if cur == beginWord:
                ans.append(path[::-1])
                return
            for precursor in prev[cur]:
                path.append(precursor)
                dfs(path, precursor)
                path.pop()

        ans = []
        words = set(wordList)
        if endWord not in words:
            return ans
        # no exception if beginWord not in set
        words.discard(beginWord)
        dist = {beginWord: 0}
        prev = defaultdict(set)
        q = deque([beginWord])
        found = False
        step = 0
        while q and not found:
            step += 1

            for _ in range(len(q), 0, -1):
                p = q.popleft()
                s = list(p)
                for i in range(len(s)):
                    ch = s[i]
                    for j in range(26):
                        s[i] = chr(ord('a') + j)
                        t = ''.join(s)
                        if dist.get(t, 0) == step:
                            prev[t].add(p) # repeated 3 lines below
                        if t not in words:
                        # if above '== step' met, then t must be removed from words[] from previous
iterations
                            continue
                        prev[t].add(p) # repeated 3 lines above
                        words.discard(t)
                        q.append(t)
                        dist[t] = step
                        if endWord == t:
                            found = True
                    s[i] = ch
        if found:
            path = [endWord]
            dfs(path, endWord)
        return ans

###########

from collections import deque

class Solution(object):
```


[*] Backtracking — Pattern Solution (matched from community answers)

Python

```
# Python solution for Word Ladder II

from collections import defaultdict, deque

def findLadders(beginWord, endWord, wordList):
    # Convert list to set for fast lookup
    wordSet = set(wordList)
    if endWord not in wordSet:
        return []

    # Dictionary to hold parent pointers for backtracking: child -> list of parents
    parents = defaultdict(list)
    # Set for words visited at current BFS level
    level = {beginWord}
    # Flag to stop BFS when endWord is reached
    found = False

    while level and not found:
        next_level = set()
        # Remove words of current level from wordSet to prevent cycles
        wordSet -= level
        for word in level:
            # Try all possible one-letter transformations
            for i in range(len(word)):
                for char in 'abcdefghijklmnopqrstuvwxyz':

                    candidate = word[:i] + char + word[i+1:]
                    if candidate in wordSet:
                        # If candidate is the endWord, note that we have found a path
                        if candidate == endWord:
                            found = True
                        # Record the current word as a parent of candidate word
                        parents[candidate].append(word)
                        next_level.add(candidate)
        # Move to the next level of the BFS
        level = next_level

    # The result list to hold all valid transformation sequences
    res = []

    # Helper function to backtrack from endWord to beginWord using the parents dictionary
    def backtrack(word, path):
        if word == beginWord:
            # Append reversed path to have the sequence from beginWord to endWord
            res.append(path[::-1])
            return
        # Recursive over all parents of the current word
        for parent in parents[word]:
            backtrack(parent, path + [parent])

    if found:
```



```
        backtrack(endWord, [endWord])
    return res

# Example usage:
# print(findLadders("hit", "cog", ["hot","dot","dog","lot","log","cog"]))
```

#996 HARD

Number of Squareful Arrays

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Math · Dynamic Programming · Backtracking

Lists: Denny Zhang

Problem:

An array is squareful if the sum of every pair of adjacent elements is a perfect square.

Given an integer array nums, return the number of permutations of nums that are squareful.

Two permutations perm1 and perm2 are different if there is some index i such that perm1[i] != perm2[i].

Example 1:

```
Input: nums = [1,17,8]
Output: 2
Explanation: [1,8,17] and [17,8,1] are the valid permutations.
```

Example 2:

```
Input: nums = [2,2,2]
Output: 1
```

Constraints:

- 1 <= nums.length <= 12
- 0 <= nums[i] <= 109

>> Brute Force [Backtracking] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def is_square(self, num):
        # Brute Force - exhaustive backtracking without pruning
        results = []
        def backtrack(start, path):
            results.append(list(path))
            for i in range(start, len(num)):
                path.append(num[i])
                backtrack(i + 1, path)
                path.pop()
        backtrack(0, [])
        return results
```

Python

Python


```
from math import isqrt

# Function to check if a number is a perfect square
def is_square(num):
    root = isqrt(num)
    return root * root == num

def numSquarefulPerms(nums):
    n = len(nums)
    nums.sort() # Sort the array to handle duplicates
    visited = [False] * n
    result = 0

    def dfs(last_index, count):
        nonlocal result
        # If all elements have been used, we found a valid permutation
        if count == n:
            result += 1
            return
        for i in range(n):
            if visited[i]:
                continue
            # Skip duplicates: if the current element is the same as the previous one and the previous one
hasn't been used
            if i > 0 and nums[i] == nums[i - 1] and not visited[i - 1]:
                continue

            # If it's the first element or the sum with the last element is a perfect square
            if last_index == -1 or is_square(nums[last_index] + nums[i]):
                visited[i] = True
                dfs(i, count + 1)
                visited[i] = False

    dfs(-1, 0)
    return result

# Example usage:
if __name__ == "__main__":
    print(numSquarefulPerms([1, 17, 8])) # Expected output: 2
    print(numSquarefulPerms([2, 2, 2])) # Expected output: 1
```

Python

```python
class Solution:
    def numSquarefulPerms(self, nums: list[int]) -> int:
        ans = 0
        used = [False] * len(nums)

        def isSquare(num: int) -> bool:
            root = math.isqrt(num)
            return root * root == num

        def dfs(path: list[int]) -> None:
            nonlocal ans
            if len(path) > 1 and not isSquare(path[-1] + path[-2]):
                return
            if len(path) == len(nums):
                ans += 1
                return

            for i, a in enumerate(nums):
                if used[i]:
                    continue
                if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:
                    continue
                used[i] = True
                dfs(path + [a])
                used[i] = False

        nums.sort()
        dfs([])
        return ans
```

Python

```python
class Solution:
    def numSquarefulPerms(self, nums: List[int]) -> int:
        n = len(nums)
        f = [[0] * n for _ in range(1 << n)]
        for j in range(n):
            f[1 << j][j] = 1
        for i in range(1 << n):
            for j in range(n):
                if i >> j & 1:
                    for k in range(n):
                        if (i >> k & 1) and k != j:
                            s = nums[j] + nums[k]
                            t = int(sqrt(s))
                            if t * t == s:
                                f[i][j] += f[i ^ (1 << j)][k]

        ans = sum(f[(1 << n) - 1][j] for j in range(n))
        for v in Counter(nums).values():
            ans //= factorial(v)
        return ans
```

Python


```python
class Solution:
    def numSquarefulPerms(self, nums: List[int]) -> int:
        n = len(nums)
        f = [[0] * n for _ in range(1 << n)]
        for j in range(n):
            f[1 << j][j] = 1
        for i in range(1 << n):
            for j in range(n):
                if i >> j & 1:
                    for k in range(n):
                        if (i >> k & 1) and k != j:
                            s = nums[j] + nums[k]
                            t = int(sqrt(s))
                            if t * t == s:
                                f[i][j] += f[i ^ (1 << j)][k]

        ans = sum(f[(1 << n) - 1][j] for j in range(n))
        for v in Counter(nums).values():
            ans //= factorial(v)
        return ans
```

[*] Backtracking — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def numSquarefulPerms(self, nums: list[int]) -> int:
        ans = 0
        used = [False] * len(nums)

        def isSquare(num: int) -> bool:
            root = math.isqrt(num)
            return root * root == num

        def dfs(path: list[int]) -> None:
            nonlocal ans
            if len(path) > 1 and not isSquare(path[-1] + path[-2]):
                return
            if len(path) == len(nums):
                ans += 1
                return

            for i, a in enumerate(nums):
                if used[i]:
                    continue
                if i > 0 and nums[i] == nums[i - 1] and not used[i - 1]:
                    continue
                used[i] = True
                dfs(path + [a])
                used[i] = False

        nums.sort()
        dfs([])
        return ans
```


Quick Review — Backtracking 10 questions · insights · complexity · solution

#17 Letter Combinations of a Phone Number [Medium]

Key Insights

- Use a mapping from digits to the respective letters as defined by a telephone keypad.
- Backtracking is an ideal approach to generate all possible combinations.
- Each digit adds a level in the recursive tree where you append one letter at a time.
- Handling edge cases such as an empty input string is crucial.

Space & Time

Time Complexity: O(4^n * n) where n is the length of the input string (each digit may map to up to 4 letters and adding a combination takes O(n) time).

Space Complexity: O(n) for the recursion call stack, not counting the space required for storing the output.

Solution

We can solve the problem using a backtracking algorithm. First, we define a dictionary mapping each digit to its corresponding letters. Then we initiate a recursive function called backtrack that takes the current combination and the index of the next digit to process. At each call, if the combination is complete (length equal to the input string), we add it to the result list. Otherwise, we iterate through the letters corresponding to the current digit, append a letter to the combination, and recursively call backtrack with the next index. This approach ensures that all possible letter combinations are generated.

#22 Generate Parentheses [Medium]

Key Insights

- Use a backtracking approach to build the valid combinations step by step.
- At any recursion step, you can add an opening parenthesis if you still have one available.
- You can add a closing parenthesis only if it would not result in an invalid sequence (i.e., the number of closing parentheses is less than the number of opening ones).
- The solution leverages recursion to explore all potential valid sequences while pruning invalid paths early.

Space & Time

Time Complexity: O(4^n / (n^(3/2)))

Space Complexity: O(n) for the recursion stack, plus space for storing all valid combinations.

Solution

The solution uses a recursive backtracking technique to generate the combinations. The algorithm maintains counters for the number of '(' and ')' added to the current string. It recursively adds a '(' if the count is less than n and adds a ')' if doing so does not make the string invalid (i.e., if the count of ')' is less than the count of '('). When a valid combination reaches a length of 2*n, it is added to the result list.

#39 Combination Sum [Medium]

Key Insights

- Backtracking is a natural fit to explore all potential combinations.
- Sorting candidates simplifies the process by allowing early termination if the current candidate exceeds the remaining target.
- Recursion is used to build combinations and backtrack once the current path exceeds or meets the target.
- The same candidate can be reused in the combination, so the recursive call does not advance the index.

Space & Time

Time Complexity: O(2^(target/min(candidates))) in the worst case where many combinations are possible.

Space Complexity: O(target/min(candidates)) for the recursion stack and temporary space used to store combinations.

Solution

The solution uses a backtracking approach. First, sort the candidates to enable pruning of recursive calls once the candidate exceeds the remaining target. A recursive helper function builds up a combination, subtracting the candidate value from the target until it reaches zero (a valid combination) or becomes negative (an invalid path). The recursion allows the same candidate to be chosen again. Each valid combination is then added to the result list.

#46 Permutations [Medium]

Key Insights

- Use backtracking to generate all possible permutations.
- Build each permutation by selecting numbers one at a time.
- When the current permutation reaches the length of nums, add it to the result.
- Backtracking allows you to undo choices and explore different orderings.

Space & Time

Time Complexity: O(n * n!) - There are n! permutations, and constructing each permutation takes O(n) time.

Space Complexity: O(n) - Recursion depth is limited to n, plus additional space for the current permutation.

Solution

The solution employs a backtracking algorithm. A helper function is used to build permutations recursively. At each step, the algorithm iterates through the available numbers, adds one to the current permutation if it has not been used yet, and recurses to handle the remaining numbers. Once the current permutation matches the length of the input array, it is added to the result list. Then, by "backtracking" (removing the last number added), the algorithm explores alternative possibilities. This systematic exploration ensures that all possible permutations are generated.

#79 Word Search [Medium]

Key Insights

- Use Depth-First Search (DFS) to explore each cell in the board as a potential starting point.
- Employ backtracking to mark cells as visited when exploring a path and revert them back when backtracking.
- Check boundary conditions and ensure the current cell's character matches the corresponding character in the word.
- Given the small grid and word sizes, a recursive solution is both feasible and straightforward.
- Search pruning can be applied by stopping a DFS branch as soon as the current path does not match the prefix of the target word.

Space & Time

Time Complexity: O(m * n * 3^L), where L is the length of the word (each DFS may branch to at most 3 further cells, excluding the coming cell).

Space Complexity: O(L) due to recursion call stack (where L is the word length).

Solution

The solution uses DFS with backtracking to traverse the grid. For each cell, we:

- Check if the cell matches the first character of the word. - Recursively explore the adjacent cells (up, down, left, right) while marking the current cell as visited. - If the path does not lead to a complete match, backtrack by resetting the visited cell. - The DFS stops if all characters in the word have been found in sequence.

The algorithm leverages in-place modifications (or an auxiliary visited array) to avoid revisiting cells while exploring a candidate path.

#113 Path Sum II [Medium]

Key Insights

- Use Depth-First Search (DFS) to explore all paths from the root to the leaves.

- Backtracking is essential: add the current node value to the path before the recursive calls and remove it (backtrack) after exploring both subtrees.
- At each leaf, check if the accumulated sum equals targetSum.
- Maintain a running sum or subtract the current node's value from the target sum as you traverse.

Space & Time

Time Complexity: O(N^2) in the worst-case scenario (where N is the number of nodes) when most nodes are leaf nodes, since each potential path copy operation can be O(N).

Space Complexity: O(N) for the recursion stack and the path storage, where N is the height of the tree.

Solution

We solve the problem using a recursive DFS approach combined with backtracking. The idea is to traverse the tree while maintaining the current path and the remaining sum needed to reach targetSum. When a leaf node is reached (node with no children), we check if the remaining sum equals the leaf's value - if yes, we record a copy of the current path.

Data structures and algorithmic approaches used:

- Recursion for DFS: to traverse to each leaf. - List (or Array) for storing the current path. - Backtracking: to remove the last added element when stepping back up the recursion stack. - Copying the path when a valid leaf is reached to store it in the final result.

A common pitfall is to forget to backtrack, which would result in incorrect paths being recorded.

#37 Sudoku Solver [Hard]

Key Insights

- Use backtracking to explore digit placements for each empty cell.
- Validate placements by ensuring that the chosen digit is not already present in the corresponding row, column, or 3x3 sub-box.
- Employ recursion to fill the board incrementally and backtrack upon encountering invalid configurations.
- Although the worst-case time complexity is exponential, the fixed board size (9x9) keeps the problem computationally manageable.

Space & Time

Time Complexity: In the worst-case scenario, the algorithm can explore up to O(9^(n)) possibilities where n is the number of empty cells. However, since n is at most 81 and many branches are pruned early, the practical runtime is far less.

Space Complexity: O(n) due to the recursion stack, where n is the number of empty cells; given the board's fixed size, this space is constant in practice.

Solution

The solution uses a backtracking approach. For each empty cell on the board, we attempt to place a digit from '1' to '9'. Before placing a digit, we check if the placement is valid by ensuring that:

- The digit is not already present in the same row. - The digit is not already present in the same column. - The digit is not already present in the corresponding 3x3 sub-box.

If a valid digit is found, we place it on the board and recursively attempt to solve the rest of the board. If no valid digit can be placed, the algorithm backtracks by reverting the last move and trying a different digit. This recursive search continues until the board is completely filled with a valid Sudoku configuration.

#51 N-Queens [Hard]

Key Insights

- Use backtracking to explore potential positions row by row.
- Use sets to track which columns and diagonals are under attack.
- Diagonals can be tracked by (row - col) and (row + col), ensuring that queens do not conflict along any diagonal.
- Recursively build each valid board configuration and backtrack when a conflict arises.

Space & Time

Time Complexity: O(n!) in the worst-case scenario, as the solution explores permutations for queen placements.

Space Complexity: O(n^2) for storing board configurations and O(n) for recursion stack and conflict-tracking sets.

Solution

We use a backtracking algorithm that places queens row by row. For each row, we try placing a queen in each column. Before placing, we check if the column, main diagonal (row - col), and anti-diagonal (row + col) are free using dedicated sets. If a queen can be safely placed, we mark the column and diagonals as occupied and recurse to the next row. Once all rows are processed, the board configuration is a valid solution and is added to the result. After that, we backtrack by unmarking the sets and removing the queen from the board. This systematic approach ensures all possible configurations are considered.

#126 Word Ladder II [Hard]

Key Insights

- Use Breadth-First Search (BFS) to traverse level-by-level and capture only the shortest paths.
- Build a parent mapping during BFS to record which words lead to each newly discovered word.
- Once the shortest path is found via BFS, use backtracking (or DFS) from endWord to reconstruct all transformation sequences.
- Remove words as they are visited within a level to avoid unnecessary revisits and cycles.
- The solution must handle multiple shortest paths, so maintaining a list of predecessors for each word is crucial.

Space & Time

Time Complexity: O(N * L * 26) where N is the number of words in the wordList and L is the length of each word. This accounts for checking all possible one-letter transformations.

Space Complexity: O(N * L) for storing the parent mapping and maintaining the BFS queue and other auxiliary data structures.

Solution

We solve the problem in two major phases:

- Use BFS to explore from beginWord to endWord. During BFS, record all parents for each word encountered. This ensures we capture all possible predecessors that yield a shortest transformation. - Once the BFS completes (once endWord is reached), use backtracking (DFS) beginning from the endWord and moving backwards via the parent mapping to reconstruct all transformation sequences in reverse. Finally, reverse each sequence to present it from beginWord to endWord.

This two-phase approach guarantees that only the shortest paths are reconstructed, as BFS ensures level-by-level exploration.

#996 Number of Squareful Arrays [Hard]

Key Insights

- The problem requires ensuring adjacent pairs sum up to a perfect square.
- Sorting the array helps in efficiently handling duplicates; when iterating in the backtracking, skip duplicate elements to avoid overcounting.
- For each valid candidate, check that the sum with the previous number (if any) is a perfect square.
- Use backtracking (DFS) to generate all valid permutations while maintaining a visited flag for each index.
- Since the array may contain duplicates, extra care is taken to only use each occurrence in a controlled order (i.e., if two identical numbers exist, only the first unvisited instance can be chosen at a given recursive call).

Space & Time

Time Complexity: O(n^2 * 2^n) in the worst-case scenario due to exploring all bitmask states with n elements and checking pairwise sums.

Space Complexity: O(n) for recursion and O(n) for the visited flag, plus potentially O(2^n * n) for memoization if implemented.

Solution

The solution uses a backtracking algorithm to explore all permutations. The array is first sorted to handle duplicates. A helper function checks if a number is a perfect square and is used to validate that the sum of every adjacent pair is a perfect square. During DFS, a visited array is maintained to track elements included so far. Skipping duplicate elements (unless the earlier duplicate was used) is essential to avoid overcounting. This approach ensures that every valid permutation is considered exactly once.

#9 BFS — 10 questions · #9 of 21

#286 MEDIUM

Walls and Gates

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · BFS · Matrix

Lists; Premium 98

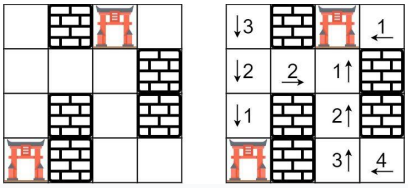
Problem:

You are given an m x n grid rooms initialized with these three possible values.

- −1 A wall or an obstacle.
- 0 A gate.
- INF Infinity means an empty room. We use the value 231 − 1 = 2147483647 to represent INF as you may assume that the distance to a gate is less than 2147483647.

Fill each empty room with the distance to its nearest gate. If it is impossible to reach a gate, it should be filled with INF.

Example 1:



```
Input: rooms = [[2147483647,-1,0,2147483647],[2147483647,2147483647,2147483647,-1],[2147483647,-1,2147483647,-1],[0,-1,2147483647,2147483647]]
Output: [[3,-1,0,1],[2,2,1,-1],[1,-1,2,-1],[0,-1,3,4]]
```

Example 2:

```
Input: rooms = [[-1]]
Output: [[-1]]
```

Constraints:

- m == rooms.length
- n == rooms[i].length

- 1 <= m, n <= 250
- rooms[i][j] is −1, 0, or 231 − 1.

>> Brute Force [BFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def wallsAndGates(self, rooms):
        # Brute Force Bfs(uff)) - BFS from every empty room independently
        from collections import deque
        m, n = len(rooms), len(rooms[0])
        INF = 2147483647
        def bfs_from(sr, sc):
            queue = deque([(sr,sc,0)])
            visited = set()
            while queue:
                r,c,d = queue.popleft()
                if (r,c) in visited: continue
                visited.add((r,c))
                if rooms[r][c]==0: return d # reached a gate
                for dr,dc in [(1,0),(-1,0),(0,1),(0,-1)]:
                    nr,nc = r+dr,c+dc
                    if 0<=nr<m and 0<=nc<n and rooms[nr][nc]!=(-1):
                        queue.append((nr,nc,d+1))
            return INF
        for r in range(m):
            for c in range(n):
                if rooms[r][c]==INF:
                    rooms[r][c]=bfs_from(r,c)
```

Python

Python


```python
from collections import deque

def wallsAndGates(rooms):
    if not rooms or not rooms[0]:
        return

    rows, cols = len(rooms), len(rooms[0])
    # Directions: up, down, left, right.
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    queue = deque()

    # Enqueue all gates (cells with 0).
    for r in range(rows):
        for c in range(cols):
            if rooms[r][c] == 0:
                queue.append((r, c))

    # Perform multi-source BFS.
    while queue:
        r, c = queue.popleft()
        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            # Process neighbor if it is within bounds and is an empty room.
            if 0 <= nr < rows and 0 <= nc < cols and rooms[nr][nc] == 2147483647:
                rooms[nr][nc] = rooms[r][c] + 1 # Update distance.

                queue.append((nr, nc))
```

Python

```python
class Solution:
  def wallsAndGates(self, rooms: list[list[int]]) -> None:
    DIRS = ((0, 1), (1, 0), (0, -1), (-1, 0))
    INF = 2**31 - 1
    m = len(rooms)
    n = len(rooms[0])
    q = collections.deque((i, j)
                          for i in range(m)
                          for j in range(n)
                          if rooms[i][j] == 0)


    while q:
      i, j = q.popleft()
      for dx, dy in DIRS:
        x = i + dx
        y = j + dy
        if x < 0 or x >= m or y < 0 or y >= n:
          continue
        if rooms[x][y] != INF:
          continue
        rooms[x][y] = rooms[i][j] + 1
        q.append((x, y))
```

Python


```python
class Solution:
    def wallsAndGates(self, rooms: List[List[int]]) -> None:
        """
        Do not return anything, modify rooms in-place instead.
        """
        m, n = len(rooms), len(rooms[0])
        inf = 2**31 - 1
        q = deque([(i, j) for i in range(m) for j in range(n) if rooms[i][j] == 0])
        d = 0
        while q:
            d += 1
            for _ in range(len(q)):
                i, j = q.popleft()
                for a, b in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                    x, y = i + a, j + b
                    if 0 <= x < m and 0 <= y < n and rooms[x][y] == inf:
                        rooms[x][y] = d
                        q.append((x, y))
```

Python

```python
class Solution:
    def wallsAndGates(self, rooms: List[List[int]]) -> None:
        """
        Do not return anything, modify rooms in-place instead.
        """
        m, n = len(rooms), len(rooms[0])
        inf = 2**31 - 1
        q = deque([(i, j) for i in range(m) for j in range(n) if rooms[i][j] == 0])
        d = 0
        while q:
            d += 1
            for _ in range(len(q)):
                i, j = q.popleft()
                for a, b in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                    x, y = i + a, j + b
                    # bfs => first reaching inf is always the shortest
                    # set its value from inf to d, same as marking as visited
                    if 0 <= x < m and 0 <= y < n and rooms[x][y] == inf:
                        rooms[x][y] = d
                        q.append((x, y))

###########

class Solution: # dfs
    def wallsAndGates(self, rooms: List[List[int]]) -> None:
```



```python
        if not rooms or not rooms[0]:
            return

        m, n = len(rooms), len(rooms[0])
        isVisited = [[False] * n for _ in range(m)] # isVisited map shared among all gates

        def fill(i: int, j: int, distance: int) -> None:
            nonlocal m, n

            # rooms[i][j] == 0 meaning == -1 or == 0, both stop
            if i < 0 or i >= m or j < 0 or j >= n or rooms[i][j] <= 0 or isVisited[i][j]:
                return

            rooms[i][j] = min(rooms[i][j], distance)
            isVisited[i][j] = True

            fill(i - 1, j, distance + 1)
            fill(i, j + 1, distance + 1)
            fill(i + 1, j, distance + 1)
            fill(i, j - 1, distance + 1)

            isVisited[i][j] = False

        for i in range(m):
            for j in range(n):

                if rooms[i][j] == 0: # start from every gate
                    fill(i, j, 0)

###########

from collections import deque

INF = 2147483647


class Solution(object):
  def wallsAndGates(self, rooms):
    """
    :type rooms: List[List[int]]
    :rtype: void Do not return anything, modify rooms in-place instead.
    """
    queue = deque([])
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]
    for i in range(0, len(rooms)):
      for j in range(0, len(rooms[0])):
        if rooms[i][j] == 0:
          queue.append((i, j))

    while queue:
      i, j = queue.popleft()


      for di, dj in directions:
        p, q = i + di, j + dj
        if 0 <= p < len(rooms) and 0 <= q < len(rooms[0]) and rooms[p][q] == INF:
          rooms[p][q] = rooms[i][j] + 1
          queue.append((p, q))
```

[*] BFS — Pattern Solution (matched from community answers)

Python

```python
from collections import deque

def wallsAndGates(rooms):
    if not rooms or not rooms[0]:
        return

    rows, cols = len(rooms), len(rooms[0])
    # Directions: up, down, left, right.
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    queue = deque()

    # Enqueue all gates (cells with 0).
    for r in range(rows):
        for c in range(cols):
            if rooms[r][c] == 0:
                queue.append((r, c))

    # Perform multi-source BFS.
    while queue:
        r, c = queue.popleft()
        for dr, dc in directions:
            nr, nc = r + dr, c + dc
            # Process neighbor if it is within bounds and is an empty room.
            if 0 <= nr < rows and 0 <= nc < cols and rooms[nr][nc] == 2147483647:
                rooms[nr][nc] = rooms[r][c] + 1 # Update distance.

                queue.append((nr, nc))
```

#322 MEDIUM

Coin Change

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming · BFS

Lists: Grind 169

Problem:

You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

Example 1:

```
Input: coins = [1,2,5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1
```

Example 2:

```
Input: coins = [2], amount = 3
Output: -1
```


Example 3:

```
Input: coins = [1], amount = 0
Output: 0
```

Constraints:

- 1 <= coins.length <= 12
- 1 <= coins[i] <= 231 - 1
- 0 <= amount <= 104

>> Brute Force [BFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def coinChange(self, coins, amount):
        # Brute force dfamount - 1*(amount/min)) - recursion without memo
        def dp(rem):
            if rem == 0: return 0
            if rem < 0: return float('inf')
            return 1 + min(dp(rem - c) for c in coins)
        res = dp(amount)
        return res if res != float('inf') else -1
```

Python

Python

```python
def coinChange(coins, amount):
    # Create a dp array with amount+1 entries set to a value greater than any possible minimum number of
coins.
    dp = [amount + 1] * (amount + 1)
    # Base case: 0 coins are needed to form amount 0.
    dp[0] = 0

    # Build up the dp table.
    for current_amount in range(1, amount + 1):
        for coin in coins:
            if coin <= current_amount:
                dp[current_amount] = min(dp[current_amount], dp[current_amount - coin] + 1)

    # If dp[amount] remains greater than amount, no solution was found.
    return dp[amount] if dp[amount] != amount + 1 else -1
```

Python

```python
class Solution:
  def coinChange(self, coins: list[int], amount: int) -> int:
    # dp[i] := the minimal number of coins to make up i
    dp = [0] + [amount + 1] * amount

    for coin in coins:
      for i in range(coin, amount + 1):
        dp[i] = min(dp[i], dp[i - coin] + 1)

    return -1 if dp[amount] == amount + 1 else dp[amount]
```

Python


```python
class Solution:
    def coinChange(self, coins: int, amount: int) -> int:
        m, n = len(coins), amount
        f = [[inf] * (n + 1) for _ in range(m + 1)]
        f[0][0] = 0
        for i, x in enumerate(coins, 1):
            for j in range(n + 1):
                f[i][j] = f[i - 1][j]
                if j >= x:
                    f[i][j] = min(f[i][j], f[i][j - x] + 1)
        return -1 if f[m][n] >= inf else f[m][n]
```

Python

```python
'''
>>> float("inf")
inf
>>> float("inf") + 1
inf
'''

class Solution(object):
  def coinChange(self, coins, amount):
    """
    :type coins: List[int]
    :type amount: int
    :rtype: int
    """

    dp = [float("inf")] + (amount + 1)
    dp[0] = 0
    for i in range(1, amount + 1):
      for coin in coins:
        if i - coin >= 0:
          dp[i] = min(dp[i], dp[i - coin] + 1)
    return dp[-1] if dp[-1] != float("inf") else -1

###########
```

```python
class Solution:
    def coinChange(self, coins: List[int], amount: int) -> int:
        dp = [amount + 1] * (amount + 1)
        dp[0] = 0
        for coin in coins:
            for j in range(coin, amount + 1):
                dp[j] = min(dp[j], dp[j - coin] + 1)
        return -1 if dp[-1] > amount else dp[-1]
```

[*] BFS — Pattern Solution (stored optimal)

Python


```python
def coinChange(coins, amount):
    # Create a dp array with amount+1 entries set to a value greater than any possible minimum number of
coins.
    dp = [amount + 1] * (amount + 1)
    # Base case: 0 coins are needed to form amount 0.
    dp[0] = 0

    # Build up the dp table.
    for current_amount in range(1, amount + 1):
        for coin in coins:
            if coin <= current_amount:
                dp[current_amount] = min(dp[current_amount], dp[current_amount - coin] + 1)

    # If dp[amount] remains greater than amount, no solution was found.
    return dp[amount] if dp[amount] != amount + 1 else -1
```

#542 MEDIUM

01 Matrix

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · BFS · Matrix

Lists: Grind 169 | Denny Zhang

Problem:

Given an m x n binary matrix mat, return the distance of the nearest 0 for each cell.

The distance between two cells sharing a common edge is 1.

Example 1:

0	0	0
0	1	0
0	0	0

```
Input: mat = [[0,0,0],[0,1,0],[0,0,0]]
Output: [[0,0,0],[0,1,0],[0,0,0]]
```

Example 2:

0	0	0
0	1	0
1	1	1

```
Input: mat = [[0,0,0],[0,1,0],[1,1,1]]
Output: [[0,0,0],[0,1,0],[1,2,1]]
```

Constraints:

- m == mat.length
- n == mat[i].length
- 1 <= m, n <= 104
- 1 <= m * n <= 104
- mat[i][j] is either 0 or 1.
- There is at least one 0 in mat.

Note: This question is the same as 1765: https://leetcode.com/problems/map-of-highest-peak/

>> Brute Force [BFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def updateMatrix(self, mat):
        # Brute Force - BFS layer by layer
        from collections import deque
        queue = deque([0])
        visited = {0}
        steps = 0
        while queue:
            for _ in range(len(queue)):
                node = queue.popleft()
                for nb in graph[node]:
                    if nb not in visited:
                        visited.add(nb); queue.append(nb)
            steps += 1
        return steps
```


Python

Python

```python
from collections import deque

def updateMatrix(mat):
    # Get the dimensions of the matrix
    rows, cols = len(mat), len(mat[0])
    # Initialize a queue for multi-source BFS
    queue = deque()

    # Initialize the matrix: use -1 to indicate unvisited cells
    for r in range(rows):
        for c in range(cols):
            if mat[r][c] == 0:
                # Enqueue all 0 cells as starting points
                queue.append((r, c))
            else:
                # Mark 1 cells as unvisited (or infinite distance)
                mat[r][c] = -1

    # Directions for neighbors (up, down, left, right)
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    # BFS traversal to update distances
    while queue:
        r, c = queue.popleft()
        for dr, dc in directions:

            nr, nc = r + dr, c + dc
            # Check boundaries and if neighbor has not been visited
            if 0 <= nr < rows and 0 <= nc < cols and mat[nr][nc] == -1:
                # Update neighbor's distance
                mat[nr][nc] = mat[r][c] + 1
                # Enqueue the neighbor for further BFS processing
                queue.append((nr, nc))

    return mat

# Example usage:
# print(updateMatrix([[0,0,0],[0,1,0],[1,1,1]]))
```

Python

```python
class Solution:
    def updateMatrix(self, mat: list[list[int]]) -> list[list[int]]:
        DIRS = ((0, 1), (1, 0), (0, -1), (-1, 0))
        m = len(mat)
        n = len(mat[0])
        q = collections.deque()
        seen = [[False] * n for _ in range(m)]

        for i in range(m):
          for j in range(n):
            if mat[i][j] == 0:
              q.append((i, j))
              seen[i][j] = True

        while q:
          i, j = q.popleft()
          for dx, dy in DIRS:
            x = i + dx
            y = j + dy
            if x < 0 or x >= m or y < 0 or y >= n:
              continue
            if seen[x][y]:
              continue
            mat[x][y] = mat[i][j] + 1
            q.append((x, y))

            seen[x][y] = True

        return mat
```

Python

```python
class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        m, n = len(mat), len(mat[0])
        ans = [[-1] * n for _ in range(m)]
        q = deque()
        for i, row in enumerate(mat):
            for j, x in enumerate(row):
                if x == 0:
                    ans[i][j] = 0
                    q.append((i, j))
        dirs = (-1, 0, 1, 0, -1)
        while q:
            i, j = q.popleft()
            for a, b in pairwise(dirs):
                x, y = i + a, j + b
                if 0 <= x < m and 0 <= y < n and ans[x][y] == -1:
                    ans[x][y] = ans[i][j] + 1
                    q.append((x, y))
        return ans
```

Python


```python
class Solution:
    def updateMatrix(self, mat: List[List[int]]) -> List[List[int]]:
        m, n = len(mat), len(mat[0])
        ans = [[-1] * n for _ in range(m)]
        q = deque()
        for i, row in enumerate(mat):
            for j, x in enumerate(row):
                if x == 0:
                    ans[i][j] = 0
                    q.append((i, j))
        dirs = (-1, 0, 1, 0, -1)
        while q:
            i, j = q.popleft()
            for a, b in pairwise(dirs):
                x, y = i + a, j + b
                if 0 <= x < m and 0 <= y < n and ans[x][y] == -1:
                    ans[x][y] = ans[i][j] + 1
                    q.append((x, y))
        return ans
```

[*] BFS — Pattern Solution (matched from community answers)

Python

```python
from collections import deque

def updateMatrix(mat):
    # Get the dimensions of the matrix
    rows, cols = len(mat), len(mat[0])
    # Initialize a queue for multi-source BFS
    queue = deque()

    # Initialize the matrix: use -1 to indicate unvisited cells
    for r in range(rows):
        for c in range(cols):
            if mat[r][c] == 0:
                # Enqueue all 0 cells as starting points
                queue.append((r, c))
            else:
                # Mark 1 cells as unvisited (or infinite distance)
                mat[r][c] = -1

    # Directions for neighbors (up, down, left, right)
    directions = [(1, 0), (-1, 0), (0, 1), (0, -1)]

    # BFS traversal to update distances
    while queue:
        r, c = queue.popleft()
        for dr, dc in directions:
```



```python
            nr, nc = r + dr, c + dc
            # Check boundaries and if neighbor has not been visited
            if 0 <= nr < rows and 0 <= nc < cols and mat[nr][nc] == -1:
                # Update neighbor's distance
                mat[nr][nc] = mat[r][c] + 1
                # Enqueue the neighbor for further BFS processing
                queue.append((nr, nc))

    return mat

# Example usage:
# print(updateMatrix([[0,0,0],[0,1,0],[1,1,1]]))
```

#994 MEDIUM

Rotting Oranges

LeetDocs · SimpleLest · WalkCC · LeetCode · Editorial

Array · BFS · Matrix

Lists: Grind 169

Problem:

You are given an m x n grid where each cell can have one of three values:

- 0 representing an empty cell,
- 1 representing a fresh orange, or
- 2 representing a rotten orange.

Every minute, any fresh orange that is 4-directionally adjacent to a rotten orange becomes rotten.

Return the minimum number of minutes that must elapse until no cell has a fresh orange. If this is impossible, return -1.

Example 1:

```
Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4
```

Example 2:

```
Input: grid = [[2,1,1],[0,1,1],[1,0,1]]
Output: -1
Explanation: The orange in the bottom left corner (row 2, column 0) is never rotten, because
rotting only happens 4-directionally.
```

Example 3:

```
Input: grid = [[0,2]]
Output: 0
```



```
Explanation: Since there are already no fresh oranges at minute 0, the answer is just 0.
```

Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 10
- grid[i][j] is 0, 1, or 2.

>> Brute Force [BFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def orangesRotting(self, grid):
        # Brute force: O(m*n*T) - repeatedly scan entire grid, spread rot each pass
        from copy import deepcopy
        m, n = len(grid), len(grid[0])
        minutes = 0
        while True:
            new_grid = deepcopy(grid)
            changed = False
            for r in range(m):
                for c in range(n):
                    if grid[r][c] == 2:
                        for dr,dc in [(1,0),(-1,0),(0,1),(0,-1)]:
                            nr,nc = r+dr,c+dc
                            if 0<=nr<m and 0<=nc<n and grid[nr][nc]==1:
                                new_grid[nr][nc]=2; changed=True

            grid = new_grid
            if not changed: break
            minutes += 1
        return minutes if not any(1 in row for row in grid) else -1
```

Python

Python


```python
from collections import deque

def orangesRotting(grid):
    # Dimensions of the grid
    rows, cols = len(grid), len(grid[0])
    # Queue to perform BFS, containing tuples of (row, col)
    rotten_queue = deque()
    fresh_count = 0

    # Step 1: Initialize the queue with all initially rotten oranges and count fresh oranges.
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 2:
                rotten_queue.append((r, c))
            elif grid[r][c] == 1:
                fresh_count += 1

    # If there are no fresh oranges, return 0 immediately.
    if fresh_count == 0:
        return 0

    minutes_passed = 0
    # Directions for adjacent cells: up, down, left, right.
    directions = [(-1,0), (1,0), (0,-1), (0,1)]

    # Step 2: BFS to rot adjacent fresh oranges.
    while rotten_queue:
        # Process oranges that are rotten in the current minute.
        for _ in range(len(rotten_queue)):
            x, y = rotten_queue.popleft()
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                # Check if the neighbor is inside the grid and is a fresh orange.
                if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 1:
                    # Mark the fresh orange as rotten.
                    grid[nx][ny] = 2
                    fresh_count -= 1
                    # Add the new rotten orange into the queue.
                    rotten_queue.append((nx, ny))
        # Increment minutes after processing one level.
        minutes_passed += 1

    # If there are still fresh oranges left, return -1.
    return minutes_passed - 1 if fresh_count == 0 else -1

# Example usage:
# grid = [[2,1,1],[1,1,0],[0,1,1]]
# print(orangesRotting(grid))
```

Python


```python
class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        cnt = 0
        q = deque()
        for i, row in enumerate(grid):
            for j, x in enumerate(row):
                if x == 2:
                    q.append((i, j))
                elif x == 1:
                    cnt += 1
        ans = 0
        dirs = (-1, 0, 1, 0, -1)
        while q and cnt:
            ans += 1
            for _ in range(len(q)):
                i, j = q.popleft()
                for a, b in pairwise(dirs):
                    x, y = i + a, j + b
                    if 0 <= x < m and 0 <= y < n and grid[x][y] == 1:
                        grid[x][y] = 2
                        q.append((x, y))
                        cnt -= 1
                        if cnt == 0:
                            return ans

        return -1 if cnt else 0
```

Python

```python
class Solution:
    def orangesRotting(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        q = deque()
        cnt = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 2:
                    q.append((i, j))
                elif grid[i][j] == 1:
                    cnt += 1
        ans = 0
        while q and cnt:
            ans += 1
            for _ in range(len(q)):
                i, j = q.popleft()
                for a, b in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                    x, y = i + a, j + b
                    if 0 <= x < m and 0 <= y < n and grid[x][y] == 1:
                        cnt -= 1
                        grid[x][y] = 2
                        q.append((x, y))
        return ans if cnt == 0 else -1
```

[*] BFS — Pattern Solution (matched from community answers)

Python

```python
from collections import deque

def orangesRotting(grid):
    # Dimensions of the grid
    rows, cols = len(grid), len(grid[0])
    # Queue to perform BFS, containing tuples of (row, col)
    rotten_queue = deque()
    fresh_count = 0

    # Step 1: Initialize the queue with all initially rotten oranges and count fresh oranges.
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 2:
                rotten_queue.append((r, c))
            elif grid[r][c] == 1:
                fresh_count += 1

    # If there are no fresh oranges, return 0 immediately.
    if fresh_count == 0:
        return 0

    minutes_passed = 0
    # Directions for adjacent cells: up, down, left, right.
    directions = [(-1,0), (1,0), (0,-1), (0,1)]

    # Step 2: BFS to rot adjacent fresh oranges.
    while rotten_queue:
        # Process oranges that are rotten in the current minute.
        for _ in range(len(rotten_queue)):
            x, y = rotten_queue.popleft()
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                # Check if the neighbor is inside the grid and is a fresh orange.
                if 0 <= nx < rows and 0 <= ny < cols and grid[nx][ny] == 1:
                    # Mark the fresh orange as rotten.
                    grid[nx][ny] = 2
                    fresh_count -= 1
                    # Add the new rotten orange into the queue.
                    rotten_queue.append((nx, ny))
        # Increment minutes after processing one level.
        minutes_passed += 1

    # If there are still fresh oranges left, return -1.
    return minutes_passed - 1 if fresh_count == 0 else -1

# Example usage:
# grid = [[2,1,1],[1,1,0],[0,1,1]]
# print(orangesRotting(grid))
```

#1197 MEDIUM

Minimum Knight Moves

LeetDoocs • SimplyLeet • WalkCC • LeetCode • Editorial

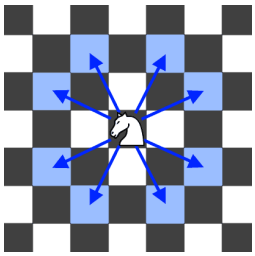
BFS

Lists: Grind 169 | Premium 98

Problem:

In an infinite chess board with coordinates from −infinity to +infinity, you have a knight at square [0, 0].

A knight has 8 possible moves it can make, as illustrated below. Each move is two squares in a cardinal direction, then one square in an orthogonal direction.



Return the minimum number of steps needed to move the knight to the square [x, y]. It is guaranteed the answer exists.

Example 1:

```
Input: x = 2, y = 1
Output: 1
Explanation: [0, 0] → [2, 1]
```

Example 2:

```
Input: x = 5, y = 5
Output: 4
Explanation: [0, 0] → [2, 1] → [4, 2] → [3, 4] → [5, 5]
```

Constraints:

- −300 <= x, y <= 300
- 0 <= |x| + |y| <= 300

>> Brute Force [BFS] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def minKnightMoves(self, x, y):
        # Brute Force - BFS layer by layer
        from collections import deque
        queue = deque([0])
        visited = {0}
        steps = 0
        while queue:
            for _ in range(len(queue)):
                node = queue.popleft()
                for nb in graph[node]:
                    if nb not in visited:
                        visited.add(nb); queue.append(nb)
            steps += 1
        return steps
```

Python

Python

```python
# Python solution using BFS:
from collections import deque

def minKnightMoves(x, y):
    # Normalize target coordinates using symmetry
    x, y = abs(x), abs(y)

    # Possible knight moves (dx, dy)
    moves = [(2, 1), (1, 2), (-1, 2), (-2, 1), (-2, -1), (-1, -2), (1, -2), (2, -1)]

    # BFS queue stores tuples of the form (current_x, current_y, move_count)
    queue = deque([(0, 0, 0)])

    # Set to track visited positions to prevent cycles
    visited = {(0, 0)}

    # Process BFS
    while queue:
        curr_x, curr_y, move_count = queue.popleft()

        # Return the current move count if target is reached
        if curr_x == x and curr_y == y:
            return move_count

        # Explore all knight moves

        for dx, dy in moves:
            next_x, next_y = curr_x + dx, curr_y + dy

            # Prune search space: only process positions with valid bounds
            if (next_x, next_y) not in visited and next_x >= -1 and next_y >= -1:
                visited.add((next_x, next_y))
                queue.append((next_x, next_y, move_count + 1))

    return -1 # Unreachable because a solution always exists

# Example test
#print(minKnightMoves(5, 5))
```


Python

```python
class Solution:
    def minKnightMoves(self, x: int, y: int) -> int:
        q = deque([(0, 0)])
        ans = 0
        vis = {(0, 0)}
        dirs = [(-2, 1), (-1, 2), (1, 2), (2, 1), (2, -1), (1, -2), (-1, -2), (-2, -1)]
        while q:
            for _ in range(len(q)):
                i, j = q.popleft()
                if (i, j) == (x, y):
                    return ans
                for a, b in dirs:
                    c, d = i + a, j + b
                    if (c, d) not in vis:
                        vis.add((c, d))
                        q.append((c, d))
            ans += 1
        return -1
```

Python

```python
class Solution:
    def minKnightMoves(self, x: int, y: int) -> int:
        def extend(m1, m2, q):
            for _ in range(len(q)):
                i, j = q.popleft()
                step = m1[(i, j)]
                for a, b in (
                    (-2, 1),
                    (-1, 2),
                    (1, 2),
                    (2, 1),
                    (2, -1),
                    (1, -2),
                    (-1, -2),
                    (-2, -1),
                ):
                    x, y = i + a, j + b
                    if (x, y) in m1:
                        continue
                    if (x, y) in m2:
                        return step + 1 + m2[(x, y)]
                    q.append((x, y))
                    m1[(x, y)] = step + 1
            return -1

        if (x, y) == (0, 0):
            return 0
        q1, q2 = deque([(0, 0)]), deque([(x, y)])
        m1, m2 = {(0, 0): 0}, {(x, y): 0}
        while q1 and q2:
            t = extend(m1, m2, q1) if len(q1) <= len(q2) else extend(m2, m1, q2)
            if t != -1:
                return t
        return -1
```

Python


```python
class Solution:
    def minKnightMoves(self, x: int, y: int) -> int:
        q = deque([(0, 0)])
        ans = 0
        vis = {(0, 0)}
        dirs = [(-2, 1), (-1, 2), (1, 2), (2, 1), (2, -1), (1, -2), (-1, -2), (-2, -1)]
        while q:
            for _ in range(len(q)):
                i, j = q.popleft()
                if (i, j) == (x, y):
                    return ans
                for a, b in dirs:
                    c, d = i + a, j + b
                    if (c, d) not in vis:
                        vis.add((c, d))
                        q.append((c, d))
            ans += 1
        return -1
```

[*] BFS — Pattern Solution (matched from community answers)

Python

```python
# Python solution using BFS:
from collections import deque

def minKnightMoves(x, y):
    # Normalize target coordinates using symmetry
    x, y = abs(x), abs(y)

    # Possible knight moves (dx, dy)
    moves = [(2, 1), (1, 2), (-1, 2), (-2, 1), (-2, -1), (-1, -2), (1, -2), (2, -1)]

    # BFS queue stores tuples of the form (current_x, current_y, move_count)
    queue = deque([(0, 0, 0)])

    # Set to track visited positions to prevent cycles
    visited = {(0, 0)}

    # Process BFS
    while queue:
        curr_x, curr_y, move_count = queue.popleft()

        # Return the current move count if target is reached
        if curr_x == x and curr_y == y:
            return move_count

        # Explore all knight moves
```



```python
        for dx, dy in moves:
            next_x, next_y = curr_x + dx, curr_y + dy

            # Prune search space: only process positions with valid bounds
            if (next_x, next_y) not in visited and next_x >= -1 and next_y >= -1:
                visited.add((next_x, next_y))
                queue.append((next_x, next_y, move_count + 1))

    return -1 # Unreachable because a solution always exists

# Example test
#print(minKnightMoves(5, 5))
```

#1730 MEDIUM

Shortest Path to Get Food

LeetDoocs • SimplyLeet • WalkCC • LeetCode • Editorial

Array • BFS • Matrix

Lists: Grind 169

Problem:

You are starving and you want to eat food as quickly as possible. You want to find the shortest path to arrive at any food cell.

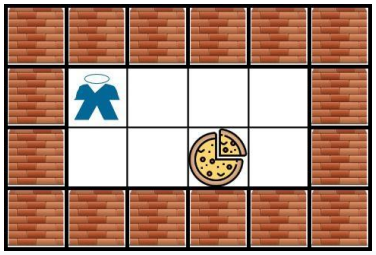
You are given an m x n character matrix, grid, of these different types of cells:

- '*' is your location. There is exactly one '*' cell.
- '#' is a food cell. There may be multiple food cells.
- 'O' is free space, and you can travel through these cells.
- 'X' is an obstacle, and you cannot travel through these cells.

You can travel to any adjacent cell north, east, south, or west of your current location if there is not an obstacle.

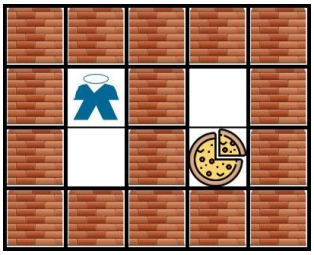
Return the length of the shortest path for you to reach any food cell. If there is no path for you to reach food, return −1.

Example 1:



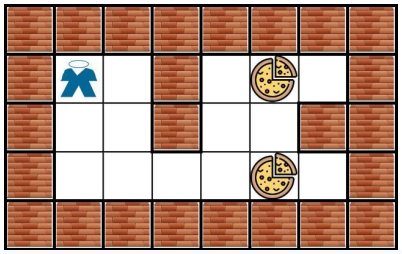
```
Input: grid = [["X","X","X","X","X","X"],["X","*","O","O","O","X"],["X","O","O","#","O","X"],
["X","X","X","X","X","X"]]
Output: 3
Explanation: It takes 3 steps to reach the food.
```

Example 2:




```
Input: grid =
[["X","X","X","X","X"],["X","*","X","O","X"],["X","O","X","#","X"],["X","X","X","X","X"]]
Output: -1
Explanation: It is not possible to reach the food.
```

Example 3:



```
Input: grid = [["X","X","X","X","X","X","X","X"],["X","*","O","X","O","#","O","X"],["X","O","
O","X","O","O","O","X","X"],["X","O","O","O","O","#","O","X"],["X","X","X","X","X","X","X","X"]]
Output: 6
Explanation: There can be multiple food cells. It only takes 6 steps to reach the bottom
food.
```

Example 4:

```
Input: grid = [["X","X","X","X","X","X","X","X"],["X","*","O","X","O","#","O","X"],["X","O","
O","X","O","O","O","X","X"],["X","O","O","O","O","#","O","X"],["O","O","O","O","O","O","O","O"]]
Output: 5
```

Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 200
- grid[row][col] is '*', 'X', 'O', or '#'.
- The grid contains exactly one '*'.

>> Brute Force [BFS] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def getFood(self, grid):
        # Brute Force - BFS layer by layer
        from collections import deque
        queue = deque([0])
        visited = {0}
        steps = 0
        while queue:
            for _ in range(len(queue)):
                node = queue.popleft()
                for nb in graph[node]:
                    if nb not in visited:
                        visited.add(nb); queue.append(nb)
            steps += 1
        return steps
```

Python

Python

```python
from collections import deque

def getFood(grid):
    # number of rows and columns
    rows = len(grid)
    cols = len(grid[0])
    # finding the start position
    start = None
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == '*':
                start = (r, c)
                break
        if start is not None:
            break

    # initialize queue for BFS, with (row, col, steps)
    queue = deque([(start[0], start[1], 0)])
    # mark visited cells to avoid reprocessing
    visited = [[False] * cols for _ in range(rows)]
    visited[start[0]][start[1]] = True

    # possible moves: up, right, down, left
    directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]
```



```python
    # begin BFS
    while queue:
        row, col, steps = queue.popleft()
        # if food is found, return current number of steps
        if grid[row][col] == '#':
            return steps
        # check all adjacent cells
        for dr, dc in directions:
            newRow, newCol = row + dr, col + dc
            # check boundaries, obstacles, and visited status
            if 0 <= newRow < rows and 0 <= newCol < cols and not visited[newRow][newCol] and
grid[newRow][newCol] != 'X':
                visited[newRow][newCol] = True
                queue.append((newRow, newCol, steps + 1))

    # if no food is reachable return -1
    return -1

# Example usage:
grid = [
  ["X","X","X","X","X","X"],
  ["X","*","O","O","O","X"],
  ["X","O","O","#","O","X"],
  ["X","X","X","X","X","X"]
]
print(getFood(grid)) # Expected output: 3
```

Python

```python
class Solution:
    def getFood(self, grid: list[list[str]]) -> int:
        DIRS = ((0, 1), (1, 0), (0, -1), (-1, 0))
        m = len(grid)
        n = len(grid[0])
        ans = 0
        q = collections.deque([self._getStartLocation(grid)])

        while q:
          for _ in range(len(q)):
            i, j = q.popleft()
            for dx, dy in DIRS:
              x = i + dx
              y = j + dy
              if x < 0 or x == m or y < 0 or y == n:
                continue
             if grid[x][y] == 'X':
               continue
             if grid[x][y] == '#':
               return ans + 1
             q.append((x, y))
             grid[x][y] = 'X' # Mark as visited.
          ans += 1

        return -1
```



```python
    def _getStartLocation(self, grid: list[list[str]]) -> tuple[int, int]:
      for i, row in enumerate(grid):
        for j, cell in enumerate(row):
          if cell == '*':
            return (i, j)
```

Python

```python
class Solution:
    def getFood(self, grid: List[List[str]]) -> int:
        m, n = len(grid), len(grid[0])
        i, j = next((i, j) for i in range(m) for j in range(n) if grid[i][j] == '*')
        q = deque([(i, j)])
        dirs = (-1, 0, 1, 0, -1)
        ans = 0
        while q:
            ans += 1
            for _ in range(len(q)):
                i, j = q.popleft()
                for a, b in pairwise(dirs):
                    x, y = i + a, j + b
                    if 0 <= x < m and 0 <= y < n:
                        if grid[x][y] == '#':
                            return ans
                        if grid[x][y] == 'O':
                            grid[x][y] = 'X'
                            q.append((x, y))
        return -1
```

Python

```python
class Solution:
    def getFood(self, grid: List[List[str]]) -> int:
        m, n = len(grid), len(grid[0])
        i, j = next((i, j) for i in range(m) for j in range(n) if grid[i][j] == '*')
        q = deque([(i, j)])
        dirs = (-1, 0, 1, 0, -1)
        ans = 0
        while q:
            ans += 1
            for _ in range(len(q)):
                i, j = q.popleft()
                for a, b in pairwise(dirs):
                    x, y = i + a, j + b
                    if 0 <= x < m and 0 <= y < n:
                        if grid[x][y] == '#':
                            return ans
                        if grid[x][y] == 'O':
                            grid[x][y] = 'X'
                            q.append((x, y))
        return -1
```

[*] BFS — Pattern Solution (matched from community answers)

Python


```python
from collections import deque

def getFood(grid):
    # number of rows and columns
    rows = len(grid)
    cols = len(grid[0])
    # finding the start position
    start = None
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == '*':
                start = (r, c)
                break
        if start is not None:
            break

    # initialize queue for BFS, with (row, col, steps)
    queue = deque([(start[0], start[1], 0)])
    # mark visited cells to avoid reprocessing
    visited = [[False] * cols for _ in range(rows)]
    visited[start[0]][start[1]] = True

    # possible moves: up, right, down, left
    directions = [(-1, 0), (0, 1), (1, 0), (0, -1)]

    # begin BFS
    while queue:
        row, col, steps = queue.popleft()
        # if food is found, return current number of steps
        if grid[row][col] == '#':
            return steps
        # check all adjacent cells
        for dr, dc in directions:
            newRow, newCol = row + dr, col + dc
            # check boundaries, obstacles, and visited status
            if 0 <= newRow < rows and 0 <= newCol < cols and not visited[newRow][newCol] and
grid[newRow][newCol] != 'X':
                visited[newRow][newCol] = True
                queue.append((newRow, newCol, steps + 1))

    # if no food is reachable return -1
    return -1

# Example usage:
grid = [
  ["X","X","X","X","X","X"],
  ["X","*","O","O","O","X"],
  ["X","O","O","#","O","X"],
  ["X","X","X","X","X","X"]
]
print(getFood(grid)) # Expected output: 3
```

#127 HARD

Word Ladder

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · BFS

Lists: Grind 169

Problem:

A transformation sequence from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

- Every adjacent pair of words differs by a single letter.
- Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
- sk == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return the number of words in the shortest transformation sequence from beginWord to endWord, or 0 if no such sequence exists.

Example 1:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]
Output: 5
Explanation: One shortest transformation sequence is "hit" -> "hot" -> "dot" -> "dog" ->
cog", which is 5 words long.
```

Example 2:

```
Input: beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log"]
Output: 0
Explanation: The endWord "cog" is not in wordList, therefore there is no valid transformation
sequence.
```

Constraints:

- 1 <= beginWord.length <= 10
- endWord.length == beginWord.length
- 1 <= wordList.length <= 5000
- wordList[i].length == beginWord.length
- beginWord, endWord, and wordList[i] consist of lowercase English letters.
- beginWord != endWord
- All the words in wordList are unique.

>> Brute Force [BFS] Interview Prep

Python


```
# Brute Force Approach
class Solution:
    def ladderLength(self, beginWord, endWord, wordList):
        # Brute Force O(n^2·L) — BFS; try all word pairs each level (no pattern precomp)
        from collections import deque
        word_set = set(wordList)
        if endWord not in word_set: return 0
        queue = deque([(beginWord, 1)])
        visited = {beginWord}
        while queue:
            word, steps = queue.popleft()
            for candidate in list(word_set):
                # Check one-letter difference by comparing character by character
                if sum(a!=b for a,b in zip(word,candidate))==1 and len(word)==len(candidate):
                    if candidate == endWord: return steps+1
                    if candidate not in visited:
                        visited.add(candidate)
                        queue.append((candidate, steps+1))
        return 0
```

Python

Python

```
# Python solution using BFS

from collections import defaultdict, deque

def ladderLength(beginWord, endWord, wordList):
    # Create a set for fast lookup
    wordSet = set(wordList)
    if endWord not in wordSet:
        return 0

    # Preprocessing: build a dictionary of intermediate states to words
    L = len(beginWord)
    all_combo_dict = defaultdict(list)
    for word in wordSet:
        for i in range(L):
            # Key with a wildcard at the i-th position
            intermediate = word[:i] + '*' + word[i+1:]
            all_combo_dict[intermediate].append(word)

    # Initialize BFS
    queue = deque([(beginWord, 1)])
    visited = set([beginWord])

    while queue:
        current_word, level = queue.popleft()
```



```
        for i in range(L):
            intermediate = current_word[:i] + '*' + current_word[i+1:]
            # Check all words sharing the same intermediate state
            for word in all_combo_dict[intermediate]:
                if word == endWord:
                    return level + 1
                if word not in visited:
                    visited.add(word)
                    queue.append((word, level + 1))
            # Once processed, clear the intermediate state list to save processing time
            all_combo_dict[intermediate] = []
    return 0

# Example usage:
# print(ladderLength("hit", "cog", ["hot","dot","dog","lot","log","cog"]))
```

Python

```
class Solution:
   def ladderLength(
      self,
      beginWord: str,
      endWord: str,
      wordList: list[str],
   ) -> int:
      wordSet = set(wordList)
      if endWord not in wordSet:
        return 0

      q = collections.deque([beginWord])

      step = 1
      while q:
        for _ in range(len(q)):
          wordList = list(q.popleft())
          for i, cache in enumerate(wordList):
            for c in string.ascii_lowercase:
              wordList[i] = c
              word = ''.join(wordList)
              if word == endWord:
                return step + 1
              if word in wordSet:
                q.append(word)
```

```
              wordSet.remove(word)
            wordList[i] = cache
        step += 1

      return 0
```

Python


```
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
        words = set(wordList)
        q = deque([beginWord])
        ans = 1
        while q:
            ans += 1
            for _ in range(len(q)):
                s = q.popleft()
                s = list(s)
                for i in range(len(s)):
                    ch = s[i]
                    for j in range(26):
                        s[i] = chr(ord('a') + j)
                        t = ''.join(s)
                        if t not in words:
                            continue
                        if t == endWord:
                            return ans
                        q.append(t)
                        words.remove(t)
                    s[i] = ch
        return 0
```

Python

```
# Similar BFS
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
        words = set(wordList)
        q = deque([beginWord])
        ans = 1
        while q:
            ans += 1
            for _ in range(len(q)):
                s = q.popleft()
                s = list(s) # must convert to list, cannot directly update s[i]='a'
                for i in range(len(s)):
                    ch = s[i]
                    for j in range(26):
                        # will re-generate s itself
                        # but s not in words-set, s removed after added to words-set
                        s[i] = chr(ord('a') + j)
                        t = ''.join(s)
                        if t not in words:
                            continue
                        if t == endWord:
                            return ans
                        q.append(t)
                        words.remove(t) # equivalent to set t as visited
                    s[i] = ch # restore
```



```
        return 0


########
'''
[*] BFS — BFS [illegible]

1. [illegible] q1, q2 [illegible] -> [illegible] -> [illegible]
2. [illegible] m1, m2 [illegible]
3. [illegible]
4. [illegible]
'''
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
        def extend(m1, m2, q):
            for _ in range(len(q)):
                s = q.popleft() # so, every time, starting from start-word... and later if t in m1 will
skip repeated part...
                step = m1[s]
                s = list(s) # s = "abc", list(s) ==> ['a', 'b', 'c']
                for i in range(len(s)):
                    ch = s[i]
                    for j in range(26):
                        s[i] = chr(ord('a') + j)
                        t = ''.join(s)
```

```
                        if t in m1 or t not in words:
                            continue
                        if t in m2:
                            return step + 1 + m2[t]
                        m1[t] = step + 1
                        q.append(t)
                    s[i] = ch
            return -1

        words = set(wordList)
        if endWord not in words:
            return 0
        q1, q2 = deque([beginWord]), deque([endWord])
        m1, m2 = {beginWord: 0}, {endWord: 0}
        while q1 and q2:
            t = extend(m1, m2, q1) if len(q1) <= len(q2) else extend(m2, m1, q2)
            if t != -1:
                return t + 1
        return 0

########

import string
from collections import deque
```

```
class Solution(object):
    def ladderLength(self, beginWord, endWord, wordList):
        """
        :type beginWord: str
```


[*] BFS — Pattern Solution (matched from community answers)

Python

```
# Similar BFS
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
        words = set(wordList)
        q = deque([beginWord])
        ans = 1
        while q:
            ans += 1
            for _ in range(len(q)):
                s = q.popleft()
                s = list(s) # must convert to list, cannot directly update s[i]='a'
                for i in range(len(s)):
                    ch = s[i]
                    for j in range(26):
                        # will re-generate s itself
                        # but s not in words-set, s removed after added to words-set
                        s[i] = chr(ord('a') + j)
                        t = ''.join(s)
                        if t not in words:
                            continue
                        if t == endWord:
                            return ans
                        q.append(t)
                        words.remove(t) # equivalent to set t as visited
                    s[i] = ch # restore
```

```
        return 0


########
'''
[*] BFS — BFS [illegible]

1. [illegible] q1, q2 [illegible] -> [illegible] -> [illegible]
2. [illegible] m1, m2 [illegible]
3. [illegible]
4. [illegible]
'''
class Solution:
    def ladderLength(self, beginWord: str, endWord: str, wordList: List[str]) -> int:
        def extend(m1, m2, q):
            for _ in range(len(q)):
                s = q.popleft() # so, every time, starting from start-word... and later if t in m1 will
skip repeated part...
                step = m1[s]
                s = list(s) # s = "abc", list(s) ==> ['a', 'b', 'c']
                for i in range(len(s)):
                    ch = s[i]
                    for j in range(26):
                        s[i] = chr(ord('a') + j)
                        t = ''.join(s)
```

```
                    if t in m1 or t not in words:
                        continue
                    if t in m2:
                        return step + 1 + m2[t]
                    m1[t] = step + 1
                    q.append(t)
                s[i] = ch
        return -1

    words = set(wordList)
    if endWord not in words:
        return 0
    q1, q2 = deque([beginWord]), deque([endWord])
    m1, m2 = {beginWord: 0}, {endWord: 0}
    while q1 and q2:
        t = extend(m1, m2, q1) if len(q1) <= len(q2) else extend(m2, m1, q2)
        if t != -1:
            return t + 1
    return 0

########

import string
from collections import deque

class Solution(object):
    def ladderLength(self, beginWord, endWord, wordList):
        """
        :type beginWord: str
```

#317 HARD

Shortest Distance from All Buildings

LeetDocs · SimpleLent · WalkCC · LeetCode · Editorial

Array · BFS · Matrix

Lists: Premium 98

Problem:

You are given an m x n grid grid of values 0, 1, or 2, where:

- each 0 marks an empty land that you can pass by freely,
- each 1 marks a building that you cannot pass through, and
- each 2 marks an obstacle that you cannot pass through.

You want to build a house on an empty land that reaches all buildings in the shortest total travel distance. You can only move up, down, left, and right.

Return the shortest travel distance for such a house. If it is not possible to build such a house according to the above rules, return -1.

The total travel distance is the sum of the distances between the houses of the friends and the meeting point.

Example 1:

1	0	2	0	1
0	0	0	0	0
0	0	1	0	0

```
Input: grid = [[1,0,2,0,1],[0,0,0,0,0],[0,0,1,0,0]]
Output: 7
Explanation: Given three buildings at (0,0), (0,4), (2,2), and an obstacle at (0,2).
The point (1,2) is an ideal empty land to build a house, as the total travel distance of
3+3+1=7 is minimal.
So return 7.
```

Example 2:

```
Input: grid = [[1,0]]
Output: 1
```

Example 3:

```
Input: grid = [[1]]
Output: -1
```

Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 50
- grid[i][j] is either 0, 1, or 2.
- There will be at least one building in the grid.

>> Brute Force [BFS] Interview Prep

Python


```
# Brute Force Approach
class Solution:
    def shortestDistance(self, grid):
        # Brute Force - BFS layer by layer
        from collections import deque
        queue = deque([0])
        visited = {0}
        steps = 0
        while queue:
            for _ in range(len(queue)):
                node = queue.popleft()
                for nb in graph[node]:
                    if nb not in visited:
                        visited.add(nb); queue.append(nb)
            steps += 1
        return steps
```

Python

Python

```
from collections import deque

def shortestDistance(grid):
    if not grid or not grid[0]:
        return -1

    m, n = len(grid), len(grid[0])
    total_buildings = 0
    total_distance = [[0] * n for _ in range(m)]
    reachable_count = [[0] * n for _ in range(m)]

    def bfs(start_i, start_j):
        visited = [[False] * n for _ in range(m)]
        q = deque([(start_i, start_j, 0)])
        visited[start_i][start_j] = True

        while q:
            i, j, dist = q.popleft()
            for x, y in [(i-1, j), (i+1, j), (i, j-1), (i, j+1)]:
                if 0 <= x < m and 0 <= y < n and not visited[x][y] and grid[x][y] == 0:
                    visited[x][y] = True
                    total_distance[x][y] += dist + 1
                    reachable_count[x][y] += 1
                    q.append((x, y, dist+1))
```



```
    for i in range(m):
        for j in range(n):
            if grid[i][j] == 1:
                total_buildings += 1
                bfs(i, j)

    min_distance = float('inf')
    for i in range(m):
        for j in range(n):
            if grid[i][j] == 0 and reachable_count[i][j] == total_buildings:
                min_distance = min(min_distance, total_distance[i][j])

    return min_distance if min_distance != float('inf') else -1

# Example test
print(shortestDistance([[1,0,2,0,1],[0,0,0,0,0],[0,0,1,0,0]]))
```

Python

```
class Solution:
    def shortestDistance(self, grid: list[list[int]]) -> int:
        DIRS = ((0, 1), (1, 0), (0, -1), (-1, 0))
        m = len(grid)
        n = len(grid[0])
        nBuildings = sum(a == 1 for row in grid for a in row)
        ans = math.inf
        # dist[i][j] -> the total distance of grid[i][j] (0) to reach all the
        # buildings (1)
        dist = [[0] * n for _ in range(m)]
        # reachCount[i][j] -> the number of buildings (1) grid[i][j] (0) can reach
        reachCount = [[0] * n for _ in range(m)]

        def bfs(row: int, col: int) -> bool:
          q = collections.deque([(row, col)])
          seen = {(row, col)}
          seenBuildings = 1

          step = 1
          while q:
            for _ in range(len(q)):
              i, j = q.popleft()
              for dx, dy in DIRS:
                x = i + dx
                y = j + dy
```



```
            if x < 0 or x >= m or y < 0 or y >= n:
              continue
            if (x, y) in seen:
              continue
            seen.add((x, y))
            if not grid[x][y]:
              dist[x][y] += step
              reachCount[x][y] += 1
              q.append((x, y))
            elif grid[x][y] == 1:
              seenBuildings += 1
        step += 1

      # True if all the buildings (1) are connected
      return seenBuildings == nBuildings

    for i in range(m):
      for j in range(n):
        if grid[i][j] == 1: # BFS from this building.
          if not bfs(i, j):
            return -1

    for i in range(m):
      for j in range(n):
        if reachCount[i][j] == nBuildings:

          ans = min(ans, dist[i][j])

    return -1 if ans == math.inf else ans
```

Python

```
class Solution:
    def shortestDistance(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        q = deque()
        total = 0 # total building count
        # cnt: how many buildings can reach (x,y)
        # if there is a column all blockers, then (x,y) cannot reach every building
        cnt = [[0] * n for _ in range(m)]
        dist = [[0] * n for _ in range(m)]
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 1:
                    total += 1
                    q.append((i, j))
                    d = 0
                    vis = set() # reset for every free-land
                    while q:
                        d += 1
                        for _ in range(len(q)):
                            r, c = q.popleft()
                            for a, b in [(0, 1), (0, -1), (1, 0), (-1, 0)]:
                                x, y = r + a, c + b
                                if (
                                    0 <= x < m
                                    and 0 <= y < n
```



```
                                    and grid[x][y] == 0
                                    and (x, y) not in vis
                                ):
                                    cnt[x][y] += 1
                                    dist[x][y] += d
                                    q.append((x, y))
                                    vis.add((x, y))

        ans = inf
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 0 and cnt[i][j] == total:
                    ans = min(ans, dist[i][j])
        return -1 if ans == inf else ans
```

[*] BFS — Pattern Solution (matched from community answers)

Python

```
from collections import deque

def shortestDistance(grid):
    if not grid or not grid[0]:
        return -1

    m, n = len(grid), len(grid[0])
    total_buildings = 0
    total_distance = [[0] * n for _ in range(m)]
    reachable_count = [[0] * n for _ in range(m)]

    def bfs(start_i, start_j):
        visited = [[False] * n for _ in range(m)]
        q = deque([(start_i, start_j, 0)])
        visited[start_i][start_j] = True

        while q:
            i, j, dist = q.popleft()
            for x, y in [(i-1, j), (i+1, j), (i, j-1), (i, j+1)]:
                if 0 <= x < m and 0 <= y < n and not visited[x][y] and grid[x][y] == 0:
                    visited[x][y] = True
                    total_distance[x][y] += dist + 1
                    reachable_count[x][y] += 1
                    q.append((x, y, dist+1))
```

```
    for i in range(n):
        for j in range(n):
            if grid[i][j] == 1:
                total_buildings += 1
                bfs(i, j)

    min_distance = float('inf')
    for i in range(n):
        for j in range(n):
            if grid[i][j] == 0 and reachable_count[i][j] == total_buildings:
                min_distance = min(min_distance, total_distance[i][j])

    return min_distance if min_distance != float('inf') else -1

# Example test:
print(shortestDistance([[1,0,2,0,1],[0,0,0,0,0],[0,0,1,0,0]]))
```

#773 HARD

Sliding Puzzle

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · BFS · Matrix

Lists: Denny Zhang

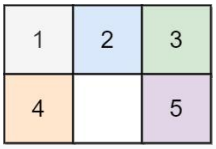
Problem:

On an 2 x 3 board, there are five tiles labeled from 1 to 5, and an empty square represented by 0. A move consists of choosing 0 and a 4-directionally adjacent number and swapping it.

The state of the board is solved if and only if the board is [[1,2,3],[4,5,0]].

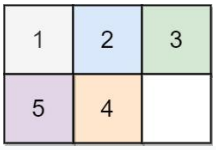
Given the puzzle board board, return the least number of moves required so that the state of the board is solved. If it is impossible for the state of the board to be solved, return -1.

Example 1:



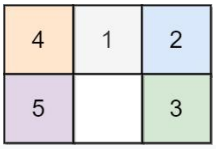

```
Input: board = [[1,2,3],[4,0,5]]
Output: 1
Explanation: Swap the 0 and the 5 in one move.
```

Example 2:




```
Input: board = [[1,2,3],[5,4,0]]
Output: -1
Explanation: No number of moves will make the board solved.
```

Example 3:




```
Input: board = [[4,1,2],[5,0,3]]
Output: 5
Explanation: 5 is the smallest number of moves that solves the board.
An example path:
After move 0: [[4,1,2],[5,0,3]]
After move 1: [[4,1,2],[0,5,3]]
After move 2: [[0,1,2],[4,5,3]]
After move 3: [[1,0,2],[4,5,3]]
After move 4: [[1,2,0],[4,5,3]]
After move 5: [[1,2,3],[4,5,0]]
```

Constraints:

- board.length == 2
- board[i].length == 3
- 0 <= board[i][j] <= 5
- Each value board[i][j] is unique.

>> Brute Force [BFS] Interview Prep

Python


```
# Brute Force Approach
class Solution:
    def slidingPuzzle(self, board):
        # Brute Force - BFS layer by layer
        from collections import deque
        queue = deque([0])
        visited = {0}
        steps = 0
        while queue:
            for _ in range(len(queue)):
                node = queue.popleft()
                for nb in graph[node]:
                    if nb not in visited:
                        visited.add(nb); queue.append(nb)
            steps += 1
        return steps
```

Python

Python

```
import collections

def slidingPuzzle(board):
    # Flatten board into a string representation.
    start = ''.join(str(num) for row in board for num in row)
    goal = "123450"
    # Mapping of indices to their neighbors in a 2x3 board.
    neighbors = {0: [1, 3], 1: [0,2,4], 2: [1,5],
                 3: [0,4], 4: [1,3,5], 5: [2,4]}

    # Initialize BFS with the starting state and move count 0.
    q = collections.deque([(start, 0)])
    visited = set([start])

    # Process states in the queue.
    while q:
        state, moves = q.popleft()
        if state == goal:
            return moves
        zero_index = state.index('0')
        # Try all valid moves by swapping 0 with its neighbours.
        for neighbor in neighbors[zero_index]:
            new_state = list(state)
            new_state[zero_index], new_state[neighbor] = new_state[neighbor], new_state[zero_index]
            new_state_str = ''.join(new_state)

            if new_state_str not in visited:
                visited.add(new_state_str)
                q.append((new_state_str, moves + 1))
    return -1

# Example usage:
print(slidingPuzzle([[1,2,3],[4,0,5]]))
```

Python


```
class Solution:
    def slidingPuzzle(self, board: List[List[int]]) -> int:
        m, n = 2, 3
        seq = []
        start, end = '', '123450'
        for i in range(m):
            for j in range(n):
                if board[i][j] != 0:
                    seq.append(board[i][j])
                start += str(board[i][j])

        def check(seq):
            n = len(seq)
            cnt = sum(seq[i] > seq[j] for i in range(n) for j in range(i, n))
            return cnt % 2 == 0

        def f(s):
            ans = 0
            for i in range(m + n):
                if s[i] != '0':
                    num = ord(s[i]) - ord('1')
                    ans += abs(i // n - num // n) + abs(i % n - num % n)
            return ans

        if not check(seq):

            return -1
        q = [(f(start), start)]
        dist = {start: 0}
        while q:
            _, state = heappop(q)
            if state == end:
                return dist[state]
            p1 = state.index('0')
            i, j = p1 // n, p1 % n
            s = list(state)
            for a, b in [(0, -1), (0, 1), (1, 0), (-1, 0)]:
                x, y = i + a, j + b
                if 0 <= x < m and 0 <= y < n:
                    p2 = x * n + y
                    s[p1], s[p2] = s[p2], s[p1]
                    next = ''.join(s)
                    s[p1], s[p2] = s[p2], s[p1]
                    if next not in dist or dist[next] > dist[state] + 1:
                        dist[next] = dist[state] + 1
                        heappush(q, (dist[next] + f(next), next))
        return -1
```

[*] BFS — Pattern Solution (matched from community answers)

Python


```
import collections

def slidingPuzzle(board):
    # Flatten board into a string representation.
    start = ''.join(str(num) for row in board for num in row)
    goal = "123450"
    # Mapping of indices to their neighbors in a 2x3 board.
    neighbors = {0: [1, 3], 1: [0,2,4], 2: [1,5],
                 3: [0,4], 4: [1,3,5], 5: [2,4]}

    # Initialize BFS with the starting state and move count 0.
    q = collections.deque([(start, 0)])
    visited = set([start])

    # Process states in the queue.
    while q:
        state, moves = q.popleft()
        if state == goal:
            return moves
        zero_index = state.index('0')
        # Try all valid moves by swapping 0 with its neighbours.
        for neighbor in neighbors[zero_index]:
            new_state = list(state)
            new_state[zero_index], new_state[neighbor] = new_state[neighbor], new_state[zero_index]
            new_state_str = ''.join(new_state)

            if new_state_str not in visited:
                visited.add(new_state_str)
                q.append((new_state_str, moves + 1))
    return -1

# Example usage:
print(slidingPuzzle([[1,2,3],[4,0,5]]))
```


#815 HARD

Bus Routes

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · BFS

Lists: Grind 169

Problem:

You are given an array routes representing bus routes where routes[i] is a bus route that the ith bus repeats forever.

- For example, if routes[0] = [1, 5, 7], this means that the 0th bus travels in the sequence 1 -> 5 -> 7 -> 1 -> 5 -> 7 -> 1 -> ... forever.

You will start at the bus stop source (You are not on any bus initially), and you want to go to the bus stop target. You can travel between bus stops by buses only.

Return the least number of buses you must take to travel from source to target. Return -1 if it is not possible.

Example 1:


```
Input: routes = [[1,2,7],[3,6,7]], source = 1, target = 6
Output: 2
Explanation: The best strategy is take the first bus to the bus stop 7, then take the second
bus to the bus stop 6.
```

Example 2:

```
Input: routes = [[7,12],[4,5,15],[6],[15,19],[9,12,13]], source = 15, target = 12
Output: -1
```

Constraints:

- 1 <= routes.length <= 500.
- 1 <= routes[i].length <= 105
- All the values of routes[i] are unique.
- sum(routes[i].length) <= 105
- 0 <= routes[i][j] < 106
- 0 <= source, target < 106

>> Brute Force [BFS] Interview Prep

Python


```
# Brute Force Approach
class Solution:
    def numBusesToDestination(self, routes, source, target):
        # Brute Force - BFS layer by layer
        from collections import deque
        queue = deque([0])
        visited = {0}
        steps = 0
        while queue:
            for _ in range(len(queue)):
                node = queue.popleft()
                for nb in graph[node]:
                    if nb not in visited:
                        visited.add(nb); queue.append(nb)
            steps += 1
        return steps
```


Python

Python

```python
from collections import defaultdict, deque

def numBusesToDestination(routes, source, target):
    # If source equals target, no buses are needed.
    if source == target:
        return 0

    # Build mapping: bus stop -> list of bus indices
    stop_to_buses = defaultdict(list)
    for bus_index, stops in enumerate(routes):
        for stop in stops:
            stop_to_buses[stop].append(bus_index)

    visited_buses = set()
    visited_stops = set([source])
    queue = deque()

    # Initialize queue with all buses that include the source stop.
    for bus in stop_to_buses[source]:
        visited_buses.add(bus)
        queue.append((bus, 1))

    # Perform BFS over bus routes.
    while queue:
        current_bus, buses_taken = queue.popleft()

        for stop in routes[current_bus]:
            if stop == target:
                return buses_taken
            if stop not in visited_stops:
                visited_stops.add(stop)
                for neighbor_bus in stop_to_buses[stop]:
                    if neighbor_bus not in visited_buses:
                        visited_buses.add(neighbor_bus)
                        queue.append((neighbor_bus, buses_taken + 1))
    return -1

# Example usage:
# routes = [[1,2,7],[3,6,7]]
# print(numBusesToDestination(routes, 1, 6))
```

Python


```python
class Solution:
  def numBusesToDestination(
      self,
      routes: list[list[int]],
      source: int,
      target: int,
  ) -> int:
    if source == target:
      return 0

    graph = collections.defaultdict(list)
    usedBuses = set()

    for i in range(len(routes)):
      for route in routes[i]:
        graph[route].append(i)

    q = collections.deque([source])

    step = 1
    while q:
      for _ in range(len(q)):
        for bus in graph[q.popleft()]:
          if bus in usedBuses:
            continue

          usedBuses.add(bus)
          for nextRoute in routes[bus]:
            if nextRoute == target:
              return step
            q.append(nextRoute)
      step += 1

    return -1
```

Python


```python
class Solution:
    def numBusesToDestination(
        self, routes: List[List[int]], source: int, target: int
    ) -> int:
        if source == target:
            return 0

        q = defaultdict(list)
        for i, route in enumerate(routes):
            for stop in route:
                q[stop].append(i)
        if source not in q or target not in q:
            return -1
        q = [(source, 0)]
        vis_bus = set()
        vis_stop = {source}
        for stop, bus_count in q:
            if stop == target:
                return bus_count
            for bus in q[stop]:
                if bus not in vis_bus:
                    vis_bus.add(bus)
                    for next_stop in routes[bus]:
                        if next_stop not in vis_stop:
                            vis_stop.add(next_stop)
                            q.append((next_stop, bus_count + 1))

        return -1
```

Python

```python
class Solution:
    def numBusesToDestination(
        self, routes: List[List[int]], source: int, target: int
    ) -> int:
        if source == target:
            return 0

        s = [set(r) for r in routes]
        d = defaultdict(list)
        for i, r in enumerate(routes):
            for v in r:
                d[v].append(i)

        q = defaultdict(list)
        for ids in d.values():
            m = len(ids)
            for i in range(m):
                for j in range(i + 1, m):
                    a, b = ids[i], ids[j]
                    q[a].append(b)
                    q[b].append(a)
        q = deque([d[source]])
        ans = 1
        vis = set(d[source])
        while q:
```



```python
            for _ in range(len(q)):
                i = q.popleft()
                if target in s[i]:
                    return ans
                for j in q[i]:
                    if j not in vis:
                        vis.add(j)
                        q.append(j)
            ans += 1
        return -1
```

[*] BFS — Pattern Solution (matched from community answers)

Python

```python
from collections import defaultdict, deque

def numBusesToDestination(routes, source, target):
    # If source equals target, no buses are needed.
    if source == target:
        return 0

    # Build mapping: bus stop -> list of bus indices
    stop_to_buses = defaultdict(list)
    for bus_index, stops in enumerate(routes):
        for stop in stops:
            stop_to_buses[stop].append(bus_index)

    visited_buses = set()
    visited_stops = set([source])
    queue = deque()

    # Initialize queue with all buses that include the source stop.
    for bus in stop_to_buses[source]:
        visited_buses.add(bus)
        queue.append((bus, 1))

    # Perform BFS over bus routes.
    while queue:
        current_bus, buses_taken = queue.popleft()

        for stop in routes[current_bus]:
            if stop == target:
                return buses_taken
            if stop not in visited_stops:
                visited_stops.add(stop)
                for neighbor_bus in stop_to_buses[stop]:
                    if neighbor_bus not in visited_buses:
                        visited_buses.add(neighbor_bus)
                        queue.append((neighbor_bus, buses_taken + 1))
    return -1

# Example usage:
# routes = [[1,2,7],[3,6,7]]
# print(numBusesToDestination(routes, 1, 6))
```


Quick Review — BFS 10 questions · insights · complexity · solution

#286 Walls and Gates [Medium]

Key Insights

- Use a multi-source Breadth-First Search (BFS) starting from all the gates, as this allows spreading the shortest distance to each empty room.
- Instead of starting from every empty room separately, starting from all gates simultaneously helps update all distances in one pass.
- Only update a room if the new calculated distance is smaller than its current value to prevent unnecessary processing.
- The problem is essentially a multi-source shortest path problem on a grid.

Space & Time

Time Complexity: O(m * n) where m and n are the dimensions of the grid, as each cell is processed at most once.

Space Complexity: O(m * n) in the worst-case scenario used by the BFS queue.

Solution

The solution uses a multi-source BFS algorithm by initially enqueuing all the gate positions (cells with value 0). Then, for each cell obtained from the queue, check its four possible neighbors (up, down, left, right). If a neighbor is an empty room (INF), update its distance to be the current cell's distance plus one, and then enqueue the neighbor to process further. This ensures that each empty room gets the minimum distance from any gate.

#322 Coin Change [Medium]

Key Insights

- Use a dynamic programming approach to build up a solution from smaller subproblems.
- Define dp[x] as the minimum number of coins required for amount x.
- The recurrence relation is: dp[x] = min(dp[x], dp[x - coin] + 1) for each coin.
- Initialize dp[0] = 0 and set other values to a number larger than the maximum possible coins (amount+1 works as an infinity substitute).
- An alternative approach is through BFS, treating each amount as a node and each coin as an edge, but the DP method is more intuitive here.

Space & Time

Time Complexity: O(amount * n) where n is the number of coins.

Space Complexity: O(amount) for the dp array.

Solution

We use a bottom-up dynamic programming approach. The idea is to use a one-dimensional dp array where each element at index x represents the minimum number of coins required to form the amount x. Starting from dp[0] which is 0, we iterate through all amounts up to the target amount and update dp[x] based on each coin denomination. If after filling the dp array the value dp[amount] remains unchanged (i.e., greater than amount), it means the amount cannot be formed and we return -1.

#542 01 Matrix [Medium]

Key Insights

- Treat all 0s as starting points for a multi-source BFS.
- Update each neighboring cell's distance if a shorter distance is found.
- Utilize a queue to process cells in order of increasing distance.
- Ensure that each cell is processed only once by marking visited cells.

Space & Time

Time Complexity: O(m

n) since each cell is processed once.

Space Complexity: O(m

n) for the queue and the answer matrix.

Solution

We use a multi-source Breadth-First Search (BFS) where every 0 in the matrix is initially added to the BFS queue. Each neighboring cell that is a 1 will have its distance updated to be one plus the distance of the current cell if it hasn't been visited yet (or if a smaller distance is found). This ensures that the first time a 1 is reached, it is reached by the shortest path. Data structures used include a queue (or deque) for BFS and the matrix itself for storing distances.

#994 Rotting Oranges [Medium]

Key Insights

- Use Breadth-First Search (BFS) starting from all initially rotten oranges.
- Process the grid level by level, where each level represents one minute.
- Keep a count of fresh oranges and decrement it as they become rotten.
- If after the BFS there are still fresh oranges, then it is impossible to rot every orange.

Space & Time

Time Complexity: O(m * n) as every cell is processed at most once.

Space Complexity: O(m * n) in the worst case for the queue and auxiliary storage.

Solution

We begin by scanning the grid to locate all rotten oranges and count the fresh ones. These rotten oranges are added to a queue for a BFS. For each minute (BFS level), we process all the current rotten oranges and try to infect their adjacent fresh oranges. If an adjacent cell contains a fresh orange, we mark it rotten, add it to the queue, and decrement our fresh count. We continue this process level by level (minute by minute) until there are no fresh oranges left. If, after processing, there are still fresh oranges remaining, we return -1, indicating that not all oranges can become rotten.

#1197 Minimum Knight Moves [Medium]

Key Insights

- Leverage the symmetry of the chessboard by converting target coordinates to their absolute values.
- Use a Breadth-First Search (BFS) because all moves have equal weight, ensuring the first encounter with the target is the minimal path.
- Prune the search space by only considering positions that are likely beneficial (e.g., positions having coordinates not far less than -1).
- Track visited states to avoid redundant processing.
- Optimize by bounding the search region relative to the target distance.

Space & Time

Time Complexity: O(n) where n is the number of positions processed during the BFS

Space Complexity: O(n) due to the storage requirements for the BFS queue and the visited set

Solution

We solve the problem using a BFS starting from the origin. First, normalize the target by taking the absolute value of x and y. BFS is then applied by exploring all eight possible knight moves at each step. Each move generates a new position that is enqueued if it hasn't been visited and falls within a reasonable bound (e.g., coordinates >= -1). The search terminates as soon as the target position is reached.

#1730 Shortest Path to Get Food [Medium]

Key Insights

- Use Breadth-First Search (BFS) to explore the grid level by level.

- The BFS guarantees that when a food cell ('#') is reached, the current distance is the shortest.
- Maintain a visited structure to avoid re-processing the same cell.
- Consider boundary and obstacle ('X') conditions while traversing.

Space & Time

Time Complexity: O(m * n), where m is the number of rows and n is the number of columns since each cell is processed at most once.

Space Complexity: O(m * n) in the worst case due to the queue and visited matrix.

Solution

We use Breadth-First Search (BFS) starting from the cell marked with '*'. The BFS uses a queue to explore neighbors in the four allowed directions (up, down, left, right). Each level of BFS represents one step in the path. As soon as a cell containing food ('#') is reached, the current step count is returned. We also maintain a visited matrix to ensure that cells are not revisited, which could otherwise result in cycles or redundant processing. This approach ensures that the first time we encounter food, we have found the shortest possible path.

#127 Word Ladder [Hard]

Key Insights

- Use Breadth-First Search (BFS) to explore transformations level by level to guarantee the shortest path.
- Preprocess words by generating intermediate representations (e.g., replacing each letter with a wildcard) to quickly find neighbors.
- Use a visited set to avoid cycles.
- Each transformation counts as a level in the BFS, so the answer is the number of levels traversed.

Space & Time

Time Complexity: O(N * M^2) where N is the number of words and M is the length of each word.

Space Complexity: O(N * M) for the storage of intermediate mappings and the BFS queue.

Solution

We solve the problem using a BFS approach. First, we preprocess the wordList by creating a mapping of all possible intermediate states. For example, for the word "dog" and wildcard position, we generate " og", "d g", and "do*". This helps us find all adjacent words (neighbors) that are only one letter different in constant time. We then initialize a queue with the beginWord and initiate the BFS. For each word dequeued, we explore all valid transformations using the precomputed intermediate mapping. If the endWord is reached during this process, we return the current level (transformation sequence length). Otherwise, we continue the BFS until all possibilities are exhausted. If no valid transformation sequence is found, we return 0.

#317 Shortest Distance from All Buildings [Hard]

Key Insights

- We need to compute the Manhattan distance from each building (cell value 1) to all empty lands (cell value 0) by performing a BFS.
- Maintain two matrices: one for cumulative total distances and another for the count of buildings that reached a particular empty cell.
- Ensure that only those empty cells reachable by every building are considered.
- The ideal cell will have a reach count equal to the total number of buildings.
- If multiple buildings cannot all reach any specific empty cell, then return -1.

Space & Time

Time Complexity: O(k * m * n) where k is the number of buildings and m,n are the grid dimensions.

Space Complexity: O(m * n) due to the auxiliary matrices and visited state tracking used in BFS.

Solution

We first iterate through the grid to locate all buildings while counting them. For each building, we perform a BFS to compute the shortest distance to each reachable empty cell. During the BFS, we update the cumulative distance for

that cell and record that it was reached by one additional building. After processing all the buildings, we scan the grid to find the empty cell that is reachable from all buildings and has the minimum cumulative distance. If no such cell exists, we return -1.

#773 Sliding Puzzle [Hard]

Key Insights

- Represent the board as a string (e.g. "123450") to simplify state comparison and manipulation.
- Use Breadth First Search (BFS) to explore all possible moves since the puzzle has a small fixed state space (6! = 720 possible states).
- Precompute the valid neighbor indices for each board position to efficiently generate new states.
- Track visited states to avoid loops and redundant work.

Space & Time

Time Complexity: O(1) in practice, since there are at most 720 states (6! permutations).

Space Complexity: O(1) for the same reason, as the state space is fixed and limited.

Solution

We solve the problem using a Breadth First Search (BFS) approach. The board configuration is flattened into a string to represent states. A mapping from each index to its possible neighbors (i.e., positions to which 0 can move) is precomputed. BFS is then used to explore every valid move from the initial state while marking states as visited to avoid cycles. When the goal state ("123450") is reached, we return the number of moves taken. If BFS completes without reaching the goal, the puzzle is unsolvable and we return -1.

#815 Bus Routes [Hard]

Key Insights

- Model the problem as a graph where nodes are bus routes.
- Two bus routes are connected if they share at least one common bus stop.
- Use a hash map to quickly look up bus routes available at any bus stop.
- Employ Breadth-First Search (BFS) on the bus routes graph to find the minimum number of bus transfers.
- Mark visited bus routes and bus stops to avoid redundant work.

Space & Time

Time Complexity: O(N * L) in the worst-case scenario, where N is the number of bus routes and L is the average number of stops per route.

Space Complexity: O(N * L) due to storage of the stop-to-buses mapping and BFS auxiliary data structures.

Solution

We begin by constructing a mapping from each bus stop to the list of buses passing through it. This allows quick retrieval of buses available at a given stop. Starting at the source stop, initialize the BFS queue with all buses that can be boarded at that stop. Then, for each bus route in the queue, inspect each stop along that route. If a stop is the target, return the number of buses taken so far. Otherwise, for each unvisited stop, add all adjacent bus routes (buses that pass through this stop) to the BFS queue with an incremented bus count. Continue until the queue is empty, which means the target cannot be reached.

#10 Trie — 12 questions · #10 of 21

#14 EASY

Longest Common Prefix

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Trie

Lists: Grind 169

Problem:

Write a function to find the longest common prefix string amongst an array of strings.

If there is no common prefix, return an empty string "".

Example 1:

```
Input: strs = ["flower","flow","flight"]
Output: "fl"
```

Example 2:

```
Input: strs = ["dog","racecar","car"]
Output: ""
Explanation: There is no common prefix among the input strings.
```

Constraints:

- 1 <= strs.length <= 200
- 0 <= strs[i].length <= 200
- strs[i] consists of only lowercase English letters if it is non-empty.

>> Brute Force [Trie] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def longestCommonPrefix(self, strs):
        # Brute Force O(n·L) - start with strs[0], trim until prefix of all
        if not strs: return ''
        prefix = strs[0]
        for word in strs[1:]:
            while not word.startswith(prefix):
                prefix = prefix[:-1]
                if not prefix: return ''
        return prefix
```

Python

Python


```python
def longestCommonPrefix(strs):
    # If no strings exist, return empty
    if not strs:
        return ""
    # Iterate over each index of the first string
    for i in range(len(strs[0])):
        current_char = strs[0][i]
        # Compare the current character with the corresponding one in the rest of the strings
        for s in strs[1:]:
            # If index exceeds length or characters differ, return the prefix found so far
            if i >= len(s) or s[i] != current_char:
                return strs[0][:i]
    # If no mismatch is found, return the complete first string
    return strs[0]

# Example usage:
print(longestCommonPrefix(["flower", "flow", "flight"]))
```

Python

```python
class Solution:
    def longestCommonPrefix(self, strs: list[str]) -> str:
        if not strs:
            return ''

        for i in range(len(strs[0])):
            for j in range(1, len(strs)):
                if i >= len(strs[j]) or strs[j][i] != strs[0][i]:
                    return strs[0][:i]

        return strs[0]
```

Python

```python
class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        for i in range(len(strs[0])):
            for s in strs[1:]:
                if len(s) <= i or s[i] != strs[0][i]:
                    return s[:i]
        return strs[0]
```

Python

```python
class Solution:
    def longestCommonPrefix(self, strs: List[str]) -> str:
        for i in range(len(strs[0])):
            for s in strs[1:]:
                if len(s) <= i or s[i] != strs[0][i]:
                    return s[:i]
        return strs[0]
```

[*] Trie — Pattern Solution (stored optimal)

Python


```python
def longestCommonPrefix(strs):
    # If no strings exist, return empty
    if not strs:
        return ""
    # Iterate over each index of the first string
    for i in range(len(strs[0])):
        current_char = strs[0][i]
        # Compare the current character with the corresponding one in the rest of the strings
        for s in strs[1:]:
            # If index exceeds length or characters differ, return the prefix found so far
            if i >= len(s) or s[i] != current_char:
                return strs[0][:i]
    # If no mismatch is found, return the complete first string
    return strs[0]

# Example usage:
print(longestCommonPrefix(["flower", "flow", "flight"]))
```

#139 MEDIUM

Word Break

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · String · Dynamic Programming · Trie · Memoization

Lists: Grind 169

Problem:

Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

```
Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".
```

Example 2:

```
Input: s = "applepenapple", wordDict = ["apple","pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
Note that you are allowed to reuse a dictionary word.
```

Example 3:

```
Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
Output: false
```

Constraints:

- 1 <= s.length <= 300
- 1 <= wordDict.length <= 1000
- 1 <= wordDict[i].length <= 20
- s and wordDict[i] consist of only lowercase English letters.
- All the strings of wordDict are unique.

>> Brute Force [Trie] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def wordBreak(self, s, wordDict):
        # Brute Force O(2^n) - try all partition points recursively
        word_set = set(wordDict)
        def can_break(start):
            if start == len(s): return True
            for end in range(start + 1, len(s) + 1):
                if s[start:end] in word_set and can_break(end):
                    return True
            return False
        return can_break(0)
```

Python

Python

```python
# Python solution for the Word Break problem

def wordBreak(s, wordDict):
    # Convert the list to a set for O(1) look-ups
    wordSet = set(wordDict)

    # dp[i] is True if s[0:i] can be segmented into words from the wordDict
    n = len(s)
    dp = [False] * (n + 1)

    # Base case: empty string is always segmentable
    dp[0] = True

    # Iterate through the string positions
    for i in range(1, n + 1):
        # Check every possible partition s[0:j] and s[j:i]
        for j in range(i):
            # If s[0:j] can be segmented and s[j:i] is in the dictionary
            if dp[j] and s[j:i] in wordSet:
                dp[i] = True # s[0:i] can be segmented
                break # No need to check further partitions
    return dp[n]

# Example usage:
s = "leetcode"

wordDict = ["leet", "code"]
print(wordBreak(s, wordDict)) # Output: True
```

Python

```
class Solution:
    def wordBreak(self, s: str, wordDict: list[str]) -> bool:
        wordSet = set(wordDict)

        @functools.lru_cache(None)
        def wordBreak(s: str) -> bool:
            """Returns True if s can be segmented."""
            if s in wordSet:
                return True
            return any(s[:i] in wordSet and wordBreak(s[i:]) for i in range(len(s)))

        return wordBreak(s)
```

Python

```
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        words = set(wordDict)
        n = len(s)
        f = [True] + [False] * n
        for i in range(1, n + 1):
            f[i] = any(f[j] and s[j:i] in words for j in range(i))
        return f[n]
```

Python

```
class Trie:
    def __init__(self):
        self.children: List[Trie | None] = [None] * 26
        self.isEnd = False

    def insert(self, w: str):
        node = self
        for c in w:
            idx = ord(c) - ord('a')
            if not node.children[idx]:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.isEnd = True


class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        trie = Trie()
        for w in wordDict:
            trie.insert(w)
        n = len(s)
        f = [False] * (n + 1)
        f[n] = True
        for i in range(n - 1, -1, -1):
            node = trie
```



```
            for j in range(i, n):
                idx = ord(s[j]) - ord('a')
                if not node.children[idx]:
                    break
                node = node.children[idx]
                if node.isEnd and f[j + 1]:
                    f[i] = True
                    break
        return f[0]
```

Python

```
class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        words = set(wordDict)
        n = len(s)

        # dp[j] meaning s from 0 to index=i-1~ is breakable
        dp = [True] + [False] * n # so actually size is n+1
        for i in range(1, n + 1):
            dp[i] = any( (dp[j] and s[j:i] in words) for j in range(i) )
        return dp[n]

############

class Solution:
    def wordBreak(self, s: str, wordDict: Set[str]) -> bool:
        if not s or not wordDict:
            return False

        length = len(s)

        # construct dp, dp[i] meaning position i-1 can from dict or not
        dp[-1] meaning at last-index plus 1, checking up to last-index
        dp = [False] * (length + 1)
        dp[0] = True # initiate
```



```
        for i in range(length + 1):
            if not dp[i]:
                continue
            for word in wordDict:
                if i + len(word) > length:
                    continue
                if s[i:i + len(word)] == word:
                    dp[i + len(word)] = True

        return dp[length]

############

class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        words = set(wordDict)
        n = len(s)
        dp = [False] * (n + 1)
        dp[0] = True # i'th char in string, is good
        for j in range(1, n + 1):
            for i in range(j): # starting at 0, including dp[0]
                if dp[i] and s[i:j] in words:
                    dp[j] = True # j is exclusive, meaning True until index i-1
                    break
        return dp[-1]
```

```
############

class Trie:
    def __init__(self):
        self.children = [None] * 26
        self.is_end = False

    def insert(self, w):
        node = self
        for c in w:
            idx = ord(c) - ord('a')
            if node.children[idx] is None:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.is_end = True

    def search(self, w):
        node = self
        for c in w:
            idx = ord(c) - ord('a')
            if node.children[idx] is None:
                return False
            node = node.children[idx]

        return node.is_end

class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        # https://docs.python.org/3/library/functools.html#functools.cache
```


[*] Trie — Pattern Solution (matched from community answers)

Python

```
class Trie:
    def __init__(self):
        self.children: List[Trie | None] = [None] * 26
        self.isEnd = False

    def insert(self, w: str):
        node = self
        for c in w:
            idx = ord(c) - ord('a')
            if not node.children[idx]:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.isEnd = True


class Solution:
    def wordBreak(self, s: str, wordDict: List[str]) -> bool:
        trie = Trie()
        for w in wordDict:
            trie.insert(w)
        n = len(s)
        f = [False] * (n + 1)
        f[n] = True
        for i in range(n - 1, -1, -1):
            node = trie
```

```
            for j in range(i, n):
                idx = ord(s[j]) - ord('a')
                if not node.children[idx]:
                    break
                node = node.children[idx]
                if node.isEnd and f[j + 1]:
                    f[i] = True
                    break
        return f[0]
```

#208 MEDIUM

Implement Trie (Prefix Tree)

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Design · Trie

Lists: Grind 169

Problem:

A trie (pronounced as "try") or prefix tree is a tree data structure used to efficiently store and retrieve keys in a dataset of strings. There are various applications of this data structure, such as autocomplete and spellchecker.

Implement the Trie class:

- Trie() Initializes the trie object.
- void insert(String word) Inserts the string word into the trie.

- boolean search(String word) Returns true if the string word is in the trie (i.e., was inserted before), and false otherwise.
- boolean startsWith(String prefix) Returns true if there is a previously inserted string word that has the prefix prefix, and false otherwise.

Example 1:

```
Input
["Trie", "insert", "search", "search", "startsWith", "insert", "search"]
[[], ["apple"], ["apple"], ["app"], ["app"], ["app"], ["app"]]
Output
[null, null, true, false, true, null, true]

Explanation
Trie trie = new Trie();
trie.insert("apple");
trie.search("apple"); // return True
trie.search("app"); // return False
trie.startsWith("app"); // return True
trie.insert("app");
trie.search("app"); // return True
```

Constraints:

- 1 <= word.length, prefix.length <= 2000
- word and prefix consist only of lowercase English letters.
- At most 3 * 104 calls in total will be made to insert, search, and startsWith.

Python

Python

```
class TrieNode:
    def __init__(self):
        # Dictionary to store child nodes and a flag for end of word.
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        # Initialize the Trie with the root node.
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        # Start from the root.
        node = self.root
        for char in word:
            # If character doesn't exist, add a new TrieNode.
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        # Mark the end of word.
        node.is_end_of_word = True

    def search(self, word: str) -> bool:
        # Traverse the Trie for each character.
        node = self.root
```



```
        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        # Returns True if word ends exactly here.
        return node.is_end_of_word

    def startsWith(self, prefix: str) -> bool:
        # Traverse checking for the prefix.
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True
```

Python

```
class TrieNode:
  def __init__(self):
    self.children: dict[str, TrieNode] = {}
    self.isWord = False


class Trie:
  def __init__(self):
    self.root = TrieNode()

  def insert(self, word: str) -> None:
    node: TrieNode = self.root
    for c in word:
      node = node.children.setdefault(c, TrieNode())
    node.isWord = True

  def search(self, word: str) -> bool:
    node = self._find(word)
    return node is not None and node.isWord

  def startsWith(self, prefix: str) -> bool:
    return self._find(prefix) is not None

  def _find(self, prefix: str) -> TrieNode | None:
    node: TrieNode = self.root
```

```
    for c in prefix:
      if c not in node.children:
        return None
      node = node.children[c]
    return node
```

Python

```python
class Trie:
    def __init__(self):
        self.children = [None] * 26
        self.is_end = False

    def insert(self, word: str) -> None:
        node = self
        for c in word:
            idx = ord(c) - ord('a')
            if node.children[idx] is None:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.is_end = True

    def search(self, word: str) -> bool:
        node = self._search_prefix(word)
        return node is not None and node.is_end

    def startsWith(self, prefix: str) -> bool:
        node = self._search_prefix(prefix)
        return node is not None

    def _search_prefix(self, prefix: str):
        node = self
        for c in prefix:
            idx = ord(c) - ord('a')
            if node.children[idx] is None:
                return None
            node = node.children[idx]
        return node


# Your Trie object will be instantiated and called as such:
# obj = Trie()
# obj.insert(word)
# param_2 = obj.search(word)
# param_3 = obj.startsWith(prefix)
```

Python


```python
import collections

class TrieNode:

    def __init__(self):
        '''
        Usually, a Python dictionary throws a KeyError if you try to get an item with a key that is not
currently in the dictionary.
        The defaultdict in contract will simply create any items that you try to access

        https://stackoverflow.com/questions/5900578/how-does-collections-defaultdict-work
        '''
        self.child = collections.defaultdict(TrieNode)
        self.is_word = False

class Trie:

    def __init__(self):
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        cur = self.root
        for letter in word:
            cur = cur.child[letter]
        cur.is_word = True

    def search(self, word: str) -> bool:
        cur = self.root
        for letter in word:
            # cur = cur.child[letter] ==> will not work, it will creat a default node for this letter
            cur = cur.child.get(letter)
            if not cur:
                return False
        return cur.is_word

    def startsWith(self, prefix: str) -> bool:
        cur = self.root
        for letter in prefix:
            # cur = cur.child[letter] ==> will not work, it will creat a default node for this letter
            cur = cur.child.get(letter)
            if not cur:
                return False
        return True

# Your Trie object will be instantiated and called as such:
# obj = Trie()
# obj.insert(word)
# param_2 = obj.search(word)
# param_3 = obj.startsWith(prefix)
```



```python
##########

class Trie:
    def __init__(self):
        self.children = [None] * 26
        self.is_end = False

    def insert(self, word: str) -> None:
        node = self
        for c in word:
            idx = ord(c) - ord('a')
            if node.children[idx] is None:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.is_end = True

    def search(self, word: str) -> bool:
        node = self._search_prefix(word)
        return node is not None and node.is_end

    def startsWith(self, prefix: str) -> bool:
        node = self._search_prefix(prefix)
        return node is not None

    def _search_prefix(self, prefix: str):

        node = self
        for c in prefix:
            idx = ord(c) - ord('a')
            if node.children[idx] is None:
                return None
```

[*] Trie — Pattern Solution (matched from community answers)

Python


```python
class TrieNode:
    def __init__(self):
        # Dictionary to store child nodes and a flag for end of word.
        self.children = {}
        self.is_end_of_word = False

class Trie:
    def __init__(self):
        # Initialize the Trie with the root node.
        self.root = TrieNode()

    def insert(self, word: str) -> None:
        # Start from the root.
        node = self.root
        for char in word:
            # If character doesn't exist, add a new TrieNode.
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        # Mark the end of word.
        node.is_end_of_word = True

    def search(self, word: str) -> bool:
        # Traverse the Trie for each character.
        node = self.root

        for char in word:
            if char not in node.children:
                return False
            node = node.children[char]
        # Return True if word exists exactly here.
        return node.is_end_of_word

    def startsWith(self, prefix: str) -> bool:
        # Traverse checking for the prefix.
        node = self.root
        for char in prefix:
            if char not in node.children:
                return False
            node = node.children[char]
        return True
```

#211 MEDIUM

Design Add and Search Words Data Structure

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · DFS · Design · Trie

Lists: Grind 169

Problem:

Design a data structure that supports adding new words and finding if a string matches any previously added string.

Implement the WordDictionary class:

- WordDictionary() Initializes the object.

- void addWord(word) Adds word to the data structure, it can be matched later.
- bool search(word) Returns true if there is any string in the data structure that matches word or false otherwise. word may contain dots '.' where dots can be matched with any letter.

Example:

```
Input
["WordDictionary","addWord","addWord","addWord","search","search","search","search"]
[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]
Output
[null,null,null,null,false,true,true,true]

Explanation
WordDictionary wordDictionary = new WordDictionary();
wordDictionary.addWord("bad");
wordDictionary.addWord("dad");
wordDictionary.addWord("mad");
wordDictionary.search("pad"); // return False
wordDictionary.search("bad"); // return True
wordDictionary.search(".ad"); // return True
wordDictionary.search("b.."); // return True
```

Constraints:

- 1 <= word.length <= 25
- word in addWord consists of lowercase English letters.
- word in search consist of '.' or lowercase English letters.
- There will be at most 2 dots in word for search queries.
- At most 104 calls will be made to addWord and search.

Python

Python


```python
class TrieNode:
    def __init__(self):
        # Dictionary to store child nodes
        self.children = {}
        # Flag to indicate if a word ends here
        self.is_end = False

class WordDictionary:
    def __init__(self):
        # Initialize the Trie with an empty root node
        self.root = TrieNode()

    def addWord(self, word: str) -> None:
        # Start at the root of the Trie
        node = self.root
        for char in word:
            # Create a new node if the character is not present
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        # Mark the end of the word
        node.is_end = True

    def search(self, word: str) -> bool:
        # DFS helper function

        def dfs(index, node):
            if index == len(word):
                return node.is_end
            char = word[index]
            if char == '.':
                # If wildcard, search all children nodes
                for child in node.children.values():
                    if dfs(index + 1, child):
                        return True
                return False
            else:
                # If character not found, word does not exist
                if char not in node.children:
                    return False
                return dfs(index + 1, node.children[char])
        return dfs(0, self.root)

# Example usage:
wd = WordDictionary()
wd.addWord("bad")
wd.addWord("dad")
wd.addWord("mad")
print(wd.search("pad")) # Output: False
print(wd.search("bad")) # Output: True
print(wd.search(".ad")) # Output: True

print(wd.search("b..")) # Output: True
```

Python

```python
class TrieNode:
    def __init__(self):
        self.children: dict[str, TrieNode] = {}
        self.isWord = False


class WordDictionary:
    def __init__(self):
        self.root = TrieNode()

    def addWord(self, word: str) -> None:
        node: TrieNode = self.root
        for c in word:
            node = node.children.setdefault(c, TrieNode())
        node.isWord = True

    def search(self, word: str) -> bool:
        return self._dfs(word, 0, self.root)

    def _dfs(self, word: str, s: int, node: TrieNode) -> bool:
        if s == len(word):
            return node.isWord
        if word[s] != '.':
            child: TrieNode = node.children.get(word[s], None)
            return self._dfs(word, s + 1, child) if child else False

        return any(self._dfs(word, s + 1, child) for child in node.children.values())
```

Python

```python
class Trie:
    def __init__(self):
        self.children = [None] * 26
        self.is_end = False


class WordDictionary:
    def __init__(self):
        self.trie = Trie()

    def addWord(self, word: str) -> None:
        node = self.trie
        for c in word:
            idx = ord(c) - ord('a')
            if node.children[idx] is None:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.is_end = True

    def search(self, word: str) -> bool:
        def search(word, node):
            for i in range(len(word)):
                c = word[i]
                idx = ord(c) - ord('a')
                if c != '.' and node.children[idx] is None:
```



```python
                    return False
                if c == '.':
                    for child in node.children:
                        if child is not None and search(word[i + 1 :], child):
                            return True
                    return False
                node = node.children[idx]
            return node.is_end

        return search(word, self.trie)


# Your WordDictionary object will be instantiated and called as such:
# obj = WordDictionary()
# obj.addWord(word)
# param_2 = obj.search(word)
```

Python

```python
class Trie:
    def __init__(self):
        self.children = [None] * 26
        self.is_end = False


class WordDictionary:
    def __init__(self):
        self.trie = Trie()

    def addWord(self, word: str) -> None:
        node = self.trie
        for c in word:
            idx = ord(c) - ord('a')
            if node.children[idx] is None:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.is_end = True

    def search(self, word: str) -> bool:
        def search(word, node):
            for i in range(len(word)):
                c = word[i]
                idx = ord(c) - ord('a')
                if c != '.' and node.children[idx] is None:
```



```python
                    return False
                if c == '.':
                    for child in node.children:
                        if child is not None and search(word[i + 1 :], child):
                            return True
                    return False
                node = node.children[idx]
            return node.is_end

        return search(word, self.trie)


# Your WordDictionary object will be instantiated and called as such:
# obj = WordDictionary()
# obj.addWord(word)
# param_2 = obj.search(word)
```

[*] Trie — Pattern Solution (matched from community answers)

Python

```python
class TrieNode:
    def __init__(self):
        # Dictionary to store child nodes
        self.children = {}
        # Flag to indicate if a word ends here
        self.is_end = False

class WordDictionary:
    def __init__(self):
        # Initialize the Trie with an empty root node
        self.root = TrieNode()

    def addWord(self, word: str) -> None:
        # Start at the root of the Trie
        node = self.root
        for char in word:
            # Create a new node if the character is not present
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
        # Mark the end of the word
        node.is_end = True

    def search(self, word: str) -> bool:
        # DFS helper function
```



```python
        def dfs(index, node):
            if index == len(word):
                return node.is_end
            char = word[index]
            if char == '.':
                # If wildcard, search all children nodes
                for child in node.children.values():
                    if dfs(index + 1, child):
                        return True
                return False
            else:
                # If character not found, word does not exist
                if char not in node.children:
                    return False
                return dfs(index + 1, node.children[char])
        return dfs(0, self.root)

# Example usage:
wd = WordDictionary()
wd.addWord("bad")
wd.addWord("dad")
wd.addWord("mad")
print(wd.search("pad")) # Output: False
print(wd.search("bad")) # Output: True
print(wd.search(".ad")) # Output: True

print(wd.search("b..")) # Output: True
```

#616 MEDIUM

Add Bold Tag in String

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · String · Trie

Lists: Premium 98

Problem:

You are given a string s and an array of strings words.

You should add a closed pair of bold tag and to wrap the substrings in s that exist in words.

- If two such substrings overlap, you should wrap them together with only one pair of closed bold-tag.
- If two substrings wrapped by bold tags are consecutive, you should combine them.

Return s after adding the bold tags.

Example 1:

```
Input: s = "abcxyz123", words = ["abc","123"]
Output: "<b>abc</b>xyz<b>123</b>"
Explanation: The two strings of words are substrings of s as following: "abcxyz123".
We add <b> before each substring and </b> after each substring.
```

Example 2:

```
Input: s = "aaabbb", words = ["aa","b"]
Output: "<b>aaabbb</b>"
Explanation:
```



```
"aa" appears as a substring two times: "aaabbb" and "aaabbb".
"b" appears as a substring three times: "aaabbb", "aaabbb", and "aaabbb".
We add <b> before each substring and </b> after each substring:
"<b>a<b>a</b>a</b><b>b</b><b>b</b><b>b</b>".
Since the first two <b>'s overlap, we merge them: "<b>aaa</b><b>b</b><b>b</b><b>b</b>".
Since now the four <b>'s are consecutive, we merge them: "<b>aaabbb</b>".
```

Constraints:

- 1 <= s.length <= 1000
- 0 <= words.length <= 100
- 1 <= words[i].length <= 1000
- s and words[i] consist of English letters and digits.
- All the values of words are unique.

Note: This question is the same as 758. Bold Words in String.

>> Brute Force [Trie] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def addBoldTag(self, s, words):
        # Brute force O(n*m) - linear scan through all words
        matches = []
        for word in words:
            if word.startswith(prefix):
                matches.append(word)
        return matches
```

Python

Python


```python
# Python Code Solution
def addBoldTag(s, words):
    n = len(s)
    # Mark positions to be bolded
    bold = [False] * n
    # Iterate each word and mark its occurrences
    for word in words:
        start = s.find(word)
        while start != -1:
            for i in range(start, start + len(word)):
                bold[i] = True
            start = s.find(word, start + 1)

    # Build the final result with bold tags
    result = []
    i = 0
    while i < n:
        if not bold[i]:
            result.append(s[i])
            i += 1
        else:
            result.append("<b>")
            while i < n and bold[i]:
                result.append(s[i])
                i += 1

            result.append("</b>")
    return "".join(result)

# Example usage:
# s = "abcxyz123"
# words = ["abc", "123"]
# print(addBoldTag(s, words)) # "<b>abc</b>xyz<b>123</b>"

# Line-by-line:
# 1. Initialize an array "bold" of length n to track which characters need bold formatting.
# 2. For each word, use string.find to locate every occurrence and mark the corresponding indices.
# 3. Traverse the string s. If the current character is marked bold and it is the start of a bold segment,
# insert the opening tag before appending characters until the end of that bold segment.
# 4. Append the closing tag after the bold segment and continue the process.
```

Python

```python
class Solution:
    def addBoldTag(self, s: str, words: list[str]) -> str:
        n = len(s)
        ans = []
        # bold[i] = True if s[i] should be bolded
        bold = [0] * n

        boldEnd = -1 # s[:boldEnd] should be bolded
        for i in range(n):
            for word in words:
                if s[i:].startswith(word):
                    boldEnd = max(boldEnd, i + len(word))
            bold[i] = boldEnd > i

        # Construct the with bold tags
        i = 0
        while i < n:
            if bold[i]:
                j = i
                while j < n and bold[j]:
                    j += 1
                # `s[i..j]` should be bolded.
                ans.append('<b>' + s[i:j] + '</b>')
                i = j
            else:

                ans.append(s[i])
                i += 1

        return ''.join(ans)
```

Python

```python
class Trie:
    def __init__(self):
        self.children = [None] * 128
        self.is_end = False

    def insert(self, word):
        node = self
        for c in word:
            idx = ord(c)
            if node.children[idx] is None:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.is_end = True


class Solution:
    def addBoldTag(self, s: str, words: List[str]) -> str:
        trie = Trie()
        for w in words:
            trie.insert(w)
        n = len(s)
        pairs = []
        for i in range(n):
            node = trie
            for j in range(i, n):
```



```python
                idx = ord(s[j])
                if node.children[idx] is None:
                    break
                node = node.children[idx]
                if node.is_end:
                    pairs.append([i, j])
        if not pairs:
            return s
        st, ed = pairs[0]
        t = []
        for a, b in pairs[1:]:
            if ed + 1 < a:
                t.append([st, ed])
                st, ed = a, b
            else:
                ed = max(ed, b)
        t.append([st, ed])

        ans = []
        i = j = 0
        while i < n:
            if j == len(t):
                ans.append(s[i:])
                break
            st, ed = t[j]

            if i < st:
                ans.append(s[i:st])
            ans.append('<b>')
            ans.append(s[st : ed + 1])
            ans.append('</b>')
            j += 1
            i = ed + 1

        return ''.join(ans)
```

[*] Trie — Pattern Solution (matched from community answers)

Python


```python
class Trie:
    def __init__(self):
        self.children = [None] * 128
        self.is_end = False

    def insert(self, word):
        node = self
        for c in word:
            idx = ord(c)
            if node.children[idx] is None:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.is_end = True


class Solution:
    def addBoldTag(self, s: str, words: List[str]) -> str:
        trie = Trie()
        for w in words:
            trie.insert(w)
        n = len(s)
        pairs = []
        for i in range(n):
            node = trie
            for j in range(i, n):
```



```python
                idx = ord(s[j])
                if node.children[idx] is None:
                    break
                node = node.children[idx]
                if node.is_end:
                    pairs.append([i, j])
        if not pairs:
            return s
        st, ed = pairs[0]
        t = []
        for a, b in pairs[1:]:
            if ed + 1 < a:
                t.append([st, ed])
                st, ed = a, b
            else:
                ed = max(ed, b)
        t.append([st, ed])

        ans = []
        i = j = 0
        while i < n:
            if j == len(t):
                ans.append(s[i:])
                break
            st, ed = t[j]

            if i < st:
                ans.append(s[i:st])
            ans.append('<b>')
            ans.append(s[st : ed + 1])
            ans.append('</b>')
            j += 1
            i = ed + 1

        return ''.join(ans)
```

#692 MEDIUM

Top K Frequent Words

LeetCode › SimpleList · WalkCC · LeetCode · Editorial

Hash Table · String · Trie · Sorting · Heap

Lists: Grind 169

Problem:

Given an array of strings words and an integer k, return the k most frequent strings.

Return the answer sorted by the frequency from highest to lowest. Sort the words with the same frequency by their lexicographical order.

Example 1:

```
Input: words = ["i","love","leetcode","i","love","coding"], k = 2
Output: ["i","love"]
Explanation: "i" and "love" are the two most frequent words.
Note that "i" comes before "love" due to a lower alphabetical order.
```

Example 2:

```
Input: words = ["the","day","is","sunny","the","the","the","sunny","is","is"], k = 4
Output: ["the","is","sunny","day"]
Explanation: "the", "is", "sunny" and "day" are the four most frequent words, with the number
of occurrence being 4, 3, 2 and 1 respectively.
```

Constraints:

- 1 <= words.length <= 500
- 1 <= words[i].length <= 10
- words[i] consists of lowercase English letters.
- k is in the range [1, The number of unique words[i]]

Follow-up: Could you solve it in O(n log(k)) time and O(n) extra space?

>> Brute Force [Trie] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def topKFrequent(self, words, k):
        # Brute Force O(n·m) - linear scan through all words
        matches = []
        for word in words:
            if word.startswith(prefix):
                matches.append(word)
        return matches
```

Python

Python

```python
import heapq
from collections import Counter

def topKFrequent(words, k):
    # Count the frequency of each word
    count = Counter(words)

    # Initialize a min-heap; Python's heapq is a min-heap.
    # The comparator is (frequency, negative word) but use word in reverse order:
    # Since we want words with lower alphabetical order to be prioritized in the result,
    # and in the min heap if frequencies equal, we want the one with lexicographically larger word to be
popped.
    heap = []

    for word, freq in count.items():
        # Push each pair into the heap. The tuple ordering:
        # First element is frequency and second element is the word (with reverse lex order trick)
        heapq.heappush(heap, (freq, -ord(word[0]), word))
        # However, using -ord(word[0]) is not robust for multi-letter words.
        # Instead, we can use tuple (freq, word) with a custom comparator by ensuring we invert the word
order by using word in reverse.
        # For simplicity, we push tuple (freq, word) and then pop if heap size > k with a custom condition.

    # Reset heap and use a custom comparator implemented by the tuple (-freq, word) to simulate max heap
behavior.
    heap = []
    for word, freq in count.items():
        # We push with frequency as negative so that larger frequencies come first;
```

```python
        # use word as is to satisfy lexicographical order when frequencies are equal.
        heapq.heappush(heap, (-freq, word))

    # Extract k elements from the heap and sort them accordingly.
    result = []
    for _ in range(k):
        freq, word = heapq.heappop(heap)
        result.append(word)
    return result

# Example usage:
print(topKFrequent(["i","love","leetcode","i","love","coding"], 2)) # Output: ['i', 'love']
```

Python


```python
class Solution:
    def topKFrequent(self, words: list[str], k: int) -> list[str]:
        ans = []
        bucket = [[] for _ in range(len(words) + 1)]

        for word, freq in collections.Counter(words).items():
            bucket[freq].append(word)

        for b in reversed(bucket):
            for word in sorted(b):
                ans.append(word)
                if len(ans) == k:
                    return ans
```

Python

```python
from dataclasses import dataclass


@dataclass(frozen=True)
class T:
    word: str
    freq: int

    def __lt__(self, other):
        if self.freq == other.freq:
            return self.word > other.word
        return self.freq < other.freq


class Solution:
    def topKFrequent(self, words: list[str], k: int) -> list[str]:
        ans = []
        heap = []

        for word, freq in collections.Counter(words).items():
            heapq.heappush(heap, T(word, freq))
            if len(heap) > k:
                heapq.heappop(heap)

        while heap:
```

```python
            ans.append(heapq.heappop(heap).word)

        return ans[::-1]
```

Python

```python
class Solution:
    def topKFrequent(self, words: List[str], k: int) -> List[str]:
        cnt = Counter(words)
        return sorted(cnt, key=lambda x: (-cnt[x], x))[:k]
```

Python

```
'''
>>> sorted(student_objects, key=attrgetter('grade', 'age'))
[('john', 'A', 15), ('dave', 'B', 10), ('jane', 'B', 12)]

ref: https://docs.python.org/3/howto/sorting.html#operator-module-functions
'''

class Solution:
    def topKFrequent(self, words: List[str], k: int) -> List[str]:
        cnt = Counter(words)
        # multiple comparators "lambda x: (-cnt[x], x)"
        return sorted(cnt, key=lambda x: (-cnt[x], x))[:k]
        # why the return/yield sorted() not a tuple (word, count), but only word?
        # it's the property of Counter() class
        # so, also pass O3: return sorted(cnt.keys(), key=lambda x: (-cnt[x], x))[:k]

'''
>>> words = ["i","love","leetcode","i","love","coding"]
>>> k = 2
>>> cnt = Counter(words)
>>> cnt
Counter({'i': 2, 'love': 2, 'leetcode': 1, 'coding': 1})

>>> sorted(cnt, key=lambda x: (-cnt[x], x))
['i', 'love', 'coding', 'leetcode']
```

```
# default, only for keys()
>>> sorted(cnt)
['coding', 'i', 'leetcode', 'love']
>>> sorted(cnt.keys())
['coding', 'i', 'leetcode', 'love']
>>> sorted(cnt.values())
[1, 1, 2, 2]
>>> sorted(cnt.items())
[('coding', 1), ('i', 2), ('leetcode', 1), ('love', 2)]
'''


###########

'''
>>> words = ["the","day","is","sunny","the","the","the","sunny","is","is"]
>>> count = collections.Counter(words)
>>> count
Counter({'the': 4, 'is': 3, 'sunny': 2, 'day': 1})
>>>
>>> count.items()
dict_items([('the', 4), ('day', 1), ('is', 3), ('sunny', 2)])
>>>
'''
```



```
'''
cmp doesn't exist in Python 3. If you really want it, you could define it yourself:

def cmp(a, b):
    return (a > b) - (a < b)

'''
class Solution(object): #python-2
    def topKFrequent(self, words, k):
        """
        :type words: List[str]
        :type k: int
        :rtype: List[str]
        """
        count = collections.Counter(words)

        # https://docs.python.org/3/howto/sorting.html#comparison-functions
        def compare(x, y):
            if x[1] == y[1]:
                return cmp(x[0], y[0])
            else:
                return -cmp(x[1], y[1]) # -cmp, so reversed order, bigger ones at front
        return [x[0] for x in sorted(count.items(), cmp = compare)[:k]]
```

[*] Trie — Pattern Solution (stored optimal)

Python

```
import heapq
from collections import Counter

def topKFrequent(words, k):
    # Count the frequency of each word
    count = Counter(words)

    # Initialize a min-heap; Python's heapq is a min-heap.
    # The comparator is (frequency, negative word) but use word in reverse order:
    # Since we want words with lower alphabetical order to be prioritized in the result,
    # and in the min heap if frequencies equal, we want the one with lexicographically larger word to be
popped.
    heap = []

    for word, freq in count.items():
        # Push each pair into the heap. The tuple ordering:
        # First element is frequency and second element is the word (with reverse lex order trick)
        heapq.heappush(heap, (freq, -ord(word[0]), word))
        # However, using -ord(word[0]) is not robust for multi-letter words.
        # Instead, we can use tuple (freq, word) with a custom comparator by ensuring we invert the word
order by using word in reverse.
        # For simplicity, we push tuple (freq, word) and then pop if heap size > k with a custom condition.

    # Reset heap and use a custom comparator implemented by the tuple (-freq, word) to simulate max heap
behavior.
    heap = []
    for word, freq in count.items():
        # We push with frequency as negative so that larger frequencies come first;
```



```
        # use word as is to satisfy lexicographical order when frequencies are equal.
        heapq.heappush(heap, (-freq, word))

    # Extract k elements from the heap and sort them accordingly.
    result = []
    for _ in range(k):
        freq, word = heapq.heappop(heap)
        result.append(word)
    return result

# Example usage:
print(topKFrequent(["i","love","leetcode","i","love","coding"], 2)) # Output: ['i', 'love']
```

#720 **MEDIUM**

Longest Word in Dictionary

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · String · Trie · Sorting

Lists: Denny Zhang

Problem:

Given an array of strings words representing an English Dictionary, return the longest word in words that can be built one character at a time by other words in words.

If there is more than one possible answer, return the longest word with the smallest lexicographical order. If there is no answer, return the empty string.

Note that the word should be built from left to right with each additional character being added to the end of a previous word.

Example 1:

```
Input: words = ["w","wo","wor","worl","world"]
Output: "world"
Explanation: The word "world" can be built one character at a time by "w", "wo", "wor", and
"worl".
```

Example 2:

```
Input: words = ["a","banana","app","appl","ap","apply","apple"]
Output: "apple"
Explanation: Both "apply" and "apple" can be built from other words in the dictionary.
However, "apple" is lexicographically smaller than "apply".
```

Constraints:

- 1 <= words.length <= 1000
- 1 <= words[i].length <= 30
- words[i] consists of lowercase English letters.

>> Brute Force [Trie] Interview Prep

Python


```
# Brute Force Approach
class Solution:
    def longestWord(self, words):
        # Brute Force O(n*m) - linear scan through all words
        matches = []
        for word in words:
            if word.startswith(prefix):
                matches.append(word)
        return matches
```

Python

Python

```
# Python solution for Longest Word in Dictionary

def longestWord(words):
    # Sort words by length and lexicographical order
    words.sort(key=lambda x: (len(x), x))

    valid_words = set()
    result = ""

    # Process each word
    for word in words:
        # For single-letter words, they are valid if they exist in the list
        if len(word) == 1 or word[:-1] in valid_words:
            valid_words.add(word)
            # Update result if the current word is longer or if equal length but lexicographically smaller
            if len(word) > len(result) or (len(word) == len(result) and word < result):
                result = word

    return result

# Example usage:
# print(longestWord(["a", "banana", "app", "appl", "ap", "apply", "apple"]))
```

Python


```
class Solution:
    def longestWord(self, words: list[str]) -> str:
        root = {}

        for word in words:
            node = root
            for c in word:
                if c not in node:
                    node[c] = {}
                node = node[c]
            node['word'] = word

        def dfs(node: dict) -> str:
            ans = node['word'] if 'word' in node else ''

            for child in node:
                if 'word' in node[child] and len(node[child]['word']) > 0:
                    childWord = dfs(node[child])
                    if len(childWord) > len(ans) or (
                            len(childWord) == len(ans) and childWord < ans):
                        ans = childWord

            return ans

        return dfs(root)
```

Python

```
class Trie:
    def __init__(self):
        self.children: List[Optional[Trie]] = [None] * 26
        self.is_end = False

    def insert(self, w: str):
        node = self
        for c in w:
            idx = ord(c) - ord("a")
            if node.children[idx] is None:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.is_end = True

    def search(self, w: str) -> bool:
        node = self
        for c in w:
            idx = ord(c) - ord("a")
            if node.children[idx] is None:
                return False
            node = node.children[idx]
            if not node.is_end:
                return False
        return True
```



```
class Solution:
    def longestWord(self, words: List[str]) -> str:
        trie = Trie()
        for w in words:
            trie.insert(w)
        ans = ""
        for w in words:
            if trie.search(w) and (
                len(ans) < len(w) or (len(ans) == len(w) and ans > w)
            ):
                ans = w
        return ans
```

Python

```
class Solution:
    def longestWord(self, words: List[str]) -> str:
        cnt, ans = 0, ''
        s = set(words)
        for w in s:
            n = len(w)
            if all(w[:i] in s for i in range(1, n)):
                if cnt < n:
                    cnt, ans = n, w
                elif cnt == n and w < ans:
                    ans = w
        return ans
```

[*] Trie — Pattern Solution (matched from community answers)

Python

```
class Trie:
    def __init__(self):
        self.children: List[Optional[Trie]] = [None] * 26
        self.is_end = False

    def insert(self, w: str):
        node = self
        for c in w:
            idx = ord(c) - ord("a")
            if node.children[idx] is None:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.is_end = True

    def search(self, w: str) -> bool:
        node = self
        for c in w:
            idx = ord(c) - ord("a")
            if node.children[idx] is None:
                return False
            node = node.children[idx]
            if not node.is_end:
                return False
        return True
```

```python
class Solution:
    def longestWord(self, words: List[str]) -> str:
        trie = Trie()
        for w in words:
            trie.insert(w)
        ans = ""
        for w in words:
            if trie.search(w) and (
                len(ans) < len(w) or (len(ans) == len(w) and ans > w)
            ):
                ans = w
        return ans
```

#1166 MEDIUM

Design File System

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Design · Trie

Lists: Denny Zhang

Problem:

You are asked to design a file system that allows you to create new paths and associate them with different values.

The format of a path is one or more concatenated strings of the form: / followed by one or more lowercase English letters. For example, "/leetcode" and "/leetcode/problems" are valid paths while an empty string "" and "/" are not.

Implement the FileSystem class:

- bool createPath(string path, int value) Creates a new path and associates a value to it if possible and returns true. Returns false if the path already exists or its parent path doesn't exist.
- int get(string path) Returns the value associated with path or returns -1 if the path doesn't exist.

Example 1:

```
Input:
["FileSystem","createPath","get"]
[[],["/a",1],["/a"]]
Output:
[null,true,1]
Explanation:
FileSystem fileSystem = new FileSystem();

fileSystem.createPath("/a", 1); // return true
fileSystem.get("/a"); // return 1
```

Example 2:

```
Input:
["FileSystem","createPath","createPath","get","createPath","get"]
[[],["/leet",1],["/leet/code",2],["/leet/code"],["/c/d",1],["/c"]]
Output:
[null,true,true,2,false,-1]
```



```
Explanation:
FileSystem fileSystem = new FileSystem();

fileSystem.createPath("/leet", 1); // return true
fileSystem.createPath("/leet/code", 2); // return true
fileSystem.get("/leet/code"); // return 2
fileSystem.createPath("/c/d", 1); // return false because the parent path "/c" doesn't exist.
fileSystem.get("/c"); // return -1 because this path doesn't exist.
```

Constraints:

- 2 <= path.length <= 100
- 1 <= value <= 109
- Each path is valid and consists of lowercase English letters and '/'.
- At most 104 calls in total will be made to createPath and get.

Python

Python

```python
class FileSystem:
    def __init__(self):
        # Initialize hash map to store paths and their values.
        self.paths = {}

    def createPath(self, path: str, value: int) -> bool:
        # Check if the path already exists.
        if path in self.paths:
            return False

        # Find the index of the last '/' to get the parent path.
        last_slash = path.rfind('/')
        # If last slash is 0, then parent is root level, which is invalid because "" is not a valid path.
        if last_slash == 0:
            parent = ""
        else:
            parent = path[:last_slash]

        # For non-root paths, the parent must exist.
        if parent and parent not in self.paths:
            return False

        # Store the new path with its value.
        self.paths[path] = value
        return True

    def get(self, path: str) -> int:
        # Return the value associated with the path, or -1 if it doesn't exist.
        return self.paths.get(path, -1)

# Example usage:
# fs = FileSystem()
# print(fs.createPath("/a", 1)) # True
# print(fs.get("/a")) # 1
```

Python


```python
class TrieNode:
    def __init__(self, value: int = 0):
        self.children: dict[str, TrieNode] = {}
        self.value = value


class FileSystem:
    def __init__(self):
        self.root = TrieNode()

    def createPath(self, path: str, value: int) -> bool:
        node: TrieNode = self.root
        subpaths = path.split('/')

        for i in range(1, len(subpaths) - 1):
            if subpaths[i] not in node.children:
                return False
            node = node.children[subpaths[i]]

        if subpaths[-1] in node.children:
            return False
        node.children[subpaths[-1]] = TrieNode(value)
        return True

    def get(self, path: str) -> int:

        node: TrieNode = self.root
        for subpath in path.split('/')[1:]:
            if subpath not in node.children:
                return -1
            node = node.children[subpath]
        return node.value
```

Python

```python
class Trie:
    def __init__(self, v: int = -1):
        self.children = {}
        self.v = v

    def insert(self, w: str, v: int) -> bool:
        node = self
        ps = w.split("/")
        for p in ps[1:-1]:
            if p not in node.children:
                return False
            node = node.children[p]
        if ps[-1] in node.children:
            return False
        node.children[ps[-1]] = Trie(v)
        return True

    def search(self, w: str) -> int:
        node = self
        for p in w.split("/")[1:]:
            if p not in node.children:
                return -1
            node = node.children[p]
        return node.v
```



```python
class FileSystem:
    def __init__(self):
        self.trie = Trie()

    def createPath(self, path: str, value: int) -> bool:
        return self.trie.insert(path, value)

    def get(self, path: str) -> int:
        return self.trie.search(path)


# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem()
# param_1 = obj.createPath(path,value)
# param_2 = obj.get(path)
```

Python

```python
'''
>>> "/a/b/c".split("/")
['', 'a', 'b', 'c']
'''


class Trie:
    def __init__(self, v: int = -1):
        self.children = {}
        self.v = v

    def insert(self, w: str, v: int) -> bool:
        node = self
        ps = w.split("/")
        for p in ps[1:-1]:
            if p not in node.children:
                return False
            node = node.children[p]
        if ps[-1] in node.children:
            return False
        node.children[ps[-1]] = Trie(v)
        return True

    def search(self, w: str) -> int:
        node = self
        for p in w.split("/")[1:]:
```



```python
            if p not in node.children:
                return -1
            node = node.children[p]
        return node.v


class FileSystem:
    def __init__(self):
        self.trie = Trie()

    def createPath(self, path: str, value: int) -> bool:
        return self.trie.insert(path, value)

    def get(self, path: str) -> int:
        return self.trie.search(path)


# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem()
# param_1 = obj.createPath(path,value)
# param_2 = obj.get(path)
```

[*] Trie — Pattern Solution (matched from community answers)

Python

```python
class Trie:
    def __init__(self, v: int = -1):
        self.children = {}
        self.v = v

    def insert(self, w: str, v: int) -> bool:
        node = self
        ps = w.split("/")
        for p in ps[1:-1]:
            if p not in node.children:
                return False
            node = node.children[p]
        if ps[-1] in node.children:
            return False
        node.children[ps[-1]] = Trie(v)
        return True

    def search(self, w: str) -> int:
        node = self
        for p in w.split("/")[1:]:
            if p not in node.children:
                return -1
            node = node.children[p]
        return node.v
```



```python
class FileSystem:
    def __init__(self):
        self.trie = Trie()

    def createPath(self, path: str, value: int) -> bool:
        return self.trie.insert(path, value)

    def get(self, path: str) -> int:
        return self.trie.search(path)


# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem()
# param_1 = obj.createPath(path,value)
# param_2 = obj.get(path)
```

#212 HARD

Word Search II

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · String · Backtracking · Trie · Matrix

Lists: Grind 169

Problem:

Given an m x n board of characters and a list of strings words, return all words on the board.

Each word must be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

Example 1:

o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v

```
Input: board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]],
words = ["oath","pea","eat","rain"]
Output: ["eat","oath"]
```

Example 2:

a	b
c	d

```
Input: board = [["a","b"],["c","d"]], words = ["abcb"]
Output: []
```

Constraints:

- m == board.length
- n == board[i].length
- 1 <= m, n <= 12
- board[i][j] is a lowercase English letter.
- 1 <= words.length <= 3 * 104
- 1 <= words[i].length <= 10

- words[i] consists of lowercase English letters.
- All the strings of words are unique.

>> Brute Force [Trie] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findWords(self, board, words):
        # Brute Force O(k · m·n · 4^L) - run word-search for each word independently
        def exist(board, word):
            m, n = len(board), len(board[0])
            def dfs(r, c, i, seen):
                if i == len(word): return True
                if r<0 or r>=m or c<0 or c>=n: return False
                if (r,c) in seen or board[r][c]!=word[i]: return False
                seen.add((r,c))
                ok = any(dfs(r+dr,c+dc,i+1,seen)
                         for dr,dc in [(1,0),(-1,0),(0,1),(0,-1)])
                seen.remove((r,c))
                return ok
            return any(dfs(r,c,0,set()) for r in range(m) for c in range(n))
        return [w for w in words if exist(board, w)]
```

Python

Python

```python
# Define a TrieNode for building the Trie.
class TrieNode:
    def __init__(self):
        self.children = {}
        self.word = None # Stores complete word at the end node

class Solution:
    def findWords(self, board, words):
        # Build the Trie from the list of words.
        root = TrieNode()
        for word in words:
            node = root
            for char in word:
                if char not in node.children:
                    node.children[char] = TrieNode()
                node = node.children[char]
            node.word = word # Mark the end of a word

        result = []
        m, n = len(board), len(board[0])

        # Define the DFS helper function.
        def dfs(i, j, node):
            char = board[i][j]
            if char not in node.children:
```



```python
                return
            next_node = node.children[char]
            if next_node.word:
                result.append(next_node.word)
                next_node.word = None # Avoid duplicate entries

            board[i][j] = '#' # Mark the current cell as visited
            # Explore all four adjacent directions.
            for dx, dy in ((0, 1), (1, 0), (0, -1), (-1, 0)):
                ni, nj = i + dx, j + dy
                if 0 <= ni < m and 0 <= nj < n and board[ni][nj] != '#':
                    dfs(ni, nj, next_node)
            board[i][j] = char # Restore the original character

        # Start DFS from each cell.
        for i in range(m):
            for j in range(n):
                dfs(i, j, root)
        return result
```

Python

```python
class TrieNode:
    def __init__(self):
        self.children: dict[str, TrieNode] = {}
        self.word: str | None = None


class Solution:
    def findWords(self, board: list[list[str]], words: list[str]) -> list[str]:
        m = len(board)
        n = len(board[0])
        ans = []
        root = TrieNode()

        def insert(word: str) -> None:
            node = root
            for c in word:
                node = node.children.setdefault(c, TrieNode())
            node.word = word

        for word in words:
            insert(word)

        def dfs(i: int, j: int, node: TrieNode) -> None:
            if i < 0 or i >= m or j < 0 or j >= n:
                return
```



```python
            if board[i][j] == '*':
                return

            c = board[i][j]
            if c not in node.children:
                return

            child = node.children[c]
            if child.word:
                ans.append(child.word)
                child.word = None

            board[i][j] = '*'
            dfs(i + 1, j, child)
            dfs(i - 1, j, child)
            dfs(i, j + 1, child)
            dfs(i, j - 1, child)
            board[i][j] = c

        for i in range(m):
            for j in range(n):
                dfs(i, j, root)

        return ans
```

Python

```python
class Trie:
    def __init__(self):
        self.children = [None] * 26
        self.w = ""
        # minor ajust, not a boolean is_end, but the whole word
        # so it's easier for dfs to save the word

    def insert(self, w):
        node = self
        for c in w:
            idx = ord(c) - ord('a')
            if node.children[idx] is None:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.w = w


class Solution:
    def findWords(self, board: List[List[str]], words: List[str]) -> List[str]:
        def dfs(node, i, j):
            idx = ord(board[i][j]) - ord('a')
            if node.children[idx] is None:
                return
            node = node.children[idx]
            if node.w:
```



```python
                ans.add(node.w)
            c = board[i][j]
            board[i][j] = '@' # 0 for visited already
            for a, b in [(0, -1), (0, 1), (1, 0), (-1, 0)]:
                x, y = i + a, j + b
                if 0 <= x < m and 0 <= y < n and board[x][y] != '@':
                    dfs(node, x, y)
            board[i][j] = c

        trie = Trie()
        for w in words:
            trie.insert(w)
        ans = set()
        m, n = len(board), len(board[0])
        for i in range(m):
            for j in range(n):
                dfs(trie, i, j)
        return list(ans)


###############


class Trie:
    def __init__(self):

        self.children: List[Trie | None] = [None] * 26
        self.ref: int = -1

    def insert(self, w: str, ref: int):
        node = self
        for c in w:
            idx = ord(c) - ord('a')
            if node.children[idx] is None:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.ref = ref


class Solution:
    def findWords(self, board: List[List[str]], words: List[str]) -> List[str]:
        def dfs(node: Trie, i: int, j: int):
            idx = ord(board[i][j]) - ord('a')
            if node.children[idx] is None:
                return
            node = node.children[idx]
            if node.ref == 0:
                ans.append(words[node.ref])
                node.ref = -1
            c = board[i][j]
            board[i][j] = '#'

            for a, b in pairwise((-1, 0, 1, 0, -1)):
                x, y = i + a, j + b
                if 0 <= x < m and 0 <= y < n and board[x][y] != '#':
                    dfs(node, x, y)
            board[i][j] = c
```


[*] Trie — Pattern Solution (matched from community answers)

Python

```python
# Define a TrieNode for building the Trie.
class TrieNode:
    def __init__(self):
        self.children = {}
        self.word = None # Stores complete word at the end node

class Solution:
    def findWords(self, board, words):
        # Build the Trie from the list of words.
        root = TrieNode()
        for word in words:
            node = root
            for char in word:
                if char not in node.children:
                    node.children[char] = TrieNode()
                node = node.children[char]
            node.word = word # Mark the end of a word

        result = []
        m, n = len(board), len(board[0])

        # Define the DFS helper function.
        def dfs(i, j, node):
            char = board[i][j]
            if char not in node.children:
```

```python
                return
            next_node = node.children[char]
            if next_node.word:
                result.append(next_node.word)
                next_node.word = None # Avoid duplicate entries

            board[i][j] = '#' # Mark the current cell as visited
            # Explore all four adjacent directions.
            for dx, dy in ((0, 1), (1, 0), (0, -1), (-1, 0)):
                ni, nj = i + dx, j + dy
                if 0 <= ni < m and 0 <= nj < n and board[ni][nj] != '#':
                    dfs(ni, nj, next_node)
            board[i][j] = char # Restore the original character

        # Start DFS from each cell.
        for i in range(m):
            for j in range(n):
                dfs(i, j, root)
        return result
```

#336 HARD

Palindrome Pairs

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · String · Trie

Lists: Grind 169

Problem:

You are given a 0-indexed array of unique strings words.

A palindrome pair is a pair of integers (i, j) such that:

- 0 <= i, j < words.length,
- i != j, and
- words[i] + words[j] (the concatenation of the two strings) is a palindrome.

Return an array of all the palindrome pairs of words.

You must write an algorithm with O(sum of words[i].length) runtime complexity.

Example 1:

```
Input: words = ["abcd","dcba","lls","s","sssll"]
Output: [[0,1],[1,0],[3,2],[2,4]]
Explanation: The palindromes are ["abcddcba","dcbaabcd","slls","llsssll"]
```

Example 2:

```
Input: words = ["bat","tab","cat"]
Output: [[0,1],[1,0]]
Explanation: The palindromes are ["battab","tabbat"]
```

Example 3:

```
Input: words = ["a",""]
Output: [[0,1],[1,0]]
Explanation: The palindromes are ["a","a"]
```

Constraints:

- 1 <= words.length <= 5000
- 0 <= words[i].length <= 300
- words[i] consists of lowercase English letters.

>> Brute Force [Trie] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def is_palindrome(self, s):
        # Brute Force O(n*m) - linear scan through all words
        matches = []
        for word in words:
            if word.startswith(prefix):
                matches.append(word)
        return matches
```

Python

Python


```python
# Python implementation for Palindrome Pairs

def is_palindrome(s):
    # Helper function to check if a string is a palindrome
    return s == s[::-1]

def palindromePairs(words):
    result = []
    # Create a dictionary mapping each word to its index
    word_dict = {word: i for i, word in enumerate(words)}

    # Iterate over each word with its index
    for i, word in enumerate(words):
        # For each possible split position (including edge empty string splits)
        for j in range(len(word) + 1):
            prefix = word[:j]
            suffix = word[j:]

            # If prefix is palindrome, check for reversed suffix in dictionary
            if is_palindrome(prefix):
                reversed_suffix = suffix[::-1]
                # Ensure that the reversed_suffix exists and is not the same word
                if reversed_suffix in word_dict and word_dict[reversed_suffix] != i:
                    result.append([word_dict[reversed_suffix], i])


            # Check the suffix; avoid duplicate pairs when j equals len(word) because
            # the case when suffix is empty was handled in the first condition
            if j != len(word) and is_palindrome(suffix):
                reversed_prefix = prefix[::-1]
                if reversed_prefix in word_dict and word_dict[reversed_prefix] != i:
                    result.append([i, word_dict[reversed_prefix]])

    return result

# Example usage:
words = ["abcd", "dcba", "lls", "s", "sssll"]
print(palindromePairs(words)) # Expected Output: [[0,1],[1,0],[3,2],[2,4]]
```

Python

```python
class Solution:
    def palindromePairs(self, words: list[str]) -> list[list[int]]:
        ans = []
        dict = {word[::-1]: i for i, word in enumerate(words)}

        for i, word in enumerate(words):
          if "" in dict and dict[""] != i and word == word[::-1]:
            ans.append([i, dict[""]])

          for j in range(1, len(word) + 1):
            l = word[:j]
            r = word[j:]
            if l in dict and dict[l] != i and r == r[::-1]:
              ans.append([i, dict[l]])
            if r in dict and dict[r] != i and l == l[::-1]:
              ans.append([dict[r], i])

        return ans
```


Python

```python
class Solution:
    def palindromePairs(self, words: List[str]) -> List[List[int]]:
        d = {w: i for i, w in enumerate(words)}
        ans = []
        for i, w in enumerate(words):
            for j in range(len(w) + 1):
                a, b = w[:j], w[j:]
                ra, rb = a[::-1], b[::-1]
                if ra in d and d[ra] != i and b == rb:
                    ans.append([i, d[ra]])
                if j and rb in d and d[rb] != i and a == ra:
                    ans.append([d[rb], i])
        return ans
```

Python

```python
class Solution:
    def palindromePairs(self, words: List[str]) -> List[List[int]]:
        d = {w: i for i, w in enumerate(words)}
        ans = []
        for i, w in enumerate(words):
            for j in range(len(w) + 1):
                a, b = w[:j], w[j:]
                ra, rb = a[::-1], b[::-1]
                if ra in d and d[ra] != i and b == rb:
                    ans.append([i, d[ra]])
                if j and rb in d and d[rb] != i and a == ra:
                    ans.append([d[rb], i])
        return ans
```

[*] Trie — Pattern Solution (stored optimal)

Python


```python
# Python implementation for Palindrome Pairs

def is_palindrome(s):
    # Helper function to check if a string is a palindrome
    return s == s[::-1]

def palindromePairs(words):
    result = []
    # Create a dictionary mapping each word to its index
    word_dict = {word: i for i, word in enumerate(words)}

    # Iterate over each word with its index
    for i, word in enumerate(words):
        # For each possible split position (including edge empty string splits)
        for j in range(len(word) + 1):
            prefix = word[:j]
            suffix = word[j:]

            # If prefix is palindrome, check for reversed suffix in dictionary
            if is_palindrome(prefix):
                reversed_suffix = suffix[::-1]
                # Ensure that the reversed_suffix exists and is not the same word
                if reversed_suffix in word_dict and word_dict[reversed_suffix] != i:
                    result.append([word_dict[reversed_suffix], i])


            # Check the suffix; avoid duplicate pairs when j equals len(word) because
            # the case when suffix is empty was handled in the first condition
            if j != len(word) and is_palindrome(suffix):
                reversed_prefix = prefix[::-1]
                if reversed_prefix in word_dict and word_dict[reversed_prefix] != i:
                    result.append([i, word_dict[reversed_prefix]])

    return result

# Example usage:
words = ["abcd", "dcba", "lls", "s", "sssll"]
print(palindromePairs(words)) # Expected Output: [[0,1],[1,0],[3,2],[2,4]]
```

#588 HARD

Design In-Memory File System

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Design · Trie

Lists: Grind 169 | Premium 98

Problem:

Design a data structure that simulates an in-memory file system.

Implement the FileSystem class:

- FileSystem() Initializes the object of the system.
- List<String> ls(String path) If path is a file path, returns a list that only contains this file's name.
- If path is a directory path, returns the list of file and directory names in this directory.
- If filePath does not exist, creates that file containing given content.

- If filePath already exists, appends the given content to original content.

Example 1:

Operation	Output	Explanation
FileSystem fs = new FileSystem()	null	The constructor returns nothing.
fs.ls("/")	[]	Initially, directory `/` has nothing. So return empty list.
fs.mkdir("/a/b/c")	null	Create directory `a` in directory `/`. Then create directory `b` in directory `a`. Finally, create directory `c` in directory `b`.
fs.addContentToFile("/a/b/c/d","hello")	null	Create a file named `d` with content `"hello"` in directory `/a/b/c`.
fs.ls("/")	["a"]	Only directory `a` is in directory `/`.
fs.readContentFromFile("/a/b/c/d")	"hello"	Output the file content.

```
Input
["FileSystem", "ls", "mkdir", "addContentToFile", "ls", "readContentFromFile"]
[[], ["/"], ["/a/b/c"], ["/a/b/c/d", "hello"], ["/"], ["/a/b/c/d"]]
Output
[null, [], null, null, ["a"], "hello"]

Explanation
FileSystem fileSystem = new FileSystem();
fileSystem.ls("/"); // return []
fileSystem.mkdir("/a/b/c");
fileSystem.addContentToFile("/a/b/c/d", "hello");
fileSystem.ls("/"); // return ["a"]
fileSystem.readContentFromFile("/a/b/c/d"); // return "hello"
```

Constraints:

- 1 <= path.length, filePath.length <= 100
- path and filePath are absolute paths which begin with '/' and do not end with '/' except that the path is just "/".
- You can assume that all directory names and file names only contain lowercase letters, and the same names will not exist in the same directory.
- You can assume that all operations will be passed valid parameters, and users will not attempt to retrieve file content or list a directory or file that does not exist.
- You can assume that the parent directory for the file in addContentToFile will exist.
- 1 <= content.length <= 50
- At most 300 calls will be made to ls, mkdir, addContentToFile, and readContentFromFile.

Python

Python


```python
class Node:
    def __init__(self):
        # Indicates whether this node represents a file or a directory
        self.is_file = False
        # For directories, mapping from name to Node
        self.children = {}
        # For files, store content
        self.content = ""

class FileSystem:

    def __init__(self):
        # Root directory node initialization
        self.root = Node()

    def ls(self, path: str) -> list:
        # Traverse the path to obtain the target node
        node, name = self.traverse(path)
        # If the node is a file, return the single element list containing its name
        if node.is_file:
            return [name]
        # Otherwise, list all keys (names) in the directory, sorted lexicographically
        return sorted(node.children.keys())

    def mkdir(self, path: str) -> None:

        # Create directory by traversing the path and creating nodes if necessary
        self.traverse(path)

    def addContentToFile(self, filePath: str, content: str) -> None:
        # Traverse to the file's node, if it does not exist, create necessary nodes along the path
        node, name = self.traverse(filePath)
        # If not already a file, mark the node as a file
        if not node.is_file:
            node.is_file = True
            node.content = ""
        # Append the new content to the file's content
        node.content += content

    def readContentFromFile(self, filePath: str) -> str:
        # Traverse to the file's node
        node, _ = self.traverse(filePath)
        # Return the content of the file
        return node.content

    def traverse(self, path: str):
        # Start from the root node
        node = self.root
        if path == "/":
            # Special case for root path
            return node, ""
```

```python
        # Split the path by "/" and traverse the nodes
        parts = path.split("/")[1:]
        for part in parts:
            # Create the node if it does not exist
            if part not in node.children:
                node.children[part] = Node()
            node = node.children[part]
        # Return the target node and its name (last part of the path)
        return node, parts[-1]

# Example usage:
# fs = FileSystem()
# print(fs.ls("/")) # printc []
# fs.mkdir("/a/b/c")
# fs.addContentToFile("/a/b/c/d", "hello")
# print(fs.ls("/")) # printc ['a']
# print(fs.readContentFromFile("/a/b/c/d")) # prints "hello"
```

Python

```python
'''
>>> "/a/b/c".split('/')
['', 'a', 'b', 'c']
'''

class Trie:
    def __init__(self):
        self.name = None
        self.isFile = False
        self.content = []
        self.children = {}

    def insert(self, path, isFile):
        node = self
        ps = path.split('/')
        for p in ps[1:]:
            if p not in node.children:
                node.children[p] = Trie()
            node = node.children[p]
        node.isFile = isFile
        if isFile:
            node.name = ps[-1]
        return node

    def search(self, path):
```



```python
        node = self
        if path == '/':
            return node
        ps = path.split('/')
        for p in ps[1:]:
            if p not in node.children:
                return None
            node = node.children[p]
        return node


class FileSystem:
    def __init__(self):
        self.root = Trie()

    def ls(self, path: str) -> List[str]:
        node = self.root.search(path)
        if node is None:
            return []
        if node.isFile:
            return [node.name]
        return sorted(node.children.keys())

    def mkdir(self, path: str) -> None:
        self.root.insert(path, False)


    def addContentToFile(self, filePath: str, content: str) -> None:
        node = self.root.insert(filePath, True)
        node.content.append(content)

    def readContentFromFile(self, filePath: str) -> str:
        node = self.root.search(filePath)
        return ''.join(node.content)


# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem()
# param_1 = obj.ls(path)
# obj.mkdir(path)
# obj.addContentToFile(filePath,content)
# param_4 = obj.readContentFromFile(filePath)
```

[³] Trie — Pattern Solution (matched from community answers)

Python


```python
'''
>>> "/a/b/c".split('/')
['', 'a', 'b', 'c']
'''

class Trie:
    def __init__(self):
        self.name = None
        self.isFile = False
        self.content = []
        self.children = {}

    def insert(self, path, isFile):
        node = self
        ps = path.split('/')
        for p in ps[1:]:
            if p not in node.children:
                node.children[p] = Trie()
            node = node.children[p]
        node.isFile = isFile
        if isFile:
            node.name = ps[-1]
        return node

    def search(self, path):
```



```python
        node = self
        if path == '/':
            return node
        ps = path.split('/')
        for p in ps[1:]:
            if p not in node.children:
                return None
            node = node.children[p]
        return node


class FileSystem:
    def __init__(self):
        self.root = Trie()

    def ls(self, path: str) -> List[str]:
        node = self.root.search(path)
        if node is None:
            return []
        if node.isFile:
            return [node.name]
        return sorted(node.children.keys())

    def mkdir(self, path: str) -> None:
        self.root.insert(path, False)


    def addContentToFile(self, filePath: str, content: str) -> None:
        node = self.root.insert(filePath, True)
        node.content.append(content)

    def readContentFromFile(self, filePath: str) -> str:
        node = self.root.search(filePath)
        return ''.join(node.content)


# Your FileSystem object will be instantiated and called as such:
# obj = FileSystem()
# param_1 = obj.ls(path)
# obj.mkdir(path)
# obj.addContentToFile(filePath,content)
# param_4 = obj.readContentFromFile(filePath)
```

#642 HARD

Design Search Autocomplete System

LeetDoocs · SimplyLeet · WallCC · LeetCode · Editorial

String · Design · Trie · Data Stream · Heap

Lists: Premium 98

Problem:

Design a search autocomplete system for a search engine. Users may input a sentence (at least one word and end with a special character '#').

You are given a string array sentences and an integer array times both of length n where sentences[i] is a previously typed sentence and times[i] is the corresponding number of times the sentence was typed. For each input character except '#', return the top 3 historical hot sentences that have the same prefix as the part of the sentence already typed.

Here are the specific rules:

- The hot degree for a sentence is defined as the number of times a user typed the exactly same sentence before.
- The returned top 3 hot sentences should be sorted by hot degree (The first is the hottest one). If several sentences have the same hot degree, use ASCII-code order (smaller one appears first).
- If less than 3 hot sentences exist, return as many as you can.
- When the input is a special character, it means the sentence ends, and in this case, you need to return an empty list.

Implement the AutocompleteSystem class:

- AutocompleteSystem(String[] sentences, int[] times) Initializes the object with the sentences and times arrays.
- List<String> input(char c) This indicates that the user typed the character c. Returns an empty array [] if c == '#' and stores the inputted sentence in the system.
- Returns the top 3 historical hot sentences that have the same prefix as the part of the sentence already typed. If there are fewer than 3 matches, return them all.

Example 1:


```
Input
["AutocompleteSystem", "input", "input", "input", "input"]
[[["i love you", "island", "iroman", "i love leetcode"], [5, 3, 2, 2]], ["i"], [" "], ["a"],
["#"]]
Output
[null, ["i love you", "island", "i love leetcode"], ["i love you", "i love leetcode"], [],
[]]

Explanation
AutocompleteSystem obj = new AutocompleteSystem(["i love you", "island", "iroman", "i love
leetcode"], [5, 3, 2, 2]);
obj.input("i"); // return ["i love you", "island", "i love leetcode"]. There are four
sentences that have prefix "i". Among them, "ironman" and "i love leetcode" have same hot
degree. Since ' ' has ASCII code 32 and 'r' has ASCII code 114, "i love leetcode" should be
in front of "ironman". Also we only need to output top 3 hot sentences, so "ironman" will be
ignored.
obj.input(" "); // return ["i love you", "i love leetcode"]. There are only two sentences
that have prefix "i ".
obj.input("a"); // return []. There are no sentences that have prefix "i a".
obj.input("#"); // return []. The user finished the input, the sentence "i a" should be saved
as a historical sentence in system. And the following input will be counted as a new search.
```

Constraints:

- n == sentences.length
- n == times.length
- 1 <= n <= 100
- 1 <= sentences[i].length <= 100
- 1 <= times[i] <= 50
- c is a lowercase English letter, a hash '#', or space ' '.
- Each tested sentence will be a sequence of characters c that end with the character '#'.
- Each tested sentence will have a length in the range [1, 200].
- The words in each input sentence are separated by single spaces.
- At most 5000 calls will be made to input.

Python

Python


```python
class TrieNode:
    def __init__(self):
        # Mapping from character to TrieNode
        self.children = {}
        # Dictionary mapping sentence to its frequency count
        self.counts = {}

class AutocompleteSystem:
    def __init__(self, sentences, times):
        self.root = TrieNode()
        self.current_node = self.root
        self.current_input = ""
        # Initialize the trie with the given sentences and their times
        for sentence, time in zip(sentences, times):
            self.addSentence(sentence, time)

    def addSentence(self, sentence, freq):
        node = self.root
        for char in sentence:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
            # Update sentence frequency at each node
            node.counts[sentence] = node.counts.get(sentence, 0) + freq

    def input(self, c):
        results = []
        if c == '#':
            # End of input: add current sentence to Trie
            self.addSentence(self.current_input, 1)
            # Reset the current input and traversal pointer
            self.current_input = ""
            self.current_node = self.root
            return results

        self.current_input += c
        # Move to the next Trie node if it exists
        if self.current_node:
            self.current_node = self.current_node.children.get(c, None)
        if not self.current_node:
            return results

        # Retrieve sentences and sort them by frequency (descending) and lexicographical order (ascending)
        data = list(self.current_node.counts.items())
        data.sort(key=lambda x: (-x[1], x[0]))
        for i in range(min(3, len(data))):
            results.append(data[i][0])
        return results

# Example usage:

# auto = AutocompleteSystem(["i love you", "island", "iroman", "i love leetcode"], [5, 3, 2, 2])
# print(auto.input("i"))
# print(auto.input(" "))
# print(auto.input("a"))
# print(auto.input("#"))
```

Python

```python
class TrieNode:
    def __init__(self):
        self.children: dict[str, TrieNode] = {}
        self.s: str | None = None
        self.time = 0
        self.top3: list[TrieNode] = []

    def __lt__(self, other):
        if self.time == other.time:
            return self.s < other.s
        return self.time > other.time

    def update(self, node) -> None:
        if node not in self.top3:
            self.top3.append(node)
        self.top3.sort()
        if len(self.top3) > 3:
            self.top3.pop()


class AutocompleteSystem:
    def __init__(self, sentences: list[str], times: list[int]):
        self.root = TrieNode()
        self.curr = self.root
        self.s: list[str] = []


        for sentence, time in zip(sentences, times):
            self._insert(sentence, time)

    def input(self, c: str) -> list[str]:
        if c == '#':
            self._insert(''.join(self.s), 1)
            self.curr = self.root
            self.s = []
            return []

        self.s.append(c)

        if self.curr:
            self.curr = self.curr.children.get(c, None)
        if not self.curr:
            return []
        return [node.s for node in self.curr.top3]

    def _insert(self, sentence: str, time: int) -> None:
        node = self.root
        for c in sentence:
            node = node.children.setdefault(c, TrieNode())
        node.s = sentence
        node.time += time


        leaf = node
        node: TrieNode = self.root
        for c in sentence:
            node = node.children[c]
            node.update(leaf)
```


Python

```python
class Trie:
    def __init__(self):
        # 26+1, extra 1 for space ' ' node
        self.children = [None] * 27
        self.v = 0
        self.w = ''

    def insert(self, w, t):
        node = self
        for c in w:
            idx = 26 if c == ' ' else ord(c) - ord('a')
            if node.children[idx] is None:
                node.children[idx] = Trie()
            node = node.children[idx]
        node.v += t
        node.w = w

    def search(self, pref):
        node = self
        for c in pref:
            idx = 26 if c == ' ' else ord(c) - ord('a')
            if node.children[idx] is None:
                return None
            node = node.children[idx]
        return node


class AutocompleteSystem:
    def __init__(self, sentences: List[str], times: List[int]):
        self.trie = Trie()
        for a, b in zip(sentences, times):
            self.trie.insert(a, b)
        self.t = [] # target

    def input(self, c: str) -> List[str]:
        def dfs(node):
            if node is None:
                return
            if node.v:
                res.append((node.v, node.w))
            for nxt in node.children:
                dfs(nxt)

        if c == '#':
            s = ''.join(self.t)
            self.trie.insert(s, 1)
            self.t = []
            return []

        res = []
```



```python
        self.t.append(c)
        node = self.trie.search(''.join(self.t))
        if node is None:
            return res
        dfs(node)
        res.sort(key=lambda x: (-x[0], x[1]))
        return [v[1] for v in res[:3]]


# Your AutocompleteSystem object will be instantiated and called as such:
# obj = AutocompleteSystem(sentences, times)
# param_1 = obj.input(c)
```

[*] Trie — Pattern Solution (matched from community answers)

Python

```python
class TrieNode:
    def __init__(self):
        # Mapping from character to TrieNode
        self.children = {}
        # Dictionary mapping sentence to its frequency count
        self.counts = {}

class AutocompleteSystem:
    def __init__(self, sentences, times):
        self.root = TrieNode()
        self.current_node = self.root
        self.current_input = ""
        # Initialize Trie with the given sentences and their times
        for sentence, time in zip(sentences, times):
            self.addSentence(sentence, time)

    def addSentence(self, sentence, freq):
        node = self.root
        for char in sentence:
            if char not in node.children:
                node.children[char] = TrieNode()
            node = node.children[char]
            # Update sentence frequency at each node
            node.counts[sentence] = node.counts.get(sentence, 0) + freq
```



```python
    def input(self, c):
        results = []
        if c == '#':
            # End of input: add current sentence to Trie
            self.addSentence(self.current_input, 1)
            # Reset the current input and traversal pointer
            self.current_input = ""
            self.current_node = self.root
            return results

        self.current_input += c
        # Move to the next Trie node if it exists
        if self.current_node:
            self.current_node = self.current_node.children.get(c, None)
        if not self.current_node:
            return results

        # Retrieve sentences and sort them by frequency (descending) and lexicographical order (ascending)
        data = list(self.current_node.counts.items())
        data.sort(key=lambda x: (-x[1], x[0]))
        for i in range(min(3, len(data))):
            results.append(data[i][0])
        return results

# Example usage:

# auto = AutocompleteSystem(["i love you", "island", "iroman", "i love leetcode"], [5, 3, 2, 2])
# print(auto.input("i"))
# print(auto.input(" "))
# print(auto.input("a"))
# print(auto.input("#"))
```


Quick Review — Trie 12 questions · insights · complexity · solution

#14 Longest Common Prefix [Easy]

Key Insights

- Compare characters of the strings one column at a time (vertical scanning).
- The longest possible prefix is limited by the length of the shortest string in the array.
- As soon as a mismatch is found, the prefix up to that point is the answer.
- Handle edge cases such as an empty input array or strings with no common prefix.

Space & Time

Time Complexity: O(N*M) where N is the number of strings and M is the length of the shortest string.

Space Complexity: O(1) additional space (ignoring output space).

Solution

We use the vertical scanning approach:

- If the input array is empty, return an empty string. - Iterate character by character over the first string. - For each character, compare it with the corresponding character in each of the other strings. - If a mismatch or end of any string is encountered, return the substring from the start up to the current index. - If no mismatch is found, the entire first string is the common prefix.

This solution ensures that we only traverse the necessary part of the strings and stops early if a discrepancy is discovered.

#139 Word Break [Medium]

Key Insights

- Use Dynamic Programming (DP) to break the problem into subproblems.
- Create a DP array where dp[i] is true if the substring s[0:i] can be segmented using words from the dictionary.
- Leverage a hash set for fast look-up of dictionary words.
- For every index i, check all possible previous break points j such that dp[j] is true and s[j:i] is in the wordDict.
- The problem can also be approached with recursion and memoization, but DP is more straightforward for iterative implementation.

Space & Time

Time Complexity: O(n^2) in the worst-case scenario, where n is the length of s (each substring check is O(1) if using a hash set, and there could be O(n^2) checks overall).

Space Complexity: O(n) for the dp array and O(k) for the hash set where k is the number of words in the dictionary.

Solution

The solution uses a Dynamic Programming (DP) approach:

- Convert wordDict to a hash set for constant time lookup. - Create a boolean dp array of size n+1 (n is the length of s), where dp[i] indicates if s[0:i] can be segmented. - Set dp[0] to true because an empty string is always segmentable. - For each index i from 1 to n, iterate through all indices j from 0 to i: If dp[j] is true and the substring s[j:i] is in the dictionary, then set dp[i] to true. Break early from the inner loop when a valid break is found. - Return dp[n] as the result.

This method efficiently builds up the solution by utilizing previous results to determine if the entire string can be segmented.

#208 Implement Trie (Prefix Tree) [Medium]

Key Insights

- Use a tree-like structure where each node represents a character.
- Each node stores its children in a dictionary (or hash map) for O(1) average time access.

- Mark the end of a word with a boolean flag.
- Insertion is done character by character, creating new nodes as needed.
- Searching and checking prefixes follow traversals down the tree.

Space & Time

Time Complexity:

- Insert: O(m) where m is the length of the word.
- Search: O(m)
- startsWith: O(m)

Space Complexity:

- O(N * M) in the worst-case scenario, where N is the number of words and M is the average length of the words, due to storing each character in the Trie.

Solution

We implement the Trie using a node-based approach. Each TrieNode contains a hash map (or dictionary) of its children and a boolean indicating if the node marks the end of a word. The Trie class uses these nodes to build out the words character by character. The insert method traverses and builds the necessary nodes; the search method verifies that all characters exist in order and that the final node marks the end of a word; the startsWith method checks if the prefix exists in the Trie. No tricky edge cases exist beyond handling null or empty inputs, as constraints guarantee valid lowercase strings.

#211 Design Add and Search Words Data Structure [Medium]

Key Insights

- Utilize a Trie (prefix tree) to efficiently store words.
- Each node in the Trie contains a dictionary for its children and a flag indicating the end of a word.
- The search operation is implemented with a depth-first search (DFS) approach.
- When the search query contains a dot, explore all possible child nodes.

Space & Time

Time Complexity:

- addWord: O(w), where w is the length of the word.
- search: Worst-case O(26^w) for words with multiple wildcards, but practical performance is better given constraints.

Space Complexity: O(n * w), where n is the number of words and w is the average word length.

Solution

The WordDictionary is implemented using a Trie. For adding, we insert each character of the word into the Trie. For searching, we use DFS. When encountering a wildcard '.', we recursively search all children nodes. This structure combined with DFS efficiently addresses the wildcard matching and ensures good performance for multiple operations.

#616 Add Bold Tag in String [Medium]

Key Insights

- We need to identify every occurrence of each word in s.
- Overlapping or consecutive intervals should be merged to form one continuous bold region.
- Use an auxiliary data structure (e.g., a boolean array) to mark positions that should be bolded.
- Once the positions are marked, build the final string by inserting bold tags at the appropriate boundaries.

Space & Time

Time Complexity: O(n * m * k) where n is the length of s, m is the number of words, and k is the average length of the words (due to substring matching).

Space Complexity: O(n) for the auxiliary boolean array used to mark bold positions.

Solution

The solution involves two main steps:

- Identify and mark the indices in the string s that are part of any matching word. We do this by iterating over each word and marking every occurrence's start and end in a boolean array. - Construct the final string by traversing the boolean array. When encountering a segment of consecutive true values, wrap that segment with and tags.

The key idea is to merge intervals by marking every index that should be bold and then using a single pass to insert tags only when transitioning from non-bold to bold and vice versa.

#692 Top K Frequent Words [Medium]

Key Insights

- Count the frequency of each unique word using a hash table or dictionary.
- Use a min-heap (priority queue) to keep track of the top k frequent words. In the heap, the ordering is based on frequency; if frequencies are equal, order by lexicographical order (inverted condition for min-heap).
- Alternatively, you can sort the unique words based on frequency and lexicographical order and then pick the top k.
- For an efficient solution, use a min-heap that maintains size k achieving O(n log k) time complexity.

Space & Time

Time Complexity: O(n log k) when using a heap, where n is the number of unique words.

Space Complexity: O(n) for storing the frequency counts and the heap.

Solution

We first calculate the frequency of each word using a hash table (or dictionary). Then, we use a min-heap that stores pairs of (frequency, word). The comparator for the heap is defined such that the root of the heap is the word with the smallest frequency; if two words have the same frequency, the word with the lexicographically larger value is considered "smaller" in our ordering so that it gets popped when the heap size exceeds k. After processing all words, the heap contains the top k frequent words. Finally, the heap elements are extracted and reversed (if needed) to produce the final list sorted from highest to lowest frequency with lexicographic tie-breaking.

#720 Longest Word in Dictionary [Medium]

Key Insights

- The word must be built character by character; hence every prefix (removing one character from the end at a time) must exist in the dictionary.
- Sorting the input list can help in ensuring that smaller words (prefixes) are processed first.
- Using a hash set to store valid words enables rapid prefix checks.
- Lexicographical order plays a role when words have the same length.

Space & Time

Time Complexity: O(n * L + n log n) where n is the number of words and L is the maximum length of a word (n * L for prefix checking, and n log n for sorting).

Space Complexity: O(n * L) for storing words in the set.

Solution

The approach begins by sorting the list of words. Sorting is done primarily by word length and secondarily by lexicographical order so that when we iterate through the words, all possible prefixes for a word have already been considered. Initialize a set to keep track of valid words that can be built one character at a time. For each word in the sorted list, check if all its prefixes (word without the last character progressively) exist in the set. If a word qualifies, add it to the set and check if it should be the new result based on length and lexicographical order. The set lookup ensures that checking each prefix is efficient.

#1166 Design File System [Medium]

Key Insights

- Use a data structure (like a hash map) to store paths and associated values.

- To create a new path, check that the path does not already exist.
- Ensure the immediate parent path exists before creation.
- When retrieving a value, simply look it up in the storage structure.
- Splitting the path by "/" allows easy identification of the parent.

Space & Time

Time Complexity:

- createPath: O(n) per call where n is the number of segments in the path (to process and validate parent path).
- get: O(1) per call if using a hash map.

Space Complexity: O(m * L) where m is the number of paths stored and L is the average length of a path.

Solution

The solution uses a hash map (dictionary) to store file paths along with their associated values. For the createPath method, the algorithm:

- Checks if the path already exists. - Extracts the parent path by removing the last segment. - Validates the existence of the parent path. - Inserts the new path with its corresponding value if validation passes. For the get method, the algorithm simply returns the stored value if the path exists; otherwise, it returns -1. This approach ensures efficient lookup and validation using string manipulation and hash map operations.

#212 Word Search II [Hard]

Key Insights

- Use Depth-First Search (DFS) to explore possible words starting from each board cell.
- Build a Trie (prefix tree) to store the words for O(1) prefix checks and efficient pruning.
- Mark visited cells and restore them after DFS to avoid revisiting in the current search path.
- Prune search paths early if the current prefix is not part of any word in the Trie.

Space & Time

Time Complexity: O(m * n * 4^L) in the worst case, where L is the maximum word length; however, using a Trie prunes unnecessary paths.

Space Complexity: O(K) for the Trie storage, where K is the total number of letters in all words, plus O(m * n) auxiliary space for recursion and board marking.

Solution

We first insert all words into a Trie data structure to allow quick lookups of prefixes. Then, for each cell in the board, we perform DFS. During DFS, if the current path matches a Trie node that represents a complete word, we add that word to the result list and mark it as found to avoid duplicates. We continue exploring adjacent cells (up, down, left, right) while marking the cell as visited. After exploring, we restore the original letter for further DFS calls. This combination of Trie and DFS with backtracking efficiently finds valid words on the board.

#336 Palindrome Pairs [Hard]

Key Insights

- A palindrome is a string that reads the same forwards and backwards.
- For any given word, if you split it into two parts, one part (either prefix or suffix) might be a palindrome.
- If the other part's reverse exists elsewhere in the array, then that reverse can be concatenated to yield a palindrome.
- Carefully handle edge cases like empty strings, since an empty string is a palindrome by default.

Space & Time

Time Complexity: O(n * k^2) in the worst case where n is the number of words and k is the maximum word length, but with the palindrome checking optimized this can be considered O(sum of word lengths): Each word is processed once and each split is checked in O(k) time.

Space Complexity: O(n * k) due to the hashmap storing all words and their indices.

Solution

We use a hashmap (dictionary) to store each word and its corresponding index. For every word, we iterate through every possible split position to divide the word into a prefix and a suffix. For each split:

- If the prefix is a palindrome, we look for the reverse of the suffix in the hashmap. If found (and not the same word), then combining the reverse with the current word forms a palindrome. - Similarly, if the suffix is a palindrome, we look for the reverse of the prefix.

We must be cautious with edge cases, especially when either part is empty, as the empty string is considered a palindrome. This method avoids checking every possible pair explicitly and leverages string reversal and palindrome checks to achieve efficiency.

#588 Design In-Memory File System [Hard]

Key Insights

- Use a tree (trie) to represent directories and files.
- Each node can be either a directory or a file; directories maintain a mapping of names to child nodes.
- For ls, if the given path is a file, return only its name. If it's a directory, return the sorted list of names of its children.
- mkdir must create all intermediate directories if they do not already exist.
- File nodes store content and support content appending.

Space & Time

Time Complexity:

- ls: O(k log k) where k is the number of entries in the directory (due to sorting), or O(n) if traversing a long path.
- mkdir, addContentToFile, readContentFromFile: O(d) where d is the depth of the file path.

Space Complexity:

- O(T) where T is the total number of characters stored in directory names and file contents.

Solution

The solution uses a tree-based approach where each node represents either a directory or a file. Each directory node stores its children in a hash table (or dictionary) mapping names to node objects. File nodes store their content as a string and carry a flag indicating they are files. For the ls operation, check if the node is a file—if so, return its name; otherwise, list and sort the names of all children in that directory. The mkdir operation traverses (or creates) nodes for each segment of the provided path. The addContentToFile method accesses or creates the file node and appends the content. Finally, readContentFromFile retrieves the stored content from the file node.

#642 Design Search Autocomplete System [Hard]

Key Insights

- Use a Trie to efficiently store and retrieve sentences based on their prefixes.
- Maintain a mapping at each Trie node to record sentences (or their frequencies) that pass through that node.
- Use sorting or a min-heap to select the top three sentences based on their frequency (hot degree) and lexicographical order if frequencies tie.
- On receiving '#', update the Trie with the current sentence and reset the current search state.

Space & Time

Time Complexity: O(L * log k) per character input query, where L is the length of the current prefix and k is 3 (constant factor), thus ensuring efficient autocompletion.

Space Complexity: O(N * L), where N is the number of unique sentences and L is the average sentence length, for maintaining the Trie.

Solution

The solution revolves around building a Trie where each node contains:

- A mapping to its child nodes. - A hash map that records all sentences passing through the node along with their occurrence counts.

For each character entered (except '#'), the system:

- Traverses the Trie to find the current prefix node. - Retrieves stored sentences at that node and sorts them based on the following criteria: descending by frequency and ascending lexicographically if frequencies are equal. - Returns the top three sentences from the sorted result.

When the user inputs '#', the current sentence (tracked as user input so far) is added or its frequency updated in the Trie, and the input state resets.

#11 Math — 12 questions · #11 of 21

#9 EASY

Palindrome Number

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math

Lists: Grind 169

Problem:

Given an integer x, return true if x is a palindrome, and false otherwise.

Example 1:

```
Input: x = 121
Output: true
Explanation: 121 reads as 121 from left to right and from right to left.
```

Example 2:

```
Input: x = -121
Output: false
Explanation: From left to right, it reads -121. From right to left, it becomes 121-.
Therefore it is not a palindrome.
```

Example 3:

```
Input: x = 10
Output: false
Explanation: Reads 01 from right to left. Therefore it is not a palindrome.
```

Constraints:

- -231 <= x <= 231 - 1

Follow up: Could you solve it without converting the integer to a string?

>> Brute Force [Math] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def isPalindrome(self, x):
        # Brute force - iterate all candidates
        for i in range(2, n + 1):
            is_valid = all(i % j != 0 for j in range(2, i))
            if is_valid: pass
        return 0
```

Python

Python


```python
# Function to check if a number is a palindrome without converting to a string.
def isPalindrome(x):
    # Negative numbers and numbers ending with 0 (but not 0 itself) are not palindromic.
    if x < 0 or (x % 10 == 0 and x != 0):
        return False
    reversed_half = 0
    # Process: Reverse half of the number.
    while x > reversed_half:
        # Add the last digit of x to reversed_half.
        reversed_half = reversed_half * 10 + x % 10
        # Remove the last digit from x.
        x //= 10
    # For numbers with odd digits, reversed_half // 10 removes the middle digit.
    return x == reversed_half or x == reversed_half // 10

# Example usage:
print(isPalindrome(121)) # Output: True
print(isPalindrome(-121)) # Output: False
print(isPalindrome(10)) # Output: False
```

Python

```python
class Solution:
    def isPalindrome(self, x: int) -> bool:
        if x < 0:
            return False

        rev = 0
        y = x

        while y:
            rev = rev * 10 + y % 10
            y //= 10

        return rev == x
```

Python

```python
class Solution:
    def isPalindrome(self, x: int) -> bool:
        if x < 0 or (x and x % 10 == 0):
            return False
        y = 0
        while y < x:
            y = y * 10 + x % 10
            x //= 10
        return x in (y, y // 10)
```

Python

```python
class Solution:
    def isPalindrome(self, x: int) -> bool:
        if x < 0 or (x and x % 10 == 0):
            return False
        y = 0
        while y < x:
            y = y * 10 + x % 10
            x //= 10
        return x in (y, y // 10)
```

[*] Math — Pattern Solution (stored optimal)

Python

```python
# Function to check if a number is a palindrome without converting to a string.
def isPalindrome(x):
    # Negative numbers and numbers ending with 0 (but not 0 itself) are not palindromic.
    if x < 0 or (x % 10 == 0 and x != 0):
        return False
    reversed_half = 0
    # Process: Reverse half of the number.
    while x > reversed_half:
        # Add the last digit of x to reversed_half.
        reversed_half = reversed_half * 10 + x % 10
        # Remove the last digit from x.
        x //= 10
    # For numbers with odd digits, reversed_half // 10 removes the middle digit.
    return x == reversed_half or x == reversed_half // 10

# Example usage:
print(isPalindrome(121)) # Output: True
print(isPalindrome(-121)) # Output: False
print(isPalindrome(10)) # Output: False
```

#13 EASY

Roman to Integer

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · Math · String

Lists: Grind 169

Problem:

Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M.

```
Symbol Value
I 1
V 5
X 10
L 50
C 100
D 500
M 1000
```

For example, 2 is written as II in Roman numeral, just two ones added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

- I can be placed before V (5) and X (10) to make 4 and 9.
- X can be placed before L (50) and C (100) to make 40 and 90.
- C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

Example 1:

```
Input: s = "III"
Output: 3
Explanation: III = 3.
```

Example 2:

```
Input: s = "LVIII"
Output: 58
Explanation: L = 50, V= 5, III = 3.
```

Example 3:

```
Input: s = "MCMXCIV"
Output: 1994
Explanation: M = 1000, CM = 900, XC = 90 and IV = 4.
```

Constraints:

- 1 <= s.length <= 15
- s contains only the characters ('I', 'V', 'X', 'L', 'C', 'D', 'M').
- It is guaranteed that s is a valid roman numeral in the range [1, 3999].

>> Brute Force [Math] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def romanToInt(self, s):
        # Brute Force - Iterate all candidates
        for i in range(2, n + 1):
            is_valid = all(i % j != 0 for j in range(2, i))
            if is_valid: pass
        return 0
```

Python

Python

```python
# Define a function to convert Roman numeral to integer
def romanToInt(s):
    # Map each Roman numeral to its integer value
    roman_values = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}
    total = 0 # Initialize the total integer value

    # Loop over the string using index
    for i in range(len(s)):
        # If this is not the last character and the current value is less than the next value,
        # subtract the current value; otherwise, add the current value.
        if i + 1 < len(s) and roman_values[s[i]] < roman_values[s[i + 1]]:
            total -= roman_values[s[i]]
        else:
            total += roman_values[s[i]]
    return total

# Example usage:
print(romanToInt("MCMXCIV")) # Output: 1994
```

Python


```python
class Solution:
    def romanToInt(self, s: str) -> int:
        ans = 0
        roman = {'I': 1, 'V': 5, 'X': 10, 'L': 50,
                 'C': 100, 'D': 500, 'M': 1000}

        for a, b in zip(s, s[1:]):
          if roman[a] < roman[b]:
            ans -= roman[a]
          else:
            ans += roman[a]

        return ans + roman[s[-1]]
```

Python

```python
class Solution:
    def romanToInt(self, s: str) -> int:
        d = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}
        return sum((-1 if d[a] < d[b] else 1) * d[a] for a, b in pairwise(s)) + d[s[-1]]
```

Python

```python
class Solution:
    def romanToInt(self, s: str) -> int:
        d = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}
        return sum((-1 if d[a] < d[b] else 1) * d[a] for a, b in pairwise(s)) + d[s[-1]]
```

[*] Math — Pattern Solution (stored optimal)

Python

```python
# Define a function to convert Roman numeral to integer
def romanToInt(s):
    # Map each Roman numeral to its integer value
    roman_values = {'I': 1, 'V': 5, 'X': 10, 'L': 50, 'C': 100, 'D': 500, 'M': 1000}
    total = 0 # Initialize the total integer value

    # Loop over the string using index
    for i in range(len(s)):
        # If this is not the last character and the current value is less than the next value,
        # subtract the current value; otherwise, add the current value.
        if i + 1 < len(s) and roman_values[s[i]] < roman_values[s[i + 1]]:
            total -= roman_values[s[i]]
        else:
            total += roman_values[s[i]]
    return total

# Example usage:
print(romanToInt("MCMXCIV")) # Output: 1994
```

#504 EASY

Base 7

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math

Lists: Denny Zhang

Problem:

Given an integer num, return a string of its base 7 representation.

Example 1:

```
Input: num = 100
Output: "202"
```

Example 2:

```
Input: num = -7
Output: "-10"
```

Constraints:

- -107 <= num <= 107

>> Brute Force [Math] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def convertToBase7(self, num):
        # Brute Force - Iterate all candidates
        for i in range(2, n + 1):
            is_valid = all(i % j != 0 for j in range(2, i))
            if is_valid: pass
        return 0
```

Python

Python

```python
# Python solution for converting an integer to its base 7 string representation.
def convertToBase7(num):
    # Special case: If num is 0, return "0"
    if num == 0:
        return "0"

    # Flag to check if the number is negative
    negative = num < 0
    # Work with the absolute value of the number
    num = abs(num)

    # List to store base 7 digits
    digits = []

    # Process the number until it becomes 0
    while num > 0:
        # Get the remainder (digit in base 7)
        remainder = num % 7
        # Append the digit as string to the list
        digits.append(str(remainder))
        # Update num by integer division by 7
        num //= 7

    # Reverse the list to get digits in correct order
    base7 = ''.join(reversed(digits))
```



```python
    # If original number was negative, attach the minus sign
    return "-" + base7 if negative else base7

# Example usage:
print(convertToBase7(100)) # Expected output: "202"
print(convertToBase7(-7)) # Expected output: "-10"
```

Python

```python
class Solution:
    def convertToBase7(self, num: int) -> str:
        if num == 0:
            return '0'
        if num < 0:
            return '-' + self.convertToBase7(-num)
        ans = []
        while num:
            ans.append(str(num % 7))
            num //= 7
        return ''.join(ans[::-1])
```

Python

```python
class Solution:
    def convertToBase7(self, num: int) -> str:
        if num == 0:
            return '0'
        if num < 0:
            return '-' + self.convertToBase7(-num)
        ans = []
        while num:
            ans.append(str(num % 7))
            num //= 7
        return ''.join(ans[::-1])
```

[*] Math — Pattern Solution (stored optimal)

Python


```python
# Python solution for converting an integer to its base 7 string representation.
def convertToBase7(num):
    # Special case: If num is 0, return "0"
    if num == 0:
        return "0"

    # Flag to check if the number is negative
    negative = num < 0
    # Work with the absolute value of the number
    num = abs(num)

    # List to store base 7 digits
    digits = []

    # Process the number until it becomes 0
    while num > 0:
        # Get the remainder (digit in base 7)
        remainder = num % 7
        # Append the digit as string to the list
        digits.append(str(remainder))
        # Update num by integer division by 7
        num //= 7

    # Reverse the list to get digits in correct order
    base7 = ''.join(reversed(digits))

    # If original number was negative, attach the minus sign
    return "-" + base7 if negative else base7

# Example usage:
print(convertToBase7(100)) # Expected output: "202"
print(convertToBase7(-7)) # Expected output: "-10"
```


#1056 EASY

Confusing Number

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math

Lists: Premium 98

Problem:

A confusing number is a number that when rotated 180 degrees becomes a different number with each digit valid.

We can rotate digits of a number by 180 degrees to form new digits.

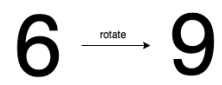
- When 0, 1, 6, 8, and 9 are rotated 180 degrees, they become 0, 1, 9, 8, and 6 respectively.
- When 2, 3, 4, 5, and 7 are rotated 180 degrees, they become invalid.

Note that after rotating a number, we can ignore leading zeros.

- For example, after rotating 8000, we have 0008 which is considered as just 8.

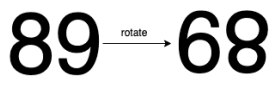
Given an integer n, return true if it is a confusing number, or false otherwise.

Example 1:




```
Input: n = 6
Output: true
Explanation: We get 9 after rotating 6, 9 is a valid number, and 9 != 6.
```

Example 2:




```
Input: n = 89
Output: true
Explanation: We get 68 after rotating 89, 68 is a valid number and 68 != 89.
```

Example 3:

```
Input: n = 11
Output: false
Explanation: We get 11 after rotating 11, 11 is a valid number but the value remains the
same, thus 11 is not a confusing number
```

Constraints:

- 0 <= n <= 109

Python

Python


```python
# Define the mapping for valid rotations
def is_confusing_number(n: int) -> bool:
    rotation_map = {'0': '0', '1': '1', '6': '9', '8': '8', '9': '6'}

    # Convert the number to a string to process each digit
    original_str = str(n)
    rotated_str = ""

    # Process each digit in reverse order to simulate 180 degree rotation
    for digit in reversed(original_str):
        # If digit is not valid for rotation, return false immediately
        if digit not in rotation_map:
            return False
        # Append the rotated digit to the rotated string
        rotated_str += rotation_map[digit]

    # Convert rotated string to an integer (leading zeros are ignored)
    rotated_num = int(rotated_str)

    # The number is confusing if the rotated number is different from the original number
    return rotated_num != n

# Example usage:
print(is_confusing_number(6)) # Expected output: True
print(is_confusing_number(89)) # Expected output: True

print(is_confusing_number(11)) # Expected output: False
```

Python

```python
class Solution:
  def confusingNumber(self, n: int) -> bool:
    s = str(n)
    rotated = {'0': '0', '1': '1', '6': '9', '8': '8', '9': '6'}
    rotatedNum = []

    for c in s[::-1]:
      if c not in rotated:
        return False
      rotatedNum.append(rotated[c])

    return ''.join(rotatedNum) != s
```

Python

```python
class Solution:
    def confusingNumber(self, n: int) -> bool:
        x, y = n, 0
        d = [0, 1, -1, -1, -1, -1, 9, -1, 8, 6]
        while x:
            x, v = divmod(x, 10)
            if d[v] < 0:
                return False
            y = y * 10 + d[v]
        return y != n
```

Python


```python
class Solution:
    def confusingNumber(self, n: int) -> bool:
        x, y = n, 0
        d = [0, 1, -1, -1, -1, -1, 9, -1, 8, 6]
        while x:
            x, v = divmod(x, 10)
            if d[v] < 0:
                return False
            y = y * 10 + d[v]
        return y != n
```

[*] Math — Pattern Solution (stored optimal)

Python

```python
# Define the mapping for valid rotations
def is_confusing_number(n: int) -> bool:
    rotation_map = {'0': '0', '1': '1', '6': '9', '8': '8', '9': '6'}

    # Convert the number to a string to process each digit
    original_str = str(n)
    rotated_str = ""

    # Process each digit in reverse order to simulate 180 degree rotation
    for digit in reversed(original_str):
        # If digit is not valid for rotation, return false immediately
        if digit not in rotation_map:
            return False
        # Append the rotated digit to the rotated string
        rotated_str += rotation_map[digit]

    # Convert rotated string to an integer (leading zeros are ignored)
    rotated_num = int(rotated_str)

    # The number is confusing if the rotated number is different from the original number
    return rotated_num != n

# Example usage:
print(is_confusing_number(6)) # Expected output: True
print(is_confusing_number(89)) # Expected output: True

print(is_confusing_number(11)) # Expected output: False
```

#1228 **EASY**

Missing Number in Arithmetic Progression

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Math

Lists: Premium 98

Problem:

In some array arr, the values were in arithmetic progression: the values arr[i + 1] − arr[i] are all equal for every 0 <= i < arr.length − 1.

A value from arr was removed that was not the first or last value in the array.

Given arr, return the removed value.

Example 1:

```
Input: arr = [5,7,11,13]
Output: 9
Explanation: The previous array was [5,7,9,11,13].
```

Example 2:

```
Input: arr = [15,13,12]
Output: 14
Explanation: The previous array was [15,14,13,12].
```

Constraints:

- 3 <= arr.length <= 1000
- 0 <= arr[i] <= 105
- The given array is guaranteed to be a valid array.

>> Brute Force [Math] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def missingNumber(self, arr):
        # Brute force - iterate all candidates
        for i in range(2, n + 1):
            is_valid = all(i % j != 0 for j in range(2, i))
            if is_valid: pass
        return 0
```

Python

Python

```python
# Function to find the missing number in the arithmetic progression
def missingNumber(arr):
    # Calculate the expected difference using the first and last elements
    n = len(arr)
    diff = (arr[-1] - arr[0]) // n # integer division because values are integers
    # Iterate over the array to find the place where the difference deviates from diff
    for i in range(len(arr) - 1):
        # If the difference between consecutive numbers is not equal to the expected diff
        if arr[i+1] - arr[i] != diff:
            # Return the missing number which should be current number + diff
            return arr[i] + diff
    # Return -1 if no missing number is found (should not happen given the problem constraints)
    return -1

# Example usage:
# print(missingNumber([5,7,11,13])) # Output should be 9
```

Python


```python
class Solution:
    def missingNumber(self, arr: list[int]) -> int:
    n = len(arr)
    delta = (arr[-1] - arr[0]) // n
    l = 0
    r = n - 1

    while l < r:
      m = (l + r) // 2
      if arr[m] == arr[0] + m * delta:
        l = m + 1
      else:
        r = m

    return arr[0] + l * delta
```

Python

```python
class Solution:
    def missingNumber(self, arr: List[int]) -> int:
        return (arr[0] + arr[-1]) * (len(arr) + 1) // 2 - sum(arr)
```

Python

```python
class Solution:
    def missingNumber(self, arr: List[int]) -> int:
        n = len(arr)
        d = (arr[-1] - arr[0]) // n
        for i in range(1, n):
            if arr[i] != arr[i - 1] + d:
                return arr[i - 1] + d
        return arr[0]
```

Python

```python
class Solution:
    def missingNumber(self, arr: List[int]) -> int:
        return (arr[0] + arr[-1]) * (len(arr) + 1) // 2 - sum(arr)
```

[*] Math — Pattern Solution (stored optimal)

Python

```python
# Function to find the missing number in the arithmetic progression
def missingNumber(arr):
    # Calculate the expected difference using the first and last elements
    n = len(arr)
    diff = (arr[-1] - arr[0]) // n # integer division because values are integers
    # Iterate over the array to find the place where the difference deviates from diff
    for i in range(len(arr) - 1):
        # If the difference between consecutive numbers is not equal to the expected diff
        if arr[i+1] - arr[i] != diff:
            # Return the missing number which should be current number + diff
            return arr[i] + diff
    # Return -1 if no missing number is found (should not happen given the problem constraints)
    return -1

# Example usage:
# print(missingNumber([5,7,11,13])) # Output should be 9
```


#1427 **EASY**

Perform String Shifts

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Math · String

Lists: Premium 98

Problem:

You are given a string s containing lowercase English letters, and a matrix shift, where shift[i] = [directioni, amounti]:

- directioni can be 0 (for left shift) or 1 (for right shift).
- amounti is the amount by which string s is to be shifted.
- A left shift by 1 means remove the first character of s and append it to the end.
- Similarly, a right shift by 1 means remove the last character of s and add it to the beginning.

Return the final string after all operations.

Example 1:

```
Input: s = "abc", shift = [[0,1],[1,2]]
Output: "cab"
Explanation:
[0,1] means shift to left by 1. "abc" -> "bca"
[1,2] means shift to right by 2. "bca" -> "cab"
```

Example 2:

```
Input: s = "abcdefg", shift = [[1,1],[1,1],[0,2],[1,3]]
Output: "efgabcd"
Explanation:
[1,1] means shift to right by 1. "abcdefg" -> "gabcdef"
[1,1] means shift to right by 1. "gabcdef" -> "fgabcde"
[0,2] means shift to left by 2. "fgabcde" -> "abcdefg"
[1,3] means shift to right by 3. "abcdefg" -> "efgabcd"
```

Constraints:

- 1 <= s.length <= 100
- s only contains lower case English letters.
- 1 <= shift.length <= 100
- shift[i].length == 2
- directioni is either 0 or 1.
- 0 <= amounti <= 100

>> Brute Force [Math] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def stringShift(self, s, shift):
        # Brute Force - iterate all candidates
        for i in range(2, n + 1):
            is_valid = all(i % j != 0 for j in range(2, i))
            if is_valid: pass
        return 0
```

Python

Python

```
# Python solution

def stringShift(s, shift):
    # Initialize the net shift counter
    net_shift = 0
    for direction, amount in shift:
        if direction == 0:
            net_shift -= amount # Left shift reduces index
        else:
            net_shift += amount # Right shift increases index

    # Normalize shift to be within bounds of the string length
    net_shift %= len(s)

    # A right shift can be simulated by taking the last net_shift characters
    # and concatenating them with the beginning of the string.
    return s[-net_shift:] + s[:-net_shift]

# Example usage
if __name__ == "__main__":
    s = "abcdefg"
    shift_operations = [[1,1],[1,1],[0,2],[1,3]]
    print(stringShift(s, shift_operations)) # Output: "efgabcd"
```

Python

```
class Solution:
   def stringShift(self, s: str, shift: list[list[int]]) -> str:
     move = 0

     for direction, amount in shift:
       if direction == 0:
         move -= amount
       else:
         move += amount

     move %= len(s)
     return s[-move:] + s[:-move]
```

Python

```
class Solution:
    def stringShift(self, s: str, shift: List[List[int]]) -> str:
        x = sum((b if a else -b) for a, b in shift)
        x %= len(s)
        return s[-x:] + s[:-x]
```


Python

```
class Solution:
    def stringShift(self, s: str, shift: List[List[int]]) -> str:
        x = sum((b if a else -b) for a, b in shift)
        x %= len(s)
        return s[-x:] + s[:-x]
```

[*] Math — Pattern Solution (stored optimal)

Python

```
# Python solution

def stringShift(s, shift):
    # Initialize the net shift counter
    net_shift = 0
    for direction, amount in shift:
        if direction == 0:
            net_shift -= amount # Left shift reduces index
        else:
            net_shift += amount # Right shift increases index

    # Normalize shift to be within bounds of the string length
    net_shift %= len(s)

    # A right shift can be simulated by taking the last net_shift characters
    # and concatenating them with the beginning of the string.
    return s[-net_shift:] + s[:-net_shift]

# Example usage
if __name__ == "__main__":
    s = "abcdefg"
    shift_operations = [[1,1],[1,1],[0,2],[1,3]]
    print(stringShift(s, shift_operations)) # Output: "efgabcd"
```

#7 MEDIUM

Reverse Integer

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math

Lists: Grind 169

Problem:

Given a signed 32-bit integer x, return x with its digits reversed. If reversing x causes the value to go outside the signed 32-bit integer range [−231, 231 − 1], then return 0.

Assume the environment does not allow you to store 64-bit integers (signed or unsigned).

Example 1:

```
Input: x = 123
Output: 321
```

Example 2:

```
Input: x = -123
Output: -321
```


Example 3:

```
Input: x = 120
Output: 21
```

Constraints:

- -231 <= x <= 231 - 1

Python

Python

```
# Function to reverse an integer
def reverse(x: int) -> int:
    # Define the maximum and minimum limits for a 32-bit signed integer
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31

    reversed_num = 0 # This will store the reversed number

    # Process each digit until x becomes 0
    while x != 0:
        # Pop the last digit from x
        # Use modulo 10 to get the last digit
        # For negative numbers, modulo may yield negative remainder
        pop = x % 10 if x > 0 else x % -10
        # Remove the last digit from x by integer division
        x = x // 10

        # Check if appending the digit would cause overflow
        if reversed_num > INT_MAX // 10 or (reversed_num == INT_MAX // 10 and pop > 7):
            return 0
        if reversed_num < INT_MIN // 10 or (reversed_num == INT_MIN // 10 and pop < -8):
            return 0

        # Append the digit to the reversed number
        reversed_num = reversed_num * 10 + pop


    return reversed_num

# Example usage:
print(reverse(123)) # Output: 321
print(reverse(-123)) # Output: -321
print(reverse(120)) # Output: 21
```

Python

```
class Solution:
   def reverse(self, x: int) -> int:
     ans = 0
     sign = -1 if x < 0 else 1
     x *= sign

     while x:
       ans = ans * 10 + x % 10
       x //= 10

     return 0 if ans < -2**31 or ans > 2**31 - 1 else sign * ans
```

Python


```
class Solution:
    def reverse(self, x: int) -> int:
        ans = 0
        mi, mx = -(2**31), 2**31 - 1
        while x:
            if ans < mi // 10 + 1 or ans > mx // 10:
                return 0
            y = x % 10
            if x < 0 and y > 0:
                y -= 10
            ans = ans * 10 + y
            x = (x - y) // 10
        return ans
```

Python

```
class Solution:
    def reverse(self, x: int) -> int:
        ans = 0
        mi, mx = -(2**31), 2**31 - 1
        while x:
            if ans < mi // 10 + 1 or ans > mx // 10:
                return 0
            y = x % 10
            if x < 0 and y > 0:
                y -= 10
            ans = ans * 10 + y
            x = (x - y) // 10
        return ans
```

[*] Math — Pattern Solution (stored optimal)

Python

```
# Function to reverse an integer
def reverse(x: int) -> int:
    # Define the maximum and minimum limits for a 32-bit signed integer
    INT_MAX = 2**31 - 1
    INT_MIN = -2**31

    reversed_num = 0 # This will store the reversed number

    # Process each digit until x becomes 0
    while x != 0:
        # Pop the last digit from x
        # Use modulo 10 to get the last digit
        # For negative numbers, modulo may yield negative remainder
        pop = x % 10 if x > 0 else x % -10
        # Remove the last digit from x by integer division
        x = x // 10

        # Check if appending the digit would cause overflow
        if reversed_num > INT_MAX // 10 or (reversed_num == INT_MAX // 10 and pop > 7):
            return 0
        if reversed_num < INT_MIN // 10 or (reversed_num == INT_MIN // 10 and pop < -8):
            return 0

        # Append the digit to the reversed number
        reversed_num = reversed_num * 10 + pop
```



```
    return reversed_num

# Example usage:
print(reverse(123)) # Output: 321
print(reverse(-123)) # Output: -321
print(reverse(120)) # Output: 21
```

#50 MEDIUM

Pow(x, n)

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math · Recursion

Lists: Grind 169

Problem:

Implement pow(x, n), which calculates x raised to the power n (i.e., xn).

Example 1:

```
Input: x = 2.00000, n = 10
Output: 1024.00000
```

Example 2:

```
Input: x = 2.10000, n = 3
Output: 9.26100
```

Example 3:

```
Input: x = 2.00000, n = -2
Output: 0.25000
Explanation: 2-2 = 1/22 = 1/4 = 0.25
```

Constraints:

- -100.0 < x < 100.0
- -231 <= n <= 231-1
- n is an integer.
- Either x is not zero or n > 0.
- -104 <= xn <= 104

>> Brute Force [Math] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def myPow(self, x, n):
        # Brute Force O(|n|) - multiply x by itself n times
        if n == 0: return 1.0
        result = 1.0
        for _ in range(abs(n)):
            result *= x
        return result if n > 0 else 1 / result
```


Python

Python

```
# Python implementation of pow(x, n) using exponentiation by squaring
def myPow(x, n):
    # if exponent is negative, invert x and make n positive
    if n < 0:
        x = 1 / x
        n = -n

    result = 1.0 # initialize result
    while n:
        # if the current exponent is odd, multiply result by the base
        if n % 2:
            result *= x
        x *= x # square the base for the next iteration
        n //= 2 # divide the exponent by 2
    return result

# Example usage:
print(myPow(2.0, 10)) # Output: 1024.0
print(myPow(2.1, 3)) # Output: 9.261000000000001
print(myPow(2.0, -2)) # Output: 0.25
```

Python

```
class Solution:
   def myPow(self, x: float, n: int) -> float:
     if n == 0:
       return 1
     if n < 0:
       return 1 / self.myPow(x, -n)
     if n % 2 == 1:
       return x * self.myPow(x, n - 1)
     return self.myPow(x * x, n // 2)
```

Python

```
class Solution:
    def myPow(self, x: float, n: int) -> float:
        def qpow(a: float, n: int) -> float:
            ans = 1
            while n:
                if n & 1:
                    ans *= a
                a *= a
                n >>= 1
            return ans

        return qpow(x, n) if n >= 0 else 1 / qpow(x, -n)
```

Python

```python
class Solution:
    def myPow(self, x: float, n: int) -> float:
        def qpow(a: float, n: int) -> float:
            ans = 1
            while n:
                if n & 1:
                    ans *= a
                a *= a
                n >>= 1
            return ans

        return qpow(x, n) if n >= 0 else 1 / qpow(x, -n)
```

[*] Math — Pattern Solution (matched from community answers)

Python

```python
# Python implementation of powf(x, n) using exponentiation by squaring
def myPow(x, n):
    # If exponent is negative, invert x and make n positive
    if n < 0:
        x = 1 / x
        n = -n

    result = 1.0 # Initialize result
    while n:
        # If the current exponent is odd, multiply result by the base
        if n % 2:
            result *= x

        x *= x # Square the base for the next iteration
        n //= 2 # Divide the exponent by 2
    return result

# Example usage:
print(myPow(2.0, 10)) # Output: 1024.0
print(myPow(2.1, 3)) # Output: 9.261000000000001
print(myPow(2.0, -2)) # Output: 0.25
```

#380 MEDIUM

Insert Delete GetRandom O(1)

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Math · Design

Lists: Grind 169

Problem:

Implement the RandomizedSet class:

- RandomizedSet() Initializes the RandomizedSet object.
- bool insert(int val) Inserts an item val into the set if not present. Returns true if the item was not present, false otherwise.
- bool remove(int val) Removes an item val from the set if present. Returns true if the item was present, false otherwise.
- int getRandom() Returns a random element from the current set of elements (it's guaranteed that at least one element exists when this method is called). Each element must have the same probability of

being returned.

You must implement the functions of the class such that each function works in average O(1) time complexity.

Example 1:

```
Input
["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getRandom"]
[[], [1], [2], [2], [], [1], [2], []]
Output
[null, true, false, true, 2, true, false, 2]

Explanation
RandomizedSet randomizedSet = new RandomizedSet();
randomizedSet.insert(1); // Inserts 1 to the set. Returns true as 1 was inserted
successfully.
randomizedSet.remove(2); // Returns false as 2 does not exist in the set.
randomizedSet.insert(2); // Inserts 2 to the set, returns true. Set now contains [1,2].
randomizedSet.getRandom(); // getRandom() should return either 1 or 2 randomly.
randomizedSet.remove(1); // Removes 1 from the set, returns true. Set now contains [2].
randomizedSet.insert(2); // 2 was already in the set, so return false.
randomizedSet.getRandom(); // Since 2 is the only number in the set, getRandom() will always
return 2.
```

Constraints:

- -231 <= val <= 231 - 1
- At most 2 * 105 calls will be made to insert, remove, and getRandom.
- There will be at least one element in the data structure when getRandom is called.

Python

Python


```python
import random

class RandomizedSet:
    def __init__(self):
        # List to store the values
        self.values = []
        # Dictionary to map a value to its index in the list
        self.val_to_index = {}

    def insert(self, val: int) -> bool:
        # If the value is already present, do not insert.
        if val in self.val_to_index:
            return False
        # Append the value and store its index.
        self.val_to_index[val] = len(self.values)
        self.values.append(val)
        return True

    def remove(self, val: int) -> bool:
        # If the value is not present, return False.
        if val not in self.val_to_index:
            return False
        # Index of element to remove.
        index = self.val_to_index[val]
        # Get the last element.
        last_element = self.values[-1]
        # Place the last element at the index of the element to remove.
        self.values[index] = last_element
        self.val_to_index[last_element] = index
        # Remove the last element from the list.
        self.values.pop()
        # Delete the mapping for the removed element.
        del self.val_to_index[val]
        return True

    def getRandom(self) -> int:
        # Return a random element from the list.
        return random.choice(self.values)
```

Python


```python
class RandomizedSet:
  def __init__(self):
    self.vals = []
    self.valToIndex = collections.defaultdict(int) # {val: index in vals}

  def insert(self, val: int) -> bool:
    if val in self.valToIndex:
      return False
    self.valToIndex[val] = len(self.vals)
    self.vals.append(val)
    return True

  def remove(self, val: int) -> bool:
    if val not in self.valToIndex:
      return False
    index = self.valToIndex[val]
    # The order of the following two lines is important when vals.size() == 1.
    self.valToIndex[self.vals[-1]] = index
    del self.valToIndex[val]
    self.vals[index] = self.vals[-1]
    self.vals.pop()
    return True

  def getRandom(self) -> int:
    index = random.randint(0, len(self.vals) - 1)

    return self.vals[index]
```

Python

```python
class RandomizedSet:
    def __init__(self):
        self.d = {}
        self.q = []

    def insert(self, val: int) -> bool:
        if val in self.d:
            return False
        self.d[val] = len(self.q)
        self.q.append(val)
        return True

    def remove(self, val: int) -> bool:
        if val not in self.d:
            return False
        i = self.d[val]
        self.d[self.q[-1]] = i
        self.q[i] = self.q[-1]
        self.q.pop()
        self.d.pop(val)
        return True

    def getRandom(self) -> int:
        return choice(self.q)
```



```python
# Your RandomizedSet object will be instantiated and called as such:
# obj = RandomizedSet()
# param_1 = obj.insert(val)
# param_2 = obj.remove(val)
# param_3 = obj.getRandom()
```

Python

```python
'''
my_dict.pop('b') # vs del d[k]

    my_dict = {'a': 1, 'b': 2, 'c': 3}
    val = my_dict.pop('b')
    print(my_dict) # {'a': 1, 'c': 3}
    print(val) # 2

    >>> item = my_dict.pop()
    Traceback (most recent call last):
      File "<stdin>", line 1, in <module>
    TypeError: pop expected at least 1 argument, got 0

    >>> my_dict = {'a': 1, 'b': 2, 'c': 3}
    >>> del my_dict['b']
    >>> my_dict
    {'a': 1, 'c': 3}
'''

from collections import defaultdict
from random import choice

class RandomizedSet(object):
  def __init__(self):
    """
```



```python
    Initialize your data structure here.
    """

    # value (map) -> its index -> value (list)
    self.d = {}
    self.a = []  # introduce it purely for random

  def insert(self, val):
    """
    Inserts a value to the set. Returns true if the set did not already contain the specified element.
    :type val: int
    :rtype: bool
    """
    if val in self.d:
      return False
    self.a.append(val)
    self.d[val] = len(self.a) - 1
    return True

  def remove(self, val):
    """
    Removes a value from the set. Returns true if the set contained the specified element.
    :type val: int
    :rtype: bool
    """

    if val not in self.d:
      return False
    index = self.d[val]

    # process last index/val
    self.a[index] = self.a[-1]
    self.d[self.a[-1]] = index

    # process to be deleted index/val
    self.a.pop() # delete in list
    del self.d[val] # delete in dict
    # ok: self.d.pop(val)
    return True

  def getRandom(self):
    """
    Get a random element from the set.
    :rtype: int
    """
    return self.a[random.randrange(0, len(self.a))]
    # return random.choice(self.a)

############

# Your RandomizedSet object will be instantiated and called as such:

# obj = RandomizedSet()
# param_1 = obj.insert(val)
# param_2 = obj.remove(val)
# param_3 = obj.getRandom()
```

[*] Math — Pattern Solution (stored optimal)

Python

```python
import random

class RandomizedSet:
    def __init__(self):
        # List to store the values
        self.values = []
        # Dictionary to map a value to its index in the list
        self.val_to_index = {}

    def insert(self, val: int) -> bool:
        # If the value is already present, do not insert.
        if val in self.val_to_index:
            return False
        # Append the value and store its index.
        self.val_to_index[val] = len(self.values)
        self.values.append(val)
        return True

    def remove(self, val: int) -> bool:
        # If the value is not present, return False.
        if val not in self.val_to_index:
            return False
        # Index of element to remove.
        index = self.val_to_index[val]
        # Get the last element.

        last_element = self.values[-1]
        # Place the last element at the index of the element to remove.
        self.values[index] = last_element
        self.val_to_index[last_element] = index
        # Remove the last element from the list.
        self.values.pop()
        # Update the mapping for the removed element.
        del self.val_to_index[val]
        return True

    def getRandom(self) -> int:
        # Return a random element from the list.
        return random.choice(self.values)
```

#1017 MEDIUM

Convert to Base -2

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math

Lists: Denny Zhang

Problem:

Given an integer n, return a binary string representing its representation in base -2.

Note that the returned string should not have leading zeros unless the string is "0".

Example 1:

```
Input: n = 2
Output: "110"
```



```
Explantion: (-2)2 + (-2)1 = 2
```

Example 2:

```
Input: n = 3
Output: "111"
Explantion: (-2)2 + (-2)1 + (-2)0 = 3
```

Example 3:

```
Input: n = 4
Output: "100"
Explantion: (-2)2 = 4
```

Constraints:

- 0 <= n <= 109

Python

Python

```python
# Python solution with detailed comments
def baseNeg2(n: int) -> str:
    # Special case for zero as input.
    if n == 0:
        return "0"

    digits = [] # List to store digits in base -2.

    while n != 0:
        n, remainder = divmod(n, -2) # Get quotient and remainder when dividing by -2.
        if remainder < 0:
            # Adjust the remainder and increment the quotient.
            remainder += 2
            n += 1
        # Append the remainder (digit) to the list.
        digits.append(str(remainder))

    # Reverse the digits to form the proper binary string and return.
    return "".join(reversed(digits))

# Example usage:
print(baseNeg2(2)) # Output: "110"
print(baseNeg2(3)) # Output: "111"
print(baseNeg2(4)) # Output: "100"
```

Python

```python
class Solution:
  def baseNeg2(self, n: int) -> str:
    ans = []

    while n != 0:
      ans.append(str(n % 2))
      n = -(n >> 1)

    return ''.join(reversed(ans)) if ans else '0'
```

Python


```python
class Solution:
    def baseNeg2(self, n: int) -> str:
        k = 1
        ans = []
        while n:
            if n % 2:
                ans.append('1')
                n -= k
            else:
                ans.append('0')
            n //= 2
            k *= -1
        return ''.join(ans[::-1]) or '0'
```

Python

```python
class Solution:
    def baseNeg2(self, n: int) -> str:
        k = 1
        ans = []
        while n:
            if n % 2:
                ans.append('1')
                n -= k
            else:
                ans.append('0')
            n //= 2
            k *= -1
        return ''.join(ans[::-1]) or '0'
```

[*] Math — Pattern Solution (matched from community answers)

Python

```python
class Solution:
  def baseNeg2(self, n: int) -> str:
    ans = []

    while n != 0:
      ans.append(str(n % 2))
      n = -(n >> 1)

    return ''.join(reversed(ans)) if ans else '0'
```

#3160 MEDIUM

Find the Number of Distinct Colors Among the Balls

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Simulation

Lists: CodeSignal

Problem:

You are given an integer limit and a 2D array queries of size n x 2.

There are limit + 1 balls with distinct labels in the range [0, limit]. Initially, all balls are uncolored. For every query in queries that is of the form [x, y], you mark ball x with the color y. After each query, you

need to find the number of colors among the balls.

Return an array result of length n, where result[i] denotes the number of colors after ith query.

Note that when answering a query, lack of a color will not be considered as a color.

Example 1:

Input: limit = 4, queries = [[1,4],[2,5],[1,3],[3,4]]

Output: [1,2,2,3]

Explanation:

- After query 0, ball 1 has color 4.
- After query 1, ball 1 has color 4, and ball 2 has color 5.
- After query 2, ball 1 has color 3, and ball 2 has color 5.
- After query 3, ball 1 has color 3, ball 2 has color 5, and ball 3 has color 4.

Example 2:

Input: limit = 4, queries = [[0,1],[1,2],[2,2],[3,4],[4,5]]

Output: [1,2,2,3,4]

Explanation:

- After query 0, ball 0 has color 1.
- After query 1, ball 0 has color 1, and ball 1 has color 2.
- After query 2, ball 0 has color 1, and balls 1 and 2 have color 2.
- After query 3, ball 0 has color 1, balls 1 and 2 have color 2, and ball 3 has color 4.
- After query 4, ball 0 has color 1, balls 1 and 2 have color 2, ball 3 has color 4, and ball 4 has color 5.

Constraints:

- 1 <= limit <= 109
- 1 <= n == queries.length <= 105
- queries[i].length == 2
- 0 <= queries[i][0] <= limit
- 1 <= queries[i][1] <= 109

Python

Python

```python
# Python solution with detailed comments

def distinctColors(limit, queries):
    # Dictionary to keep track of the current color of each ball
    ball_color = {}
    # Dictionary to count the frequency of each color applied
    color_count = {}
    # List to store the results after each query
    results = []

    # Process each query one by one
    for ball, color in queries:
        # If the ball has been colored before, update the old color frequency
        if ball in ball_color:
            old_color = ball_color[ball]
            color_count[old_color] -= 1 # Decrement the count of the old color
            # If the frequency becomes zero, remove the color from the dictionary
            if color_count[old_color] == 0:
                del color_count[old_color]

        # Update ball's color to the new color
        ball_color[ball] = color
        # Increase the frequency count of the new color
        color_count[color] = color_count.get(color, 0) + 1

        # The number of distinct colors is the number of keys in color_count
        results.append(len(color_count))

    return results

# Example usage:
# limit = 4
# queries = [[1,4],[2,5],[1,3],[3,4]]
# print(distinctColors(limit, queries)) # Output: [1,2,2,3]
```

Python


```python
class Solution:
  def queryResults(self, limit: int, queries: list[list[int]]) -> list[int]:
    ans = []
    ballToColor = {}
    colorCount = collections.Counter()

    for ball, color in queries:
      if ball in ballToColor:
        prevColor = ballToColor[ball]
        colorCount[prevColor] -= 1
        if colorCount[prevColor] == 0:
          del colorCount[prevColor]
      ballToColor[ball] = color
      colorCount[color] += 1
      ans.append(len(colorCount))

    return ans
```

Python

```python
class Solution:
    def queryResults(self, limit: int, queries: List[List[int]]) -> List[int]:
        g = {}
        cnt = Counter()
        ans = []
        for x, y in queries:
            cnt[y] += 1
            if x in g:
                cnt[g[x]] -= 1
                if cnt[g[x]] == 0:
                    cnt.pop(g[x])
            g[x] = y
            ans.append(len(cnt))
        return ans
```

Python

```python
class Solution:
    def queryResults(self, limit: int, queries: List[List[int]]) -> List[int]:
        g = {}
        cnt = Counter()
        ans = []
        for x, y in queries:
            cnt[y] += 1
            if x in g:
                cnt[g[x]] -= 1
                if cnt[g[x]] == 0:
                    cnt.pop(g[x])
            g[x] = y
            ans.append(len(cnt))
        return ans
```

[*] Math — Pattern Solution (stored optimal)

Python

```python
class Solution:
    def queryResults(self, n: int, queries: List[List[int]]) -> List[int]:
        node_color = {}
        color_count = {}
        ans = []
        for node, color in queries:
            if node in node_color:
                old = node_color[node]
                color_count[old] -= 1
                if color_count[old] == 0:
                    del color_count[old]
            node_color[node] = color
            color_count[color] = color_count.get(color, 0) + 1
            ans.append(len(color_count))
        return ans
```

#68 HARD

Text Justification

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · String · Simulation

Lists: CodeSignal

Problem:

Given an array of strings words and a width maxWidth, format the text such that each line has exactly maxWidth characters and is fully (left and right) justified.

You should pack your words in a greedy approach; that is, pack as many words as you can in each line. Pad extra spaces ' ' when necessary so that each line has exactly maxWidth characters.

Extra spaces between words should be distributed as evenly as possible. If the number of spaces on a line does not divide evenly between words, the empty slots on the left will be assigned more spaces than the slots on the right.

For the last line of text, it should be left-justified, and no extra space is inserted between words.

Note:

- A word is defined as a character sequence consisting of non-space characters only.
- Each word's length is guaranteed to be greater than 0 and not exceed maxWidth.
- The input array words contains at least one word.

Example 1:

```
Input: words = ["This", "is", "an", "example", "of", "text", "justification."], maxWidth = 16
Output:
[
   "This    is    an",
   "example  of text",
   "justification.  "
]
```

Example 2:

```
Input: words = ["What","must","be","acknowledgment","shall","be"], maxWidth = 16
Output:
```



```
[
  "What   must   be",
  "acknowledgment  ",
  "shall be        "
]
Explanation: Note that the last line is "shall be " instead of "shall be", because the last
line must be left-justified instead of fully-justified.
Note that the second line is also left-justified because it contains only one word.
```

Example 3:

```
Input: words = ["Science","is","what","we","understand","well","enough","to","explain","to","
a","computer.","Art","is","everything","else","we","do"], maxWidth = 20
Output:
[
  "Science  is  what we",
 "understand      well",
  "enough to explain to",
  "a  computer.  Art is",
  "everything  else  we",
  "do                  "
]
```

Constraints:

- 1 <= words.length <= 300
- 1 <= words[i].length <= 20
- words[i] consists of only English letters and symbols.
- 1 <= maxWidth <= 100
- words[i].length <= maxWidth

>> Brute Force [Math] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def fullJustify(self, words, maxWidth):
        # Brute Force - iterate all candidates
        for i in range(2, n + 1):
            is_valid = all(i % j != 0 for j in range(2, i))
            if is_valid: pass
        return 0
```

Python

Python


```python
def fullJustify(words, maxWidth):
    result = [] # Final list of justified text lines.
    current_words = [] # Current line's words.
    current_len = 0 # Total length of current line's words (excluding spaces).

    # Process all words.
    for word in words:
        # Check if adding the next word would exceed the maxWidth when including minimum one space between
words.
        if current_words and (current_len + len(word) + len(current_words)) > maxWidth:
            total_spaces = maxWidth - current_len # Total spaces to distribute.
            gaps = len(current_words) - 1 # Number of gaps between words.

            # If there is only one word, left justify.
            if gaps == 0:
                line = current_words[0] + ' ' * total_spaces
            else:
                # Calculate base spaces and extras to add from the left.
                space_per_gap, extra = divmod(total_spaces, gaps)
                line = ''
                for i in range(gaps):
                    # Distribute an extra space to the leftmost gaps.
                    spaces = space_per_gap + (1 if i < extra else 0)
                    line += current_words[i] + ' ' * spaces
                line += current_words[-1] # Append the last word.
            result.append(line)

            # Reset the current line to start with the current word.
            current_words = []
            current_len = 0

        # Add the word to the current line.
        current_words.append(word)
        current_len += len(word)

    # Process the last line, which is left-justified.
    line = ' '.join(current_words)
    line += ' ' * (maxWidth - len(line))
    result.append(line)
    return result

# Example usage:
words = ["This", "is", "an", "example", "of", "text", "justification."]
maxWidth = 16
print(fullJustify(words, maxWidth))
```

Python


```python
class Solution:
  def fullJustify(self, words: list[str], maxWidth: int) -> list[str]:
    ans = []
    row = []
    rowLetters = 0

    for word in words:
      # If we place the word in this row, it will exceed the maximum width.
      # Therefore, we cannot put the word in this row and have to pad spaces
      # for each word in this row.
      if rowLetters + len(word) + len(row) > maxWidth:
        for i in range(maxWidth - rowLetters):
          row[i % len(row) - 1 or 1] += ' '
        ans.append(''.join(row))
        row = []
        rowLetters = 0
      row.append(word)
      rowLetters += len(word)

    return ans + [' '.join(row).ljust(maxWidth)]
```

Python

```python
class Solution:
    def fullJustify(self, words: List[str], maxWidth: int) -> List[str]:
        ans = []
        i, n = 0, len(words)
        while i < n:
            t = []
            cnt = len(words[i])
            t.append(words[i])
            i += 1
            while i < n and cnt + 1 + len(words[i]) <= maxWidth:
                cnt += 1 + len(words[i])
                t.append(words[i])
                i += 1
            if i == n or len(t) == 1:
                left = ' '.join(t)
                right = ' ' * (maxWidth - len(left))
                ans.append(left + right)
                continue
            space_width = maxWidth - (cnt - len(t) + 1)
            w, m = divmod(space_width, len(t) - 1)
            row = []
            for j, s in enumerate(t[:-1]):
                row.append(s)
                row.append(' ' * (w + (1 if j < m else 0)))
            row.append(t[-1])

            ans.append(''.join(row))
        return ans
```

Python


```python
'''
Explanation:

- The function `fullJustify` takes a list of words and a maximum width `maxWidth` as inputs.
- It iterates over each word, deciding whether to add the current word to the current line (`cur`) or to
start a new line.
- If adding the current word to the line would exceed `maxWidth`, it justifies the current line by adding
extra spaces between words as needed, then starts a new line.
- Once all words are processed, it left-justifies the last line by joining the remaining words in `cur` with
a single space and then using `.ljust(maxWidth)` to ensure the line is of maximum width.
- The result is a list of strings, where each string represents a justified line of text.

This solution carefully handles edge cases, such as when there's only one word in the line (avoiding
division by zero) and ensuring the last line is left-justified instead of fully justified.
'''

'''
>>> cur = [1]
>>> len(cur) - 1
0
>>> ( len(cur) - 1 or 1 )
1
'''

'''
# api: string.ljust(width[, fillchar])

>>> "Hello".ljust(10)
'Hello  '
```

```python
>>> "Hello".ljust(10, '-')
'Hello-----'
>>> "Hello".ljust(2)
'Hello'

>>> "Hello World".ljust(20)
'Hello World '
>>> "Hello World".ljust(20, '-')
'Hello World---------'


# api: string.rjust(width[, fillchar])
>>> "Hello".rjust(10)
'  Hello'
>>> "Hello".rjust(10, '-')
'-----Hello'
>>> "Hello".rjust(2)
'Hello'

>>> "Hello World".rjust(20)
'  Hello World'
>>> "Hello World".rjust(20, '-')
'---------Hello World'

'''
```



```python
class Solution:
    def fullJustify(self, words: List[str], maxWidth: int) -> List[str]:
        res, cur, num_of_letters = [], [], 0

        for w in words:
            # len(cur) is the space count, from last round, right one less after adding 'w'
            if num_of_letters + len(cur) + len(w) > maxWidth:
                for i in range(maxWidth - num_of_letters):
                    # The 'or 1' part is for dealing with the edge case 'len(cur) == 1'
                    cur[i % ( len(cur) - 1 or 1 )] += ' '
                res.append(''.join(cur))
                cur, num_of_letters = [], 0
            cur += [w]
            num_of_letters += len(w)

        return res + [' '.join(cur).ljust(maxWidth)]


############
'''
>>> 17 % 3
2
>>> divmod(17,3)
'''
(5, 2)
'''

'''
>>> ['This', 'is', 'an']
```

[*] Math — Pattern Solution (stored optimal)

Python

```python
def fullJustify(words, maxWidth):
    result = [] # Final list of justified text lines.
    current_words = [] # Current line's words.
    current_len = 0 # Total length of current line's words (excluding spaces).

    # Process all words.
    for word in words:
        # Check if adding the next word would exceed the maxWidth when including minimum one space between words.
        if current_words and (current_len + len(word) + len(current_words)) > maxWidth:
            total_spaces = maxWidth - current_len # Total spaces to distribute.
            gaps = len(current_words) - 1 # Number of gaps between words.

            # If there is only one word, left justify.
            if gaps == 0:
                line = current_words[0] + ' ' + total_spaces
            else:
                # Calculate even spaces and extras to add from the left.
                space_per_gap, extra = divmod(total_spaces, gaps)
                line = ''
                for i in range(gaps):
                    # Distribute an extra space to the leftmost gaps.
                    spaces = space_per_gap + (1 if i < extra else 0)
                    line += current_words[i] + ' ' * spaces
            line += current_words[-1] # Append the last word.
            result.append(line)

            # Reset the current line to start with the current word.
            current_words = []
            current_len = 0

        # Add the word to the current line.
        current_words.append(word)
        current_len += len(word)

    # Process the last line, which is left-justified.
    line = ' '.join(current_words)
    line += ' ' * (maxWidth - len(line))
    result.append(line)
    return result

# Example usage.
words = ["This", "is", "an", "example", "of", "text", "justification."]
maxWidth = 16
print(fullJustify(words, maxWidth))
```


Quick Review — Math 12 questions · insights · complexity · solution

#9 Palindrome Number [Easy]

Key Insights

- Negative numbers are not palindromic.
- A number ending in 0 (except 0 itself) cannot be a palindrome.
- Instead of reversing the entire number (risking overflow), reverse only half of the number and compare with the other half.
- For numbers with an odd number of digits, the middle digit can be disregarded.

Space & Time

Time Complexity: O(log10(n)) - Each iteration reduces the number by a factor of 10.

Space Complexity: O(1) - Only constant extra space is used.

Solution

The solution checks for obvious non-palindrome cases first (negative numbers and numbers ending with 0 that are not 0). Then, it reverses half of the digits of the number while the original number is larger than the reversed half. Finally, it compares the remaining part of the original number with the reversed half. In the case of an odd number of digits, the middle digit is removed by dividing the reversed half by 10 before the final comparison.

#13 Roman to Integer [Easy]

Key Insights

- Map each Roman numeral character to its integer value using a hash table or dictionary.
- Loop through the string and check if the current numeral is less than the next numeral; if so, subtract its value.
- Otherwise, add its value to the total.
- This approach takes advantage of the subtractive notation used in Roman numerals.

Space & Time

Time Complexity: O(n), where n is the length of the input string.

Space Complexity: O(1), since the mapping and additional variables use constant space.

Solution

We use a dictionary (or hash map) to store the values of each Roman numeral. As we iterate over the string, we compare the current numeral's value with the next numeral's value. When a numeral of smaller value appears before a numeral of larger value, we subtract the current numeral's value; otherwise, we add it. This technique efficiently handles both the standard and subtractive cases in one pass through the string.

#504 Base 7 [Easy]

Key Insights

- Handle the edge case where num is 0.
- Use division and modulo operations to extract base 7 digits.
- Manage negative numbers by converting to positive and reattaching the minus sign.
- Build the base 7 digits in reverse order during iteration.

Space & Time

Time Complexity: O(log_7(n)) where n is the absolute value of the number.

Space Complexity: O(log_7(n)) to store the digits of the number.

Solution

The algorithm follows these steps:

- Check if num equals 0 and return "0". - Determine if the number is negative. If so, work with its absolute value but remember the negative flag. - Iteratively compute the remainder when divided by 7 (using modulo) and update the number by integer division by 7. - Append each remainder (converted to character) to a list representing the digits. - The digits are computed in reverse order; reverse the list for the correct order. - If the original number was negative, append the "-" sign to the front. - Join the digits to form the resulting base 7 string.

  This approach uses basic arithmetic (division and modulo) and a list (or equivalent structure) to accumulate the computed digits.

#1056 Confusing Number [Easy]

Key Insights

- Use a mapping dictionary for valid rotations: {0:0, 1:1, 6:9, 8:8, 9:6}.
- Process the digits of n from right to left to form the rotated number.
- If any digit is not present in the mapping, the number is instantly not confusing.
- Compare the rotated number with the original number to check if they are different.

Space & Time

Time Complexity: O(d), where d is the number of digits in n (approximately O(log n)).

Space Complexity: O(d), needed to store the rotated number string.

Solution

The approach is to convert the number to a string and iterate through it in reverse. For each digit, check if it is valid using the predefined mapping. If it is, append its rotated counterpart to a new string. After processing, convert the rotated string to an integer (ignoring any leading zeros). Finally, compare the rotated number to the original number. If they are different, the number is confusing.

#1228 Missing Number in Arithmetic Progression [Easy]

Key Insights

- The complete arithmetic progression has equal differences between consecutive numbers.
- The missing element can be found by calculating the expected common difference using the first and last elements.
- Since exactly one number is missing, iterate through the provided array to detect where the difference deviates from the expected common difference.

Space & Time

Time Complexity: O(n)

Space Complexity: O(1)

Solution

We first compute the expected common difference (diff) for the original arithmetic progression. Since the actual progression had one more element than the current array, the formula used is: diff = (last element - first element) / (n) where n is the length of the current array. Next, iterate over the array and check the difference between each pair of consecutive elements. When the actual difference deviates from diff, it indicates that the missing number is the previous element plus diff. This approach uses a simple loop and constant extra space.

#1427 Perform String Shifts [Easy]

Key Insights

- Consolidate all shifts by calculating a net shift: subtract for left shifts and add for right shifts.
- Use modulo arithmetic with the string length to avoid unnecessary full rotations.
- Slice the string based on the effective shift to construct the final string.

Space & Time

Time Complexity: O(n + m), where n is the length of the string and m is the number of shift operations.

Space Complexity: O(n) for the output string.

Solution

The solution involves computing the net shift value by iterating through the shift operations. Left shifts subtract from the net value, and right shifts add to the net value. After obtaining the cumulative shift, use modulo with the string length to normalize the shift, ensuring it lies within bounds. The final string is constructed by slicing the string into two parts and concatenating them appropriately. This approach leverages string slicing as the primary operation for shifting.

#7 Reverse Integer [Medium]

Key Insights

- The problem requires reversing the digits while preserving the sign.
- Watch out for integer overflow due to the reversed number exceeding the 32-bit signed integer limits.
- The approach involves repeatedly extracting the last digit from the number and appending it to a new reversed integer.
- Overflow checks are done during each iteration before actually appending the digit.

Space & Time

Time Complexity: O(n), where n is the number of digits in the integer.

Space Complexity: O(1), only constant extra space is used.

Solution

The solution uses a while loop to extract the last digit from the original number using modulo operation (x % 10) and then removes this digit using integer division (x //= 10). A reversed number is constructed by multiplying the current reversed number by 10 and adding the extracted digit. Before appending each digit, the algorithm checks if the new reversed number will overflow a 32-bit signed integer by comparing it with the thresholds derived from INT_MAX and INT_MIN. The algorithm handles negative values by working with x directly since the modulo and division operations work properly with negatives in most languages.

#50 Pow(x, n) [Medium]

Key Insights

- Use exponentiation by squaring to reduce the number of multiplications.
- For a negative exponent, compute the power for |n| and then take the reciprocal.
- Handle the edge case when n is 0 (return 1).
- The algorithm works in O(log n) time complexity.

Space & Time

Time Complexity: O(log |n|)

Space Complexity: O(1) for the iterative solution, or O(log |n|) if implemented recursively

Solution

We use exponentiation by squaring, an algorithm that reduces the number of multiplications by repeatedly squaring the base and halving the exponent. Here's how it works:

- If n is negative, switch x to 1/x and n to -n. - Initialize the result as 1. - Loop while n is greater than 0: If the current n is odd, multiply the result by x. Square the base x. Divide n by 2 (using integer division). This iterative approach efficiently computes the power in logarithmic steps. The same approach can be implemented recursively.

#380 Insert Delete GetRandom O(1) [Medium]

Key Insights

- Use a dynamic array (or list) to store the elements to allow O(1) access by index.
- Use a hash table (or dictionary/map) to store the mapping from value to its index in the array.
- For removal, swap the element to be removed with the last element in the array, then delete the last element. This avoids expensive shifting of elements.
- getRandom can be implemented by generating a random index into the array.

Space & Time

Time Complexity: O(1) on average for insert, remove, and getRandom operations.

Space Complexity: O(n) where n is the number of elements stored in the data structure.

Solution

The solution involves maintaining a list to store the values and a hash map to quickly check for the presence of a value and to know its index in the list. When an element is removed, swap it with the last element of the list and update the map accordingly; then, remove the last element in O(1) time. This approach ensures that insert, remove, and getRandom can all be achieved in constant time on average.

#1017 Convert to Base -2 [Medium]

Key Insights

- Use division and remainder operations to simulate converting a number to a different base.
- When working with a negative base (-2), the remainder can become negative; adjust the remainder (by adding the base's absolute value) and update the quotient accordingly.
- For n = 0, simply return "0" since there is nothing to convert.
- Build the answer by collecting digits (remainders) and then reversing the order since the conversion provides digits from least significant to most significant.

Space & Time

Time Complexity: O(log|n|) - the loop runs for about the number of digits in the base -2 representation.

Space Complexity: O(log|n|) - extra space for storing the computed digits.

Solution

To solve the problem, we simulate the conversion process similar to how you would convert an integer to a positive base. However, since the base here is -2, special handling of negative remainders is needed. In each iteration, we:

- Divide the current number by -2. - Calculate the remainder. If the remainder is negative, adjust it by adding 2, and increment the quotient to compensate. - Append the computed remainder to a list (which represents the binary digit). - Continue until the number reduces to zero. Finally, reverse the list to form the correct string representation from the most significant to the least significant digit.

#3160 Find the Number of Distinct Colors Among the Balls [Medium]

Key Insights

- Use a hash map to store the current color of each ball that has been updated.
- Use another hash map to store how many balls currently use each color.
- When a ball is recolored, decrement the count of its old color first.
- If an old color count becomes zero, remove it from the color-frequency map.
- Add the new color to the ball and increment its frequency.
- After each query, the number of distinct colors is the number of keys in the color-frequency map.

Space & Time

Time Complexity: O(q) average for q queries, since each query does only O(1) average-time hash map work.

Space Complexity: O(min(q, n)) for colored balls, plus O(q) in the worst case for distinct active colors.

Solution

We solve the problem using two hash maps. The first map stores the current color of each ball, and the second map stores the frequency of each active color. For every query, if the ball already has a color, we decrement the old color's count and remove that color from the frequency map if its count reaches zero. Then we update the ball with the new color and increment the new color's frequency. After processing the query, the number of distinct colors is simply the number of keys remaining in the frequency map.

#68 Text Justification [Hard]

Key Insights

- Use a greedy approach to fill each line with as many words as possible without exceeding maxWidth.
- Calculate total space to distribute by subtracting the combined words length from maxWidth.
- If the line has more than one word, distribute extra spaces evenly; if not, pad the line with spaces at the end.
- Handle the last line separately by left-justifying the text (i.e., adding a single space between words and padding the end).

Space & Time

Time Complexity: O(n) where n is the number of words, since each word is processed once.

Space Complexity: O(n) for storing the result and temporary buffers.

Solution

The solution iterates over the list of words and groups them into lines such that the combined length of the words and minimum required spaces does not exceed maxWidth. For each completed group (line):

- If it's not the last line, calculate extra spaces required. - If the line has multiple words, distribute the spaces evenly, adding any extra spaces from left to right. - If the line contains a single word, append spaces to the right. - For the last line, join words with a single space and pad the remaining width with spaces.

  Data structures used include lists (or arrays) for current words accumulation and the final result. The logic is implemented using a simulation approach to mimic text formatting.

#12 Arrays & Hashing — 18 questions · #12 of 21

#1 EASY

Two Sum

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table

Lists: Grind 169

Problem:

Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target.

You may assume that each input would have exactly one solution, and you may not use the same element twice.

You can return the answer in any order.

Example 1:

```
Input: nums = [2,7,11,15], target = 9
Output: [0,1]
Explanation: Because nums[0] + nums[1] == 9, we return [0, 1].
```

Example 2:

```
Input: nums = [3,2,4], target = 6
Output: [1,2]
```

Example 3:

```
Input: nums = [3,3], target = 6
Output: [0,1]
```

Constraints:

- 2 <= nums.length <= 104
- -109 <= nums[i] <= 109
- -109 <= target <= 109
- Only one valid answer exists.

Follow-up: Can you come up with an algorithm that is less than O(n2) time complexity?

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def two_sum(self, nums, target):
        # Brute force O(n^2) - check every pair; O(1) extra space
        for i in range(len(nums)):
            for j in range(i + 1, len(nums)):
                if nums[i] + nums[j] == target:
                    return [i, j]
```


Python

```python
def two_sum(nums, target):
    # Create a dictionary to store the number and its index
    num_to_index = {}
    # Iterate over the list with index
    for index, num in enumerate(nums):
        # Calculate the complement needed to reach the target
        complement = target - num
        # Check if the complement is in the dictionary
        if complement in num_to_index:
            # Return the index of the complement and the current index
            return [num_to_index[complement], index]
        # Store the number with its index
        num_to_index[num] = index
    # Since the problem guarantees one solution, we don't need a return statement here.

# Example usage:
# print(two_sum([2, 7, 11, 15], 9)) # Output: [0, 1]
```

Python

```python
class Solution:
  def twoSum(self, nums: list[int], target: int) -> list[int]:
    numToIndex = {}

    for i, num in enumerate(nums):
      if target - num in numToIndex:
        return numToIndex[target - num], i
      numToIndex[num] = i
```

Python

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        d = {}
        for i, x in enumerate(nums):
            if (y := target - x) in d:
                return [d[y], i]
            d[x] = i
```

Python

```python
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        m = {}
        for i, x in enumerate(nums):
            y = target - x
            if y in m:
                return [m[y], i]
            m[x] = i
```

[*] Arrays & Hashing — Pattern Solution (stored optimal)

Python


```python
def two_sum(nums, target):
    # Create a dictionary to store the number and its index
    num_to_index = {}
    # Iterate over the list with index
    for index, num in enumerate(nums):
        # Calculate the complement needed to reach the target
        complement = target - num
        # Check if the complement is in the dictionary
        if complement in num_to_index:
            # Return the index of the complement and the current index
            return [num_to_index[complement], index]
        # Store the number with its index
        num_to_index[num] = index
    # Since the problem guarantees one solution, we don't need a return statement here.

# Example usage:
# print(two_sum([2, 7, 11, 15], 9)) # Output: [0, 1]
```


#163 EASY

Missing Ranges

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array

Lists: Premium 98

Problem:

You are given an inclusive range [lower, upper] and a sorted unique integer array nums, where all elements are within the inclusive range.

A number x is considered missing if x is in the range [lower, upper] and x is not in nums.

Return the shortest sorted list of ranges that exactly covers all the missing numbers. That is, no element of nums is included in any of the ranges, and each missing number is covered by one of the ranges.

Example 1:

```
Input: nums = [0,1,3,50,75], lower = 0, upper = 99
Output: [[2,2],[4,49],[51,74],[76,99]]
Explanation: The ranges are:
[2,2]
[4,49]
[51,74]
[76,99]
```

Example 2:

```
Input: nums = [-1], lower = -1, upper = -1
Output: []
Explanation: There are no missing ranges since there are no missing numbers.
```

Constraints:

- -109 <= lower <= upper <= 109
- 0 <= nums.length <= 100
- lower <= nums[i] <= upper
- All the values of nums are unique.

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findMissingRanges(self, nums, lower, upper):
        # Brute Force O(n^2) - check all pairs without a hash map
        n = len(nums)
        for i in range(n):
            for j in range(i + 1, n):
                # process pair nums[i], nums[j]
                pass
        return [] # adjust return
```

Python

```python
# Python solution for Missing Ranges problem

def findMissingRanges(nums, lower, upper):
    missing_ranges = [] # List to store the missing range intervals
    prev = lower - 1 # Initialize previous value to one less than lower

    # Append a sentinel at the end for easier gap handling
    nums.append(upper + 1)

    # Loop through each number, including the sentinel
    for num in nums:
        # If there is a gap between previous and current number
        if num - prev >= 2:
            # If gap is of single number, add that number as a range
            if num - prev == 2:
                missing_ranges.append([prev + 1, prev + 1])
            else:
                missing_ranges.append([prev + 1, num - 1])
        prev = num # Update previous to current
    return missing_ranges

# Example usage:
print(findMissingRanges([0,1,3,50,75], 0, 99))
```

Python


```python
class Solution:
    def findMissingRanges(
        self,
        nums: list[int],
        lower: int,
        upper: int,
    ) -> list[list[int]]:
        def getRange(lo: int, hi: int) -> list[int]:
            if lo == hi:
                return [lo, lo]
            return [lo, hi]

        if not nums:
            return [getRange(lower, upper)]

        ans = []

        if nums[0] > lower:
            ans.append(getRange(lower, nums[0] - 1))

        for prev, curr in zip(nums, nums[1:]):
            if curr > prev + 1:
                ans.append(getRange(prev + 1, curr - 1))

        if nums[-1] < upper:

            ans.append(getRange(nums[-1] + 1, upper))

        return ans
```

Python

```python
class Solution:
    def findMissingRanges(
        self, nums: List[int], lower: int, upper: int
    ) -> List[List[int]]:
        n = len(nums)
        if n == 0:
            return [[lower, upper]]
        ans = []
        if nums[0] > lower:
            ans.append([lower, nums[0] - 1])
        for a, b in pairwise(nums):
            if b - a > 1:
                ans.append([a + 1, b - 1])
        if nums[-1] < upper:
            ans.append([nums[-1] + 1, upper])
        return ans
```



```python
class Solution:
    def findMissingRanges(
        self, nums: List[int], lower: int, upper: int
    ) -> List[List[int]]:
        n = len(nums)
        if n == 0:
            return [[lower, upper]]
        ans = []
        if nums[0] > lower:
            ans.append([lower, nums[0] - 1])
        for a, b in pairwise(nums):
            if b - a > 1:
                ans.append([a + 1, b - 1])
        if nums[-1] < upper:
            ans.append([nums[-1] + 1, upper])
        return ans
```


[*] Arrays & Hashing — Pattern Solution (stored optimal)

Python


```python
# Python solution for Missing Ranges problem

def findMissingRanges(nums, lower, upper):
    missing_ranges = [] # List to store the missing range intervals
    prev = lower - 1 # Initialize previous value to one less than lower

    # Append a sentinel at the end for easier gap handling
    nums.append(upper + 1)

    # Loop through each number, including the sentinel
    for num in nums:
        # If there is a gap between previous and current number
        if num - prev >= 2:
            # If gap is of single number, add that number as a range
            if num - prev == 2:
                missing_ranges.append([prev + 1, prev + 1])
            else:
                missing_ranges.append([prev + 1, num - 1])
        prev = num # Update previous to current
    return missing_ranges

# Example usage:
print(findMissingRanges([0,1,3,50,75], 0, 99))
```


#346 EASY

Moving Average from Data Stream

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Design · Queue · Data Stream

Lists: Premium 98

Problem:

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

Implement the MovingAverage class:

- MovingAverage(int size) Initializes the object with the size of the window size.
- double next(int val) Returns the moving average of the last size values of the stream.

Example 1:

```
Input
["MovingAverage", "next", "next", "next", "next"]
[[3], [1], [10], [3], [5]]
Output
[null, 1.0, 5.5, 4.66667, 6.0]

Explanation
MovingAverage movingAverage = new MovingAverage(3);
movingAverage.next(1); // return 1.0 = 1 / 1
movingAverage.next(10); // return 5.5 = (1 + 10) / 2
movingAverage.next(3); // return 4.66667 = (1 + 10 + 3) / 3
movingAverage.next(5); // return 6.0 = (10 + 3 + 5) / 3
```

Constraints:

- 1 <= size <= 1000
- -105 <= val <= 105
- At most 104 calls will be made to next.

Python

Python

```python
# Python solution for Moving Average from Data Stream

from collections import deque # Import deque for efficient queue operations

class MovingAverage:
    def __init__(self, size):
        # Initialize with the maximum window size
        self.size = size # Maximum number of elements in the window
        self.queue = deque() # Queue to store current window values
        self.current_sum = 0 # Running sum of the current window

    def next(self, val):
        # Add new value to the window
        if len(self.queue) == self.size:
            # If the window is full, remove the oldest element and update the sum
            removed_val = self.queue.popleft() # Remove the front element
            self.current_sum -= removed_val # Subtract its value from the running sum
        # Append the new value to the queue and update the sum
        self.queue.append(val)
        self.current_sum += val

        # Calculate and return the moving average by dividing the running sum by the current window size
        return self.current_sum / len(self.queue)
```

Python


```python
class MovingAverage:
  def __init__(self, size: int):
    self.size = size
    self.sum = 0
    self.q = collections.deque()

  def next(self, val: int) -> float:
    if len(self.q) == self.size:
      self.sum -= self.q.popleft()
    self.sum += val
    self.q.append(val)
    return self.sum / len(self.q)
```

Python

```python
class MovingAverage:

    def __init__(self, size: int):
        self.s = 0
        self.data = [0] * size
        self.cnt = 0

    def next(self, val: int) -> float:
        i = self.cnt % len(self.data)
        self.s += val - self.data[i]
        self.data[i] = val
        self.cnt += 1
        return self.s / min(self.cnt, len(self.data))


# Your MovingAverage object will be instantiated and called as such:
# obj = MovingAverage(size)
# param_1 = obj.next(val)
```

Python

```python
class MovingAverage:
    def __init__(self, size: int):
        self.n = size
        self.s = 0
        self.q = deque()

    def next(self, val: int) -> float:
        if len(self.q) == self.n:
            self.s -= self.q.popleft()
        self.q.append(val)
        self.s += val
        return self.s / len(self.q)


# Your MovingAverage object will be instantiated and called as such:
# obj = MovingAverage(size)
# param_1 = obj.next(val)
```

Python


```python
class MovingAverage:
    def __init__(self, size: int):
        self.arr = [0] * size
        self.s = 0
        self.cnt = 0

    def next(self, val: int) -> float:
        idx = self.cnt % len(self.arr) # circular array
        self.s += val - self.arr[idx]
        self.arr[idx] = val
        self.cnt += 1
        return self.s / min(self.cnt, len(self.arr))


# Your MovingAverage object will be instantiated and called as such:
# obj = MovingAverage(size)
# param_1 = obj.next(val)

###########

from collections import deque

# with no sum variable, calculate on the fly
class MovingAverage:
    def __init__(self, size: int):

        self.queue = deque()
        self.size = size
        self.avg = 0

    def next(self, val: int) -> float:
        if len(self.queue) < self.size:
            self.queue.append(val)
            self.avg = sum(self.queue) / len(self.queue)
            return self.avg
        else:
            head = self.queue.popleft()
            self.queue.append(val)
            minus = head / self.size
            add = val / self.size
            self.avg = self.avg + add - minus
            return self.avg

###########

from collections import deque

class MovingAverage(object):

  def __init__(self, size):
    """
```



```
    Initialize your data structure here.
    :type size: int
    """
    self.windowSize = size
    self.windowSum = 0.0
    self.data = deque([])

  def next(self, val):
    """
    :type val: int
    :rtype: float
    """
    self.windowSum += val
    data = self.data

    leftTop = 0
    if len(data) >= self.windowSize:
      leftTop = data.popleft()
    data.append(val)

    self.windowSum -= leftTop
    if len(data) < self.windowSize:
      return self.windowSum / len(data)
    return self.windowSum / self.windowSize

# Your MovingAverage object will be instantiated and called as such:
# obj = MovingAverage(size)
# param_1 = obj.next(val)
```

[*] Arrays & Hashing — Pattern Solution (stored optimal)

Python

```python
# Python solution for Moving Average from Data Stream

from collections import deque # Import deque for efficient queue operations

class MovingAverage:
    def __init__(self, size):
        # Initialize with the maximum window size
        self.size = size # Maximum number of elements in the window
        self.queue = deque() # Queue to store current window values
        self.current_sum = 0 # Running sum of the current window

    def next(self, val):
        # Add new value to the window
        if len(self.queue) == self.size:
            # If the window is full, remove the oldest element and update the sum
            removed_val = self.queue.popleft() # Remove the front element
            self.current_sum -= removed_val # Subtract its value from the running sum
        # Append the new value to the queue and update the sum
        self.queue.append(val)
        self.current_sum += val

        # Calculate and return the moving average by dividing the running sum by the current window size
        return self.current_sum / len(self.queue)
```


#760 EASY

Find Anagram Mappings

LeetDocs · SimpleLeet · WalkCC · LeetCode · Editorial

Array · Hash Table

Lists: Premium 98

Problem:

You are given two integer arrays nums1 and nums2 where nums2 is an anagram of nums1. Both arrays may contain duplicates.

Return an index mapping array mapping from nums1 to nums2 where mapping[i] = j means the ith element in nums1 appears in nums2 at index j. If there are multiple answers, return any of them.

An array a is an anagram of an array b means b is made by randomizing the order of the elements in a.

Example 1:

```
Input: nums1 = [12,28,46,32,50], nums2 = [50,12,32,46,28]
Output: [1,4,3,2,0]
Explanation: As mapping[0] = 1 because the 0th element of nums1 appears at nums2[1], and
mapping[1] = 4 because the 1st element of nums1 appears at nums2[4], and so on.
```

Example 2:

```
Input: nums1 = [84,46], nums2 = [84,46]
Output: [0,1]
```

Constraints:

- 1 <= nums1.length <= 100
- nums2.length == nums1.length
- 0 <= nums1[i], nums2[i] <= 105
- nums2 is an anagram of nums1.

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def anagramMappings(self, nums1, nums2):
        # Brute Force O(n^2) - check all pairs without a hash map
        n = len(nums1)
        for i in range(n):
            for j in range(i + 1, n):
                # process pair nums1[i], nums1[j]
                pass
        return [] # adjust return
```

Python

Python


```python
# Build a dictionary mapping each number in nums2 to a list of its indices.
def anagramMappings(nums1, nums2):
    # Dictionary to store number : list of indices
    index_dict = {}
    for index, num in enumerate(nums2):
        if num in index_dict:
            index_dict[num].append(index)
        else:
            index_dict[num] = [index]

    mapping = []
    # Generate the mapping for each element in nums1.
    for num in nums1:
        # Pop the first available index for num and add it to mapping.
        mapping.append(index_dict[num].pop(0))
    return mapping

# Example usage:
nums1 = [12,28,46,32,50]
nums2 = [50,12,32,46,28]
print(anagramMappings(nums1, nums2)) # Output: [1,4,3,2,0] or any valid mapping
```

Python

```python
class Solution:
  def anagramMappings(self, nums1: list[int], nums2: list[int]) -> list[int]:
    numToIndices = collections.defaultdict(list)

    for i, num in enumerate(nums2):
      numToIndices[num].append(i)

    return [numToIndices[num].pop() for num in nums1]
```

Python

```python
class Solution:
    def anagramMappings(self, nums1: List[int], nums2: List[int]) -> List[int]:
        d = {x: i for i, x in enumerate(nums2)}
        return [d[x] for x in nums1]
```

Python

```python
class Solution:
    def anagramMappings(self, nums1: List[int], nums2: List[int]) -> List[int]:
        mapper = defaultdict(set)
        for i, num in enumerate(nums2):
            mapper[num].add(i)
        return [mapper[num].pop() for num in nums1]
```

[*] Arrays & Hashing — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def anagramMappings(self, nums1: list[int], nums2: list[int]) -> list[int]:
        numToIndices = collections.defaultdict(list)

        for i, num in enumerate(nums2):
            numToIndices[num].append(i)

        return [numToIndices[num].pop() for num in nums1]
```

#1426 EASY

Counting Elements

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table

Lists: Premium 98

Problem:

Given an integer array arr, count how many elements x there are, such that x + 1 is also in arr. If there are duplicates in arr, count them separately.

Example 1:

```
Input: arr = [1,2,3]
Output: 2
Explanation: 1 and 2 are counted cause 2 and 3 are in arr.
```

Example 2:

```
Input: arr = [1,1,3,3,5,5,7,7]
Output: 0
Explanation: No numbers are counted, cause there is no 2, 4, 6, or 8 in arr.
```

Constraints:

- 1 <= arr.length <= 1000
- 0 <= arr[i] <= 1000

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def count_elements(self, arr):
        # Brute Force O(n^2) - check all pairs without a hash map
        n = len(arr)
        for i in range(n):
            for j in range(i + 1, n):
                # process pair arr[i], arr[j]
                pass
        return [] # adjust return
```

Python

Python


```python
# Function to count elements where (x + 1) is in the array
def count_elements(arr):
    # Create a set of unique numbers for O(1) lookups
    unique_numbers = set(arr)
    count = 0
    # Iterate through each number in the array
    for number in arr:
        # If (number + 1) exists in the unique set, increment count
        if number + 1 in unique_numbers:
            count += 1
    return count

# Example usage:
# arr = [1, 2, 3]
# print(count_elements(arr)) # Expected output: 2
```

Python

```python
class Solution:
    def countElements(self, arr: list[int]) -> int:
        cnt = collections.Counter(arr)
        return sum(freq
                   for a, freq in cnt.items()
                   if cnt[a + 1] > 0)
```

Python

```python
class Solution:
    def countElements(self, arr: List[int]) -> int:
        cnt = Counter(arr)
        return sum(v for x, v in cnt.items() if cnt[x + 1])
```

Python

```python
class Solution:
    def countElements(self, arr: List[int]) -> int:
        cnt = Counter(arr)
        return sum(v for x, v in cnt.items() if cnt[x + 1])
```

[*] Arrays & Hashing — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def countElements(self, arr: list[int]) -> int:
        cnt = collections.Counter(arr)
        return sum(freq
                   for a, freq in cnt.items()
                   if cnt[a + 1] > 0)
```

#1534 EASY

Count Good Triplets

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Enumeration

Lists: CodeSignal

Problem:

Given an array of integers arr, and three integers a, b and c. You need to find the number of good triplets.

A triplet (arr[i], arr[j], arr[k]) is good if the following conditions are true:

- 0 <= i < j < k < arr.length
- |arr[i] − arr[j]| <= a
- |arr[j] − arr[k]| <= b
- |arr[i] − arr[k]| <= c

Where |x| denotes the absolute value of x.

Return the number of good triplets.

Example 1:

```
Input: arr = [3,0,1,1,9,7], a = 7, b = 2, c = 3

Output: 4

Explanation: There are 4 good triplets: [(3,0,1), (3,0,1), (3,1,1), (0,1,1)].
```

Example 2:

```
Input: arr = [1,1,2,2,3], a = 0, b = 0, c = 1

Output: 0

Explanation: No triplet satisfies all conditions.
```

Constraints:

- 3 <= arr.length <= 100
- 0 <= arr[i] <= 1000
- 0 <= a, b, c <= 1000

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def countGoodTriplets(self, arr, a, b, c):
        # Brute Force O(n^2) - check all pairs without a hash map
        n = len(arr)
        for i in range(n):
            for j in range(i + 1, n):
                # process pair arr[i], arr[j]
                pass
        return [] # adjust return
```

Python

Python


```python
# Python solution for Count Good Triplets
def countGoodTriplets(arr, a, b, c):
    # Initialize counter for good triplets
    good_triplets_count = 0
    n = len(arr)
    # Iterate over all triplet combinations with indices i, j, k where i < j < k
    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                # Check if absolute differences satisfy the conditions
                if abs(arr[i] - arr[j]) <= a and abs(arr[j] - arr[k]) <= b and abs(arr[i] - arr[k]) <= c:
                    good_triplets_count += 1
    # Return the count of good triplets
    return good_triplets_count

# Example usage:
arr = [3, 0, 1, 1, 9, 7]
a, b, c = 7, 2, 3
print(countGoodTriplets(arr, a, b, c)) # Expected output: 4
```

Python

```python
class Solution:
    def countGoodTriplets(self, arr: list[int], a: int, b: int, c: int) -> int:
        return sum(abs(arr[i] - arr[j]) <= a and
                   abs(arr[j] - arr[k]) <= b and
                   abs(arr[i] - arr[k]) <= c
                   for i, j, k in itertools.combinations(range(len(arr)), 3))
```

Python

```python
class Solution:
    def countGoodTriplets(self, arr: List[int], a: int, b: int, c: int) -> int:
        ans, n = 0, len(arr)
        for i in range(n):
            for j in range(i + 1, n):
                for k in range(j + 1, n):
                    ans += (
                        abs(arr[i] - arr[j]) <= a
                        and abs(arr[j] - arr[k]) <= b
                        and abs(arr[i] - arr[k]) <= c
                    )
        return ans
```

Python

```python
class Solution:
    def countGoodTriplets(self, arr: List[int], a: int, b: int, c: int) -> int:
        ans, n = 0, len(arr)
        for i in range(n):
            for j in range(i + 1, n):
                for k in range(j + 1, n):
                    ans += (
                        abs(arr[i] - arr[j]) <= a
                        and abs(arr[j] - arr[k]) <= b
                        and abs(arr[i] - arr[k]) <= c
                    )
        return ans
```


[*] Arrays & Hashing — Pattern Solution (stored optimal)

Python

```python
# Python solution for Count Good Triplets
def countGoodTriplets(arr, a, b, c):
    # Initialize counter for good triplets
    good_triplets_count = 0
    n = len(arr)
    # Iterate over all triplet combinations with indices i, j, k where i < j < k
    for i in range(n):
        for j in range(i + 1, n):
            for k in range(j + 1, n):
                # Check if absolute differences satisfy the conditions
                if abs(arr[i] - arr[j]) <= a and abs(arr[j] - arr[k]) <= b and abs(arr[i] - arr[k]) <= c:
                    good_triplets_count += 1
    # Return the count of good triplets
    return good_triplets_count

# Example usage:
arr = [3, 0, 1, 1, 9, 7]
a, b, c = 7, 2, 3
print(countGoodTriplets(arr, a, b, c)) # Expected output: 4
```

#57 MEDIUM

Insert Interval

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array

Lists: Grind 169

Problem:

You are given an array of non-overlapping intervals intervals where intervals[i] = [starti, endi] represent the start and the end of the ith interval and intervals is sorted in ascending order by starti. You are also given an interval newInterval = [start, end] that represents the start and end of another interval.

Insert newInterval into intervals such that intervals is still sorted in ascending order by starti and intervals still does not have any overlapping intervals (merge overlapping intervals if necessary).

Return intervals after the insertion.

Note that you don't need to modify intervals in-place. You can make a new array and return it.

Example 1:

```
Input: intervals = [[1,3],[6,9]], newInterval = [2,5]
Output: [[1,5],[6,9]]
```

Example 2:

```
Input: intervals = [[1,2],[3,5],[6,7],[8,10],[12,16]], newInterval = [4,8]
Output: [[1,2],[3,10],[12,16]]
Explanation: Because the new interval [4,8] overlaps with [3,5],[6,7],[8,10].
```

Constraints:

- 0 <= intervals.length <= 104
- intervals[i].length == 2

- 0 <= starti <= endi <= 105
- intervals is sorted by starti in ascending order.
- newInterval.length == 2
- 0 <= start <= end <= 105

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def insert(self, intervals, newInterval):
        # Brute Force O(n log n) - insert, sort, then merge like merge-intervals
        intervals.append(newInterval)
        intervals.sort(key=lambda x: x[0])
        result = [intervals[0]]
        for iv in intervals[1:]:
            if iv[0] <= result[-1][1]:
                result[-1][1] = max(result[-1][1], iv[1])
            else:
                result.append(iv)
        return result
```

Python

Python

```python
# Python solution for Insert Interval

class Solution:
    def insert(self, intervals, newInterval):
        # List to store the resulting intervals
        merged_intervals = []
        i = 0
        n = len(intervals)

        # Add all intervals that end before newInterval starts
        while i < n and intervals[i][1] < newInterval[0]:
            merged_intervals.append(intervals[i])
            i += 1

        # Merge overlapping intervals with newInterval
        while i < n and intervals[i][0] <= newInterval[1]:
            # Update the start and end of newInterval by merging with overlapping intervals
            newInterval[0] = min(newInterval[0], intervals[i][0])
            newInterval[1] = max(newInterval[1], intervals[i][1])
            i += 1
        # Add the merged newInterval
        merged_intervals.append(newInterval)

        # Add any remaining intervals that come after newInterval
        while i < n:

            merged_intervals.append(intervals[i])
            i += 1

        return merged_intervals
```

Python

```python
class Solution:
    def insert(self, intervals: list[list[int]],
               newInterval: list[int]) -> list[list[int]]:
        n = len(intervals)
        ans = []
        i = 0

        while i < n and intervals[i][1] < newInterval[0]:
            ans.append(intervals[i])
            i += 1

        # Merge overlapping intervals.
        while i < n and intervals[i][0] <= newInterval[1]:
            newInterval[0] = min(newInterval[0], intervals[i][0])
            newInterval[1] = max(newInterval[1], intervals[i][1])
            i += 1

        ans.append(newInterval)

        while i < n:
            ans.append(intervals[i])
            i += 1

        return ans
```

Python

```python
class Solution:
    def insert(
        self, intervals: List[List[int]], newInterval: List[int]
    ) -> List[List[int]]:
        def merge(intervals: List[List[int]]) -> List[List[int]]:
            intervals.sort()
            ans = [intervals[0]]
            for s, e in intervals[1:]:
                if ans[-1][1] < s:
                    ans.append([s, e])
                else:
                    ans[-1][1] = max(ans[-1][1], e)
            return ans

        intervals.append(newInterval)
        return merge(intervals)
```

Python


```python
'''
>>> a = [1,2,3,4,5]
>>> a[1]
2
>>> a[-1]
4
'''
class Solution:
    def insert(self, intervals: List[List[int]], newInterval: List[int]) -> List[List[int]]:
        start = newInterval[0]
        end = newInterval[1]

        left = list(filter(lambda x: x[1] < start, intervals)) # cast via list()
        right = list(filter(lambda x: x[0] > end, intervals))

        if left + right != intervals:
            start = min(start, intervals[len(left)][0]) # note, left not -1, because index starts at 0
            end = max(end, intervals[-len(right)][1]) # same, starting at right with index=0

        return left + [[start, end]] + right

######

class Solution: # re-use Leetcode-56's merge solution
    def insert(self, intervals: List[List[int]], newInterval: List[int]) -> List[List[int]]:
        intervals.append(newInterval)
        return self.merge(intervals)

    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        ans = []
        for intv in sorted(intervals, key=lambda x: x[0]):
            if ans and ans[-1][1] >= intv[0]:
                ans[-1][1] = max(ans[-1][1], intv[1])
            else:
                ans.append(intv)
        return ans

######

class Solution:
    def insert(
        self, intervals: List[List[int]], newInterval: List[int]
    ) -> List[List[int]]:
        def merge(intervals: List[List[int]]) -> List[List[int]]:
            intervals.sort()
            ans = [intervals[0]]
            for s, e in intervals[1:]:
                if ans[-1][1] < s:
                    ans.append([s, e])
                else:
                    ans[-1][1] = max(ans[-1][1], e)
            return ans

        intervals.append(newInterval)
        return merge(intervals)
```


[*] Arrays & Hashing — Pattern Solution (stored optimal)

Python

```python
# Python solution for Insert Interval

class Solution:
    def insert(self, intervals, newInterval):
        # List to store the resulting intervals
        merged_intervals = []
        i = 0
        n = len(intervals)

        # Add all intervals that end before newInterval starts
        while i < n and intervals[i][1] < newInterval[0]:
            merged_intervals.append(intervals[i])
            i += 1

        # Merge overlapping intervals with newInterval
        while i < n and intervals[i][0] <= newInterval[1]:
            # Update the start and end of newInterval by merging with overlapping intervals
            newInterval[0] = min(newInterval[0], intervals[i][0])
            newInterval[1] = max(newInterval[1], intervals[i][1])
            i += 1
        # Add the merged newInterval
        merged_intervals.append(newInterval)

        # Add any remaining intervals that come after newInterval
        while i < n:
            merged_intervals.append(intervals[i])
            i += 1

        return merged_intervals
```

#238 MEDIUM

Product of Array Except Self

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Prefix Sum

Lists: Grind 169

Problem:

Given an integer array nums, return an array answer such that answer[i] is equal to the product of all the elements of nums except nums[i].

The product of any prefix or suffix of nums is guaranteed to fit in a 32-bit integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

Example 1:

```
Input: nums = [1,2,3,4]
Output: [24,12,8,6]
```

Example 2:


```
Input: nums = [-1,1,0,-3,3]
Output: [0,0,9,0,0]
```

Constraints:

- 2 <= nums.length <= 105
- -30 <= nums[i] <= 30
- The input is generated such that answer[i] is guaranteed to fit in a 32-bit integer.

Follow up: Can you solve the problem in O(1) extra space complexity? (The output array does not count as extra space for space complexity analysis.)

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def productExceptSelf(self, nums):
        # Brute-force O(n^2) - for each index multiply all other elements
        n = len(nums)
        result = []
        for i in range(n):
            prod = 1
            for j in range(n):
                if i != j:
                    prod *= nums[j]
            result.append(prod)
        return result
```

Python

Python

```python
# Function to compute product of array except self
def productExceptSelf(nums):
    n = len(nums)
    # Initialize the output array with 1s for prefix multiplication
    answer = [1] * n

    # Compute prefix products: for each element answer[i] holds the product of all numbers left of i
    prefix_product = 1
    for i in range(n):
        answer[i] = prefix_product # store prefix product so far
        prefix_product *= nums[i] # update the prefix product

    # Compute suffix products and multiply with prefix products in answer array
    suffix_product = 1
    for i in range(n - 1, -1, -1):
        answer[i] *= suffix_product # multiply current answer with suffix product
        suffix_product *= nums[i] # update the suffix product

    return answer

# Example usage
nums = [1, 2, 3, 4]
print(productExceptSelf(nums)) # Output: [24, 12, 8, 6]
```

Python


```python
class Solution:
    def productExceptSelf(self, nums: list[int]) -> list[int]:
        n = len(nums)
        prefix = [1] * n # prefix product
        suffix = [1] * n # suffix product

        for i in range(1, n):
            prefix[i] = prefix[i - 1] * nums[i - 1]

        for i in reversed(range(n - 1)):
            suffix[i] = suffix[i + 1] * nums[i + 1]

        return [prefix[i] * suffix[i] for i in range(n)]
```

Python

```python
class Solution:
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        n = len(nums)
        ans = [0] * n
        left = right = 1
        for i, v in enumerate(nums):
            ans[i] = left
            left *= v
        for i in range(n - 1, -1, -1):
            ans[i] *= right
            right *= nums[i]
        return ans
```

Python

```python
class Solution: # 2 extra arrays
    def productExceptSelf(self, nums):
        if nums is None or len(nums) == 0:
            return nums

        # Product from index=0 to index=i-1
        from_left = [1] * len(nums)
        for i in range(1, len(nums)):
            from_left[i] = nums[i - 1] * from_left[i - 1]

        # Product from index=n-1 to current position
        from_right = [1] * len(nums)
        for i in range(len(nums) - 2, -1, -1):
            from_right[i] = nums[i + 1] * from_right[i + 1]

        # Calculate result
        result = [0] * len(nums)
        for i in range(len(nums)):
            result[i] = from_left[i] * from_right[i]

        return result

###########

class Solution: # follow up, no extra space
```



```python
    def productExceptSelf(self, nums: List[int]) -> List[int]:
        n = len(nums)
        ans = [0] * n
        left = right = 1
        for i, v in enumerate(nums):
            ans[i] = left
            left *= v
        for i in range(n - 1, -1, -1):
            ans[i] *= right
            right *= nums[i]
        return ans


###########


class Solution(object):
    def productExceptSelf(self, nums):
        """
        :type nums: List[int]
        :rtype: List[int]
        """
        dp = [1] * len(nums)
        for i in range(1, len(nums)):
            dp[i] = dp[i - 1] * nums[i - 1]

        prod = 1
        for i in reversed(range(0, len(nums))):
            dp[i] = dp[i] * prod
            prod *= nums[i]
        return dp
```

[*] Arrays & Hashing — Pattern Solution (stored optimal)

Python

```python
# Function to compute product of array except self
def productExceptSelf(nums):
    n = len(nums)
    # Initialize the output array with 1s for prefix multiplication
    answer = [1] * n

    # Compute prefix products: for each element answer[i] holds the product of all numbers left of i
    prefix_product = 1
    for i in range(n):
        answer[i] = prefix_product # store prefix product so far
        prefix_product *= nums[i] # update the prefix product

    # Compute suffix products and multiply with prefix products in answer array
    suffix_product = 1
    for i in range(n - 1, -1, -1):
        answer[i] *= suffix_product # multiply current answer with suffix product
        suffix_product *= nums[i] # update the suffix product

    return answer

# Example usage
nums = [1, 2, 3, 4]
print(productExceptSelf(nums)) # Output: [24, 12, 8, 6]
```

#281 MEDIUM

Zigzag Iterator

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Design · Queue · Iterator

Lists: Denny Zhang

Problem:

Given two vectors of integers v1 and v2, implement an iterator to return their elements alternately.

Implement the ZigzagIterator class:

- ZigzagIterator(List<int> v1, List<int> v2) initializes the object with the two vectors v1 and v2.
- boolean hasNext() returns true if the iterator still has elements, and false otherwise.
- int next() returns the current element of the iterator and moves the iterator to the next element.

Example 1:

```
Input: v1 = [1,2], v2 = [3,4,5,6]
Output: [1,3,2,4,5,6]
Explanation: By calling next repeatedly until hasNext returns false, the order of elements
returned by next should be: [1,3,2,4,5,6].
```

Example 2:

```
Input: v1 = [1], v2 = []
Output: [1]
```

Example 3:

```
Input: v1 = [], v2 = [1]
Output: [1]
```

Constraints:

- 0 <= v1.length, v2.length <= 1000
- 1 <= v1.length + v2.length <= 2000
- -231 <= v1[i], v2[i] <= 231 - 1

Follow up: What if you are given k vectors? How well can your code be extended to such cases?

Clarification for the follow-up question:

The "Zigzag" order is not clearly defined and is ambiguous for k > 2 cases. If "Zigzag" does not look right to you, replace "Zigzag" with "Cyclic".

Follow-up Example:

```
Input: v1 = [1,2,3], v2 = [4,5,6,7], v3 = [8,9]
Output: [1,4,8,2,5,9,3,6,7]
```

Python

Python


```python
class ZigzagIterator:
    def __init__(self, v1, v2):
        # Initialize the queue with non-empty vectors paired with starting index 0
        from collections import deque
        self.queue = deque()
        if v1:
            self.queue.append((v1, 0))
        if v2:
            self.queue.append((v2, 0))

    def next(self):
        # Pop from the left
        vec, idx = self.queue.popleft()
        result = vec[idx]
        # If there are more elements, put it back with the next index
        if idx + 1 < len(vec):
            self.queue.append((vec, idx + 1))
        return result

    def hasNext(self):
        return len(self.queue) > 0

# Example usage:
# v1 = [1,2]; v2 = [3,4,5,6]
# iterator = ZigzagIterator(v1, v2)

# while iterator.hasNext():
# print(iterator.next(), end=" ")
```

Python

```python
class ZigzagIterator:
  def __init__(self, v1: list[int], v2: list[int]):
    def vals():
      for i in itertools.count():
        for v in v1, v2:
          if i < len(v):
            yield v[i]
    self.vals = vals()
    self.n = len(v1) + len(v2)

  def next(self):
    self.n -= 1
    return next(self.vals)

  def hasNext(self):
    return self.n > 0
```

Python


```python
class ZigzagIterator:
    def __init__(self, v1: List[int], v2: List[int]):
        self.cur = 0
        self.size = 2
        self.indexes = [0] * self.size
        self.vectors = [v1, v2]

    def next(self) -> int:
        vector = self.vectors[self.cur]
        index = self.indexes[self.cur]
        res = vector[index]
        self.indexes[self.cur] = index + 1
        self.cur = (self.cur + 1) % self.size
        return res

    def hasNext(self) -> bool:
        start = self.cur
        while self.indexes[self.cur] == len(self.vectors[self.cur]):
            self.cur = (self.cur + 1) % self.size
            if self.cur == start:
                return False
        return True


# Your ZigzagIterator object will be instantiated and called as such:
# i, v = ZigzagIterator(v1, v2), []
# while i.hasNext(): v.append(i.next())
```

Python

```python
class ZigzagIteratorK:

    # int[0] is index of vector of vectors list, int[1] is start of a vector, int[2] is end
    q = []

    # no map needed, if move 'vectors' a class variable
    vectors = []

    def __init__(self, vectors):
        self.vectors = vectors
        for i in range(len(vectors)):
            self.q.append([i, 0, len(vectors[i])])

    def next(self):
        current = self.q.pop(0)

        i = current[0]
        start = current[1]
        end = current[2]

        if start + 1 < end:
            self.q.append([i, start + 1, end])

        return self.vectors[i][start]
```



```python
    def hasNext(self):
        return len(self.q) != 0

###########

class ZigzagIterator:
    def __init__(self, v1: List[int], v2: List[int]):
        self.cur = 0
        self.size = 2
        self.indexes = [0] * self.size
        self.vectors = [v1, v2]

    def next(self) -> int:
        vector = self.vectors[self.cur]
        index = self.indexes[self.cur]
        res = vector[index]
        self.indexes[self.cur] = index + 1
        self.cur = (self.cur + 1) % self.size
        return res

    def hasNext(self) -> bool:
        start = self.cur
        while self.indexes[self.cur] == len(self.vectors[self.cur]):
            self.cur = (self.cur + 1) % self.size
            if self.cur == start:
                return False
        return True


# Your ZigzagIterator object will be instantiated and called as such:
# i, v = ZigzagIterator(v1, v2), []
# while i.hasNext(): v.append(i.next())

###########

'''
iter() - builtin function, return an iterator object
    https://docs.python.org/3/library/functions.html#iter


    >>> a = map(iter, [[1,2],[3,4]])
    >>>
    >>> b = next(a)
    >>> b
    <list_iterator object at 0x101630280>
    >>> next(b)
    1
    >>> next(b)
    2
    >>> next(b)

    Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
    StopIteration
```

[*] Arrays & Hashing — Pattern Solution (stored optimal)

Python


```python
class ZigzagIterator:
    def __init__(self, v1, v2):
        # Initialize the queue with non-empty vectors paired with starting index 0
        from collections import deque
        self.queue = deque()
        if v1:
            self.queue.append((v1, 0))
        if v2:
            self.queue.append((v2, 0))

    def next(self):
        # Pop from the left
        vec, idx = self.queue.popleft()
        result = vec[idx]
        # If there are more elements, put it back with the next index
        if idx + 1 < len(vec):
            self.queue.append((vec, idx + 1))
        return result

    def hasNext(self):
        return len(self.queue) > 0

# Example usage:
# v1 = [1,2]; v2 = [3,4,5,6]
# iterator = ZigzagIterator(v1, v2)

# while iterator.hasNext():
# print(iterator.next(), end=" ")
```

#307 MEDIUM

Range Sum Query - Mutable

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Design · Binary Indexed Tree · Segment Tree

Lists: Denny Zhang

Problem:

Given an integer array nums, handle multiple queries of the following types:

1. Update the value of an element in nums.
2. Calculate the sum of the elements of nums between indices left and right inclusive where left <= right.

Implement the NumArray class:

- NumArray(int[] nums) Initializes the object with the integer array nums.
- void update(int index, int val) Updates the value of nums[index] to be val.
- int sumRange(int left, int right) Returns the sum of the elements of nums between indices left and right inclusive (i.e. nums[left] + nums[left + 1] + ... + nums[right]).

Example 1:

```
Input
["NumArray", "sumRange", "update", "sumRange"]
[[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]
Output
```



```
[null, 9, null, 8]

Explanation
NumArray numArray = new NumArray([1, 3, 5]);
numArray.sumRange(0, 2); // return 1 + 3 + 5 = 9
numArray.update(1, 2); // nums = [1, 2, 5]
numArray.sumRange(0, 2); // return 1 + 2 + 5 = 8
```

Constraints:

- 1 <= nums.length <= 3 * 104
- -100 <= nums[i] <= 100
- 0 <= index < nums.length
- -100 <= val <= 100
- 0 <= left <= right < nums.length
- At most 3 * 104 calls will be made to update and sumRange.

Python

Python

```python
# Python implementation using a Binary Indexed Tree (Fenwick Tree)
class NumArray:
    def __init__(self, nums):
        # Store the original array
        self.nums = nums[:]
        self.n = len(nums)
        # Initialize the BIT with 0s, size is (n+1) because BIT is 1-indexed
        self.tree = [0] * (self.n + 1)
        # Build the BIT: update tree for each value in the nums array
        for i, num in enumerate(nums):
            self._update_tree(i, num)

    def _update_tree(self, index, delta):
        # Update the BIT for index with delta change
        i = index + 1 # Convert to 1-indexed for BIT operations
        while i <= self.n:
            self.tree[i] += delta # Add delta to current tree node
            i += i & -i # Move to the next responsible node

    def update(self, index, val):
        # Calculate the difference between new value and current value
        delta = val - self.nums[index]
        self.nums[index] = val # Update the original array
        self._update_tree(index, delta) # Update the BIT to reflect this change
```

```python
    def _prefix_sum(self, index):
        # Compute the prefix sum from start up to and including index
        result = 0
        i = index + 1 # Convert to 1-indexed
        while i:
            result += self.tree[i]
            i -= i & -i # Move to parent node
        return result

    def sumRange(self, left, right):
        # Return the sum of the range [left, right]
        return self._prefix_sum(right) - self._prefix_sum(left - 1) if left > 0 else
self._prefix_sum(right)

# Example usage:
# numArray = NumArray([1, 3, 5])
# print(numArray.sumRange(0, 2))
# numArray.update(1, 2)
# print(numArray.sumRange(0, 2))
```

Python

```python
class FenwickTree:
    def __init__(self, n: int):
        self.sums = [0] * (n + 1)

    def add(self, i: int, delta: int) -> None:
        while i < len(self.sums):
            self.sums[i] += delta
            i += FenwickTree.lowbit(i)

    def get(self, i: int) -> int:
        sum = 0
        while i > 0:
            sum += self.sums[i]
            i -= FenwickTree.lowbit(i)
        return sum

    @staticmethod
    def lowbit(i: int) -> int:
        return i & -i


class NumArray:
    def __init__(self, nums: list[int]):
        self.nums = nums
        self.tree = FenwickTree(len(nums))

        for i, num in enumerate(nums):
            self.tree.add(i + 1, num)

    def update(self, index: int, val: int) -> None:
        self.tree.add(index + 1, val - self.nums[index])
        self.nums[index] = val

    def sumRange(self, left: int, right: int) -> int:
        return self.tree.get(right + 1) - self.tree.get(left)
```

Python


```python
class BinaryIndexedTree:
    __slots__ = ("n", "c")

    def __init__(self, n):
        self.n = n
        self.c = [0] * (n + 1)

    def update(self, x: int, delta: int):
        while x <= self.n:
            self.c[x] += delta
            x += x & -x

    def query(self, x: int) -> int:
        s = 0
        while x > 0:
            s += self.c[x]
            x -= x & -x
        return s


class NumArray:
    __slots__ = ("tree")

    def __init__(self, nums: List[int]):
        self.tree = BinaryIndexedTree(len(nums))

        for i, v in enumerate(nums, 1):
            self.tree.update(i, v)

    def update(self, index: int, val: int) -> None:
        prev = self.sumRange(index, index)
        self.tree.update(index + 1, val - prev)

    def sumRange(self, left: int, right: int) -> int:
        return self.tree.query(right + 1) - self.tree.query(left)


# Your NumArray object will be instantiated and called as such:
# obj = NumArray(nums)
# obj.update(index,val)
# param_2 = obj.sumRange(left,right)
```

Python


```python
class BinaryIndexedTree:
    def __init__(self, n):
        self.n = n
        self.c = [0] * (n + 1)

    @staticmethod
    def lowbit(x):
        return x & -x

    def update(self, x, delta):
        while x <= self.n:
            self.c[x] += delta
            x += BinaryIndexedTree.lowbit(x)

    def query(self, x):
        s = 0
        while x > 0:
            s += self.c[x]
            x -= BinaryIndexedTree.lowbit(x)
        return s


class NumArray:
    def __init__(self, nums: List[int]):
        self.tree = BinaryIndexedTree(len(nums))

        for i, v in enumerate(nums, 1):
            self.tree.update(i, v)

    def update(self, index: int, val: int) -> None:
        prev = self.sumRange(index, index)
        self.tree.update(index + 1, val - prev)

    def sumRange(self, left: int, right: int) -> int:
        return self.tree.query(right + 1) - self.tree.query(left)


# Your NumArray object will be instantiated and called as such:
# obj = NumArray(nums)
# obj.update(index,val)
# param_2 = obj.sumRange(left,right)

###########

# Segment tree node
class STNode(object):
    def __init__(self, start, end):
        self.start = start
        self.end = end
        self.total = 0
        self.left = None
```



```python
        self.right = None


class SegmentedTree(object):
    def __init__(self, nums, start, end):
        self.root = self.buildTree(nums, start, end)

    def buildTree(self, nums, start, end):
        if start > end:
            return None

        if start == end:
            node = STNode(start, end)
            node.total = nums[start]
            return node

        mid = start + (end - start) / 2

        root = STNode(start, end)
        root.left = self.buildTree(nums, start, mid)
        root.right = self.buildTree(nums, mid + 1, end)
        root.total = root.left.total + root.right.total
        return root

    def updateVal(self, i, val):

        def updateVal(root, i, val):
            if root.start == root.end:
                root.total = val
                return val
            mid = root.start + (root.end - root.start) / 2
```

[*] Arrays & Hashing — Pattern Solution (stored optimal)

Python


```python
# Python implementation using a Binary Indexed Tree (Fenwick Tree)
class NumArray:
    def __init__(self, nums):
        # Store the original array
        self.nums = nums[:]
        self.n = len(nums)
        # Initialize the BIT with 0s, size is (n+1) because BIT is 1-indexed
        self.tree = [0] * (self.n + 1)
        # Build the BIT: update tree for each value in the nums array
        for i, num in enumerate(nums):
            self._update_tree(i, num)

    def _update_tree(self, index, delta):
        # Update the BIT for index with delta change
        i = index + 1 # Convert to 1-indexed for BIT operations
        while i <= self.n:
            self.tree[i] += delta # Add delta to current tree node
            i += i & -i # Move to the next responsible node

    def update(self, index, val):
        # Calculate the difference between new value and current value
        delta = val - self.nums[index]
        self.nums[index] = val # Update the original array
        self._update_tree(index, delta) # Update the BIT to reflect this change

    def _prefix_sum(self, index):
        # Compute the prefix sum from start up to and including index
        result = 0
        i = index + 1 # Convert to 1-indexed
        while i:
            result += self.tree[i]
            i -= i & -i # Move to parent node
        return result

    def sumRange(self, left, right):
        # Return the sum of the range [left, right]
        return self._prefix_sum(right) - self._prefix_sum(left - 1) if left > 0 else
self._prefix_sum(right)

# Example usage:
# numArray = NumArray([1, 3, 5])
# print(numArray.sumRange(0, 2))
# numArray.update(1, 2)
# print(numArray.sumRange(0, 2))
```

#454 MEDIUM

4Sum II

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table

Lists: CodeSignal

Problem:

Given four integer arrays nums1, nums2, nums3, and nums4 all of length n, return the number of tuples (i, j, k, l) such that:

- 0 <= i, j, k, l < n
- nums1[i] + nums2[j] + nums3[k] + nums4[l] == 0

Example 1:

```
Input: nums1 = [1,2], nums2 = [-2,-1], nums3 = [-1,2], nums4 = [0,2]
Output: 2
Explanation:
The two tuples are:
1. (0, 0, 0, 1) -> nums1[0] + nums2[0] + nums3[0] + nums4[1] = 1 + (-2) + (-1) + 2 = 0
2. (1, 1, 0, 0) -> nums1[1] + nums2[1] + nums3[0] + nums4[0] = 2 + (-1) + (-1) + 0 = 0
```

Example 2:

```
Input: nums1 = [0], nums2 = [0], nums3 = [0], nums4 = [0]
Output: 1
```

Constraints:

- n == nums1.length
- n == nums2.length
- n == nums3.length
- n == nums4.length
- 1 <= n <= 200
- -228 <= nums1[i], nums2[i], nums3[i], nums4[i] <= 228

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def four_sum_count(self, nums1, nums2, nums3, nums4):
        # Brute Force O(n^2) - check all pairs without a hash map
        n = len(nums1)
        for i in range(n):
            for j in range(i + 1, n):
                # process pair nums1[i], nums1[j]
                pass
        return [] # adjust return
```

Python

Python

```python
# Function to count quadruplets that sum to zero using a hash map approach
def four_sum_count(nums1, nums2, nums3, nums4):
    # Dictionary to store the frequency of sums of elements from nums1 and nums2
    sum_count = {}
    for num1 in nums1:
        for num2 in nums2:
            current_sum = num1 + num2
            sum_count[current_sum] = sum_count.get(current_sum, 0) + 1

    count = 0
    # For each pair from nums3 and nums4, find the complement that makes the total sum zero
    for num3 in nums3:
        for num4 in nums4:
            complement = -(num3 + num4)
            count += sum_count.get(complement, 0)
    return count

# Example usage
nums1 = [1,2]
nums2 = [-2,-1]
nums3 = [-1,2]
nums4 = [0,2]
print(four_sum_count(nums1, nums2, nums3, nums4)) # Expected Output: 2
```

Python

```python
class Solution:
   def fourSumCount(self, nums1: list[int], nums2: list[int],
                    nums3: list[int], nums4: list[int]) -> int:
    count = collections.Counter(a + b for a in nums1 for b in nums2)
    return sum(count[-c - d] for c in nums3 for d in nums4)
```

Python

```python
class Solution:
    def fourSumCount(
        self, nums1: List[int], nums2: List[int], nums3: List[int], nums4: List[int]
    ) -> int:
        cnt = Counter(a + b for a in nums1 for b in nums2)
        return sum(cnt[-(c + d)] for c in nums3 for d in nums4)
```

Python


```python
from collections import Counter

class Solution:
    def fourSumCount(
        self, nums1: List[int], nums2: List[int], nums3: List[int], nums4: List[int]
    ) -> int:
        cnt = Counter(a + b for a in nums1 for b in nums2)
        return sum(cnt[-(c + d)] for c in nums3 for d in nums4)

###########

class Solution:
    def fourSumCount(
        self, nums1: List[int], nums2: List[int], nums3: List[int], nums4: List[int]
    ) -> int:
        counter = Counter()
        for a in nums1:
            for b in nums2:
                counter[a + b] += 1

        ans = 0
        for c in nums3:
            for d in nums4:
                # adding for every time -(c+d)
                ans += counter[-(c + d)]
        return ans

###########

class Solution(object):
  def fourSumCount(self, A, B, C, D):
    """
    :type A: List[int]
    :type B: List[int]
    :type C: List[int]
    :type D: List[int]
    :rtype: int
    """
    ans = 0
    abDict = {}
    for i in range(len(A)):
      for j in range(len(B)):
        if A[i] + B[j] not in abDict:
          abDict[A[i] + B[j]] = 1
        else:
          abDict[A[i] + B[j]] += 1

    for i in range(len(C)):
      for j in range(len(D)):
        if -C[i] - D[j] in abDict:
          ans += abDict[-C[i] - D[j]]

    return ans
```

[*] Arrays & Hashing — Pattern Solution (matched from community answers)

Python


```python
class Solution:
   def fourSumCount(self, nums1: list[int], nums2: list[int],
                    nums3: list[int], nums4: list[int]) -> int:
    count = collections.Counter(a + b for a in nums1 for b in nums2)
    return sum(count[-c - d] for c in nums3 for d in nums4)
```

#525 MEDIUM

Contiguous Array

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Prefix Sum

Lists: Grind 169

Problem:

Given a binary array nums, return the maximum length of a contiguous subarray with an equal number of 0 and 1.

Example 1:

```
Input: nums = [0,1]
Output: 2
Explanation: [0, 1] is the longest contiguous subarray with an equal number of 0 and 1.
```

Example 2:

```
Input: nums = [0,1,0]
Output: 2
Explanation: [0, 1] (or [1, 0]) is a longest contiguous subarray with equal number of 0 and 1.
```

Example 3:

```
Input: nums = [0,1,1,1,1,1,0,0,0]
Output: 6
Explanation: [1,1,1,0,0,0] is the longest contiguous subarray with equal number of 0 and 1.
```

Constraints:

- 1 <= nums.length <= 105
- nums[i] is either 0 or 1.

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findMaxLength(self, nums):
        # Brute Force O(n^2) - check all pairs without a hash map
        n = len(nums)
        for i in range(n):
            for j in range(i + 1, n):
                # process pair nums[i], nums[j]
                pass
        return [] # adjust return
```

Python

Python

```python
# Python solution for Contiguous Array

def findMaxLength(nums):
    # Dictionary to store the earliest occurrence of each prefix sum
    prefix_indices = {0: -1}
    prefix_sum = 0
    max_length = 0

    # Iterate over the array
    for index, num in enumerate(nums):
        # Convert 0 to -1, keep 1 as is
        if num == 0:
            prefix_sum -= 1
        else:
            prefix_sum += 1

        # If the prefix sum has been seen before, a balanced subarray exists
        if prefix_sum in prefix_indices:
            # Calculate the length of the subarray
            current_length = index - prefix_indices[prefix_sum]
            max_length = max(max_length, current_length)
        else:
            # Store the index for this prefix sum if not already seen
            prefix_indices[prefix_sum] = index

    # Return the maximum length of balanced subarray
    return max_length

# Example usage:
# print(findMaxLength([0,1,0])) # Expected output: 2
```

Python

```python
class Solution:
   def findMaxLength(self, nums: list[int]) -> int:
    ans = 0
    prefix = 0
    prefixToIndex = {0: -1}

    for i, num in enumerate(nums):
      prefix += 1 if num else -1
      ans = max(ans, i - prefixToIndex.setdefault(prefix, i))

    return ans
```

Python


```python
class Solution:
    def findMaxLength(self, nums: List[int]) -> int:
        d = {0: -1}
        ans = i = 0
        for i, x in enumerate(nums):
            s += 1 if x else -1
            if s in d:
                ans = max(ans, i - d[s])
            else:
                d[s] = i
        return ans
```

Python

```python
class Solution:
    def findMaxLength(self, nums: List[int]) -> int:
        s = ans = 0
        mp = {0: -1}
        for i, v in enumerate(nums):
            s += 1 if v == 1 else -1
            if s in mp:
                ans = max(ans, i - mp[s])
            else:
                mp[s] = i
        return ans
```

[*] Arrays & Hashing — Pattern Solution (stored optimal)

Python

```python
# Python solution for Contiguous Array

def findMaxLength(nums):
    # Dictionary to store the earliest occurrence of each prefix sum
    prefix_indices = {0: -1}
    prefix_sum = 0
    max_length = 0

    # Iterate over the array
    for index, num in enumerate(nums):
        # Convert 0 to -1, keep 1 as is
        if num == 0:
            prefix_sum -= 1
        else:
            prefix_sum += 1

        # If the prefix sum has been seen before, a balanced subarray exists
        if prefix_sum in prefix_indices:
            # Calculate the length of the subarray
            current_length = index - prefix_indices[prefix_sum]
            max_length = max(max_length, current_length)
        else:
            # Store the index for this prefix sum if not already seen
            prefix_indices[prefix_sum] = index
```



```python
    # Return the maximum length of balanced subarray
    return max_length

# Example usage:
# print(findMaxLength([0,1,0])) # Expected output: 2
```

#560 MEDIUM

Subarray Sum Equals K

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Prefix Sum

Lists: Grind 169

Problem:

Given an array of integers nums and an integer k, return the total number of subarrays whose sum equals to k.

A subarray is a contiguous non-empty sequence of elements within an array.

Example 1:

```
Input: nums = [1,1,1], k = 2
Output: 2
```

Example 2:

```
Input: nums = [1,2,3], k = 3
Output: 2
```

Constraints:

- 1 <= nums.length <= 2 * 104
- -1000 <= nums[i] <= 1000
- -107 <= k <= 107

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def subarraySum(self, nums, k):
        # Brute Force O(n^2) - compute sum of every contiguous subarray
        count = 0
        for i in range(len(nums)):
            total = 0
            for j in range(i, len(nums)):
                total += nums[j]
                if total == k:
                    count += 1
        return count
```

Python

Python

```python
# Function to calculate the number of subarrays that sum to k
def subarraySum(nums, k):
    # Dictionary to store frequency of prefix sums
    prefix_counts = {0: 1} # Initialize with prefix sum 0
    current_sum = 0 # Running sum initialization
    count = 0 # To count the valid subarrays
    # Iterate over the array
    for num in nums:
        current_sum += num # Update the running (prefix) sum
        # Check if there's a prefix sum that differs from current_sum by k
        count += prefix_counts.get(current_sum - k, 0)
        # Update the frequency count for the current sum
        prefix_counts[current_sum] = prefix_counts.get(current_sum, 0) + 1
    return count

# Example usage:
print(subarraySum([1,1,1], 2)) # Expected output: 2
```

Python

```python
class Solution:
  def subarraySum(self, nums: list[int], k: int) -> int:
    ans = 0
    prefix = 0
    count = collections.Counter({0: 1})

    for num in nums:
      prefix += num
      ans += count[prefix - k]
      count[prefix] += 1

    return ans
```

Python

```python
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        cnt = Counter({0: 1})
        ans = s = 0
        for x in nums:
            s += x
            ans += cnt[s - k]
            cnt[s] += 1
        return ans
```

Python


```python
class Solution:
    def subarraySum(self, nums: List[int], k: int) -> int:
        dp = Counter({0: 1})
        ans = s = 0
        for num in nums:
            s += num
            ans += dp[s - k] # not matter if the same num like [1,1]
            dp[s] += 1
        return ans

############

class Solution: # over time limit, not fast enough
    def subarraySum(self, nums: List[int], k: int) -> int:
        count = 0

        # sum value from 1 to i-th element
        sum_arr = [0] * (len(nums) + 1)

        # cached sum
        for i in range(1, len(nums) + 1):
            sum_arr[i] = sum_arr[i - 1] + nums[i - 1]

        for start in range(len(nums)):
            for end in range(start + 1, len(nums) + 1):

                if sum_arr[end] - sum_arr[start] == k:
                    count += 1

        return count
```

[*] Arrays & Hashing — Pattern Solution (matched from community answers)

Python

```python
class Solution:
  def subarraySum(self, nums: list[int], k: int) -> int:
    ans = 0
    prefix = 0
    count = collections.Counter({0: 1})

    for num in nums:
      prefix += num
      ans += count[prefix - k]
      count[prefix] += 1

    return ans
```

#1010 MEDIUM

Pairs of Songs With Total Durations Divisible by 60

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Counting

Lists: CodeSignal

Problem:

You are given a list of songs where the ith song has a duration of time[i] seconds.

Return the number of pairs of songs for which their total duration in seconds is divisible by 60. Formally, we want the number of indices i, j such that i < j with (time[i] + time[j]) % 60 == 0.

Example 1:

```
Input: time = [30,20,150,100,40]
Output: 3
Explanation: Three pairs have a total duration divisible by 60:
(time[0] = 30, time[2] = 150): total duration 180
(time[1] = 20, time[3] = 100): total duration 120
(time[1] = 20, time[4] = 40): total duration 60
```

Example 2:

```
Input: time = [60,60,60]
Output: 3
Explanation: All three pairs have a total duration of 120, which is divisible by 60.
```

Constraints:

- 1 <= time.length <= 6 * 104
- 1 <= time[i] <= 500

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def numPairsDivisibleBy60(self, time):
        # Brute Force O(n^2) - check all pairs without a hash map
        n = len(time)
        for i in range(n):
            for j in range(i + 1, n):
                # process pair time[i], time[j]
                pass
        return [] # adjust return
```

Python

Python


```python
# Python solution for counting pairs of songs whose duration sums are divisible by 60
def numPairsDivisibleBy60(time):
    # Frequency array for remainders 0-59
    remainder_freq = [0] * 60
    pair_count = 0

    # Loop over each song duration
    for t in time:
        # Calculate the remainder when dividing by 60
        remainder = t % 60
        # Calculate the complementary remainder needed for the sum to be divisible by 60
        complement = (60 - remainder) % 60
        # Increase pair count by frequency of complement seen so far
        pair_count += remainder_freq[complement]
        # Update the frequency counter for the current remainder
        remainder_freq[remainder] += 1

    return pair_count

# Example usage
# print(numPairsDivisibleBy60([30,20,150,100,40])) # Expected output: 3
```

Python

```python
class Solution:
  def numPairsDivisibleBy60(self, time: list[int]) -> int:
    ans = 0
    count = [0] * 60

    for t in time:
      t %= 60
      ans += count[(60 - t) % 60]
      count[t] += 1

    return ans
```

Python

```python
class Solution:
    def numPairsDivisibleBy60(self, time: List[int]) -> int:
        cnt = Counter()
        ans = 0
        for x in time:
            x %= 60
            y = (60 - x) % 60
            ans += cnt[y]
            cnt[x] += 1
        return ans
```

Python

```python
class Solution:
    def numPairsDivisibleBy60(self, time: List[int]) -> int:
        cnt = Counter(t % 60 for t in time)
        ans = sum(cnt[x] * cnt[60 - x] for x in range(1, 30))
        ans += cnt[0] * (cnt[0] - 1) // 2
        ans += cnt[30] * (cnt[30] - 1) // 2
        return ans
```


[*] Arrays & Hashing — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def numPairsDivisibleBy60(self, time: List[int]) -> int:
        cnt = Counter()
        ans = 0
        for x in time:
            x %= 60
            y = (60 - x) % 60
            ans += cnt[y]
            cnt[x] += 1
        return ans
```

#1272 MEDIUM

Remove Interval

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array

Lists: Premium 98

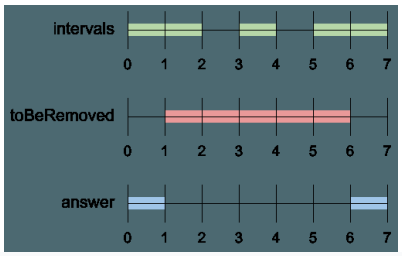
Problem:

A set of real numbers can be represented as the union of several disjoint intervals, where each interval is in the form [a, b). A real number x is in the set if one of its intervals [a, b) contains x (i.e. a <= x < b).

You are given a sorted list of disjoint intervals intervals representing a set of real numbers as described above, where intervals[i] = [ai, bi] represents the interval [ai, bi). You are also given another interval toBeRemoved.

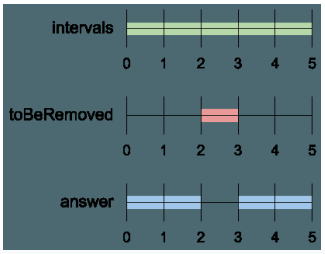
Return the set of real numbers with the interval toBeRemoved removed from intervals. In other words, return the set of real numbers such that every x in the set is in intervals but not in toBeRemoved. Your answer should be a sorted list of disjoint intervals as described above.

Example 1:




```
Input: intervals = [[0,2],[3,4],[5,7]], toBeRemoved = [1,6]
Output: [[0,1],[6,7]]
```

Example 2:




```
Input: intervals = [[0,5]], toBeRemoved = [2,3]
Output: [[0,2],[3,5]]
```

Example 3:

```
Input: intervals = [[-5,-4],[-3,-2],[1,2],[3,5],[8,9]], toBeRemoved = [-1,4]
Output: [[-5,-4],[-3,-2],[4,5],[8,9]]
```

Constraints:

- 1 <= intervals.length <= 104
- -109 <= ai < bi <= 109

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def removeInterval(self, intervals, toBeRemoved):
        # Brute Force O(n^2) - check all pairs without a hash map
        n = len(intervals)
        for i in range(n):
            for j in range(i + 1, n):
                # process pair intervals[i], intervals[j]
                pass
        return [] # adjust return
```

Python

Python

```python
def removeInterval(intervals, toBeRemoved):
    # Unpack the removal interval boundaries.
    remove_start, remove_end = toBeRemoved
    result = []

    # Iterate over each interval.
    for interval in intervals:
        start, end = interval

        # If the interval is completely outside toBeRemoved, add as is.
        if end <= remove_start or start >= remove_end:
            result.append([start, end])
        else:
            # If there is a non-overlapping left part, add it.
            if start < remove_start:
                result.append([start, remove_start])
            # If there is a non-overlapping right part, add it.
            if end > remove_end:
                result.append([remove_end, end])

    return result

# Example usage:
intervals = [[0,2],[3,4],[5,7]]
toBeRemoved = [1,6]

print(removeInterval(intervals, toBeRemoved)) # Expected output: [[0,1],[6,7]]
```

Python


```python
class Solution:
    def removeInterval(self, intervals: list[list[int]],
                       toBeRemoved: list[int]) -> list[list[int]]:

    ans = []

    for a, b in intervals:
      if a >= toBeRemoved[1] or b <= toBeRemoved[0]:
        ans.append([a, b])
      else: # a < toBeRemoved[1] and b > toBeRemoved[0]
        if a < toBeRemoved[0]:
          ans.append([a, toBeRemoved[0]])
        if b > toBeRemoved[1]:
          ans.append([toBeRemoved[1], b])

    return ans
```

Python

```python
class Solution:
    def removeInterval(
        self, intervals: List[List[int]], toBeRemoved: List[int]
    ) -> List[List[int]]:
        x, y = toBeRemoved
        ans = []
        for a, b in intervals:
            if a >= y or b <= x:
                ans.append([a, b])
            else:
                if a < x:
                    ans.append([a, x])
                if b > y:
                    ans.append([y, b])
        return ans
```

Python


```
'''
>>> a
[1, 2, 3, 4, 5]
>>>
>>> x, y = a
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: too many values to unpack
>>> x,y,z,p,q = a
>>> x
1
>>> y
2
>>> z
3
'''
class Solution:
    def removeInterval(
        self, intervals: List[List[int]], toBeRemoved: List[int]
    ) -> List[List[int]]:
        x, y = toBeRemoved
        ans = []
        for a, b in intervals:
            if a >= y or b <= x:
                ans.append([a, b])

            else:
                if a < x:
                    ans.append([a, x]) # in this else block, meaning b>x, so no need to append
[a,min(b,x)]

                if b > y:
                    ans.append([y, b])
                # if x < a < b < y, then delete (a,b), do nothing and skip here
        return ans

###########

class Solution:
    def removeInterval(
        self, intervals: List[List[int]], toBeRemoved: List[int]
    ) -> List[List[int]]:
        result = []

        for ind, intv in enumerate(intervals):
            if intv[1] < toBeRemoved[0] or toBeRemoved[1] < intv[0]:
                result.append(intv)
            else:
                if intv[0] < toBeRemoved[0]:
                    result.append([intv[0], min(intv[1], toBeRemoved[0])]) # min() is not needed
                if intv[1] > toBeRemoved[1]:
                    result.append([max(intv[0], toBeRemoved[1]), intv[1]])

        return result
```

[*] Arrays & Hashing — Pattern Solution (stored optimal)

Python


```python
def removeInterval(intervals, toBeRemoved):
    # Unpack the removal interval boundaries.
    remove_start, remove_end = toBeRemoved
    result = []

    # Iterate over each interval.
    for interval in intervals:
        start, end = interval

        # If the interval is completely outside toBeRemoved, add as is.
        if end <= remove_start or start >= remove_end:
            result.append([start, end])
        else:
            # If there is a non-overlapping left part, add it.
            if start < remove_start:
                result.append([start, remove_start])
            # If there is a non-overlapping right part, add it.
            if end > remove_end:
                result.append([remove_end, end])

    return result

# Example usage:
intervals = [[0,2],[3,4],[5,7]]
toBeRemoved = [1,6]

print(removeInterval(intervals, toBeRemoved)) # Expected output: [[0,1],[6,7]]
```

#1429 MEDIUM

First Unique Number

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Design · Queue · Data Stream

Lists: Premium 98

Problem:

You have a queue of integers, you need to retrieve the first unique integer in the queue.

Implement the FirstUnique class:

- FirstUnique(int[] nums) Initializes the object with the numbers in the queue.
- int showFirstUnique() returns the value of the first unique integer of the queue, and returns -1 if there is no such integer.
- void add(int value) insert value to the queue.

Example 1:

```
Input:
["FirstUnique","showFirstUnique","add","showFirstUnique","add","showFirstUnique","add","showF
irstUnique"]
[[[2,3,5]],[],[5],[],[2],[],[3],[]]
Output:
[null,2,null,2,null,3,null,-1]
Explanation:
FirstUnique firstUnique = new FirstUnique([2,3,5]);
```



```
firstUnique.showFirstUnique(); // return 2
firstUnique.add(5); // the queue is now [2,3,5,5]
firstUnique.showFirstUnique(); // return 2
firstUnique.add(2);            // the queue is now [2,3,5,5,2]
firstUnique.showFirstUnique(); // return 3
firstUnique.add(3);            // the queue is now [2,3,5,5,2,3]
firstUnique.showFirstUnique(); // return -1
```

Example 2:

```
Input:
["FirstUnique","showFirstUnique","add","add","add","add","add","showFirstUnique"]
[[[7,7,7,7,7,7]],[],[7],[3],[3],[7],[17],[]]
Output:
[null,-1,null,null,null,null,null,17]
Explanation:
FirstUnique firstUnique = new FirstUnique([7,7,7,7,7,7]);
firstUnique.showFirstUnique(); // return -1
firstUnique.add(7); // the queue is now [7,7,7,7,7,7,7]
firstUnique.add(3);            // the queue is now [7,7,7,7,7,7,7,3]
firstUnique.add(3);            // the queue is now [7,7,7,7,7,7,7,3,3]
firstUnique.add(7);            // the queue is now [7,7,7,7,7,7,7,3,3,7]
firstUnique.add(17);           // the queue is now [7,7,7,7,7,7,7,3,3,7,17]
firstUnique.showFirstUnique(); // return 17
```

Example 3:

```
Input:
["FirstUnique","showFirstUnique","add","showFirstUnique"]
[[[809]],[],[809],[]]
Output:
[null,809,null,-1]
Explanation:
FirstUnique firstUnique = new FirstUnique([809]);
firstUnique.showFirstUnique(); // return 809
firstUnique.add(809); // the queue is now [809,809]
firstUnique.showFirstUnique(); // return -1
```

Constraints:

- 1 <= nums.length <= 10^5
- 1 <= nums[i] <= 10^8
- 1 <= value <= 10^8
- At most 50000 calls will be made to showFirstUnique and add.

Python

Python


```python
import collections

class FirstUnique:
    # Initialize the object with the numbers in the queue.
    def __init__(self, nums):
        # Using deque for efficient popleft operations.
        self.queue = collections.deque()
        # Dictionary to store frequency count.
        self.count = collections.defaultdict(int)
        # Process the initial list of numbers.
        for num in nums:
            self.add(num)

    # Returns the value of the first unique integer of the queue.
    def showFirstUnique(self):
        # Remove numbers from the front if they are no longer unique.
        while self.queue and self.count[self.queue[0]] > 1:
            self.queue.popleft()
        # Return the first unique number if available.
        return self.queue[0] if self.queue else -1

    # Inserts value to the queue.
    def add(self, value):
        self.count[value] += 1
        # If it is the first occurrence, append it to the queue.

        if self.count[value] == 1:
            self.queue.append(value)
```

Python

```python
class FirstUnique:
  def __init__(self, nums: list[int]):
    self.seen = set()
    self.unique = {}
    for num in nums:
      self.add(num)

  def showFirstUnique(self) -> int:
    return next(iter(self.unique), -1)

  def add(self, value: int) -> None:
    if value not in self.seen:
      self.seen.add(value)
      self.unique[value] = 1
    elif value in self.unique:
      # We have added this value before, and this is the second time we're
      # adding it. So, erase the value from `unique`.
      self.unique.pop(value)
```

Python

```python
class FirstUnique:
    def __init__(self, nums: List[int]):
        self.cnt = Counter(nums)
        self.unique = OrderedDict({v: 1 for v in nums if self.cnt[v] == 1})

    def showFirstUnique(self) -> int:
        return -1 if not self.unique else next(v for v in self.unique.keys())

    def add(self, value: int) -> None:
        self.cnt[value] += 1
        if self.cnt[value] == 1:
            self.unique[value] = 1
        elif value in self.unique:
            self.unique.pop(value)


# Your FirstUnique object will be instantiated and called as such:
# obj = FirstUnique(nums)
# param_1 = obj.showFirstUnique()
# obj.add(value)
```

Python

```python
class FirstUnique:
    def __init__(self, nums: List[int]):
        self.cnt = Counter(nums)
        self.unique = OrderedDict({v: 1 for v in nums if self.cnt[v] == 1})

    def showFirstUnique(self) -> int:
        return -1 if not self.unique else next(v for v in self.unique.keys())

    def add(self, value: int) -> None:
        self.cnt[value] += 1
        if self.cnt[value] == 1:
            self.unique[value] = 1
        elif value in self.unique:
            self.unique.pop(value)


# Your FirstUnique object will be instantiated and called as such:
# obj = FirstUnique(nums)
# param_1 = obj.showFirstUnique()
# obj.add(value)
```

[*] Arrays & Hashing — Pattern Solution (matched from community answers)

Python


```python
import collections

class FirstUnique:
    # Initialize the object with the numbers in the queue.
    def __init__(self, nums):
        # Using deque for efficient popleft operations.
        self.queue = collections.deque()
        # Dictionary to store frequency count.
        self.count = collections.defaultdict(int)
        # Process the initial list of numbers.
        for num in nums:
            self.add(num)

    # Returns the value of the first unique integer of the queue.
    def showFirstUnique(self):
        # Remove numbers from the front if they are no longer unique.
        while self.queue and self.count[self.queue[0]] > 1:
            self.queue.popleft()
        # Return the first unique number if available.
        return self.queue[0] if self.queue else -1

    # Inserts value to the queue.
    def add(self, value):
        self.count[value] += 1
        # If it is the first occurrence, append it to the queue.

        if self.count[value] == 1:
            self.queue.append(value)
```

#3159 MEDIUM

Find Occurrences of an Element in an Array

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table

Lists: CodeSignal

Problem:

You are given an integer array nums, an integer array queries, and an integer x.

For each queries[i], you need to find the index of the queries[i]th occurrence of x in the nums array. If there are fewer than queries[i] occurrences of x, the answer should be −1 for that query.

Return an integer array answer containing the answers to all queries.

Example 1:

Input: nums = [1,3,1,7], queries = [1,3,2,4], x = 1

Output: [0,−1,2,−1]

Explanation:

- For the 1st query, the first occurrence of 1 is at index 0.
- For the 2nd query, there are only two occurrences of 1 in nums, so the answer is −1.
- For the 3rd query, the second occurrence of 1 is at index 2.
- For the 4th query, there are only two occurrences of 1 in nums, so the answer is −1.

Example 2:

Input: nums = [1,2,3], queries = [10], x = 5

Output: [−1]

Explanation:

- For the 1st query, 5 doesn't exist in nums, so the answer is −1.

Constraints:

- 1 <= nums.length, queries.length <= 105
- 1 <= queries[i] <= 105
- 1 <= nums[i], x <= 104

Python

Python

```python
# Function to find occurrences of x at specified query indices
def find_occurrences(nums, queries, x):
    # Preprocess the indices where x occurs in nums
    positions = []
    for index, num in enumerate(nums):
        if num == x:
            positions.append(index)

    # Process each query
    results = []
    for occurrence in queries:
        # Check if the requested occurrence exists
        if occurrence <= len(positions):
            results.append(positions[occurrence - 1]) # Convert to zero-based index
        else:
            results.append(-1)
    return results

# Example usage:
nums = [1, 3, 1, 7]
queries = [1, 3, 2, 4]
x = 1
print(find_occurrences(nums, queries, x)) # Output: [0, -1, 2, -1]
```

Python

```python
class Solution:
    def occurrencesOfElement(
        self,
        nums: list[int],
        queries: list[int],
        x: int,
    ) -> list[int]:
        indices = [i for i, num in enumerate(nums) if num == x]
        return [indices[query - 1] if query <= len(indices) else -1
                for query in queries]
```

Python


```python
class Solution:
    def occurrencesOfElement(
        self, nums: List[int], queries: List[int], x: int
    ) -> List[int]:
        ids = [i for i, v in enumerate(nums) if v == x]
        return [ids[i - 1] if i - 1 < len(ids) else -1 for i in queries]
```

Python

```python
class Solution:
    def occurrencesOfElement(
        self, nums: List[int], queries: List[int], x: int
    ) -> List[int]:
        ids = [i for i, v in enumerate(nums) if v == x]
        return [ids[i - 1] if i - 1 < len(ids) else -1 for i in queries]
```

[*] Arrays & Hashing — Pattern Solution (stored optimal)

Python

```python
class Solution:
    def occurrencesOfElement(self, nums: List[int], queries: List[int], x: int) -> List[int]:
        indices = [i for i, v in enumerate(nums) if v == x]
        return [indices[q - 1] if q <= len(indices) else -1 for q in queries]
```

#41 HARD

First Missing Positive

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table

Lists: Grind 169

Problem:

Given an unsorted integer array nums. Return the smallest positive integer that is not present in nums.

You must implement an algorithm that runs in O(n) time and uses O(1) auxiliary space.

Example 1:

```
Input: nums = [1,2,0]
Output: 3
Explanation: The numbers in the range [1,2] are all in the array.
```

Example 2:

```
Input: nums = [3,4,-1,1]
Output: 2
Explanation: 1 is in the array but 2 is missing.
```

Example 3:

```
Input: nums = [7,8,9,11,12]
Output: 1
Explanation: The smallest positive integer 1 is missing.
```

Constraints:

- 1 <= nums.length <= 105

- −231 <= nums[i] <= 231 − 1

>> Brute Force [Arrays & Hashing] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def firstMissingPositive(self, nums):
        # Brute Force O(n^2) - check 1,2,3,... until one is missing
        num_set = set(nums)
        i = 1
        while i in num_set:
            i += 1
        return i
```

Python

Python

```python
# Python implementation of First Missing Positive

def firstMissingPositive(nums):
    n = len(nums)
    # Rearrange numbers to their correct positions
    for i in range(n):
        while 1 <= nums[i] <= n and nums[nums[i] - 1] != nums[i]:
            correct_index = nums[i] - 1
            nums[i], nums[correct_index] = nums[correct_index], nums[i]

    # Identify the first missing positive integer
    for i in range(n):
        if nums[i] != i + 1:
            return i + 1
    return n + 1

# Example usage:
print(firstMissingPositive([3, 4, -1, 1])) # Output should be 2
```

Python

```python
class Solution:
    def firstMissingPositive(self, nums: list[int]) -> int:
        n = len(nums)

        # Correct slot:
        # nums[i] = i + 1
        # nums[i] - 1 = i
        # nums[nums[i] - 1] = nums[i]
        for i in range(n):
          while nums[i] > 0 and nums[i] <= n and nums[nums[i] - 1] != nums[i]:
            nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]

        for i, num in enumerate(nums):
          if num != i + 1:
            return i + 1

        return n + 1
```

Python


```python
class Solution:
    def firstMissingPositive(self, nums: List[int]) -> int:
        n = len(nums)
        for i in range(n):
            while 1 <= nums[i] <= n and nums[i] != nums[nums[i] - 1]:
                j = nums[i] - 1
                nums[i], nums[j] = nums[j], nums[i]
        for i in range(n):
            if nums[i] != i + 1:
                return i + 1
        return n + 1
```

Python

```python
class Solution:
    def firstMissingPositive(self, nums: List[int]) -> int:
        def swap(i, j):
            nums[i], nums[j] = nums[j], nums[i]

        n = len(nums)
        for i in range(n):
            while 1 <= nums[i] <= n and nums[i] != nums[nums[i] - 1]:
                swap(i, nums[i] - 1)
        for i in range(n):
            if i + 1 != nums[i]:
                return i + 1
        return n + 1
```

[*] Arrays & Hashing — Pattern Solution (stored optimal)

Python

```python
# Python implementation of First Missing Positive

def firstMissingPositive(nums):
    n = len(nums)
    # Rearrange numbers to their correct positions
    for i in range(n):
        while 1 <= nums[i] <= n and nums[nums[i] - 1] != nums[i]:
            correct_index = nums[i] - 1
            nums[i], nums[correct_index] = nums[correct_index], nums[i]

    # Identify the first missing positive integer
    for i in range(n):
        if nums[i] != i + 1:
            return i + 1
    return n + 1

# Example usage:
print(firstMissingPositive([3, 4, -1, 1])) # Output should be 2
```

Quick Review — Arrays & Hashing 18 questions · insights · complexity · solution

#1 Two Sum [Easy]

Key Insights

- Use a hash table to store the elements and their indices for fast lookup.
- For each element, calculate its complement: target minus the current element.
- Check if the complement exists in the hash table.
- If found, return the indices of the current element and its complement.
- This approach only requires a single pass through the array.

Space & Time

Time Complexity: O(n) - We traverse the list only once.

Space Complexity: O(n) - In the worst-case, we store all elements in the hash table.

Solution

We use a hash table (dictionary) to map each array element to its index as we iterate. For each number, we compute the complement (target - number) and check if it already exists in our hash table. If the complement is found, we return the indices. This one-pass solution ensures linear time complexity and avoids the need for nested loops.

#163 Missing Ranges [Easy]

Key Insights

- Use a pointer (or iterate index) to traverse the nums array while keeping track of the previous number considered.
- Check for missing numbers before the first element, between consecutive elements, and after the last element.
- When a gap is found, convert it to a range string that is either a single number (if the gap is exactly one number) or an interval.
- Handle edge cases when the nums array is empty.

Space & Time

Time Complexity: O(n), where n is the length of the nums array, since we iterate through the array once.

Space Complexity: O(1) additional space (ignoring the output list), as we only use a few extra variables.

Solution

The solution leverages a single pass through the input array and calculates the missing ranges by comparing the expected number with the current number in nums. First, initialize a variable "prev" to one less than the lower bound so that missing numbers from the very start of the range can be handled uniformly. For each number in the array, if there is a gap between prev and the current number (i.e., current number is at least 2 more than prev), then add the missing range (prev+1 to current number-1) to the output list. Update "prev" to the current number and finally handle any gap between the last number and the upper bound. The primary data structures used are the list for the output; the approach is iterative.

#346 Moving Average from Data Stream [Easy]

Key Insights

- Use a queue (or circular buffer) to keep track of the current window elements.
- Maintain a running sum to quickly update the window's total when a new element comes in.
- When the window is full, subtract the value of the element that falls out as a new one is added.
- Each call to next() is executed in constant time, O(1).

Space & Time

Time Complexity: O(1) per call to next()

Space Complexity: O(window size)

Solution

The solution leverages a queue data structure to store the elements of the sliding window. A running sum variable is maintained for efficient calculation of the moving average. When a new integer is added:

- If the size of the queue is equal to the maximum window size, remove the oldest element and subtract its value from the running sum. - Add the new element to both the queue and the running sum. - Compute and return the moving average as the running sum divided by the number of elements in the queue. This approach ensures that each update is done in constant time without having to sum all elements in the window repeatedly.

#760 Find Anagram Mappings [Easy]

Key Insights

- Since nums2 is an anagram of nums1, every element in nums1 has a corresponding position in nums2.
- A dictionary (or hash table) can be used to map each number in nums2 to its indices.
- When iterating through nums1, we can assign the next available index from the dictionary for that number.

Space & Time

Time Complexity: O(n) where n is the number of elements in the arrays (single pass to build the dictionary and another pass to form the mapping).

Space Complexity: O(n) for storing the dictionary that holds indices of elements in nums2.

Solution

The approach consists of two primary steps:

- Build a dictionary to record all indices where each number appears in nums2. This dictionary maps a number to a list of indices. - Iterate over nums1 and for each element, remove (or pop) one index from the corresponding list in the dictionary, and add that index to the final mapping array. This guarantees that each number from nums1 is correctly mapped to one of its positions in nums2 while efficiently handling duplicates.

#1426 Counting Elements [Easy]

Key Insights

- Use a hash table or set to store unique elements from arr for constant time lookups.
- Iterate through each element in arr and check if x + 1 exists in the set.
- Count each valid occurrence, including duplicates.
- This approach efficiently handles the problem without needing nested loops.

Space & Time

Time Complexity: O(n), where n is the number of elements in arr (set lookups are O(1) on average).

Space Complexity: O(n), due to the use of an auxiliary set to store unique elements.

Solution

The solution leverages a set to achieve constant time membership checks. First, we convert arr to a set of unique numbers. Then, iterate through arr, and for each element, check if (element + 1) exists in the set. If it does, increment the count. This approach correctly handles duplicates by counting each occurrence during iteration.

#1534 Count Good Triplets [Easy]

Key Insights

- Brute force approach using three nested loops is acceptable since the maximum array length is 100.
- Check all possible triplets and count only those meeting the specified conditions.
- Absolute difference comparisons are the key condition validations.

Space & Time

Time Complexity: O(n^3) where n is the length of the array.

Space Complexity: O(1) extra space (ignoring input space).

Solution

We use a brute force algorithm with three nested loops, iterating over all combinations of triplets (i, j, k) ensuring i < j < k. For each triplet, we calculate the absolute differences:

- |arr[i] - arr[j]| - |arr[j] - arr[k]| - |arr[i] - arr[k]| The triplet is counted as good if all absolute differences satisfy the conditions (<= a, <= b, and <= c respectively). This simple approach leverages the small input size, eliminating the need for more complex data structures or optimizations.

#57 Insert Interval [Medium]

Key Insights

- The intervals array is sorted by start times.
- Three main steps are used:
- Add intervals that come completely before the new interval.
- Merge overlapping intervals with the new interval.
- Append intervals that start after the new (merged) interval.
- Iterating through the list once allows an O(n) solution.

Space & Time

Time Complexity: O(n) since we scan the list once.

Space Complexity: O(n) to store the resulting list of intervals.

Solution

Traverse the intervals list and perform the following:

- Append all intervals that end before the new interval starts. - For intervals that overlap with newInterval, update newInterval to merge them (using min for start and max for end). - Append the new merged interval. - Append all remaining intervals. This ensures that all intervals remain non-overlapping and sorted with a single pass.

#238 Product of Array Except Self [Medium]

Key Insights

- The product for each index can be computed by multiplying the product of all numbers to the left and the product of all numbers to the right of that index.
- Division is not allowed, so we must compute the cumulative products explicitly.
- Create two passes: one forward pass for prefix products and one backward pass for suffix products.
- Use the output array to store the cumulative product values, achieving O(1) extra space by not using additional arrays.

Space & Time

Time Complexity: O(n) because we iterate through the array twice.

Space Complexity: O(1) extra space (excluding the output array).

Solution

The approach uses two passes over the input array:

- Initialize the answer array and fill it with prefix products. For each index i, answer[i] contains the product of all elements before i. - Traverse the array from the end using a running suffix product. Update the answer array by multiplying the current suffix product with the prefix product already stored. This method leverages an output array to store intermediate results and uses a temporary variable for the suffix product, ensuring constant extra space.

#281 Zigzag Iterator [Medium]

Key Insights

- Use a FIFO (first-in-first-out) structure to maintain the order of vectors which still have remaining elements.
- For each call to next(), select the next element from the current vector and then move that vector to the end of the queue if it still contains elements.
- The approach ensures proper zigzag (or cyclic) ordering among the vectors.
- When extending to k vectors, the same approach works: initialize the queue with each non-empty vector.

Space & Time

Time Complexity: O(1) per next() call on average, with an overall O(n) for n total elements.

Space Complexity: O(k) for the queue structure (where k is the number of vectors), plus the storage for the input vectors.

Solution

We can solve the Zigzag Iterator problem by using a queue (or deque) to hold pairs of the vector's current index and the vector itself. For each call to next(), we remove the front element from the queue, return the current item from its associated vector, and if the vector has more elements, reinsert the pair (with an updated index) into the back of the queue. This cyclic process naturally interleaves the elements from the provided vectors.

#307 Range Sum Query - Mutable [Medium]

Key Insights

- To efficiently support both point updates and range sum queries, a specialized data structure like a Binary Indexed Tree (Fenwick Tree) or a Segment Tree is ideal.
- Both data structures provide update and sum query operations in O(log n) time.
- The challenge is to maintain a mutable array while enabling fast sum computation over any specified range.

Space & Time

Time Complexity: O(log n) for both update and sumRange operations

Space Complexity: O(n) to store the additional tree structure

Solution

We can use a Binary Indexed Tree (Fenwick Tree) to solve this problem. The Fenwick Tree allows for efficient prefix sum calculations and point updates, both in O(log n) time. The approach involves:

- Building the Fenwick Tree using the initial array. - On an update, computing the difference between the new value and the current value, then propagating the change appropriately in the tree. - For the sum range query, computing the prefix sum up to right and subtracting the prefix sum up to left-1.

#454 4Sum II [Medium]

Key Insights

- Use a hash table to store the frequency of all possible sums from nums1 and nums2.
- For each pair from nums3 and nums4, compute the complement (negative sum) and look it up in the hash table.
- This reduces the time complexity from O(n^4) to O(n^2).

Space & Time

Time Complexity: O(n^2) where n is the length of each array.

Space Complexity: O(n^2) due to the storage of pair sums in the hash table.

Solution

The solution employs a hash table to precompute and store the frequency of sums of all pairs from the first two arrays. Then, by iterating over all pairs from the third and fourth arrays, we calculate the required complement that would result in a total sum of zero. This complement is checked in the hash table, and the frequency (number of valid pairs) is added to the result. This method efficiently finds the number of valid quadruplets without resorting to a brute-force O(n^4) approach.

#525 Contiguous Array [Medium]

Key Insights

- Convert 0s to -1 so that a balanced subarray (equal number of 0s and 1s) sums to zero.
- Use a prefix sum to track the cumulative sum as you iterate through the array.
- Use a hash table (or dictionary) to store the first occurrence of each prefix sum.
- When the same prefix sum is encountered again, it indicates that the subarray between these indices is balanced.

Space & Time

Time Complexity: O(n) - We iterate through the array once.

Space Complexity: O(n) - In the worst case, we store an entry for each prefix sum.

Solution

We use a technique where we convert each 0 in the array to -1. Then, we compute a running prefix sum as we iterate through the array. A prefix sum of 0 at any point indicates that the subarray from the beginning to the current index is balanced. For each prefix sum, if we have seen it before, the subarray between the previous index (where the prefix sum was first seen) and the current index sums to 0. We update the maximum length accordingly. We use a hash table to store the first index where each prefix sum occurs.

#560 Subarray Sum Equals K [Medium]

Key Insights

- Utilize the prefix sum technique to simplify sum calculation for subarrays.
- Use a hash table (or dictionary/map) to store the frequency of prefix sums encountered.
- For each element, check if (current prefix sum - k) exists in the hash table; if it does, it indicates a valid subarray.
- Initialize the hash table with {0: 1} to manage cases where a subarray's sum is exactly k.

Space & Time

Time Complexity: O(n)

Space Complexity: O(n)

Solution

We solve the problem by maintaining a running sum (prefix sum) and using a hash table to record the number of times each sum has occurred. As we iterate through the array, we update the running sum. For each new sum, we check if (current sum - k) exists in our hash table. If it does, the value associated with (current sum - k) gives the number of subarrays ending at the current index that sum to k. This method efficiently computes the result in a single pass through the array.

#1010 Pairs of Songs With Total Durations Divisible by 60 [Medium]

Key Insights

- Each song duration's effect is determined by its remainder when divided by 60.
- For any given module value r (where 0 ≤ r < 60), the complementary module needed to complete 60 is (60 - r) % 60.
- Special handling is needed when r is 0 (or 30, since 30 + 30 = 60) to avoid double-counting.
- A frequency counter for remainders can be efficiently used to count valid pairs in one pass through the list.

Space & Time

Time Complexity: O(n), where n is the number of songs.

Space Complexity: O(60) → O(1), since we only need to keep track of 60 possible remainders.

Solution

We iterate over the list of song durations and calculate the remainder of each duration modulo 60. A frequency array (or hash table) is used to record how many times each remainder has been seen so far. For each song, we look up the frequency of its complementary remainder (i.e., (60 - current_remainder) % 60) already encountered. This frequency indicates how many valid pairs can be formed with the current song. After counting pairs with the current song, we update the frequency of its remainder for future pairings.

#1272 Remove Interval [Medium]

Key Insights

- Iterate over each interval and check if it overlaps with the toBeRemoved interval.
- If the current interval is completely outside of toBeRemoved, add it to the result as is.
- For overlapping intervals, compute the non-overlapping portions:
- If the left side of the interval lies before toBeRemoved, include [start, min(end, toBeRemoved.start)].

- If the right side of the interval lies after toBeRemoved, include [max(start, toBeRemoved.end), end].

Space & Time

Time Complexity: O(n), where n is the number of intervals.

Space Complexity: O(n), for storing the resulting intervals.

Solution

The approach involves iterating through the list of sorted disjoint intervals and checking for overlap with the toBeRemoved interval. For each interval:

- If it is completely before or after the removal interval, add it directly to the result. - If it overlaps with toBeRemoved, determine the remaining portion(s): The left side of the interval that does not intersect (if any). The right side of the interval that does not intersect (if any). Use simple conditional checks and list manipulation to form the final set of intervals.

#1429 First Unique Number [Medium]

Key Insights

- Use a queue to maintain the order of potential unique numbers.
- Use a hash table (or map) to count the occurrences of each number.
- When showing the first unique number, remove numbers from the front of the queue if their frequency is more than one.
- All operations (add and showFirstUnique) can be done in O(1) amortized time.

Space & Time

Time Complexity: O(1) amortized for each add and showFirstUnique call.

Space Complexity: O(n), where n is the number of distinct integers encountered.

Solution

We design the FirstUnique class using two main data structures: a queue and a hash map (or dictionary). The queue preserves the order of insertion for unique candidates, while the hash map holds the frequency count for each integer.

When a new number is added:

- Increment its count in the hash map. - If the count is one, append it to the queue.

To retrieve the first unique number:

- While the queue is not empty and the count for the element at the front is greater than one, remove it from the queue. - Return the front element of the queue if it exists, otherwise return -1.

This approach ensures that we are always up-to-date with the current first unique number.

#3159 Find Occurrences of an Element in an Array [Medium]

Key Insights

- Scan the array once and record every index where `nums[i] == x`.
- Store those indices in order inside a positions array.
- For each query `k`, check whether the `k`-th occurrence exists.
- If it exists, return `positions[k - 1]`.
- Otherwise, return `-1`.
- This lets every query be answered in constant time after the preprocessing pass.

Space & Time

Time Complexity: O(n + q), where n is the length of `nums` and q is the number of queries.

Space Complexity: O(m), where m is the number of occurrences of `x`.

Solution

We solve the problem by first precomputing all occurrences of `x`. As we iterate through `nums`, every time we see `x`, we append its index to an array called `positions`. Then, for each query value `k`, we check whether `k` is

within the bounds of `positions`. If it is, the answer is `positions[k - 1]`; otherwise, the required occurrence does not exist, so the answer is `-1`. This avoids rescanning the array for every query.

#41 First Missing Positive [Hard]

Key Insights

- The missing positive number must be in the range [1, n+1] where n is the length of the array.
- Use the array indices to place numbers in their correct positions (i.e., number i should be at index i−1) with in-place swapping.
- After rearrangement, scan the array: the first index where the number is not equal to index+1 indicates the missing positive integer.

Space & Time

Time Complexity: O(n) as each number is swapped at most once.

Space Complexity: O(1) since the rearrangement occurs in-place.

Solution

The approach involves reordering the array so that each positive integer in the range [1, n] is placed at the index corresponding to its value minus one. This means for any valid number i, nums[i−1] should be i. We iterate through the array and perform in-place swaps to achieve this order. After this process, another pass through the array reveals the first position where the number does not match the expected value (i+1), which is the smallest missing positive. If every position is correct, then the missing value is n+1.

#13 Two Pointers — 19 questions · #13 of 21

#88 EASY

Merge Sorted Array

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers · Sorting

Lists: Denny Zhang

Problem:

You are given two integer arrays nums1 and nums2, sorted in non-decreasing order, and two integers m and n, representing the number of elements in nums1 and nums2 respectively.

Merge nums1 and nums2 into a single array sorted in non-decreasing order.

The final sorted array should not be returned by the function, but instead be stored inside the array nums1. To accommodate this, nums1 has a length of m + n, where the first m elements denote the elements that should be merged, and the last n elements are set to 0 and should be ignored. nums2 has a length of n.

Example 1:

```
Input: nums1 = [1,2,3,0,0,0], m = 3, nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]
Explanation: The arrays we are merging are [1,2,3] and [2,5,6].
The result of the merge is [1,2,2,3,5,6] with the underlined elements coming from nums1.
```

Example 2:

```
Input: nums1 = [1], m = 1, nums2 = [], n = 0
Output: [1]
Explanation: The arrays we are merging are [1] and [].
The result of the merge is [1].
```

Example 3:

```
Input: nums1 = [0], m = 0, nums2 = [1], n = 1
Output: [1]
Explanation: The arrays we are merging are [] and [1].
The result of the merge is [1].
Note that because m = 0, there are no elements in nums1. The 0 is only there to ensure the
merge result can fit in nums1.
```

Constraints:

- nums1.length == m + n
- nums2.length == n
- 0 <= m, n <= 200
- 1 <= m + n <= 200
- −109 <= nums1[i], nums2[j] <= 109

Follow up: Can you come up with an algorithm that runs in O(m + n) time?

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def merge(self, nums1, m, nums2, n):
        # Brute Force O(n^2) - nested loops instead of two pointers
        n = len(nums1)
        for i in range(n):
            for j in range(i + 1, n):
                # check nums1[i] and nums1[j]
                pass
```

Python

Python

```python
def merge(nums1, m, nums2, n):
    # Initialize pointers for nums1, nums2 and the tail index
    p1 = m - 1
    p2 = n - 1
    tail = m + n - 1

    # Merge arrays from the end by choosing the larger element
    while p1 >= 0 and p2 >= 0:
        if nums1[p1] > nums2[p2]:
            nums1[tail] = nums1[p1]
            p1 -= 1 # Move pointer in nums1
        else:
            nums1[tail] = nums2[p2]
            p2 -= 1 # Move pointer in nums2
        tail -= 1 # Move tail pointer

    # Copy any remaining elements from nums2 into nums1
    while p2 >= 0:
        nums1[tail] = nums2[p2]
        p2 -= 1
        tail -= 1
```

Python

```python
class Solution:
    def merge(self, nums1: list[int], m: int, nums2: list[int], n: int) -> None:
        i = m - 1 # nums1's index (the actual nums)
        j = n - 1 # nums2's index
        k = m + n - 1 # nums1's index (the next filled position)

        while j >= 0:
            if i >= 0 and nums1[i] > nums2[j]:
                nums1[k] = nums1[i]
                k -= 1
                i -= 1
            else:
                nums1[k] = nums2[j]
                k -= 1
                j -= 1
```

Python


```python
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        k = m + n - 1
        i, j = m - 1, n - 1
        while j >= 0:
            if i >= 0 and nums1[i] > nums2[j]:
                nums1[k] = nums1[i]
                i -= 1
            else:
                nums1[k] = nums2[j]
                j -= 1
            k -= 1
```

Python

```python
class Solution:
    def merge(self, nums1: List[int], m: int, nums2: List[int], n: int) -> None:
        """
        Do not return anything, modify nums1 in-place instead.
        """
        i, j, k = m - 1, n - 1, m + n - 1
        # what if j==-1 already but k is not 0 yet? e.g. m=[1,2,0], n=[3]
        # => then no more ops needed, rest of m[] already sorted, so this while is good enough
        while j >= 0:
            # i could be -1, eg. [7,8,9, ] and [1]
            if i >= 0 and nums1[i] > nums2[j]:
                nums1[k] = nums1[i]
                i -= 1
            else:
                nums1[k] = nums2[j]
                j -= 1
            k -= 1
```

[*] Two Pointers — Pattern Solution (stored optimal)

Python

```python
def merge(nums1, m, nums2, n):
    # Initialize pointers for nums1, nums2 and the tail index
    p1 = m - 1
    p2 = n - 1
    tail = m + n - 1

    # Merge arrays from the end by choosing the larger element
    while p1 >= 0 and p2 >= 0:
        if nums1[p1] > nums2[p2]:
            nums1[tail] = nums1[p1]
            p1 -= 1 # Move pointer in nums1
        else:
            nums1[tail] = nums2[p2]
            p2 -= 1 # Move pointer in nums2
        tail -= 1 # Move tail pointer

    # Copy any remaining elements from nums2 into nums1
    while p2 >= 0:
        nums1[tail] = nums2[p2]
        p2 -= 1
        tail -= 1
```


#125 EASY

Valid Palindrome

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Two Pointers · String

Lists: Grind 169

Problem:

A phrase is a palindrome if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string s, return true if it is a palindrome, or false otherwise.

Example 1:

```
Input: s = "A man, a plan, a canal: Panama"
Output: true
Explanation: "amanaplanacanalpanama" is a palindrome.
```

Example 2:

```
Input: s = "race a car"
Output: false
Explanation: "raceacar" is not a palindrome.
```

Example 3:

```
Input: s = " "
Output: true
Explanations: s is an empty string "" after removing non-alphanumeric characters.
Since an empty string reads the same forward and backward, it is a palindrome.
```

Constraints:

- 1 <= s.length <= 2 * 105
- s consists only of printable ASCII characters.

Python

Python

```python
class Solution:
    def isPalindrome(self, s: str) -> bool:
        # Initialize two pointers at the beginning and end of the string
        left, right = 0, len(s) - 1
        while left < right:
            # Skip non-alphanumeric characters from the left
            while left < right and not s[left].isalnum():
                left += 1
            # Skip non-alphanumeric characters from the right
            while left < right and not s[right].isalnum():
                right -= 1
            # Compare the characters in lowercase
            if s[left].lower() != s[right].lower():
                return False
            left += 1
            right -= 1
        return True
```


Python

```python
class Solution:
    def isPalindrome(self, s: str) -> bool:
        l = 0
        r = len(s) - 1

        while l < r:
          while l < r and not s[l].isalnum():
            l += 1
          while l < r and not s[r].isalnum():
            r -= 1
          if s[l].lower() != s[r].lower():
            return False
          l += 1
          r -= 1

        return True
```

Python

```python
class Solution:
    def isPalindrome(self, s: str) -> bool:
        i, j = 0, len(s) - 1
        while i < j:
            if not s[i].isalnum():
                i += 1
            elif not s[j].isalnum():
                j -= 1
            elif s[i].lower() != s[j].lower():
                return False
            else:
                i, j = i + 1, j - 1
        return True
```

Python

```python
class Solution:
    def isPalindrome(self, s: str) -> bool:
        i, j = 0, len(s) - 1
        while i < j:
            if not s[i].isalnum():
                i += 1
            elif not s[j].isalnum():
                j -= 1
            elif s[i].lower() != s[j].lower():
                return False
            else:
                i, j = i + 1, j - 1
        return True
```

[*] Two Pointers — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def isPalindrome(self, s: str) -> bool:
        # Initialize two pointers at the beginning and end of the string
        left, right = 0, len(s) - 1
        while left < right:
            # Skip non-alphanumeric characters from the left
            while left < right and not s[left].isalnum():
                left += 1
            # Skip non-alphanumeric characters from the right
            while left < right and not s[right].isalnum():
                right -= 1
            # Compare the characters in lowercase
            if s[left].lower() != s[right].lower():
                return False
            left += 1
            right -= 1
        return True
```

#283 EASY

Move Zeroes

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers

Lists: Grind 169

Problem:

Given an integer array nums, move all 0's to the end of it while maintaining the relative order of the non-zero elements.

Note that you must do this in-place without making a copy of the array.

Example 1:

```
Input: nums = [0,1,0,3,12]
Output: [1,3,12,0,0]
```

Example 2:

```
Input: nums = [0]
Output: [0]
```

Constraints:

- 1 <= nums.length <= 104
- -231 <= nums[i] <= 231 - 1

Follow up: Could you minimize the total number of operations done?

>> Brute Force [Two Pointers] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def moveZeroes(self, nums):
        # Brute Force O(n^2) - nested loops instead of two pointers
        n = len(nums)
        for i in range(n):
            for j in range(i + 1, n):
                # check nums[i] and nums[j]
                pass
```

Python

Python

```python
# Moves all zeros in the list 'nums' to the end in-place.
def moveZeroes(nums):
    # 'j' is the pointer where the next non-zero element should be placed.
    j = 0
    # Loop through all the elements with index 'i'
    for i in range(len(nums)):
        # Check if the current element is non-zero.
        if nums[i] != 0:
            # Swap only if i is not equal to j to avoid unnecessary operations.
            nums[j], nums[i] = nums[i], nums[j]
            # Increment j to move to the next insertion index.
            j += 1

# Example usage:
nums = [0, 1, 0, 3, 12]
moveZeroes(nums)
print(nums) # Expected output: [1, 3, 12, 0, 0]
```

Python

```python
class Solution:
    def moveZeroes(self, nums: list[int]) -> None:
        j = 0
        for num in nums:
            if num != 0:
                nums[j] = num
                j += 1

        for i in range(j, len(nums)):
            nums[i] = 0
```

Python

```python
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        k = 0
        for i, x in enumerate(nums):
            if x:
                nums[k], nums[i] = nums[i], nums[k]
                k += 1
```

Python


```python
class Solution:
    def moveZeroes(self, nums: List[int]) -> None:
        j = -1
        for i, x in enumerate(nums):
            if x:
                i += 1
                nums[i], nums[j] = nums[j], nums[i]
```

[*] Two Pointers — Pattern Solution (stored optimal)

Python

```python
# Moves all zeros in the list 'nums' to the end in-place.
def moveZeroes(nums):
    # 'j' is the pointer where the next non-zero element should be placed.
    j = 0
    # Loop through all the elements with index 'i'
    for i in range(len(nums)):
        # Check if the current element is non-zero.
        if nums[i] != 0:
            # Swap only if i is not equal to j to avoid unnecessary operations.
            nums[j], nums[i] = nums[i], nums[j]
            # Increment j to move to the next insertion index.
            j += 1

# Example usage:
nums = [0, 1, 0, 3, 12]
moveZeroes(nums)
print(nums) # Expected output: [1, 3, 12, 0, 0]
```

#408 EASY

Valid Word Abbreviation

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Two Pointers · String

Lists: Premium 98

Problem:

A string can be abbreviated by replacing any number of non-adjacent, non-empty substrings with their lengths. The lengths should not have leading zeros.

For example, a string such as "substitution" could be abbreviated as (but not limited to):

- "s10n" ("s ubstitutio n")
- "sub4u4" ("sub stit u tion")
- "12" ("substitution")
- "su3i1u2on" ("su bst i t u ti on")
- "substitution" (no substrings replaced)

The following are not valid abbreviations:

- "s55n" ("s ubsti tutio n", the replaced substrings are adjacent)
- "s010n" (has leading zeros)
- "s0ubstitution" (replaces an empty substring)

Given a string word and an abbreviation abbr, return whether the string matches the given abbreviation.

A substring is a contiguous non-empty sequence of characters within a string.

Example 1:

```
Input: word = "internationalization", abbr = "i12iz4n"
Output: true
Explanation: The word "internationalization" can be abbreviated as "i12iz4n" ("i nternational
iz atio n").
```

Example 2:

```
Input: word = "apple", abbr = "a2e"
Output: false
Explanation: The word "apple" cannot be abbreviated as "a2e".
```

Constraints:

- 1 <= word.length <= 20
- word consists of only lowercase English letters.
- 1 <= abbr.length <= 10
- abbr consists of lowercase English letters and digits.
- All the integers in abbr will fit in a 32-bit integer.

Python

Python

```python
# Python solution for valid word abbreviation

def validWordAbbreviation(word: str, abbr: str) -> bool:
    word_index = 0 # pointer for the word
    abbr_index = 0 # pointer for the abbreviation

    while abbr_index < len(abbr):
        # if current abbrev char is a digit
        if abbr[abbr_index].isdigit():
            # leading zero check: if the digit is '0', it is invalid
            if abbr[abbr_index] == '0':
                return False
            # build the entire number
            num = 0
            while abbr_index < len(abbr) and abbr[abbr_index].isdigit():
                num = num * 10 + int(abbr[abbr_index])
                abbr_index += 1
            # skip num characters in word
            word_index += num
        else:
            # if letter doesn't match or word index is out of bounds, invalid
            if word_index >= len(word) or word[word_index] != abbr[abbr_index]:
                return False
            word_index += 1
            abbr_index += 1
```



```python
    # Valid if and only if the entire word was covered
    return word_index == len(word)

# Example usage:
#print(validWordAbbreviation("internationalization", "i12iz4n")) # Expected output: True
#print(validWordAbbreviation("apple", "a2e")) # Expected output: False
```

Python

```python
class Solution:
    def validWordAbbreviation(self, word: str, abbr: str) -> bool:
        i = 0 # word's index
        j = 0 # abbr's index

        while i < len(word) and j < len(abbr):
          if word[i] == abbr[j]:
            i += 1
            j += 1
            continue
          if not abbr[j].isdigit() or abbr[j] == '0':
            return False
          num = 0
          while j < len(abbr) and abbr[j].isdigit():
            num = num * 10 + int(abbr[j])
            j += 1
          i += num

        return i == len(word) and j == len(abbr)
```

Python

```python
class Solution:
    def validWordAbbreviation(self, word: str, abbr: str) -> bool:
        m, n = len(word), len(abbr)
        i = j = x = 0
        while i < m and j < n:
            if abbr[j].isdigit():
                if abbr[j] == "0" and x == 0:
                    return False
                x = x * 10 + int(abbr[j])
            else:
                i += x
                x = 0
                if i >= m or word[i] != abbr[j]:
                    return False
                i += 1
                j += 1
        return i + x == m and j == n
```

Python


```python
class Solution:
    def validWordAbbreviation(self, word: str, abbr: str) -> bool:
        m, n = len(word), len(abbr)
        i = j = x = 0
        while i < m and j < n:
            if abbr[j].isdigit():
                if abbr[j] == "0" and x == 0:
                    return False
                x = x * 10 + int(abbr[j])
            else:
                i += x
                x = 0
                if i >= m or word[i] != abbr[j]:
                    return False
                i += 1
                j += 1
        return i + x == m and j == n
```

[*] Two Pointers — Pattern Solution (stored optimal)

Python

```python
# Python solution for valid word abbreviation

def validWordAbbreviation(word: str, abbr: str) -> bool:
    word_index = 0 # pointer for the word
    abbr_index = 0 # pointer for the abbreviation

    while abbr_index < len(abbr):
        # if current abbrev char is a digit
        if abbr[abbr_index].isdigit():
            # leading zero check: if the digit is '0', it is invalid
            if abbr[abbr_index] == '0':
                return False
            # build the entire number
            num = 0
            while abbr_index < len(abbr) and abbr[abbr_index].isdigit():
                num = num * 10 + int(abbr[abbr_index])
                abbr_index += 1
            # skip num characters in word
            word_index += num
        else:
            # if letter doesn't match or word index is out of bounds, invalid
            if word_index >= len(word) or word[word_index] != abbr[abbr_index]:
                return False
            word_index += 1
            abbr_index += 1

    # Valid if and only if the entire word was covered
    return word_index == len(word)

# Example usage:
#print(validWordAbbreviation("internationalization", "i12iz4n")) # Expected output: True
#print(validWordAbbreviation("apple", "a2e")) # Expected output: False
```

#977 EASY

Squares of a Sorted Array

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers · Sorting

Lists: Grind 169

Problem:

Given an integer array nums sorted in non-decreasing order, return an array of the squares of each number sorted in non-decreasing order.

Example 1:

```
Input: nums = [-4,-1,0,3,10]
Output: [0,1,9,16,100]
Explanation: After squaring, the array becomes [16,1,0,9,100].
After sorting, it becomes [0,1,9,16,100].
```

Example 2:

```
Input: nums = [-7,-3,2,3,11]
Output: [4,9,9,49,121]
```

Constraints:

- 1 <= nums.length <= 104
- -104 <= nums[i] <= 104
- nums is sorted in non-decreasing order.

Follow up: Squaring each element and sorting the new array is very trivial, could you find an O(n) solution using a different approach?

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def sortedSquares(self, nums):
        # Brute Force O(n^2) - nested loops instead of two pointers
        n = len(nums)
        for i in range(n):
            for j in range(i + 1, n):
                # check nums[i] and nums[j]
                pass
```

Python

Python


```python
# Function to return squares of a sorted array in non-decreasing order.
def sortedSquares(nums):
    # Initialize two pointers at the start and end of the array.
    left, right = 0, len(nums) - 1
    # Create a result array of the same length as nums.
    result = [0] * len(nums)
    # Index to insert the largest square at the current position.
    position = len(nums) - 1

    # Loop until the left and right pointers cross.
    while left <= right:
        # Compare absolute values at the left and right pointers.
        if abs(nums[left]) > abs(nums[right]):
            result[position] = nums[left] * nums[left]
            left += 1 # Move left pointer rightward.
        else:
            result[position] = nums[right] * nums[right]
            right -= 1 # Move right pointer leftward.
        position -= 1 # Move to the next insertion position.
    return result

# Example usage:
print(sortedSquares([-4, -1, 0, 3, 10])) # Output: [0, 1, 9, 16, 100]
```

Python

```python
class Solution:
    def sortedSquares(self, nums: list[int]) -> list[int]:
        n = len(nums)
        l = 0
        r = n - 1
        ans = [0] * n

        while n:
          n -= 1
          if abs(nums[l]) > abs(nums[r]):
            ans[n] = nums[l] * nums[l]
            l += 1
          else:
            ans[n] = nums[r] * nums[r]
            r -= 1

        return ans
```

Python

```python
class Solution:
    def sortedSquares(self, nums: List[int]) -> List[int]:
        ans = []
        i, j = 0, len(nums) - 1
        while i <= j:
            a = nums[i] * nums[i]
            b = nums[j] * nums[j]
            if a > b:
                ans.append(a)
                i += 1
            else:
                ans.append(b)
                j -= 1
        return ans[::-1]
```


Python

```python
class Solution:
    def sortedSquares(self, nums: List[int]) -> List[int]:
        n = len(nums)
        res = [0] * n
        i, j, k = 0, n - 1, n - 1
        while i <= j:
            if nums[i] * nums[i] > nums[j] * nums[j]:
                res[k] = nums[i] * nums[i]
                i += 1
            else:
                res[k] = nums[j] * nums[j]
                j -= 1
            k -= 1
        return res
```

[*] Two Pointers — Pattern Solution (matched from community answers)

Python

```python
# Function to return squares of a sorted array in non-decreasing order.
def sortedSquares(nums):
    # Initialize two pointers at the start and end of the array.
    left, right = 0, len(nums) - 1
    # Create a result array of the same length as nums.
    result = [0] * len(nums)
    # Index to insert the largest square at the current position.
    position = len(nums) - 1

    # Loop until the left and right pointers cross.
    while left <= right:
        # Compare absolute values at the left and right pointers.
        if abs(nums[left]) > abs(nums[right]):
            result[position] = nums[left] * nums[left]
            left += 1 # Move left pointer rightward.
        else:
            result[position] = nums[right] * nums[right]
            right -= 1 # Move right pointer leftward.
        position -= 1 # Move to the next insertion position.
    return result

# Example usage:
print(sortedSquares([-4, -1, 0, 3, 10])) # Output: [0, 1, 9, 16, 100]
```

#11 MEDIUM

Container With Most Water

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers · Greedy

Lists: Grind 169

Problem:

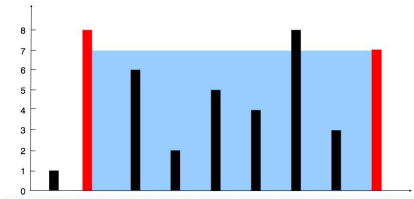
You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return the maximum amount of water a container can store.

Notice that you may not slant the container.

Example 1:




```
Input: height = [1,8,6,2,5,4,8,3,7]
Output: 49
Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this
case, the max area of water (blue section) the container can contain is 49.
```

Example 2:

```
Input: height = [1,1]
Output: 1
```

Constraints:

- n == height.length
- 2 <= n <= 105
- 0 <= height[i] <= 104

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def maxArea(self, height):
        # Brute Force O(n^2) - try every pair of lines
        res = 0
        for i in range(len(height)):
            for j in range(i + 1, len(height)):
                water = min(height[i], height[j]) * (j - i)
                res = max(res, water)
        return res
```


Python

Python

```python
# Function to calculate the maximum water container area
def maxArea(height):
    left, right = 0, len(height) - 1 # Initialize two pointers
    max_water = 0 # Variable to store the maximum area

    # Loop until pointers meet
    while left < right:
        # Calculate current area based on smaller height and distance between pointers
        current_area = min(height[left], height[right]) * (right - left)
        max_water = max(max_water, current_area) # Update maximum area if current is larger

        # Move the pointer corresponding to the smaller height
        if height[left] < height[right]:
            left += 1
        else:
            right -= 1
    return max_water

# Test the function with an example
if __name__ == "__main__":
    input_heights = [1,8,6,2,5,4,8,3,7]
    print(maxArea(input_heights)) # Expected output: 49
```

Python

```python
class Solution:
   def maxArea(self, height: list[int]) -> int:
      ans = 0
      l = 0
      r = len(height) - 1

      while l < r:
        minHeight = min(height[l], height[r])
        ans = max(ans, minHeight * (r - l))
        if height[l] < height[r]:
          l += 1
        else:
          r -= 1

      return ans
```

Python

```python
class Solution:
    def maxArea(self, height: List[int]) -> int:
        l, r = 0, len(height) - 1
        ans = 0
        while l < r:
            t = min(height[l], height[r]) * (r - l)
            ans = max(ans, t)
            if height[l] < height[r]:
                l += 1
            else:
                r -= 1
        return ans
```

Python


```python
class Solution:
    def maxArea(self, height: List[int]) -> int:
        i, j = 0, len(height) - 1
        ans = 0
        while i < j:
            t = (j - i) * min(height[i], height[j])
            ans = max(ans, t)
            if height[i] < height[j]:
                i += 1
            else:
                j -= 1
        return ans
```

[*] Two Pointers — Pattern Solution (matched from community answers)

Python

```python
# Function to calculate the maximum water container area
def maxArea(height):
    left, right = 0, len(height) - 1 # Initialize two pointers
    max_water = 0 # Variable to store the maximum area

    # Loop until pointers meet
    while left < right:
        # Calculate current area based on smaller height and distance between pointers
        current_area = min(height[left], height[right]) * (right - left)
        max_water = max(max_water, current_area) # Update maximum area if current is larger

        # Move the pointer corresponding to the smaller height
        if height[left] < height[right]:
            left += 1
        else:
            right -= 1
    return max_water

# Test the function with an example
if __name__ == "__main__":
    input_heights = [1,8,6,2,5,4,8,3,7]
    print(maxArea(input_heights)) # Expected output: 49
```

#15 MEDIUM

3Sum

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers · Sorting

Lists: Grind 169

Problem:

Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

Example 1:

```
Input: nums = [-1,0,1,2,-1,-4]
Output: [[-1,-1,2],[-1,0,1]]
```

```
Explanation:
nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.
nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.
nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.
The distinct triplets are [-1,0,1] and [-1,-1,2].
Notice that the order of the output and the order of the triplets does not matter.
```

Example 2:

```
Input: nums = [0,1,1]
Output: []
Explanation: The only possible triplet does not sum up to 0.
```

Example 3:

```
Input: nums = [0,0,0]
Output: [[0,0,0]]
Explanation: The only possible triplet sums up to 0.
```

Constraints:

- 3 <= nums.length <= 3000
- -105 <= nums[i] <= 105

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute-Force Approach
class Solution:
    def threeSum(self, nums):
        # Brute-Force O(n^3) - check all triples, deduplicate via set
        nums.sort()
        res = set()
        for i in range(len(nums)):
            for j in range(i + 1, len(nums)):
                for k in range(j + 1, len(nums)):
                    if nums[i] + nums[j] + nums[k] == 0:
                        res.add((nums[i], nums[j], nums[k]))
        return [list(t) for t in res]
```

Python

Python


```python
# Function to find all unique triplets that sum to zero.
def threeSum(nums):
    # Sort the array to facilitate finding duplicates and using two pointers.
    nums.sort()
    result = []
    n = len(nums)

    # Iterate through the array, fixing one element at a time.
    for i in range(n - 2):
        # Skip duplicate elements for the first element.
        if i > 0 and nums[i] == nums[i - 1]:
            continue

        left, right = i + 1, n - 1
        while left < right:
            current_sum = nums[i] + nums[left] + nums[right]
            # Check if the triplet adds up to zero.
            if current_sum == 0:
                result.append([nums[i], nums[left], nums[right]])
                # Skip duplicates for the second element.
                while left < right and nums[left] == nums[left + 1]:
                    left += 1
                # Skip duplicates for the third element.
                while left < right and nums[right] == nums[right - 1]:
                    right -= 1

                left += 1
                right -= 1
            elif current_sum < 0:
                left += 1
            else:
                right -= 1

    return result

# Example usage:
#print(threeSum([-1,0,1,2,-1,-4]))
```

Python


```python
class Solution:
   def threeSum(self, nums: list[int]) -> list[list[int]]:
      if len(nums) < 3:
        return []

      ans = []

      nums.sort()

      for i in range(len(nums) - 2):
        if i > 0 and nums[i] == nums[i - 1]:
          continue
        # Choose nums[i] as the first number in the triplet, then search the
        # remaining numbers in [i + 1, n - 1].
        l = i + 1
        r = len(nums) - 1
        while l < r:
          sum = nums[i] + nums[l] + nums[r]
          if sum == 0:
            ans.append((nums[i], nums[l], nums[r]))
            l += 1
            r -= 1
            while nums[l] == nums[l - 1] and l < r:
              l += 1
            while nums[r] == nums[r + 1] and l < r:

              r -= 1
          elif sum < 0:
            l += 1
          else:
            r -= 1

      return ans
```

Python


```python
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        n = len(nums)
        ans = []
        for i in range(n - 2):
            if nums[i] > 0:
                break
            if i and nums[i] == nums[i - 1]:
                continue
            j, k = i + 1, n - 1
            while j < k:
                x = nums[i] + nums[j] + nums[k]
                if x < 0:
                    j += 1
                elif x > 0:
                    k -= 1
                else:
                    ans.append([nums[i], nums[j], nums[k]])
                    j, k = j + 1, k - 1
                    while j < k and nums[j] == nums[j - 1]:
                        j += 1
                    while j < k and nums[k] == nums[k + 1]:
                        k -= 1
        return ans
```

Python

```python
class Solution:
    def threeSum(self, nums: List[int]) -> List[List[int]]:
        nums.sort()
        n = len(nums)
        ans = []
        for i in range(n - 2):
            if nums[i] > 0:
                break
            if i and nums[i] == nums[i - 1]:
                continue
            j, k = i + 1, n - 1
            while j < k:
                x = nums[i] + nums[j] + nums[k]
                if x < 0:
                    j += 1
                elif x > 0:
                    k -= 1
                else:
                    ans.append([nums[i], nums[j], nums[k]])
                    j, k = j + 1, k - 1
                    while j < k and nums[j] == nums[j - 1]:
                        j += 1
                    while j < k and nums[k] == nums[k + 1]:
                        k -= 1
        return ans
```

[*] Two Pointers — Pattern Solution (matched from community answers)

Python


```python
# Function to find all unique triplets that sum to zero.
def threeSum(nums):
    # Sort the array to facilitate finding duplicates and using two pointers.
    nums.sort()
    result = []
    n = len(nums)

    # Iterate through the array, fixing one element at a time.
    for i in range(n - 2):
        # Skip duplicate elements for the first element.
        if i > 0 and nums[i] == nums[i - 1]:
            continue

        left, right = i + 1, n - 1
        while left < right:
            current_sum = nums[i] + nums[left] + nums[right]
            # Check if the triplet adds up to zero.
            if current_sum == 0:
                result.append([nums[i], nums[left], nums[right]])
                # Skip duplicates for the second element.
                while left < right and nums[left] == nums[left + 1]:
                    left += 1
                # Skip duplicates for the third element.
                while left < right and nums[right] == nums[right - 1]:
                    right -= 1

                left += 1
                right -= 1
            elif current_sum < 0:
                left += 1
            else:
                right -= 1

    return result

# Example usage:
#print(threeSum([-1,0,1,2,-1,-4]))
```

#16 MEDIUM

3Sum Closest

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers · Sorting

Lists: Grind 169

Problem:

Given an integer array nums of length n and an integer target, find three integers at distinct indices in nums such that the sum is closest to target.

Return the sum of the three integers.

You may assume that each input would have exactly one solution.

Example 1:

```
Input: nums = [-1,2,1,-4], target = 1
Output: 2
```



```
Explanation: The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).
```

Example 2:

```
Input: nums = [0,0,0], target = 1
Output: 0
Explanation: The sum that is closest to the target is 0. (0 + 0 + 0 = 0).
```

Constraints:

- 3 <= nums.length <= 500
- -1000 <= nums[i] <= 1000
- -104 <= target <= 104

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute-Force Approach
class Solution:
    def threeSumClosest(self, nums, target):
        # Brute-Force O(n^2) - nested loops instead of two pointers
        n = len(nums)
        for i in range(n):
            for j in range(i + 1, n):
                # check nums[i] and nums[j]
                pass
```

Python

Python

```python
# Python solution for 3Sum Closest
def threeSumClosest(nums, target):
    # Sort the array to allow two pointer technique
    nums.sort()
    # Initialize best_sum with the sum of the first three elements
    best_sum = sum(nums[:3])

    # Iterate through each number as the fixed element
    for i in range(len(nums) - 2):
        left, right = i + 1, len(nums) - 1
        # Use two pointers to check pairs
        while left < right:
            current_sum = nums[i] + nums[left] + nums[right]
            # Update best_sum if current_sum is closer to target
            if abs(target - current_sum) < abs(target - best_sum):
                best_sum = current_sum
            # Move left pointer if current_sum is less than target to increase sum
            if current_sum < target:
                left += 1
            # Move right pointer if current_sum is greater than target to decrease sum
            elif current_sum > target:
                right -= 1
            else:
                # If the current_sum exactly equals target, return the target sum
                return current_sum
```

```
    # Return the closest sum found
    return best_sum

# Example usage:
# print(threeSumClosest([-1,2,1,-4], 1))
```

Python

```
class Solution:
  def threeSumClosest(self, nums: list[int], target: int) -> int:
    ans = nums[0] + nums[1] + nums[2]

    nums.sort()

    for i in range(len(nums) - 2):
      if i > 0 and nums[i] == nums[i - 1]:
        continue
      # Choose nums[i] as the first number in the triplet, then search the
      # remaining numbers in [i + 1, n - 1].
      l = i + 1
      r = len(nums) - 1
      while l < r:
        sum = nums[i] + nums[l] + nums[r]
        if sum == target:
          return sum
        if abs(sum - target) < abs(ans - target):
          ans = sum
        if sum < target:
          l += 1
        else:
          r -= 1

    return ans
```

Python

```
class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        nums.sort()
        n = len(nums)
        ans = inf
        for i, v in enumerate(nums):
            j, k = i + 1, n - 1
            while j < k:
                t = v + nums[j] + nums[k]
                if t == target:
                    return t
                if abs(t - target) < abs(ans - target):
                    ans = t
                if t > target:
                    k -= 1
                else:
                    j += 1
        return ans
```

Python


```
class Solution:
    def threeSumClosest(self, nums: List[int], target: int) -> int:
        nums.sort()
        n = len(nums)
        ans = inf
        for i, v in enumerate(nums):
            j, k = i + 1, n - 1
            while j < k:
                t = v + nums[j] + nums[k]
                if t == target:
                    return t
                if abs(t - target) < abs(ans - target):
                    ans = t
                if t > target:
                    k -= 1
                else:
                    j += 1
        return ans
```

[*] Two Pointers — Pattern Solution (matched from community answers)

Python

```
# Python solution for 3Sum Closest
def threeSumClosest(nums, target):
    # Sort the array to allow two pointer technique
    nums.sort()
    # Initialize best_sum with the sum of the first three elements
    best_sum = sum(nums[:3])

    # Iterate through each number as the fixed element
    for i in range(len(nums) - 2):
        left, right = i + 1, len(nums) - 1
        # Use two pointers to check pairs
        while left < right:
            current_sum = nums[i] + nums[left] + nums[right]
            # Update best_sum if current_sum is closer to target
            if abs(target - current_sum) < abs(target - best_sum):
                best_sum = current_sum
            # Move left pointer if current_sum is less than target to increase sum
            if current_sum < target:
                left += 1
            # Move right pointer if current_sum is greater than target to decrease sum
            elif current_sum > target:
                right -= 1
            else:
                # If the current_sum exactly equals target, return the target sum
                return current_sum

    # Return the closest sum found
    return best_sum

# Example usage:
# print(threeSumClosest([-1,2,1,-4], 1))
```

#31 MEDIUM

Next Permutation

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers

Lists: Grind 169

Problem:

A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

- For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

- For example, the next permutation of arr = [1,2,3] is [1,3,2].
- Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
- While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, find the next permutation of nums.

The replacement must be in place and use only constant extra memory.

Example 1:

```
Input: nums = [1,2,3]
Output: [1,3,2]
```

Example 2:

```
Input: nums = [3,2,1]
Output: [1,2,3]
```

Example 3:

```
Input: nums = [1,1,5]
Output: [1,5,1]
```

Constraints:

- 1 <= nums.length <= 100
- 0 <= nums[i] <= 100

>> Brute Force [Two Pointers] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def nextPermutation(self, nums):
        # Brute force O(n^2) - nested loops instead of two pointers
        n = len(nums)
        for i in range(n):
            for j in range(i + 1, n):
                # check nums[i] and nums[j]
                pass
```

Python

Python

```
def nextPermutation(nums):
    # Length of the array
    n = len(nums)
    # Find the first index from the right where the current number is less than the next number
    i = n - 2
    while i >= 0 and nums[i] >= nums[i + 1]:
        i -= 1
    # If such an index is found, find the number just larger than nums[i] from the right side to swap
    if i >= 0:
        j = n - 1
        while nums[j] <= nums[i]:
            j -= 1
        nums[i], nums[j] = nums[j], nums[i] # Swap the two numbers
    # Reverse the subarray from i+1 to the end to get the smallest sequence
    left, right = i + 1, n - 1
    while left < right:
        nums[left], nums[right] = nums[right], nums[left]
        left += 1
        right -= 1

# Example usage:
nums = [1, 2, 3]
nextPermutation(nums)
print(nums) # Output: [1, 3, 2]
```

Python

```
class Solution:
  def nextPermutation(self, nums: list[int]) -> None:
    n = len(nums)

    # From back to front, find the first number < nums[i + 1].
    i = n - 2
    while i >= 0:
      if nums[i] < nums[i + 1]:
        break
      i -= 1

    # From back to front, find the first number > nums[i], swap it with nums[i].
    if i >= 0:
      for j in range(n - 1, i, -1):
        if nums[j] > nums[i]:
          nums[i], nums[j] = nums[j], nums[i]
          break

    def reverse(nums: list[int], l: int, r: int) -> None:
      while l < r:
        nums[l], nums[r] = nums[r], nums[l]
        l += 1
        r -= 1

    # Reverse nums[i + 1..n - 1].

    reverse(nums, i + 1, len(nums) - 1)
```

Python


```
class Solution:
    def nextPermutation(self, nums: List[int]) -> None:
        n = len(nums)
        i = next((i for i in range(n - 2, -1, -1) if nums[i] < nums[i + 1]), -1)
        if ~i:
            j = next((j for j in range(n - 1, i, -1) if nums[j] > nums[i]))
            nums[i], nums[j] = nums[j], nums[i]
        nums[i + 1 :] = nums[i + 1 :][::-1]
```

Python

```
'''
>>> i = 3
>>> ~i
-4
>>> bool(~i)
True
#####################

>>> j = -1
>>> ~j
0
>>> bool(~j)
False
#####################

>>> a = (i for i in range (10, -1, -1) if i < 6)
>>> a
<generator object <genexpr> at 0x10a17ec0b0>
>>> next(a)
5
>>>
>>>
>>> b = (i for i in range (10, -1, -1) if i < 0)
>>> next(b)
Traceback (most recent call last):
```



```
  File "<stdin>", line 1, in <module>
StopIteration
>>> next(b, -1)
-1
>>>
'''
class Solution:
    def nextPermutation(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        n = len(nums)
        # next(iterator, 1) == default value -1
        i = next((i for i in range(n - 2, -1, -1) if nums[i] < nums[i + 1]), -1)
        if i != -1:
            j = next((j for j in range(n - 1, i, -1) if nums[j] > nums[i]))
            nums[i], nums[j] = nums[j], nums[i]
        nums[i + 1:] = nums[i + 1:][::-1]
        # wrong reverse: nums[i + 1:] = nums[i + 1:-1]

#############

'''
>>> a=[1,2,3]
>>> reversed(a)

<list_reverseiterator object at 0x10045Bbe0>
>>> list(reversed(a))
[3, 2, 1]

# but, use reversed(a) to directly assign values is ok
>>> b=[4,5,6,7,8]
>>> b[:3] = reversed(a)
>>> b
[3, 2, 1, 7, 8]
'''
class Solution:
    def nextPermutation(self, nums: List[int]) -> None:
        if not nums:
            return


        # □□□□□□□□□□□□□□□□□□□□□□□□□□□□next
        for i in range(len(nums)-2, -1, -1): # 3, 4, 5, 2, 1
            if nums[i] < nums[i+1]: # □□□□□□ => 4
                j = next(j for j in range(len(nums)-1, i, -1) if nums[j] > nums[i]) # □□□□□□nums-1□□□ => 5
                nums[i], nums[j] = nums[j], nums[i] # 3,5,4,2,1

                # reverse □□□□□□□□□□□□□
                # □□□□□□nums-1□□□□□□□□□□□□□□insert position
                nums[i+1:] = reversed(nums[i+1:]) # [4,2,1] reverse to [1,2,4] => 3, 5, 1, 2, 4
                return


        nums.reverse() # for□□return□□□□□□

##########
```

[*] Two Pointers — Pattern Solution (matched from community answers)

Python

```python
def nextPermutation(nums):
    # Length of the array
    n = len(nums)
    # Find the first index from the right where the current number is less than the next number
    i = n - 2
    while i >= 0 and nums[i] >= nums[i + 1]:
        i -= 1
    # If such an index is found, find the number just larger than nums[i] from the right side to swap
    if i >= 0:
        j = n - 1
        while nums[j] <= nums[i]:
            j -= 1
        nums[i], nums[j] = nums[j], nums[i] # Swap the two numbers
    # Reverse the subarray from i+1 to the end to get the smallest sequence
    left, right = i + 1, n - 1
    while left < right:
        nums[left], nums[right] = nums[right], nums[left]
        left += 1
        right -= 1

# Example usage:
nums = [1, 2, 3]
nextPermutation(nums)
print(nums) # Output: [1, 3, 2]
```

#75 MEDIUM

Sort Colors

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers · Sorting

Lists: Grind 169

Problem:

Given an array nums with n objects colored red, white, or blue, sort them in-place so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Example 1:

```
Input: nums = [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
```

Example 2:

```
Input: nums = [2,0,1]
Output: [0,1,2]
```

Constraints:

- n == nums.length
- 1 <= n <= 300
- nums[i] is either 0, 1, or 2.

Follow up: Could you come up with a one-pass algorithm using only constant extra space?

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def sortColors(self, nums):
        # Brute Force O(n log n) - just sort; Dutch-flag is the optimal O(n) approach
        nums.sort()
```

Python

Python

```python
# Function to sort the colors using Dutch National Flag algorithm
def sortColors(nums):
    # Initialize pointers for the start, current, and end positions
    low, mid, high = 0, 0, len(nums) - 1

    # Process elements until mid pointer exceeds high pointer
    while mid <= high:
        # If the current element is 0 (red)
        if nums[mid] == 0:
            # Swap current element with the element at low pointer
            nums[low], nums[mid] = nums[mid], nums[low]
            # Increment both low and mid pointers
            low += 1
            mid += 1
        # If the current element is 1 (white)
        elif nums[mid] == 1:
            # Move mid pointer forward since it's already in the correct place
            mid += 1
        # If the current element is 2 (blue)
        else:
            # Swap current element with the element at high pointer
            nums[mid], nums[high] = nums[high], nums[mid]
            # Decrement high pointer
            high -= 1

# Example usage:
nums = [2,0,2,1,1,0]
sortColors(nums)
print(nums) # Output: [0,0,1,1,2,2]
```

Python


```python
class Solution:
    def sortColors(self, nums: list[int]) -> None:
        zero = -1
        one = -1
        two = -1

        for num in nums:
            if num == 0:
                two += 1
                one += 1
                zero += 1
                nums[two] = 2
                nums[one] = 1
                nums[zero] = 0
            elif num == 1:
                two += 1
                one += 1
                nums[two] = 2
                nums[one] = 1
            else:
                two += 1
                nums[two] = 2
```

Python

```python
class Solution:
    def sortColors(self, nums: list[int]) -> None:
        l = 0 # The next 0 should be placed in l.
        r = len(nums) - 1 # The next 2 should be placed in r.

        i = 0
        while i <= r:
            if nums[i] == 0:
                nums[i], nums[l] = nums[l], nums[i]
                i += 1
                l += 1
            elif nums[i] == 1:
                i += 1
            else:
                # We may swap a 0 to index i, but we're still not sure whether this 0
                # is placed in the correct index, so we can't move pointer i.
                nums[i], nums[r] = nums[r], nums[i]
                r -= 1
```

Python

```python
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        i, j, k = -1, len(nums), 0
        while k < j:
            if nums[k] == 0:
                i += 1
                nums[i], nums[k] = nums[k], nums[i]
                k += 1
            elif nums[k] == 2:
                j -= 1
                nums[j], nums[k] = nums[k], nums[j]
            else:
                k += 1
```


Python

```python
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        i, j, k = -1, len(nums), 0
        while k < j:
            if nums[k] == 0:
                i += 1
                nums[i], nums[k] = nums[k], nums[i]
                k += 1
            elif nums[k] == 2:
                j -= 1
                nums[j], nums[k] = nums[k], nums[j]
            else:
                k += 1
```

[*] Two Pointers — Pattern Solution (stored optimal)

Python

```python
# Function to sort the colors using Dutch National Flag algorithm
def sortColors(nums):
    # Initialize pointers for the start, current, and end positions
    low, mid, high = 0, 0, len(nums) - 1

    # Process elements until mid pointer exceeds high pointer
    while mid <= high:
        # If the current element is 0 (red)
        if nums[mid] == 0:
            # Swap current element with the element at low pointer
            nums[low], nums[mid] = nums[mid], nums[low]
            # Increment both low and mid pointers
            low += 1
            mid += 1
        # If the current element is 1 (white)
        elif nums[mid] == 1:
            # Move mid pointer forward since it's already in the correct place
            mid += 1
        # If the current element is 2 (blue)
        else:
            # Swap current element with the element at high pointer
            nums[mid], nums[high] = nums[high], nums[mid]
            # Decrement high pointer
            high -= 1

# Example usage:
nums = [2,0,2,1,1,0]
sortColors(nums)
print(nums) # Output: [0,0,1,1,2,2]
```

#161 MEDIUM

One Edit Distance

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Two Pointers · String

Lists: Premium 98

Problem:

Given two strings s and t, return true if they are both one edit distance apart, otherwise return false.

A string s is said to be one distance apart from a string t if you can:

- Insert exactly one character into s to get t.
- Delete exactly one character from s to get t.
- Replace exactly one character of s with a different character to get t.

Example 1:

```
Input: s = "ab", t = "acb"
Output: true
Explanation: We can insert 'c' into s to get t.
```

Example 2:

```
Input: s = "", t = ""
Output: false
Explanation: We cannot get t from s by only one step.
```

Constraints:

- 0 <= s.length, t.length <= 104
- s and t consist of lowercase letters, uppercase letters, and digits.

Python

Python

```python
# Function to check if two strings are one edit distance apart
def isOneEditDistance(s: str, t: str) -> bool:
    # Get lengths of both strings
    m, n = len(s), len(t)

    # The difference in length must not be greater than 1
    if abs(m - n) > 1:
        return False

    # Ensure s is the shorter string, or they are equal in length
    if m > n:
        return isOneEditDistance(t, s)

    # Initialize variables: found difference flag and indices for both strings
    found_difference = False
    i = j = 0

    while i < m and j < n:
        if s[i] != t[j]:
            # If a difference has already been found, return False
            if found_difference:
                return False
            found_difference = True
            # If lengths are equal, move both pointers
            if m == n:
```



```python
                i += 1
        else:
            # If characters are the same, move the pointer in the shorter string
            i += 1
        # Always move pointer for longer string
        j += 1

    # If no difference was found during iteration, then one extra character at the end is the edit
    return found_difference or (j - i == 1)

# Example usage:
#print(isOneEditDistance("ab", "acb")) # Output: True
```

Python

```python
class Solution:
    def isOneEditDistance(self, s: str, t: str) -> bool:
        m = len(s)
        n = len(t)
        if m > n: # Make sure that |s| <= |t|.
            return self.isOneEditDistance(t, s)

        for i in range(m):
            if s[i] != t[i]:
                if m == n:
                    return s[i + 1:] == t[i + 1:] # Replace s[i] with t[i].
                return s[i:] == t[i + 1:] # Delete t[i].

        return m + 1 == n # Delete t[-1].
```

Python

```python
class Solution:
    def isOneEditDistance(self, s: str, t: str) -> bool:
        if len(s) < len(t):
            return self.isOneEditDistance(t, s)
        m, n = len(s), len(t)
        if m - n > 1:
            return False
        for i, c in enumerate(t):
            if c != s[i]:
                return s[i + 1 :] == t[i + 1 :] if m == n else s[i + 1 :] == t[i:]
        return m == n + 1
```

Python

```python
class Solution:
    def isOneEditDistance(self, s: str, t: str) -> bool:
        if len(s) < len(t): # goal is to ensure t is always the shorter one
            return self.isOneEditDistance(t, s)
        m, n = len(s), len(t)
        if m - n > 1:
            # case: abc, abcdefg, return false
            return False
        for i, c in enumerate(t):
            if c != s[i]:
                return s[i + 1 :] == t[i + 1 :] if m == n else s[i + 1 :] == t[i:]
        # case: abc, abcd, return true
        # case: abc, abc, the same 2 strings should return false
        # case: abc, abcdefg, return false
        return m == n + 1

###########

class Solution(object):
    def isOneEditDistance(self, s, t):
        """
        :type s: str
        :type t: str
        :rtype: bool
        """

        if len(s) >= len(t):
            i = 0
            while i < len(t) and s[i] == t[i]:
                i += 1
            return s != t and (s[i + 1:] == t[i:] if len(s) != len(t) else s[i + 1:] == t[i + 1:])
        else:
            return self.isOneEditDistance(t, s)
```

[*] Two Pointers — Pattern Solution (stored optimal)

Python


```python
# Function to check if two strings are one edit distance apart
def isOneEditDistance(s: str, t: str) -> bool:
    # Get lengths of both strings
    m, n = len(s), len(t)

    # The difference in length must not be greater than 1
    if abs(m - n) > 1:
        return False

    # Ensure s is the shorter string, or they are equal in length
    if m > n:
        return isOneEditDistance(t, s)

    # Initialize variables: found difference flag and indices for both strings
    found_difference = False
    i = j = 0

    while i < m and j < n:
        if s[i] != t[j]:
            # If a difference has already been found, return False
            if found_difference:
                return False
            found_difference = True
            # If lengths are equal, move both pointers
            if m == n:
                i += 1
        else:
            # If characters are the same, move the pointer in the shorter string
            i += 1
        # Always move pointer for longer string
        j += 1

    # If no difference was found during iteration, then one extra character at the end is the edit
    return found_difference or (j - i == 1)

# Example usage:
#print(isOneEditDistance("ab", "acb")) # Output: True
```

#167 MEDIUM

Two Sum II - Input Array Is Sorted

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers · Binary Search

Lists: Denny Zhang

Problem:

Given a 1-indexed array of integers numbers that is already sorted in non-decreasing order, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index1] and numbers[index2] where 1 <= index1 < index2 <= numbers.length.

Return the indices of the two numbers index1 and index2, each incremented by one, as an integer array [index1, index2] of length 2.

The tests are generated such that there is exactly one solution. You may not use the same element twice.

Your solution must use only constant extra space.

Example 1:

```
Input: numbers = [2,7,11,15], target = 9
Output: [1,2]
Explanation: The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1, 2].
```

Example 2:

```
Input: numbers = [2,3,4], target = 6
Output: [1,3]
Explanation: The sum of 2 and 4 is 6. Therefore index1 = 1, index2 = 3. We return [1, 3].
```

Example 3:

```
Input: numbers = [-1,0], target = -1
Output: [1,2]
Explanation: The sum of -1 and 0 is -1. Therefore index1 = 1, index2 = 2. We return [1, 2].
```

Constraints:

- 2 <= numbers.length <= 3 * 104
- -1000 <= numbers[i] <= 1000
- numbers is sorted in non-decreasing order.
- -1000 <= target <= 1000
- The tests are generated such that there is exactly one solution.

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def twoSum(self, numbers, target):
        # Brute Force O(n^2) - check every pair (ignore sorted property)
        for i in range(len(numbers)):
            for j in range(i + 1, len(numbers)):
                if numbers[i] + numbers[j] == target:
                    return [i + 1, j + 1]
```

Python

Python


```python
# Python solution for Two Sum II - Input Array Is Sorted
def twoSum(numbers, target):
    # Initialize two pointers at the beginning and end of the array
    left = 0
    right = len(numbers) - 1

    # Continue until the two pointers meet
    while left < right:
        # Calculate the sum of the two pointed values
        current_sum = numbers[left] + numbers[right]
        # If the sum matches the target, return the 1-indexed results
        if current_sum == target:
            return [left + 1, right + 1]
        # If the current sum is less than the target, move the left pointer to the right
        elif current_sum < target:
            left += 1
        # Otherwise, move the right pointer to the left since the current sum is too high
        else:
            right -= 1

# Example call
print(twoSum([2, 7, 11, 15], 9)) # Output: [1, 2]
```

Python

```python
class Solution:
    def twoSum(self, numbers: list[int], target: int) -> list[int]:
        l = 0
        r = len(numbers) - 1

        while l < r:
          sum = numbers[l] + numbers[r]
          if sum == target:
            return [l + 1, r + 1]
          if sum < target:
            l += 1
          else:
            r -= 1
```

Python

```python
class Solution:
    def twoSum(self, numbers: List[int], target: int) -> List[int]:
        n = len(numbers)
        for i in range(n - 1):
            x = target - numbers[i]
            j = bisect_left(numbers, x, lo=i + 1)
            if j < n and numbers[j] == x:
                return [i + 1, j + 1]
```

Python


```python
class Solution:
    def twoSum(self, numbers: List[int], target: int) -> List[int]:
        i, j = 0, len(numbers) - 1
        while i < j:
            x = numbers[i] + numbers[j]
            if x == target:
                return [i + 1, j + 1]
            if x < target:
                i += 1
            else:
                j -= 1
```

[*] Two Pointers — Pattern Solution (matched from community answers)

Python

```python
# Python solution for Two Sum II - Input Array Is Sorted
def twoSum(numbers, target):
    # Initialize two pointers at the beginning and end of the array
    left = 0
    right = len(numbers) - 1

    # Continue until the two pointers meet
    while left < right:
        # Calculate the sum of the two pointed values
        current_sum = numbers[left] + numbers[right]
        # If the sum matches the target, return the 1-indexed results
        if current_sum == target:
            return [left + 1, right + 1]
        # If the current sum is less than the target, move the left pointer to the right
        elif current_sum < target:
            left += 1
        # Otherwise, move the right pointer to the left since the current sum is too high
        else:
            right -= 1

# Example call
print(twoSum([2, 7, 11, 15], 9)) # Output: [1, 2]
```

#186 MEDIUM

Reverse Words in a String II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Two Pointers · String

Lists: Premium 98

Problem:

Given a character array s, reverse the order of the words.

A word is defined as a sequence of non-space characters. The words in s will be separated by a single space.

Your code must solve the problem in-place, i.e. without allocating extra space.

Example 1:


```
Input: s = ["t","h","e"," ","s","k","y"," ","i","s"," ","b","l","u","e"]
Output: ["b","l","u","e"," ","i","s"," ","s","k","y"," ","t","h","e"]
```

Example 2:

```
Input: s = ["a"]
Output: ["a"]
```

Constraints:

- 1 <= s.length <= 105
- s[i] is an English letter (uppercase or lowercase), digit, or space ' '.
- There is at least one word in s.
- s does not contain leading or trailing spaces.
- All the words in s are guaranteed to be separated by a single space.

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def reverse_range(self, s, left, right):
        # Brute Force O(n^2) - nested loops instead of two pointers
        n = len(s)
        for i in range(n):
            for j in range(i + 1, n):
                # check s[i] and s[j]
                pass
```

Python

Python

```python
def reverse_range(s, left, right):
    # Reverse the elements in s from index left to right.
    while left < right:
        s[left], s[right] = s[right], s[left]
        left += 1
        right -= 1

def reverseWords(s):
    # Step 1: Reverse the entire array.
    reverse_range(s, 0, len(s) - 1)

    n = len(s)
    start = 0
    # Step 2: Reverse each word to correct their order.
    for i in range(n + 1):
        if i == n or s[i] == ' ':
            reverse_range(s, start, i - 1)
            start = i + 1
```

Python

```python
class Solution:
    def reverseWords(self, s: list[str]) -> None:
        def reverse(l: int, r: int) -> None:
            while l < r:
                s[l], s[r] = s[r], s[l]
                l += 1
                r -= 1

        def reverseWords(n: int) -> None:
            i = 0
            j = 0

            while i < n:
                while i < j or (i < n and s[i] == ' '): # Skip the spaces.
                    i += 1
                while j < i or (j < n and s[j] != ' '): # Skip the spaces.
                    j += 1
                reverse(i, j - 1) # Reverse the word.

        reverse(0, len(s) - 1) # Reverse the whole string.
        reverseWords(len(s)) # Reverse each word.
```

Python

```python
class Solution:
    def reverseWords(self, s: List[str]) -> None:
        def reverse(i: int, j: int):
            while i < j:
                s[i], s[j] = s[j], s[i]
                i, j = i + 1, j - 1

        i, n = 0, len(s)
        for j, c in enumerate(s):
            if c == " ":
                reverse(i, j - 1)
                i = j + 1
            elif j == n - 1:
                reverse(i, j)
        reverse(0, n - 1)
```

Python


```python
class Solution:
    def reverseWords(self, s: List[str]) -> None:
        """
        Do not return anything, modify s in-place instead.
        """

        def reverse(s, i, j):
            while i < j:
                s[i], s[j] = s[j], s[i]
                i += 1
                j -= 1

        i, j, n = 0, 0, len(s)
        while j < n:
            if s[j] == ' ':
                reverse(s, i, j - 1)
                i = j + 1
            elif j == n - 1:
                reverse(s, i, j)
            j += 1
        reverse(s, 0, n - 1)
```

[*] Two Pointers — Pattern Solution (matched from community answers)

Python

```python
def reverse_range(s, left, right):
    # Reverse the elements in s from index left to right.
    while left < right:
        s[left], s[right] = s[right], s[left]
        left += 1
        right -= 1

def reverseWords(s):
    # Step 1: Reverse the entire array.
    reverse_range(s, 0, len(s) - 1)

    n = len(s)
    start = 0
    # Step 2: Reverse each word to correct their order.
    for i in range(n + 1):
        if i == n or s[i] == ' ':
            reverse_range(s, start, i - 1)
            start = i + 1
```

#189 MEDIUM

Rotate Array

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Math · Two Pointers

Lists: Grind 169

Problem:

Given an integer array nums, rotate the array to the right by k steps, where k is non-negative.

Example 1:


```
Input: nums = [1,2,3,4,5,6,7], k = 3
Output: [5,6,7,1,2,3,4]
Explanation:
rotate 1 steps to the right: [7,1,2,3,4,5,6]
rotate 2 steps to the right: [6,7,1,2,3,4,5]
rotate 3 steps to the right: [5,6,7,1,2,3,4]
```

Example 2:

```
Input: nums = [-1,-100,3,99], k = 2
Output: [3,99,-1,-100]
Explanation:
rotate 1 steps to the right: [99,-1,-100,3]
rotate 2 steps to the right: [3,99,-1,-100]
```

Constraints:

- 1 <= nums.length <= 105
- -231 <= nums[i] <= 231 - 1
- 0 <= k <= 105

Follow up:

- Try to come up with as many solutions as you can. There are at least three different ways to solve this problem.
- Could you do it in-place with O(1) extra space?

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def rotate(self, nums, k):
        # Brute Force O(n^2) - nested loops instead of two pointers
        n = len(nums)
        for i in range(n):
            for j in range(i + 1, n):
                # check nums[i] and nums[j]
                pass
```

Python

Python


```python
# Python solution using the reverse method

def rotate(nums, k):
    # Calculate the effective rotations needed
    n = len(nums)
    k = k % n # in case k is greater than the length of the array

    # Helper function to reverse a portion of the list in place
    def reverse(left, right):
        while left < right:
            # Swap the elements at indices left and right
            nums[left], nums[right] = nums[right], nums[left]
            left += 1
            right -= 1

    # Reverse the entire array
    reverse(0, n - 1)
    # Reverse the first k elements to restore their order
    reverse(0, k - 1)
    # Reverse the remaining n - k elements
    reverse(k, n - 1)

# Example usage:
# nums = [1,2,3,4,5,6,7]
# rotate(nums, 3)

# print(nums) # Output: [5,6,7,1,2,3,4]
```

Python

```python
class Solution:
    def rotate(self, nums: list[int], k: int) -> None:
        k %= len(nums)
        self.reverse(nums, 0, len(nums) - 1)
        self.reverse(nums, 0, k - 1)
        self.reverse(nums, k, len(nums) - 1)

    def reverse(self, nums: list[int], l: int, r: int) -> None:
        while l < r:
            nums[l], nums[r] = nums[r], nums[l]
            l += 1
            r -= 1
```

Python

```python
class Solution:
    def rotate(self, nums: List[int], k: int) -> None:
        def reverse(i: int, j: int):
            while i < j:
                nums[i], nums[j] = nums[j], nums[i]
                i, j = i + 1, j - 1

        n = len(nums)
        k %= n
        reverse(0, n - 1)
        reverse(0, k - 1)
        reverse(k, n - 1)
```

Python


```python
class Solution:
    def rotate(self, nums: List[int], k: int) -> None:
        k %= len(nums)
        nums[:] = nums[-k:] + nums[:-k]

###########

class Solution:
    def rotate(self, nums: List[int], k: int) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        n = len(nums)
        k %= n
        if n < 2 or k == 0:
            return
        nums[:] = nums[::-1]
        nums[:k] = nums[:k][::-1]
        nums[k:] = nums[k:][::-1]

###########

class Solution(object):
    def rotate(self, nums, k):
        """
        :type nums: List[int]
        :type k: int
        :rtype: void Do not return anything, modify nums in-place instead.
        """
        if len(nums) == 0 or k == 0:
            return

        def reverse(start, end, s):
            while start < end:
                s[start], s[end] = s[end], s[start]
                start += 1
                end -= 1

        n = len(nums) - 1
        k = k % len(nums)
        reverse(0, n - k, nums)
        reverse(n - k + 1, n, nums)
        reverse(0, n, nums)
```

[*] Two Pointers — Pattern Solution (matched from community answers)

Python


```python
# Python solution using the reverse method

def rotate(nums, k):
    # Calculate the effective rotations needed
    n = len(nums)
    k = k % n # in case k is greater than the length of the array

    # Helper function to reverse a portion of the list in place
    def reverse(left, right):
        while left < right:
            # Swap the elements at indices left and right
            nums[left], nums[right] = nums[right], nums[left]
            left += 1
            right -= 1

    # Reverse the entire array
    reverse(0, n - 1)
    # Reverse the first k elements to restore their order
    reverse(0, k - 1)
    # Reverse the remaining n - k elements
    reverse(k, n - 1)

# Example usage:
# nums = [1,2,3,4,5,6,7]
# rotate(nums, 3)

# print(nums) # Output: [5,6,7,1,2,3,4]
```


#532 MEDIUM

K-diff Pairs in an Array

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Two Pointers · Binary Search · Sorting

Lists: CodeSignal

Problem:

Given an array of integers nums and an integer k, return the number of unique k-diff pairs in the array.

A k-diff pair is an integer pair (nums[i], nums[j]), where the following are true:

- 0 <= i, j < nums.length
- i != j
- |nums[i] - nums[j]| == k

Notice that |val| denotes the absolute value of val.

Example 1:

```
Input: nums = [3,1,4,1,5], k = 2
Output: 2
Explanation: There are two 2-diff pairs in the array, (1, 3) and (3, 5).
Although we have two 1s in the input, we should only return the number of unique pairs.
```

Example 2:

```
Input: nums = [1,2,3,4,5], k = 1
Output: 4
```

```
Explanation: There are four 1-diff pairs in the array, (1, 2), (2, 3), (3, 4) and (4, 5).
```

Example 3:

```
Input: nums = [1,3,1,5,4], k = 0
Output: 1
Explanation: There is one 0-diff pair in the array, (1, 1).
```

Constraints:

- 1 <= nums.length <= 104
- -107 <= nums[i] <= 107
- 0 <= k <= 107

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def find_pairs(self, nums, k):
        # Brute Force O(n^2) - nested loops instead of two pointers
        n = len(nums)
        for i in range(n):
            for j in range(i + 1, n):
                # check nums[i] and nums[j]
                pass
```

Python

Python

```python
# Define a function to count k-diff pairs in the array.
def find_pairs(nums, k):
    # If k is negative, return 0 as absolute difference can't be negative.
    if k < 0:
        return 0

    count = 0

    # k == 0: Need to find numbers that appear more than once.
    if k == 0:
        # Create a frequency map for nums.
        freq = {}
        for num in nums:
            freq[num] = freq.get(num, 0) + 1
        # Count numbers with frequency greater than 1.
        for val in freq.values():
            if val > 1:
                count += 1
    else:
        # k > 0: Use a set to check for the existence of each number's complement.
        unique_nums = set(nums)
        for num in unique_nums:
            if num + k in unique_nums:
                count += 1

    return count
```



```python
# Example usage:
# print(find_pairs([3,1,4,1,5], 2)) # Expected output: 2
```

Python

```python
class Solution:
    def findPairs(self, nums: list[int], k: int) -> int:
        ans = 0
        numToIndex = {num: i for i, num in enumerate(nums)}

        for i, num in enumerate(nums):
            target = num + k
            if target in numToIndex and numToIndex[target] != i:
                ans += 1
                del numToIndex[target]

        return ans
```

Python

```python
class Solution:
    def findPairs(self, nums: List[int], k: int) -> int:
        ans = set()
        vis = set()
        for x in nums:
            if x - k in vis:
                ans.add(x - k)
            if x + k in vis:
                ans.add(x)
            vis.add(x)
        return len(ans)
```

Python

```python
class Solution:
    def findPairs(self, nums: List[int], k: int) -> int:
        vis, ans = set(), set()
        for v in nums:
            if v - k in vis:
                ans.add(v - k)
            if v + k in vis:
                ans.add(v)
            vis.add(v)
        return len(ans)
```

[*] Two Pointers — Pattern Solution (stored optimal)

Python


```python
# Define a function to count k-diff pairs in the array.
def find_pairs(nums, k):
    # If k is negative, return 0 as absolute difference can't be negative.
    if k < 0:
        return 0

    count = 0

    # k == 0: Need to find numbers that appear more than once.
    if k == 0:
        # Create a frequency map for nums.
        freq = {}
        for num in nums:
            freq[num] = freq.get(num, 0) + 1
        # Count numbers with frequency greater than 1.
        for val in freq.values():
            if val > 1:
                count += 1
    else:
        # k > 0: Use a set to check for the existence of each number's complement.
        unique_nums = set(nums)
        for num in unique_nums:
            if num + k in unique_nums:
                count += 1

    return count
```

```python
# Example usage:
# print(find_pairs([3,1,4,1,5], 2)) # Expected output: 2
```


#923 MEDIUM

3Sum With Multiplicity

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Two Pointers · Sorting · Counting

Lists: CodeSignal

Problem:

Given an integer array arr, and an integer target, return the number of tuples i, j, k such that i < j < k and arr[i] + arr[j] + arr[k] == target.

As the answer can be very large, return it modulo 109 + 7.

Example 1:

```
Input: arr = [1,1,2,2,3,3,4,4,5,5], target = 8
Output: 20
Explanation:
Enumerating by the values (arr[i], arr[j], arr[k]):
(1, 2, 5) occurs 8 times;
(1, 3, 4) occurs 8 times;
(2, 2, 4) occurs 2 times;
(2, 3, 3) occurs 2 times.
```

Example 2:


```
Input: arr = [1,1,2,2,2,2,2], target = 5
Output: 12
Explanation:
arr[i] = 1, arr[j] = arr[k] = 2 occurs 12 times:
We choose one 1 from [1,1] in 2 ways,
and two 2s from [2,2,2,2] in 6 ways.
```

Example 3:

```
Input: arr = [2,1,3], target = 6
Output: 1
Explanation: (1, 2, 3) occured one time in the array so we return 1.
```

Constraints:

- 3 <= arr.length <= 3000
- 0 <= arr[i] <= 100
- 0 <= target <= 300

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def threeSumMulti(self, arr, target):
        # Brute Force O(n^2) - nested loops instead of two pointers
        n = len(arr)
        for i in range(n):
            for j in range(i + 1, n):
                # check arr[i] and arr[j]
                pass
```

Python

Python


```python
MOD = 10**9 + 7

def threeSumMulti(arr, target):
    # Create a frequency array for numbers 0 to 100
    freq = [0] * 101
    for num in arr:
        freq[num] += 1

    count = 0
    # Iterate through all possible values for x, y, and implicitly z
    for x in range(101):
        if freq[x] == 0:
            continue
        for y in range(x, 101):
            if freq[y] == 0:
                continue
            z = target - x - y
            if z < 0 or z > 100 or freq[z] == 0:
                continue
            if x == y and y == z:
                # All numbers are the same: nC3 = n*(n-1)*(n-2)/6
                count += freq[x] * (freq[x] - 1) * (freq[x] - 2) // 6
            elif x == y and y != z:
                # x and y are the same, z is different: nC2 for x multiplied by count[z]
                count += (freq[x] * (freq[x] - 1) // 2) * freq[z]

            elif x < y and y < z:
                # All distinct: product of counts
                count += freq[x] * freq[y] * freq[z]
            elif x < y and y == z:
                # x different, y and z are the same: count[x] multiplied by nC2 for y
                count += freq[x] * (freq[y] * (freq[y] - 1) // 2)
            count %= MOD
    return count

# Example usage:
print(threeSumMulti([1,1,2,2,3,3,4,4,5,5], 8))
```

Python

```python
class Solution:
    def threeSumMulti(self, arr: list[int], target: int) -> int:
        MOD = 1_000_000_007
        ans = 0
        count = collections.Counter(arr)

        for i, x in count.items():
            for j, y in count.items():
                k = target - i - j
                if k not in count:
                    continue
                if i == j and j == k:
                    ans = (ans + x * (x - 1) * (x - 2) // 6) % MOD
                elif i == j and j != k:
                    ans = (ans + x * (x - 1) // 2 * count[k]) % MOD
                elif i < j and j < k:
                    ans = (ans + x * y * count[k]) % MOD

        return ans % MOD
```


Python

```python
class Solution:
    def threeSumMulti(self, arr: List[int], target: int) -> int:
        mod = 10**9 + 7
        cnt = Counter(arr)
        ans = 0
        for j, b in enumerate(arr):
            cnt[b] -= 1
            for a in arr[:j]:
                c = target - a - b
                ans = (ans + cnt[c]) % mod
        return ans
```

Python

```python
class Solution:
    def threeSumMulti(self, arr: List[int], target: int) -> int:
        cnt = Counter(arr)
        ans = 0
        mod = 10**9 + 7
        for j, b in enumerate(arr):
            cnt[b] -= 1
            for i in range(j):
                a = arr[i]
                c = target - a - b
                ans = (ans + cnt[c]) % mod
        return ans
```

[*] Two Pointers — Pattern Solution (stored optimal)

Python

```python
MOD = 10**9 + 7

def threeSumMulti(arr, target):
    # Create a frequency array for numbers 0 to 100
    freq = [0] * 101
    for num in arr:
        freq[num] += 1

    count = 0
    # Iterate through all possible values for x, y, and implicitly z
    for x in range(101):
        if freq[x] == 0:
            continue
        for y in range(x, 101):
            if freq[y] == 0:
                continue
            z = target - x - y
            if z < 0 or z > 100 or freq[z] == 0:
                continue
            if x == y and y == z:
                # All numbers are the same: nC3 = n*(n-1)*(n-2)/6
                count += freq[x] * (freq[x] - 1) * (freq[x] - 2) // 6
            elif x == y and y != z:
                # x and y are the same, z is different: nC2 for x multiplied by count[z]
                count += (freq[x] * (freq[x] - 1) // 2) * freq[z]
```

```
            elif x < y and y < z:
                # Mix: multiply product of counts
                count += freq[x] * freq[y] * freq[z]
            elif x < y and y == z:
                # x different, y and z are the same: count[x] multiplied by nC2 for y
                count += freq[x] * (freq[y] * (freq[y] - 1) // 2)
            count %= MOD
    return count

# Example usage:
print(threeSumMulti([1,1,2,2,3,3,4,4,5,5], 8))
```

#986 MEDIUM

Interval List Intersections

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers

Lists: Denny Zhang

Problem:

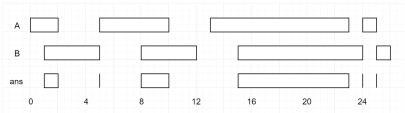
You are given two lists of closed intervals, firstList and secondList, where firstList[i] = [starti, endi] and secondList[j] = [startj, endj]. Each list of intervals is pairwise disjoint and in sorted order.

Return the intersection of these two interval lists.

A closed interval [a, b] (with a <= b) denotes the set of real numbers x with a <= x <= b.

The intersection of two closed intervals is a set of real numbers that are either empty or represented as a closed interval. For example, the intersection of [1, 3] and [2, 4] is [2, 3].

Example 1:




```
Input: firstList = [[0,2],[5,10],[13,23],[24,25]], secondList =
[[1,5],[8,12],[15,24],[25,26]]
Output: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
```

Example 2:

```
Input: firstList = [[1,3],[5,9]], secondList = []
Output: []
```

Constraints:

- 0 <= firstList.length, secondList.length <= 1000
- firstList.length + secondList.length >= 1

- 0 <= starti < endi <= 109
- endi < starti+1
- 0 <= startj < endj <= 109
- endj < startj+1

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def intervalIntersection(self, firstList, secondList):
        # Brute Force O(n^2) - nested loops instead of two pointers
        n = len(firstList)
        for i in range(n):
            for j in range(i + 1, n):
                # check firstList[i] and firstList[j]
                pass
```

Python

Python

```python
# Function to compute the interval intersections using two pointers
def intervalIntersection(firstList, secondList):
    intersections = [] # List to store the resulting intersections
    i, j = 0, 0 # Pointers for firstList and secondList

    # Loop until either list is fully traversed
    while i < len(firstList) and j < len(secondList):
        start1, end1 = firstList[i]
        start2, end2 = secondList[j]

        # Find the intersection boundaries
        start_overlap = max(start1, start2)
        end_overlap = min(end1, end2)

        # Check if there is an intersection
        if start_overlap <= end_overlap:
            intersections.append([start_overlap, end_overlap])

        # Move the pointer that has the smaller interval ending
        if end1 < end2:
            i += 1
        else:
            j += 1

    return intersections


# Example usage:
firstList = [[0,2],[5,10],[13,23],[24,25]]
secondList = [[1,5],[8,12],[15,24],[25,26]]
print(intervalIntersection(firstList, secondList)) # Output: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
```

Python


```
class Solution:
   def intervalIntersection(self, firstList: list[list[int]],
                            secondList: list[list[int]]) -> list[list[int]]:

    ans = []
    i = 0
    j = 0

    while i < len(firstList) and j < len(secondList):
      # lo := the start of the intersection
      # hi := the end of the intersection
      lo = max(firstList[i][0], secondList[j][0])
      hi = min(firstList[i][1], secondList[j][1])
      if lo <= hi:
        ans.append([lo, hi])
      if firstList[i][1] < secondList[j][1]:
        i += 1
      else:
        j += 1

    return ans
```

Python

```
class Solution:
    def intervalIntersection(
        self, firstList: List[List[int]], secondList: List[List[int]]
    ) -> List[List[int]]:
        i = j = 0
        ans = []
        while i < len(firstList) and j < len(secondList):
            s1, e1, s2, e2 = *firstList[i], *secondList[j]
            l, r = max(s1, s2), min(e1, e2)
            if l <= r:
                ans.append([l, r])
            if e1 < e2:
                i += 1
            else:
                j += 1
        return ans
```

Python

```
class Solution:
    def intervalIntersection(
        self, firstList: List[List[int]], secondList: List[List[int]]
    ) -> List[List[int]]:
        i = j = 0
        ans = []
        while i < len(firstList) and j < len(secondList):
            s1, e1, s2, e2 = *firstList[i], *secondList[j]
            l, r = max(s1, s2), min(e1, e2)
            if l <= r:
                ans.append([l, r])
            if e1 < e2:
                i += 1
            else:
                j += 1
        return ans
```


[*] Two Pointers — Pattern Solution (stored optimal)

Python

```python
# Function to compute the interval intersections using two pointers
def intervalIntersection(firstList, secondList):
    intersections = [] # List to store the resulting intersections
    i, j = 0, 0 # Pointers for firstList and secondList

    # Loop until either list is fully traversed
    while i < len(firstList) and j < len(secondList):
        start1, end1 = firstList[i]
        start2, end2 = secondList[j]

        # Find the intersection boundaries
        start_overlap = max(start1, start2)
        end_overlap = min(end1, end2)

        # Check if there is an intersection
        if start_overlap <= end_overlap:
            intersections.append([start_overlap, end_overlap])

        # Move the pointer that has the smaller interval ending
        if end1 < end2:
            i += 1
        else:
            j += 1

    return intersections


# Example usage:
firstList = [[0,2],[5,10],[13,23],[24,25]]
secondList = [[1,5],[8,12],[15,24],[25,26]]
print(intervalIntersection(firstList, secondList)) # Output: [[1,2],[5,5],[8,10],[15,23],[24,24],[25,25]]
```

#1055 MEDIUM

Shortest Way to Form String

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Two Pointers · String · Binary Search

Lists: Premium 98

Problem:

A subsequence of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not).

Given two strings source and target, return the minimum number of subsequences of source such that their concatenation equals target. If the task is impossible, return -1.

Example 1:

```
Input: source = "abc", target = "abcbc"
Output: 2
Explanation: The target "abcbc" can be formed by "abc" and "bc", which are subsequences of
source "abc".
```


Example 2:

```
Input: source = "abc", target = "acdbc"
Output: -1
Explanation: The target string cannot be constructed from the subsequences of source string
due to the character "d" in target string.
```

Example 3:

```
Input: source = "xyz", target = "xzyxz"
Output: 3
Explanation: The target string can be constructed as follows "xz" + "y" + "xz".
```

Constraints:

- 1 <= source.length, target.length <= 1000
- source and target consist of lowercase English letters.

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def shortestWay(self, source, target):
        # Brute Force O(n^2) - nested loops instead of two pointers
        n = len(source)
        for i in range(n):
            for j in range(i + 1, n):
                # check source[i] and source[j]
                pass
```

Python

Python

```python
# Python solution using greedy two-pointer approach

def shortestWay(source, target):
    count = 0 # Count of subsequences used
    t_index = 0
    t_len = len(target)

    # Pre-check: if any character in target is not in source, return -1
    if set(target) - set(source):
        return -1

    # While there are still characters in target to match
    while t_index < t_len:
        s_index = 0 # Pointer for source
        progress = False # Flag to check if we made progress this pass

        # Iterate through source and try to match target characters
        while s_index < len(source) and t_index < t_len:
            if source[s_index] == target[t_index]:
                t_index += 1 # Move to next character in target
                progress = True
            s_index += 1

        count += 1 # One subsequence used for this pass
```



```
        # If no progress was made, then it is impossible to form the target
        if not progress:
            return -1

    return count

# Example usage:
# print(shortestWay("abc", "abcbc")) # Output: 2
```

Python

```
class Solution:
    def shortestWay(self, source: str, target: str) -> int:
        def f(i, j):
            while i < m and j < n:
                if source[i] == target[j]:
                    j += 1
                i += 1
            return j

        m, n = len(source), len(target)
        ans = j = 0
        while j < n:
            k = f(0, j)
            if k == j:
                return -1
            j = k
            ans += 1
        return ans
```

Python

```
class Solution:
    def shortestWay(self, source: str, target: str) -> int:
        def f(i, j):
            while i < m and j < n:
                if source[i] == target[j]:
                    j += 1
                i += 1
            return j

        m, n = len(source), len(target)
        ans = j = 0
        while j < n:
            k = f(0, j)
            if k == j:
                return -1
            j = k
            ans += 1
        return ans
```

[*] Two Pointers — Pattern Solution (stored optimal)

Python

```python
# Python solution using greedy two-pointer approach

def shortestWay(source, target):
    count = 0 # Count of subsequences used
    t_index = 0
    t_len = len(target)

    # Pre-check: if any character in target is not in source, return -1
    if set(target) - set(source):
        return -1

    # While there are still characters in target to match
    while t_index < t_len:
        s_index = 0 # Pointer for source
        progress = False # Flag to check if we made progress this pass

        # Iterate through source and try to match target characters
        while s_index < len(source) and t_index < t_len:
            if source[s_index] == target[t_index]:
                t_index += 1 # Move to next character in target
                progress = True
            s_index += 1

        count += 1 # One subsequence used for this pass

        # If no progress was made, then it is impossible to form the target
        if not progress:
            return -1

    return count

# Example usage:
# print(shortestWay("abc", "abcbc")) # Output: 2
```

#2563 MEDIUM

Count the Number of Fair Pairs

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers · Binary Search · Sorting

Lists: CodeSignal

Problem:

Given a 0-indexed integer array nums of size n and two integers lower and upper, return the number of fair pairs.

A pair (i, j) is fair if:

- 0 <= i < j < n, and
- lower <= nums[i] + nums[j] <= upper

Example 1:

```
Input: nums = [0,1,7,4,4,5], lower = 3, upper = 6
Output: 6
Explanation: There are 6 fair pairs: (0,3), (0,4), (0,5), (1,3), (1,4), and (1,5).
```


Example 2:

```
Input: nums = [1,7,9,2,5], lower = 11, upper = 11
Output: 1
Explanation: There is a single fair pair: (2,3).
```

Constraints:

- 1 <= nums.length <= 105
- nums.length == n
- -109 <= nums[i] <= 109
- -109 <= lower <= upper <= 109

>> Brute Force [Two Pointers] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def countFairPairs(self, nums, lower, upper):
        # Brute Force O(n^2) - nested loops instead of two pointers
        n = len(nums)
        for i in range(n):
            for j in range(i + 1, n):
                # check nums[i] and nums[j]
                pass
```

Python

Python

```python
import bisect

def countFairPairs(nums, lower, upper):
    # Sort the array to allow binary search
    nums.sort()
    fair_pairs = 0
    n = len(nums)

    for i in range(n):
        # Find lower bound index for the required complement so that nums[i] + nums[j] >= lower
        left = bisect.bisect_left(nums, lower - nums[i], i + 1, n)
        # Find upper bound index for the required complement so that nums[i] + nums[j] <= upper
        right = bisect.bisect_right(nums, upper - nums[i], i + 1, n)
        fair_pairs += (right - left)

    return fair_pairs

# Example usage:
print(countFairPairs([0,1,7,4,4,5], 3, 6)) # Expected output: 6
```

Python


```python
class Solution:
    def countFairPairs(self, nums: list[int], lower: int, upper: int) -> int:
        # nums[i] + nums[j] >= nums[j] + nums[i], so the condition that i < j
        # degrades to i != j and we can sort the array.
        nums.sort()

        def countLess(summ: int) -> int:
          res = 0
          i = 0
          j = len(nums) - 1
          while i < j:
            while i < j and nums[i] + nums[j] > summ:
              j -= 1
            res += j - i
            i += 1
          return res

        return countLess(upper) - countLess(lower - 1)
```

Python

```python
class Solution:
    def countFairPairs(self, nums: List[int], lower: int, upper: int) -> int:
        nums.sort()
        ans = 0
        for i, x in enumerate(nums):
            j = bisect_left(nums, lower - x, lo=i + 1)
            k = bisect_left(nums, upper - x + 1, lo=i + 1)
            ans += k - j
        return ans
```

Python

```python
class Solution:
    def countFairPairs(self, nums: List[int], lower: int, upper: int) -> int:
        nums.sort()
        ans = 0
        for i, x in enumerate(nums):
            j = bisect_left(nums, lower - x, lo=i + 1)
            k = bisect_left(nums, upper - x + 1, lo=i + 1)
            ans += k - j
        return ans
```

[*] Two Pointers — Pattern Solution (stored optimal)

Python


```python
import bisect

def countFairPairs(nums, lower, upper):
    # Sort the array to allow binary search
    nums.sort()
    fair_pairs = 0
    n = len(nums)

    for i in range(n):
        # Find lower bound index for the required complement so that nums[i] + nums[j] >= lower
        left = bisect.bisect_left(nums, lower - nums[i], i + 1, n)
        # Find upper bound index for the required complement so that nums[i] + nums[j] <= upper
        right = bisect.bisect_right(nums, upper - nums[i], i + 1, n)
        fair_pairs += (right - left)

    return fair_pairs

# Example usage:
print(countFairPairs([0,1,7,4,4,5], 3, 6)) # Expected output: 6
```


Quick Review — Two Pointers 19 questions · insights · complexity · solution

#88 Merge Sorted Array [Easy]

Key Insights

- Both arrays are already sorted in non-decreasing order.
- The merge should be done in-place into nums1.
- Start merging from the end to avoid overwriting elements in nums1.
- Use two pointers, one for nums1 and one for nums2, and a tail pointer for placement.

Space & Time

Time Complexity: O(m + n)

Space Complexity: O(1)

Solution

We use a two-pointer approach starting from the end of both arrays. Initialize three pointers:

- p1 pointing to the last valid element in nums1. - p2 pointing to the last element in nums2. - tail pointing to the last index of nums1.

While both pointers are valid, compare the elements pointed by p1 and p2, placing the larger element at the tail position. Decrement the corresponding pointer and the tail pointer. Finally, if any elements remain in nums2 (i.e., p2 >= 0), copy them over into nums1. This solution leverages the extra space at the end of nums1 and ensures the merge is done in-place in linear time.

#125 Valid Palindrome [Easy]

Key Insights

- Use two pointers (one starting at the beginning and one at the end) to compare characters.
- Skip characters that are not alphanumeric.
- Compare characters in a case-insensitive manner.
- Continue until the pointers meet or cross.

Space & Time

Time Complexity: O(n), where n is the length of the string, as each character is processed at most once.

Space Complexity: O(1), using only a few extra variables for pointers and temporary comparisons.

Solution

The solution employs a two-pointer approach:

- Initialize two pointers: one at the start (left) and one at the end (right) of the string. - Skip any non-alphanumeric characters using these pointers. - Convert the remaining characters at each pointer to lowercase and compare them. - If any mismatch is found, return false; otherwise, continue until the pointers cross. - Return true if all corresponding characters match. This method efficiently checks for a palindrome without needing additional space for a filtered string.

#283 Move Zeroes [Easy]

Key Insights

- Use a two-pointer technique: one pointer to iterate through the array and another pointer to track the position for the next non-zero element.
- For each non-zero element encountered, swap it with the element at the tracking pointer.
- This strategy minimizes operations by ensuring each non-zero element is moved at most once.
- After repositioning all non-zero elements, zeros will naturally occupy the remaining positions.
- The in-place requirement ensures space complexity remains O(1).

Space & Time

Time Complexity: O(n), where n is the number of elements in the array, since only one pass (or at most two simple passes) is required.

Space Complexity: O(1) extra space, as the reordering is accomplished in-place.

Solution

The solution leverages a two-pointer approach. The left pointer (j) keeps track of the index where the next non-zero element should be placed, while the right pointer (i) iterates through the array. Whenever a non-zero element is encountered, it is swapped with the element at index j, and j is then incremented. This effectively gathers all non-zero elements at the beginning while moving zeros to the end. A key optimization is to perform the swap only when necessary (i.e., when i and j are not the same) to reduce the number of write operations.

#408 Valid Word Abbreviation [Easy]

Key Insights

- Use two pointers: one for looping through the word and one for the abbreviation.
- When encountering a digit in abbr, accumulate the number and skip that many characters in the word.
- Check for invalid cases such as numbers with leading zeros or consecutive digit segments that might imply adjacent skipped substrings.
- Compare letters directly when they are not digits.

Space & Time

Time Complexity: O(n + m) where n is the length of the word and m is the length of the abbreviation.

Space Complexity: O(1) (constant extra space)

Solution

The approach uses two pointers to iterate over the word and abbreviation. For each character in the abbreviation:

- If it is a letter, compare it with the current character in the word. - If it is a digit, first ensure that it does not start with '0'. Then, convert the sequence of digits into a number which represents how many characters in the word to skip. After processing, both pointers should have reached the end of their respective strings for the abbreviation to be valid. This approach is efficient due to its linear pass with constant extra space.

#977 Squares of a Sorted Array [Easy]

Key Insights

- Squaring numbers can disrupt the original order since negative numbers become positive.
- A two-pointer approach can efficiently retrieve the largest square from either end of the array.
- By comparing the absolute values at the two pointers, the largest square is inserted from the back of the result array.
- This method achieves an O(n) time complexity without needing to sort the squared values afterward.

Space & Time

Time Complexity: O(n) - We traverse the array only once.

Space Complexity: O(n) - We use an extra array to store the output.

Solution

We use a two-pointer technique to traverse the array from both ends. Initialize pointers (left and right) at the beginning and end of the array, respectively, and an index pointer starting at the end of a new results array. At each step, compare the absolute values of the elements at the left and right pointers. The larger absolute value, when squared, is placed in the result array at the current index, and the corresponding pointer is adjusted. This continues until the left pointer exceeds the right pointer. Filling the result array from the back ensures that the array remains sorted in non-decreasing order.

#11 Container With Most Water [Medium]

Key Insights

- The container's area is determined by the distance between two lines multiplied by the shorter line's height.
- A brute-force approach checking all possible pairs would result in O(n^2) time complexity, which is inefficient for large inputs.
- A two-pointer approach allows us to solve the problem in O(n) time by initializing pointers at both ends and gradually moving them inward.
- By always moving the pointer corresponding to the shorter line, we are more likely to find a taller line that could potentially form a container with a larger capacity.

Space & Time

Time Complexity: O(n) - The algorithm makes a single pass through the array with two pointers.

Space Complexity: O(1) - Only a few extra variables are used, regardless of the input size.

Solution

The optimal approach uses a two-pointer strategy:

- Start with two pointers at each end of the array. - Calculate the area formed between the two lines at these pointers. - Update the maximum area if the calculated value is larger. - Move the pointer pointing to the shorter line inward, as this is the only possible way to potentially increase the height of the shorter line and thus the area. - Continue until the two pointers meet.

This method effectively reduces the number of comparisons and ensures that every possible pair that can yield a larger area is considered.

#15 3Sum [Medium]

Key Insights

- Sort the array to efficiently skip duplicates and facilitate the two pointers approach.
- Use a fixed pointer for the first element, and then use two additional pointers (left and right) to find pairs that sum to the negation of the fixed element.
- Skip duplicate elements to ensure unique triplets.

Space & Time

Time Complexity: $O(n^2)$ where n is the number of elements in the array.

Space Complexity: O(1) extra space (ignoring the space required for the output).

Solution

We first sort the array to bring duplicates together and allow the two-pointer technique to work efficiently. Then, iterate over the array and for each element, use two pointers (one at the beginning right after the current element and one at the end of the array) to find a pair that sums with the current element to zero. Skip duplicates for the current element and within the inner loop to avoid duplicate triplets. This ensures that each valid triplet is unique and the overall runtime remains efficient.

#16 3Sum Closest [Medium]

Key Insights

- Sort the array to make it easier to use the two pointers technique.
- Fix one element and then use two pointers (left and right) to find the best pair that, along with the fixed element, produces a sum closest to the target.
- Update the best sum whenever a closer sum is found.
- The sorted order helps eliminate unnecessary iterations and allows efficient movement of pointers.

Space & Time

Time Complexity: O(n^2)

Space Complexity: O(log n) to O(n) depending on the sorting algorithm used (typically in-place sort, so extra space might be minimal)

Solution

The solution starts by sorting the array. For each index in the array (considered as the first element of the triple), we use two pointers: one starting just after the fixed element and the other at the end of the array. We calculate the sum of the fixed element and the two pointers. If this sum is closer to the target than the current best, we update our best sum. Depending on whether the current sum is less than or greater than the target, we adjust the left or right pointer to try and get closer to the target. This two-pointer method leverages the sorted order of the array for efficient summing.

#31 Next Permutation [Medium]

Key Insights

- The problem is solved by identifying the first pair from the right where the left element is less than the right element.
- Once this pivot is found, find the rightmost element that is greater than this pivot.
- Swap these two elements.
- Reverse the subarray that comes after the pivot to ensure the next permutation is the next lexicographically larger sequence.
- If no such pivot exists, the array is in descending order and should be reversed to get the smallest permutation.

Space & Time

Time Complexity: O(n)

Space Complexity: O(1)

Solution

We start by scanning the array from the right to find the first index (i) such that nums[i] is less than nums[i+1]. This index 'i' represents the pivot point where the next permutation can be formed. If we find such an index, we then scan from the right again to locate the first number that is greater than nums[i] (let this index be j), and swap these two numbers. Finally, we reverse the subarray starting from index i+1 to the end of the array, which gives us the next permutation in lexicographical order. This approach leverages in-place modifications using basic operations like swapping and reversing, thus meeting the constant space requirement.

#75 Sort Colors [Medium]

Key Insights

- Use the Dutch National Flag algorithm to partition the array into three sections.
- Maintain three pointers: one for the next position for 0 (low), one for the next position for 2 (high), and one to traverse the array (mid).
- Swap elements appropriately to ensure that all 0's are at the beginning, all 2's are at the end, and 1's remain in the middle.

Space & Time

Time Complexity: O(n)

Space Complexity: O(1)

Solution

The solution uses a three-pointer approach (Dutch National Flag algorithm). Initialize three pointers: low, mid, and high. As you iterate through the array:

- If the element is 0, swap it with the element at the low pointer and increment both low and mid. - If the element is 1, simply move mid forward. - If the element is 2, swap it with the element at the high pointer and decrement high. This single pass method ensures that the array is sorted in-place with constant extra space. Key details include correctly handling swaps without losing any data and ensuring that the pointers are updated in the proper order.

#161 One Edit Distance [Medium]

Key Insights

- The difference in length between s and t must be at most 1.

- When lengths are equal, the only possibility is that exactly one character is different (replacement).
- When one string is longer by one character, the difference can be fixed by one insertion (or deletion in the other direction).
- Use two pointers to compare corresponding characters and carefully handle the first difference.

Space & Time

Time Complexity: O(n), where n is the length of the shorter string, as we traverse both strings at most once.

Space Complexity: O(1), using only constant extra space.

Solution

The solution uses a two-pointer approach. First, we check if the length difference is greater than 1, in which case they cannot be one edit apart. For the two primary cases:

- When strings are of equal length, iterate through both strings and count the number of differing characters. There should be exactly one difference. - When lengths differ by one, use pointers for both strings. Upon encountering a difference, increment the pointer for the longer string only and ensure that no previous difference has been encountered. If a second discrepancy is found, return false. The key is to handle the edge case where the difference is at the end of the string.

#167 Two Sum II - Input Array Is Sorted [Medium]

Key Insights

- The array is sorted in non-decreasing order, which makes the two pointers technique very effective.
- Using two pointers (one at the start and one at the end) allows checking pairs and adjusting the pointers based on the sum.
- Since the problem guarantees exactly one solution and requires constant space, using two pointers satisfies both constraints.

Space & Time

Time Complexity: O(n), where n is the number of elements in the array, because in the worst-case scenario each element is visited at most once.

Space Complexity: O(1), as no additional data structure is used that scales with the input size.

Solution

We use the two pointers approach:

- Initialize two pointers, left at the beginning (index 0) and right at the end (last index) of the array. - While left is less than right, compute the sum of the elements at the two pointers. - If the sum equals the target, return the indices as [left+1, right+1] (to adjust for the 1-indexed array). - If the sum is less than the target, move the left pointer to the right to increase the sum. - If the sum is greater than the target, move the right pointer to the left to decrease the sum. This approach leverages the sorted property and uses constant extra space.

#186 Reverse Words in a String II [Medium]

Key Insights

- Reverse the entire array first to obtain the reversed order of characters.
- Then, reverse each individual word to restore the correct letter order.
- Doing so leverages the two-pointer technique for in-place reversals.
- This method accomplishes the goal in O(n) time while using O(1) extra space.

Space & Time

Time Complexity: O(n)

Space Complexity: O(1)

Solution

The solution employs a two-step reversal process using the two-pointer technique. First, reverse the entire character array to reverse the overall order of characters. Although this reverses both the words and the letters in each word, it sets up the scenario where each word's letters are in reverse order. The next step is to iterate through

the array and locate each word (using the space character as a delimiter) and then reverse each word individually to correct its letter order. This process is done in-place without any additional data structures, conserving space.

#189 Rotate Array [Medium]

Key Insights

- The rotation amount k might be greater than the length of the array, so use k % n (where n is the array length) to compute the effective rotation.
- A smart in-place solution involves reversing sections of the array:
- Reverse the entire array.
- Reverse the first k elements.
- Reverse the remaining n-k elements.
- Alternative approaches include using extra space or performing k single-step rotations (which is less optimal).

Space & Time

Time Complexity: O(n) where n is the number of elements in the array.

Space Complexity: O(1) if done in-place.

Solution

We solve the problem using the "reverse" technique to perform the rotation in-place. First, we adjust k by taking k modulo the length of the array to handle cases where k exceeds the array length. Then, we reverse the entire array. Next, we reverse the first k elements to restore their correct order. Finally, we reverse the remaining n-k elements. This sequence yields the array rotated to the right by k positions while strictly using O(1) extra space.

#532 K-diff Pairs in an Array [Medium]

Key Insights

- For k < 0, the answer is always 0 because the absolute difference cannot be negative.
- When k is 0, you are looking for numbers that appear at least twice.
- For k > 0, it is efficient to use a hash set or a hash map to check if each number's complement (number + k) exists.
- Uniqueness is ensured by processing each number only once from the set of unique numbers.

Space & Time

Time Complexity: O(n), where n is the length of the array since we iterate over the array and then over the unique numbers.

Space Complexity: O(n), for storing the frequency count or the unique elements in a hash map or hash set.

Solution

The solution first checks if k is negative, returning 0 immediately if true. For k equals 0, we count numbers that appear more than once using a frequency map. For k greater than 0, a set of unique numbers is created and for each number, we check if (number + k) is also present in the set. Each valid case contributes to the result count, ensuring that only unique pairs are considered.

#923 3Sum With Multiplicity [Medium]

Key Insights

- Use a frequency counter for elements since arr[i] is in the range [0, 100].
- Iterate through all possible combinations of numbers x, y, and z with x <= y <= z.
- For each valid triplet (x, y, z) that sums to target, count the number of ways using combinatorial formulas:
- All distinct: count[x] * count[y] * count[z].
- Two numbers the same: use nC2 for the repeated element.
- All three the same: use nC3.
- Use modulo arithmetic (MOD = 10^9 + 7) at every step to avoid overflow.

Space & Time

Time Complexity: O(M^2), where M = 101 (the range of possible numbers), which is efficient given the constraints.

Space Complexity: O(M) for the frequency array.

Solution

The solution first counts the frequency of each number in the array. Then it iterates through all valid triplets (x, y, z) (with x <= y <= z) and checks if they sum up to the target. Depending on whether x, y, and z are distinct or have duplicates, the combination count is computed accordingly:

- If x, y, and z are all the same, compute the combination as nC3. - If only two are the same, compute as nC2 multiplied by the count of the third number. - If all are distinct, use the direct product of the counts. This covers all cases while applying the modulo operation to the result.

#986 Interval List Intersections [Medium]

Key Insights

- Use two pointers to traverse both lists simultaneously.
- Determine the overlapping interval by taking the maximum of the start points and the minimum of the end points.
- If the overlapping condition (start <= end) is met, add the intersection to the result.
- Move the pointer that has the smaller end value to explore further potential intersections efficiently.
- Achieve an overall time complexity of O(m+n), where m and n are the lengths of the two interval lists.

Space & Time

Time Complexity: O(m + n) where m and n are the lengths of firstList and secondList respectively.

Space Complexity: O(1) extra space (excluding the space used for the output).

Solution

The solution uses a two pointers approach. Initialize two indices, one for each list, and iterate through both lists. For each pair of intervals, the potential intersection is computed by:

- Calculating the maximum of the two start values. - Calculating the minimum of the two end values. If the maximum start is less than or equal to the minimum end, there is an intersection, and it is added to the result list. The pointer corresponding to the interval with the smaller end value is then incremented since that interval cannot intersect with any further intervals from the other list. This process repeats until one of the lists is fully traversed.

#1055 Shortest Way to Form String [Medium]

Key Insights

- Check if every character in target exists in source; if not, the answer is -1.
- Use a greedy two-pointer strategy: iterate over target while scanning source for matching characters.
- During each pass over source, match as many characters as possible from target.
- Count the number of passes (subsequences) required until the target is completely matched.
- An optimized approach may precompute indices for binary search; however, the basic greedy simulation is efficient for the given constraints.

Space & Time

Time Complexity: O(n * m) in the worst-case, where n = length of source and m = length of target.

Space Complexity: O(1) for the greedy simulation (O(n) if additional data structures for binary search are used).

Solution

The solution uses a greedy two-pointer approach. We simulate the process of reading the target by repeatedly scanning through the source string. Each scan through source is treated as one subsequence concatenated to form the target. A pointer in the target moves forward whenever a matching character is found in source. If a complete pass through source does not advance the target pointer, then it is impossible to form the target and we return -1. The algorithm counts the number of passes required until the target is fully matched.

#2563 Count the Number of Fair Pairs [Medium]

Key Insights

- Sorting the array simplifies searching for pairs with required sum limits.
- For each element, binary search is used to locate:
- The first index where the complement makes the sum >= lower.
- The last index where the complement makes the sum <= upper.
- This avoids the need for a nested loop through the entire array, yielding a more efficient solution.

Space & Time

Time Complexity: O(n log n), primarily due to sorting and (for each element) binary search operations.

Space Complexity: O(n) if sorting creates a copy; O(1) additional space if sorting in-place.

Solution

The solution starts by sorting the input array. For each element at index i in the sorted array, we search for indices j (where j > i) such that the sum of nums[i] and nums[j] falls between lower and upper. Two binary search operations are performed:

- To find the smallest index j where nums[i] + nums[j] >= lower. - To find the largest index j where nums[i] + nums[j] <= upper. Subtracting the two positions (and summing the counts for each i) gives the total number of fair pairs. Edge cases (like no valid pair) are automatically handled by the binary search approach.

#14 Heap — 20 questions · #14 of 21

#215 MEDIUM

Kth Largest Element in an Array

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Divide and Conquer · Sorting · Heap · Quickselect

Lists: Grind 169

Problem:

Given an integer array nums and an integer k, return the kth largest element in the array.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

Can you solve it without sorting?

Example 1:

```
Input: nums = [3,2,1,5,6,4], k = 2
Output: 5
```

Example 2:

```
Input: nums = [3,2,3,1,2,4,5,5,6], k = 4
Output: 4
```

Constraints:

- 1 <= k <= nums.length <= 105
- -104 <= nums[i] <= 104

>> Brute Force [Heap] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findKthLargest(self, nums, k):
        # Brute Force O(n log n) - sort descending, index at k-1
        nums.sort(reverse=True)
        return nums[k - 1]
```

Python

Python


```python
import heapq

def findKthLargest(nums, k):
    # Create a min-heap with the first k elements
    min_heap = nums[:k]
    heapq.heapify(min_heap) # O(k) time to build the initial heap

    # Process the remaining elements
    for num in nums[k:]:
        # If the current element is larger than the smallest in the heap, replace it
        if num > min_heap[0]:
            heapq.heapreplace(min_heap, num) # O(log k)
    # The smallest element in the heap is the kth largest overall
    return min_heap[0]

# Example usage:
nums = [3, 2, 1, 5, 6, 4]
k = 2
print(findKthLargest(nums, k)) # Expected output: 5
```

Python

```python
class Solution:
  def findKthLargest(self, nums: list[int], k: int) -> int:
    minHeap = []

    for num in nums:
      heapq.heappush(minHeap, num)
      if len(minHeap) > k:
        heapq.heappop(minHeap)

    return minHeap[0]
```

Python

```python
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        def quick_sort(l: int, r: int) -> int:
            if l == r:
                return nums[l]
            i, j = l - 1, r + 1
            x = nums[(l + r) >> 1]
            while i < j:
                while 1:
                    i += 1
                    if nums[i] >= x:
                        break
                while 1:
                    j -= 1
                    if nums[j] <= x:
                        break
                if i < j:
                    nums[i], nums[j] = nums[j], nums[i]
            if j < k:
                return quick_sort(j + 1, r)
            return quick_sort(l, j)

        n = len(nums)
        k = n - k
        return quick_sort(0, n - 1)
```


Python

```python
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        lo, hi = 0, len(nums) - 1

        while lo <= hi:
            pos = self.partition(nums, lo, hi)
            if pos == k - 1: # pos starting from 0, so -1
                return nums[pos] # partially sorted, desc
            elif pos > k - 1:
                hi = pos - 1
            else: # pos < k - 1
                lo = pos + 1

        return -1 # or raise exception

    def partition(self, nums: List[int], lo: int, hi: int) -> int:
        pivot, l, r = nums[lo], lo + 1, hi
        while l <= r:
            if nums[l] < pivot and nums[r] > pivot:
                # larger num at left of pivot, easier to count for k-th lagest
                nums[l], nums[r] = nums[r], nums[l]
                l += 1
                r -= 1
            if nums[l] >= pivot:
                l += 1

            if nums[r] <= pivot:
                r -= 1
        # use nums[l] will lead to infinate looping
        nums[lo], nums[r] = nums[r], nums[lo]
        # possible there is duplicated num, but will be covered here
        return r

#############

class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        def quick_sort(left, right, k):
            if left == right:
                return nums[left]
            i, j = left - 1, right + 1
            x = nums[(left + right) >> 1]
            while i < j:
                while 1:
                    i += 1
                    if nums[i] >= x:
                        break
                while 1:
                    j -= 1
                    if nums[j] <= x:
                        break
```



```python
                if i < j:
                    nums[i], nums[j] = nums[j], nums[i]
            if j < k:
                return quick_sort(j + 1, right, k)
            return quick_sort(left, j, k)

        n = len(nums)
        return quick_sort(0, n - 1, n - k)

###########


class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        def quick_sort(left, right, k):
            if left == right:
                return nums[left]
            i, j = left - 1, right + 1
            x = nums[(left + right) >> 1]
            while i < j:
                while 1:
                    i += 1
                    if nums[i] >= x:
                        break
                while 1:
                    j -= 1
                    if nums[j] <= x:
                        break
                if i < j:
                    nums[i], nums[j] = nums[j], nums[i]
```

[*] Heap — Pattern Solution (matched from community answers)

Python

```python
class Solution:
  def findKthLargest(self, nums: list[int], k: int) -> int:
    minHeap = []

    for num in nums:
      heapq.heappush(minHeap, num)
      if len(minHeap) > k:
        heapq.heappop(minHeap)

    return minHeap[0]
```

#253 MEDIUM

Meeting Rooms II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers · Greedy · Sorting · Heap

Lists: Grind 169 | Premium 98

Problem:

Given an array of meeting time intervals intervals where intervals[i] = [starti, endi], return the minimum number of conference rooms required.

Example 1:

```
Input: intervals = [[0,30],[5,10],[15,20]]
Output: 2
```

Example 2:

```
Input: intervals = [[7,10],[2,4]]
Output: 1
```

Constraints:

- 1 <= intervals.length <= 104
- 0 <= starti < endi <= 106

>> Brute Force [Heap] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def minMeetingRooms(self, intervals):
        # Brute Force O(n^2) - simulate: assign each meeting to first free room
        intervals.sort()
        rooms = [] # end times of each room
        for start, end in intervals:
            # find earliest-ending free room
            placed = False
            for i in range(len(rooms)):
                if rooms[i] <= start:
                    rooms[i] = end
                    placed = True
                    break
            if not placed:
                rooms.append(end)
        return len(rooms)
```

Python

Python


```python
import heapq

def minMeetingRooms(intervals):
    # Sort intervals based on start time
    intervals.sort(key=lambda x: x[0])

    # Initialize a min-heap to store end times of meetings that are currently using rooms
    min_heap = []

    # Process each meeting interval
    for interval in intervals:
        start, end = interval

        # If the heap is not empty and the current meeting starts after the earliest ended meeting
        if min_heap and start >= min_heap[0]:
            heapq.heappop(min_heap) # Reuse the room

        # Allocate the current meeting end time into the heap (new or reused room)
        heapq.heappush(min_heap, end)

    # The number of rooms is the number of simultaneous meetings
    return len(min_heap)

# Example usage:
intervals = [[0,30],[5,10],[15,20]]

print(minMeetingRooms(intervals)) # Output: 2
```

Python

```python
class Solution:
  def minMeetingRooms(self, intervals: list[list[int]]) -> int:
    minHeap = [] # Store the end times of each room.

    for start, end in sorted(intervals):
      # There's no overlap, so we can reuse the same room.
      if minHeap and start >= minHeap[0]:
        heapq.heappop(minHeap)
      heapq.heappush(minHeap, end)

    return len(minHeap)
```

Python

```python
class Solution:
    def minMeetingRooms(self, intervals: List[List[int]]) -> int:
        m = max(e[1] for e in intervals)
        d = [0] * (m + 1)
        for l, r in intervals:
            d[l] += 1
            d[r] -= 1
        ans = s = 0
        for v in d:
            s += v
            ans = max(ans, s)
        return ans
```

Python

```
'''
data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]

list(accumulate(data, operator.mul)) # running product
[3, 12, 72, 144, 144, 1296, 0, 0, 0, 0]

list(accumulate(data, max)) # running maximum
[3, 4, 6, 6, 6, 9, 9, 9, 9, 9]


>>> from itertools import accumulate
>>> data = [3, 4, 6, 2, 1, 9, 0, 7, 5, 8]
>>> accumulate(data)
<itertools.accumulate object at 0x10fd70440>
>>> list(accumulate(data))
[3, 7, 13, 15, 16, 25, 25, 32, 37, 45]
>>> max(accumulate(data))
45

https://docs.python.org/3/library/itertools.html#itertools.accumulate


# Amortize a 5% loan of 1000 with 4 annual payments of 90
cashflows = [1000, -90, -90, -90, -90]
list(accumulate(cashflows, lambda bal, pmt: bal*1.05 + pmt))

[1000, 960.0, 918.0, 873.9000000000001, 827.5950000000001]
'''
class Solution:
    def minMeetingRooms(self, intervals: List[List[int]]) -> int:
        delta = [0] * 1000010
        for start, end in intervals:
            delta[start] += 1
            delta[end] -= 1
        return max(accumulate(delta))
        # why not delta.sort()
        # because accumulate() will go by order from index 0 to index final
        # just like, from sortedcontainers import SortedDict

###########

class Solution(object):
    def minMeetingRooms(self, intervals):
        """
        :type intervals: List[Interval]
        :rtype: int
        """
        meetings = []
        for i in intervals:
          meetings.append((i.start, 1))
          meetings.append((i.end, 0))
```



```
        meetings.sort()
        ans = 0
        count = 0
        for meeting in meetings:
          if meeting[1] == 1:
            count += 1
          else:
            count -= 1
          ans = max(ans, count)
        return ans
```

[*] Heap — Pattern Solution (matched from community answers)

Python

```
import heapq

def minMeetingRooms(intervals):
    # Sort intervals based on start time
    intervals.sort(key=lambda x: x[0])

    # Initialize a min-heap to store end times of meetings that are currently using rooms
    min_heap = []

    # Process each meeting interval
    for interval in intervals:
        start, end = interval

        # If the heap is not empty and the current meeting starts after the earliest ended meeting
        if min_heap and start >= min_heap[0]:
            heapq.heappop(min_heap) # Reuse the room

        # Allocate the current meeting end time into the heap (new or reused room)
        heapq.heappush(min_heap, end)

    # The number of rooms is the number of simultaneous meetings
    return len(min_heap)

# Example usage:
intervals = [[0,30],[5,10],[15,20]]

print(minMeetingRooms(intervals)) # Output: 2
```

#347 **MEDIUM**

Top K Frequent Elements

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Divide and Conquer · Sorting · Heap · Bucket Sort · Counting · Quickselect

Lists: Denny Zhang

Problem:

Given an integer array nums and an integer k, return the k most frequent elements. You may return the answer in any order.

Example 1:

Input: nums = [1,1,1,2,2,3], k = 2

Output: [1,2]

Example 2:

Input: nums = [1], k = 1

Output: [1]

Example 3:

Input: nums = [1,2,1,2,1,2,3,1,3,2], k = 2

Output: [1,2]

Constraints:

- 1 <= nums.length <= 105
- -104 <= nums[i] <= 104
- k is in the range [1, the number of unique elements in the array].
- It is guaranteed that the answer is unique.

Follow up: Your algorithm's time complexity must be better than O(n log n), where n is the array's size.

>> Brute Force [Heap] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def topKFrequent(self, nums, k):
        # Brute Force O(n log n) - count then sort by frequency
        count = {}
        for num in nums:
            count[num] = count.get(num, 0) + 1
        return sorted(count, key=lambda x: -count[x])[:k]
```

Python

Python


```
# Python solution using bucket sort

def topKFrequent(nums, k):
    # Count the frequency of each element in nums
    frequency = {}
    for num in nums:
        frequency[num] = frequency.get(num, 0) + 1

    # Create buckets where index represents frequency
    n = len(nums)
    buckets = [[] for _ in range(n + 1)]
    for num, count in frequency.items():
        buckets[count].append(num)

    # Gather the k most frequent elements
    result = []
    # Iterate over buckets in reverse order (highest frequency first)
    for freq in range(n, 0, -1):
        for num in buckets[freq]:
            result.append(num)
            if len(result) == k:
                return result

# Example usage:
# print(topKFrequent([1,1,1,2,2,3], 2)) -> Output: [1, 2]
```

Python

```
class Solution:
  def topKFrequent(self, nums: list[int], k: int) -> list[int]:
    ans = []
    bucket = [[] for _ in range(len(nums) + 1)]

    for num, freq in collections.Counter(nums).items():
      bucket[freq].append(num)

    for b in reversed(bucket):
      ans += b
      if len(ans) == k:
        return ans
```

Python

```
class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        cnt = Counter(nums)
        return [x for x, _ in cnt.most_common(k)]
```

Python


```
'''
>>> nums = ["a", "a", "b", "c", "c"]

>>> cnt = Counter(nums)
>>> cnt
Counter({'a': 2, 'c': 2, 'b': 1})

# default to keys
>>> sorted_freqs = sorted(cnt, key=lambda x: (-cnt[x], x))
>>> sorted_freqs
['a', 'c', 'b']
'''
from collections import Counter

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # Count the frequency of each element in the list
        cnt = Counter(nums)

        # Sort the elements by frequency in decreasing order
        # diff from below solution: sorted(cnt), not sorted(cnt.items())
        # more in https://leetcode.ca/2017-10-22-692-Top-K-Frequent-Words/
        sorted_freqs = sorted(cnt, key=lambda x: (-cnt[x], x))

        return sorted_freqs[:k]

#############

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # Count the frequency of each element in the list
        freqs = Counter(nums)

        # Sort the elements by frequency in decreasing order
        sorted_freqs = sorted(freqs.items(), key=lambda x: x[1], reverse=True)
        # also passing O2: sorted_freqs = sorted(freqs.items(), key=lambda x: -x[1])

        # Take the top k elements
        top_k = [num for num, freq in sorted_freqs[:k]]

        return top_k

#############

'''
>>> from heapq import heappush
>>> h = []
>>> heappush(h, (3,1))
>>> heappush(h, (1,1))
>>> heappush(h, (2,1))
```



```
>>> h
[(1, 1), (3, 1), (2, 1)]

>>> from heapq import heappop
>>> heappop(h)
(1, 1)
>>> heappop(h)
(2, 1)
>>> heappop(h)
(3, 1)
'''

from collections import Counter

class Solution: # heap
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        cnt = Counter(nums)
        hp = []
        for num, freq in cnt.items():
            heappush(hp, (freq, num)) # freq first, default sort by 1st element
            if len(hp) > k:
                heappop(hp)
        return [v[1] for v in hp]

#############


from typing import List
import random
# O(n log(n)) in the average case
# O(n^2) worst case
```

[*] Heap — Pattern Solution (matched from community answers)

Python

```
'''
>>> nums = ["a", "a", "b", "c", "c"]

>>> cnt = Counter(nums)
>>> cnt
Counter({'a': 2, 'c': 2, 'b': 1})

# default to keys
>>> sorted_freqs = sorted(cnt, key=lambda x: (-cnt[x], x))
>>> sorted_freqs
['a', 'c', 'b']
'''

from collections import Counter

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # Count the frequency of each element in the list
        cnt = Counter(nums)

        # Sort the elements by frequency in decreasing order
        # diff from below solution: sorted(cnt), not sorted(cnt.items())
        # more in https://leetcode.ca/2017-10-22-692-Top-K-Frequent-Words/
        sorted_freqs = sorted(cnt, key=lambda x: (-cnt[x], x))

        return sorted_freqs[:k]
```

```
#############

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # Count the frequency of each element in the list
        freqs = Counter(nums)

        # Sort the elements by frequency in decreasing order
        sorted_freqs = sorted(freqs.items(), key=lambda x: x[1], reverse=True)
        # also passing all sorted_freqs = sorted(freqs.items(), key=lambda x: -x[1])

        # Take the top k elements
        top_k = [num for num, freq in sorted_freqs[:k]]

        return top_k

#############

'''
>>> from heapq import heappush
>>> h = []
>>> heappush(h, (3,1))
>>> heappush(h, (1,1))
>>> heappush(h, (2,1))
```



```
>>> h
[(1, 1), (3, 1), (2, 1)]

>>> from heapq import heappop
>>> heappop(h)
(1, 1)
>>> heappop(h)
(2, 1)
>>> heappop(h)
(3, 1)
'''

from collections import Counter

class Solution: # heap
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        cnt = Counter(nums)
        hp = []
        for num, freq in cnt.items():
            heappush(hp, (freq, num)) # freq first, default sort by 1st element
            if len(hp) > k:
                heappop(hp)
        return [v[1] for v in hp]

#############

from typing import List
import random
# O(n log(n)) in the average case
# O(n^2) worst case
```

#505 MEDIUM

The Maze II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · DFS · BFS · Graph · Matrix · Heap · Shortest Path

Lists: Premium 98

Problem:

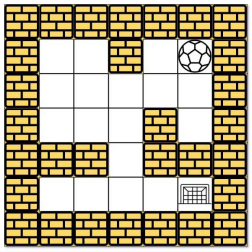
There is a ball in a maze with empty spaces (represented as 0) and walls (represented as 1). The ball can go through the empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction.

Given the m x n maze, the ball's start position and the destination, where start = [startrow, startcol] and destination = [destinationrow, destinationcol], return the shortest distance for the ball to stop at the destination. If the ball cannot stop at destination, return −1.

The distance is the number of empty spaces traveled by the ball from the start position (excluded) to the destination (included).

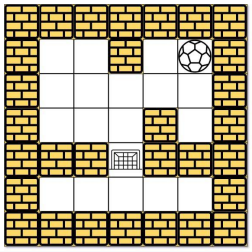
You may assume that the borders of the maze are all walls (see examples).

Example 1:



```
Input: maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4],
destination = [4,4]
Output: 12
Explanation: One possible way is : left -> down -> left -> down -> right -> down -> right.
The length of the path is 1 + 1 + 3 + 1 + 2 + 2 + 2 = 12.
```

Example 2:



```
Input: maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4],
destination = [3,2]
Output: -1
Explanation: There is no way for the ball to stop at the destination. Notice that you can
pass through the destination but you cannot stop there.
```

Example 3:

```
Input: maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]], start = [4,3],
destination = [0,1]
Output: -1
```

Constraints:

- m == maze.length
- n == maze[i].length
- 1 <= m, n <= 100
- maze[i][j] is 0 or 1.
- start.length == 2
- destination.length == 2
- 0 <= startrow, destinationrow < m
- 0 <= startcol, destinationcol < n
- Both the ball and the destination exist in an empty space, and they will not be in the same position initially.
- The maze contains at least 2 empty spaces.

>> Brute Force [Heap] Interview Prep

Python


```
# Brute Force Approach
class Solution:
    def shortestDistance(self, maze, start, destination):
        # Brute Force O(k *log n) - sort repeatedly instead of heap
        data = list(maze)
        result = []
        for _ in range(k):
            data.sort()
            result.append(data.pop(0))
        return result
```

Python

Python

```
import heapq

def shortestDistance(maze, start, destination):
    m, n = len(maze), len(maze[0])
    # Initialize distance matrix with infinity
    dist = [[float('inf')] * n for _ in range(m)]
    dist[start[0]][start[1]] = 0
    # Priority queue (min-heap) storing tuples (distance, row, col)
    heap = [(0, start[0], start[1])]
    # Directions: up, down, left, right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    while heap:
        d, row, col = heapq.heappop(heap)
        # If we have reached the destination, return the corresponding distance
        if [row, col] == destination:
            return d
        # Skip if we already found a better path to (row, col)
        if d > dist[row][col]:
            continue
        for dr, dc in directions:
            next_row, next_col, count = row, col, 0
            # Roll the ball until it hits a wall
            while 0 <= next_row + dr < m and 0 <= next_col + dc < n and maze[next_row + dr][next_col + dc]
== 0:
                next_row += dr
                next_col += dc
                count += 1
            # Check if the new distance is shorter
            if d + count < dist[next_row][next_col]:
                dist[next_row][next_col] = d + count
                heapq.heappush(heap, (dist[next_row][next_col], next_row, next_col))
    return -1
```

Python


```
class Solution:
    def shortestDistance(
        self, maze: List[List[int]], start: List[int], destination: List[int]
    ) -> int:
        m, n = len(maze), len(maze[0])
        dirs = (-1, 0, 1, 0, -1)
        si, sj = start
        di, dj = destination
        q = deque([(si, sj)])
        dist = [[inf] * n for _ in range(m)]
        dist[si][sj] = 0
        while q:
            i, j = q.popleft()
            for a, b in pairwise(dirs):
                x, y, k = i, j, dist[i][j]
                while 0 <= x + a < m and 0 <= y + b < n and maze[x + a][y + b] == 0:
                    x, y, k = x + a, y + b, k + 1
                if k < dist[x][y]:
                    dist[x][y] = k
                    q.append((x, y))
        return -1 if dist[di][dj] == inf else dist[di][dj]
```

Python

```
class Solution:
    def shortestDistance(
        self, maze: List[List[int]], start: List[int], destination: List[int]
    ) -> int:
        m, n = len(maze), len(maze[0])
        dirs = (-1, 0, 1, 0, -1)
        si, sj = start
        di, dj = destination
        q = deque([(si, sj)])
        dist = [[inf] * n for _ in range(m)]
        dist[si][sj] = 0
        while q:
            i, j = q.popleft()
            for a, b in pairwise(dirs):
                x, y, k = i, j, dist[i][j]
                while 0 <= x + a < m and 0 <= y + b < n and maze[x + a][y + b] == 0:
                    x, y, k = x + a, y + b, k + 1
                if k < dist[x][y]:
                    dist[x][y] = k
                    q.append((x, y))
        return -1 if dist[di][dj] == inf else dist[di][dj]
```

[*] Heap — Pattern Solution (matched from community answers)

Python

```python
import heapq

def shortestDistance(maze, start, destination):
    m, n = len(maze), len(maze[0])
    # Initialize distance matrix with infinity
    dist = [[float('inf')] * n for _ in range(m)]
    dist[start[0]][start[1]] = 0
    # Priority queue (min-heap) storing tuples (distance, row, col)
    heap = [(0, start[0], start[1])]
    # Directions: up, down, left, right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    while heap:
        d, row, col = heapq.heappop(heap)
        # If we have reached the destination, return the corresponding distance
        if [row, col] == destination:
            return d
        # Skip if we already found a better path to (row, col)
        if d > dist[row][col]:
            continue
        for dr, dc in directions:
            next_row, next_col, count = row, col, 0
            # Roll the ball until it hits a wall
            while 0 <= next_row + dr < m and 0 <= next_col + dc < n and maze[next_row + dr][next_col + dc]
== 0:
                next_row += dr
                next_col += dc
                count += 1
            # Check if the new distance is shorter
            if d + count < dist[next_row][next_col]:
                dist[next_row][next_col] = d + count
                heapq.heappush(heap, (dist[next_row][next_col], next_row, next_col))
    return -1
```

#621 MEDIUM

Task Scheduler

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Greedy · Heap · Counting

Lists: Grind 169

Problem:

You are given an array of CPU tasks, each labeled with a letter from A to Z, and a number n. Each CPU interval can be idle or allow the completion of one task. Tasks can be completed in any order, but there's a constraint: there has to be a gap of at least n intervals between two tasks with the same label.

Return the minimum number of CPU intervals required to complete all tasks.

Example 1:

Input: tasks = ["A","A","A","B","B","B"], n = 2

Output: 8

Explanation: A possible sequence is: A -> B -> idle -> A -> B -> idle -> A -> B.

After completing task A, you must wait two intervals before doing A again. The same applies to task B. In the 3rd interval, neither A nor B can be done, so you idle. By the 4th interval, you can do A again as 2 intervals have passed.

Example 2:

Input: tasks = ["A","C","A","B","D","B"], n = 1

Output: 6

Explanation: A possible sequence is: A -> B -> C -> D -> A -> B.

With a cooling interval of 1, you can repeat a task after just one other task.

Example 3:

Input: tasks = ["A","A","A","B","B","B"], n = 3

Output: 10

Explanation: A possible sequence is: A -> B -> idle -> idle -> A -> B -> idle -> idle -> A -> B.

There are only two types of tasks, A and B, which need to be separated by 3 intervals. This leads to idling twice between repetitions of these tasks.

Constraints:

- 1 <= tasks.length <= 104
- tasks[i] is an uppercase English letter.
- 0 <= n <= 100

>> Brute Force [Heap] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def leastInterval(self, tasks, n):
        # Brute Force O(n · intervals) - simulate the CPU schedule tick by tick
        from collections import Counter
        counts = Counter(tasks)
        time = 0
        while counts:
            # Each cycle: pick up to (n+1) most-frequent tasks
            cycle = sorted(counts.keys(), key=lambda t: -counts[t])[:n+1]
            for t in cycle:
                counts[t] -= 1
                if counts[t] == 0: del counts[t]
                time += 1
            if counts: # Idle the rest of the cycle
                time += max(0, n + 1 - len(cycle))
        return time
```

Python

Python


```python
# Python solution for Task Scheduler
from collections import Counter

def leastInterval(tasks, n):
    # Count the frequency of each task
    task_counts = Counter(tasks)
    # Find the maximum frequency among tasks
    max_freq = max(task_counts.values())
    # Count how many tasks have the maximum frequency
    max_count = sum(1 for count in task_counts.values() if count == max_freq)

    # Calculate the minimum intervals required based on the most frequent tasks
    part_count = max_freq - 1 # Number of parts between the most frequent task
    part_length = n - (max_count - 1) # Effective empty slots per part after placing max frequency tasks
    empty_slots = part_count * part_length # Total empty slots to fill with idle or other tasks
    available_tasks = len(tasks) - max_freq + max_count # Remaining tasks to fill the empty slots
    idles = max(0, empty_slots - available_tasks) # Remaining idle slots if available tasks are not enough

    # The total intervals is the sum of all tasks and idle slots inserted
    return len(tasks) + idles

# Example usage:
if __name__ == "__main__":
    tasks1 = ["A","A","A","B","B","B"]
    n1 = 2

    print(leastInterval(tasks1, n1)) # Output: 8
```

Python

```python
class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        count = collections.Counter(tasks)
        maxFreq = max(count.values())
        # Put the most frequent task in the slot first.
        maxFreqTaskOccupy = (maxFreq - 1) * (n + 1)
        # Get the number of tasks with same frequency as maxFreq, we'll append them after the
        # `maxFreqTaskOccupy`.
        nMaxFreq = sum(value == maxFreq for value in count.values())
        # max(
        # the most frequent task is frequent enough to force some idle slots,
        # the most frequent task is not frequent enough to force idle slots
        # )
        return max(maxFreqTaskOccupy + nMaxFreq, len(tasks))
```

Python

```python
class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        cnt = Counter(tasks)
        x = max(cnt.values())
        s = sum(v == x for v in cnt.values())
        return max(len(tasks), (x - 1) * (n + 1) + s)
```

Python


```python
class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        cnt = Counter(tasks)
        x = max(cnt.values())
        s = sum(v == x for v in cnt.values())
        return max(len(tasks), (x - 1) * (n + 1) + s)
```

[*] Heap — Pattern Solution (stored optimal)

Python

```python
# Python solution for Task Scheduler
from collections import Counter

def leastInterval(tasks, n):
    # Count the frequency of each task
    task_counts = Counter(tasks)
    # Find the maximum frequency among tasks
    max_freq = max(task_counts.values())
    # Count how many tasks have the maximum frequency
    max_count = sum(1 for count in task_counts.values() if count == max_freq)

    # Calculate the minimum intervals required based on the most frequent tasks
    part_count = max_freq - 1 # Number of parts between the most frequent task
    part_length = n - (max_count - 1) # Effective empty slots per part after placing max frequency tasks
    empty_slots = part_count * part_length # Total empty slots to fill with idle or other tasks
    available_tasks = len(tasks) - max_freq + max_count # Remaining tasks to fill the empty slots
    idles = max(0, empty_slots - available_tasks) # Remaining idle slots if available tasks are not enough

    # The total intervals is the sum of all tasks and idle slots inserted
    return len(tasks) + idles

# Example usage:
if __name__ == "__main__":
    tasks1 = ["A","A","A","B","B","B"]
    n1 = 2

    print(leastInterval(tasks1, n1)) # Output: 8
```

#658 MEDIUM

Find K Closest Elements

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search · Sorting · Heap · Two Pointers

Lists: Grind 169

Problem:

Given a sorted integer array arr, two integers k and x, return the k closest integers to x in the array. The result should also be sorted in ascending order.

An integer a is closer to x than an integer b if:

- |a − x| < |b − x|, or
- |a − x| == |b − x| and a < b

Example 1:

Input: arr = [1,2,3,4,5], k = 4, x = 3

Output: [1,2,3,4]

Example 2:

Input: arr = [1,1,2,3,4,5], k = 4, x = −1

Output: [1,1,2,3]

Constraints:

- 1 <= k <= arr.length
- 1 <= arr.length <= 104
- arr is sorted in ascending order.
- −104 <= arr[i], x <= 104

>> Brute Force [Heap] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findClosestElements(self, arr, k, x):
        # Brute force O(n log n) - sort repeatedly instead of heap
        data = list(arr)
        result = []
        for _ in range(k):
            data.sort()
            result.append(data.pop(0))
        return result
```

Python

Python

```python
# Function to find k closest elements to x in a sorted array
def findClosestElements(arr, k, x):
    n = len(arr)
    # Initialize the binary search bounds for the starting index of the window
    left, right = 0, n - k
    # Binary search to find the best starting index
    while left < right:
        mid = left + (right - left) // 2
        # Compare the distance from x to arr[mid] and arr[mid+k]
        # If arr[mid] is farther from x than arr[mid+k], move window to the right
        if x - arr[mid] > arr[mid + k] - x:
            left = mid + 1
        else:
            # Otherwise, continue searching towards the left
            right = mid
    # Return the k elements starting from index `left`
    return arr[left:left + k]

# Example usage:
# arr = [1,2,3,4,5], k = 4, x = 3
# print(findClosestElements(arr, 4, 3)) # Output: [1,2,3,4]
```

Python


```python
class Solution:
    def findClosestElements(self, arr: list[int], k: int, x: int) -> list[int]:
        l = 0
        r = len(arr) - k

        while l < r:
            m = (l + r) // 2
            if x - arr[m] <= arr[m + k] - x:
                r = m
            else:
                l = m + 1

        return arr[l:l + k]
```

Python

```python
class Solution:
    def findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:
        arr.sort(key=lambda v: abs(v - x))
        return sorted(arr[:k])
```

Python

```python
class Solution:
    def findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:
        left, right = 0, len(arr) - k
        while left < right:
            mid = (left + right) >> 1
            if x - arr[mid] <= arr[mid + k] - x:
                right = mid
            else:
                left = mid + 1
        return arr[left : left + k]
```

[*] Heap — Pattern Solution (stored optimal)

Python

```python
# Function to find k closest elements to x in a sorted array
def findClosestElements(arr, k, x):
    n = len(arr)
    # Initialize the binary search bounds for the starting index of the window
    left, right = 0, n - k
    # Binary search to find the best starting index
    while left < right:
        mid = left + (right - left) // 2
        # Compare the distance from x to arr[mid] and arr[mid+k]
        # If arr[mid] is farther from x than arr[mid+k], move window to the right
        if x - arr[mid] > arr[mid + k] - x:
            left = mid + 1
        else:
            # Otherwise, continue searching towards the left
            right = mid
    # Return the k elements starting from index `left`
    return arr[left:left + k]

# Example usage:
# arr = [1,2,3,4,5], k = 4, x = 3
# print(findClosestElements(arr, 4, 3)) # Output: [1,2,3,4]
```

#787 MEDIUM

Cheapest Flights Within K Stops

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Dynamic Programming · DFS · BFS · Graph · Heap

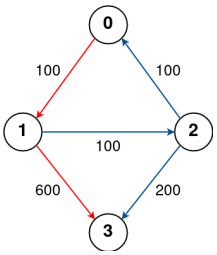
Lists: Grind 169

Problem:

There are n cities connected by some number of flights. You are given an array flights where flights[i] = [fromi, toi, pricei] indicates that there is a flight from city fromi to city toi with cost pricei.

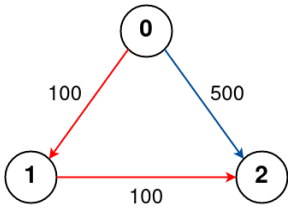
You are also given three integers src, dst, and k, return the cheapest price from src to dst with at most k stops. If there is no such route, return -1.

Example 1:



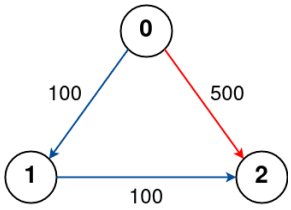

```
Input: n = 4, flights = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]], src = 0, dst =
3, k = 1
Output: 700
Explanation:
The graph is shown above.
The optimal path with at most 1 stop from city 0 to 3 is marked in red and has cost 100 + 600
= 700.
Note that the path through cities [0,1,2,3] is cheaper but is invalid because it uses 2
stops.
```

Example 2:




```
Input: n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 1
Output: 200
Explanation:
The graph is shown above.
The optimal path with at most 1 stop from city 0 to 2 is marked in red and has cost 100 + 100
= 200.
```

Example 3:




```
Input: n = 3, flights = [[0,1,100],[1,2,100],[0,2,500]], src = 0, dst = 2, k = 0
Output: 500
Explanation:
The graph is shown above.
The optimal path with no stops from city 0 to 2 is marked in red and has cost 500.
```


Constraints:

- 2 <= n <= 100
- 0 <= flights.length <= (n * (n - 1) / 2)
- flights[i].length == 3
- 0 <= fromi, toi < n
- fromi != toi
- 1 <= pricei <= 104
- There will not be any multiple flights between two cities.
- 0 <= src, dst, k < n
- src != dst

>> Brute Force [Heap] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findCheapestPrice(self, n, flights, src, dst, k):
        # Brute Force O(V^k*E) - BFS trying all paths with at most k stops
        from collections import defaultdict
        graph = defaultdict(list)
        for u,v,w in flights:
            graph[u].append((v,w))
        self.res = float('inf')
        def dfs(node, stops, cost):
            if node == dst:
                self.res = min(self.res, cost); return
            if stops > k: return
            for nei, price in graph[node]:
                if cost + price < self.res: # prune only on cost
                    dfs(nei, stops+1, cost+price)
        dfs(src, 0, 0)
        return self.res if self.res < float('inf') else -1
```

Python

Python


```python
import heapq

def findCheapestPrice(n, flights, src, dst, k):
    # Build graph: each city maps to a list of (neighboring city, flight price)
    graph = {i: [] for i in range(n)}
    for from_city, to_city, price in flights:
        graph[from_city].append((to_city, price))

    # Priority queue: (total_cost, current_city, stops so far)
    heap = [(0, src, 0)]

    # Dictionary to store the best cost to reach a city with a given stop count
    best = {}

    while heap:
        cost, city, stops = heapq.heappop(heap)
        # Return cost if destination is reached
        if city == dst:
            return cost

        # If the current number of stops exceeds k, skip further processing
        if stops > k:
            continue

        # Expand the neighboring flights

        for neighbor, price in graph[city]:
            new_cost = cost + price
            # Only proceed if this route is cheaper or not yet explored for this many stops
            if (neighbor, stops + 1) not in best or new_cost < best[(neighbor, stops + 1)]:
                best[(neighbor, stops + 1)] = new_cost
                heapq.heappush(heap, (new_cost, neighbor, stops + 1))
    return -1

# Example usage:
print(findCheapestPrice(4, [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]], 0, 3, 1))
```

Python


```python
class Solution:
    def findCheapestPrice(
        self,
        n: int,
        flights: list[list[int]],
        src: int,
        dst: int,
        k: int,
    ) -> int:
        graph = [[] for _ in range(n)]

        for u, v, w in flights:
            graph[u].append((v, w))

        return self._dijkstra(graph, src, dst, k)

    def _dijkstra(
        self,
        graph: list[list[tuple[int, int]]],
        src: int,
        dst: int,
        k: int,
    ) -> int:
        dist = [[math.inf] * (k + 2) for _ in range(len(graph))]

        dist[src][k + 1] = 0
        minHeap = [(dist[src][k + 1], src, k + 1)] # (d, u, stops)

        while minHeap:
            d, u, stops = heapq.heappop(minHeap)
            if u == dst:
                return d
            if stops == 0 or d > dist[u][stops]:
                continue
            for v, w in graph[u]:
                if d + w < dist[v][stops - 1]:
                    dist[v][stops - 1] = d + w
                    heapq.heappush(minHeap, (dist[v][stops - 1], v, stops - 1))

        return -1
```

Python

```python
class Solution:
    def findCheapestPrice(
        self, n: int, flights: List[List[int]], src: int, dst: int, k: int
    ) -> int:
        INF = 0x3F3F3F3F
        dist = [INF] * n
        dist[src] = 0
        for _ in range(k + 1):
            backup = dist.copy()
            for f, t, p in flights:
                dist[t] = min(dist[t], backup[f] + p)
        return -1 if dist[dst] == INF else dist[dst]
```

Python


```python
class Solution:
    def findCheapestPrice(
        self, n: int, flights: List[List[int]], src: int, dst: int, k: int
    ) -> int:
        INF = 0x3F3F3F3F
        dist = [INF] * n
        dist[src] = 0
        for _ in range(k + 1):
            backup = dist.copy()
            for f, t, p in flights:
                dist[t] = min(dist[t], backup[f] + p)
        return -1 if dist[dst] == INF else dist[dst]
```

[*] Heap — Pattern Solution (matched from community answers)

Python

```python
import heapq

def findCheapestPrice(n, flights, src, dst, k):
    # Build graph: each city maps to a list of (neighboring city, flight price)
    graph = {i: [] for i in range(n)}
    for from_city, to_city, price in flights:
        graph[from_city].append((to_city, price))

    # Priority queue: (total_cost, current_city, stops so far)
    heap = [(0, src, 0)]

    # Dictionary to store the best cost to reach a city with a given stop count
    best = {}

    while heap:
        cost, city, stops = heapq.heappop(heap)
        # Return cost if destination is reached
        if city == dst:
            return cost

        # If the current number of stops exceeds k, skip further processing
        if stops > k:
            continue

        # Expand the neighboring flights

        for neighbor, price in graph[city]:
            new_cost = cost + price
            # Only proceed if this route is cheaper or not yet explored for this many stops
            if (neighbor, stops + 1) not in best or new_cost < best[(neighbor, stops + 1)]:
                best[(neighbor, stops + 1)] = new_cost
                heapq.heappush(heap, (new_cost, neighbor, stops + 1))
    return -1

# Example usage:
print(findCheapestPrice(4, [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]], 0, 3, 1))
```

#973 MEDIUM

K Closest Points to Origin

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Math · Divide and Conquer · Sorting · Heap

Lists: Grind 169

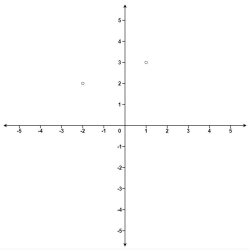
Problem:

Given an array of points where points[i] = [xi, yi] represents a point on the X–Y plane and an integer k, return the k closest points to the origin (0, 0).

The distance between two points on the X–Y plane is the Euclidean distance (i.e., √(x1 − x2)2 + (y1 − y2)2).

You may return the answer in any order. The answer is guaranteed to be unique (except for the order that it is in).

Example 1:



```
Input: points = [[1,3],[-2,2]], k = 1
Output: [[-2,2]]
Explanation:
The distance between (1, 3) and the origin is sqrt(10).
The distance between (-2, 2) and the origin is sqrt(8).
Since sqrt(8) < sqrt(10), (-2, 2) is closer to the origin.
We only want the closest k = 1 points from the origin, so the answer is just [[-2,2]].
```

Example 2:

```
Input: points = [[3,3],[5,-1],[-2,4]], k = 2
Output: [[3,3],[-2,4]]
Explanation: The answer [[-2,4],[3,3]] would also be accepted.
```

Constraints:

- 1 <= k <= points.length <= 104
- -104 <= xi, yi <= 104

>> Brute Force [Heap] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def kClosest(self, points, k):
        # Brute Force O(n log n) - sort all points by Euclidean distance
        points.sort(key=lambda p: p[0]**2 + p[1]**2)
        return points[:k]
```

Python

Python

```python
# Python Solution
def kClosest(points, k):
    # Sort the points based on their squared Euclidean distance from the origin.
    points.sort(key=lambda point: point[0]**2 + point[1]**2)
    # Return the first k points from the sorted list.
    return points[:k]

# Example usage:
# points = [[1,3], [-2,2]]
# k = 1
# print(kClosest(points, k))
```

Python

```python
class Solution:
  def kClosest(self, points: list[list[int]], k: int) -> list[list[int]]:
    maxHeap = []

    for x, y in points:
      heapq.heappush(maxHeap, (- x * x - y * y, [x, y]))
      if len(maxHeap) > k:
        heapq.heappop(maxHeap)

    return [pair[1] for pair in maxHeap]
```

Python

```python
class Solution:
    def kClosest(self, points: List[List[int]], k: int) -> List[List[int]]:
        points.sort(key=lambda p: hypot(p[0], p[1]))
        return points[:k]
```

Python

```python
class Solution:
    def kClosest(self, points: List[List[int]], k: int) -> List[List[int]]:
        points.sort(key=lambda p: p[0] * p[0] + p[1] * p[1])
        return points[:k]
```

[*] Heap — Pattern Solution (matched from community answers)

Python


```python
class Solution:
  def kClosest(self, points: list[list[int]], k: int) -> list[list[int]]:
    maxHeap = []

    for x, y in points:
      heapq.heappush(maxHeap, (- x * x - y * y, [x, y]))
      if len(maxHeap) > k:
        heapq.heappop(maxHeap)

    return [pair[1] for pair in maxHeap]
```

#1057 MEDIUM

Campus Bikes

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Greedy · Sorting · Heap

Lists: Premium 98

Problem:

On a campus represented on the X–Y plane, there are n workers and m bikes, with n <= m.

You are given an array workers of length n where workers[i] = [xi, yi] is the position of the ith worker. You are also given an array bikes of length m where bikes[j] = [xj, yj] is the position of the jth bike. All the given positions are unique.

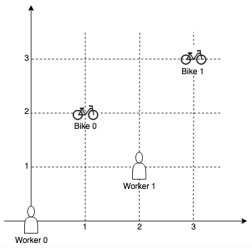
Assign a bike to each worker. Among the available bikes and workers, we choose the (workeri, bikej) pair with the shortest Manhattan distance between each other and assign the bike to that worker.

If there are multiple (workeri, bikej) pairs with the same shortest Manhattan distance, we choose the pair with the smallest worker index. If there are multiple ways to do that, we choose the pair with the smallest bike index. Repeat this process until there are no available workers.

Return an array answer of length n, where answer[i] is the index (0-indexed) of the bike that the ith worker is assigned to.

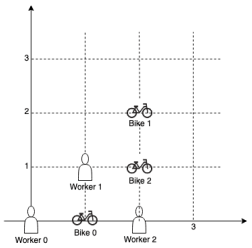
The Manhattan distance between two points p1 and p2 is Manhattan(p1, p2) = |p1.x − p2.x| + |p1.y − p2.y|.

Example 1:




```
Input: workers = [[0,0],[2,1]], bikes = [[1,2],[3,3]]
Output: [1,0]
Explanation: Worker 1 grabs Bike 0 as they are closest (without ties), and Worker 0 is
assigned Bike 1. So the output is [1, 0].
```

Example 2:




```
Input: workers = [[0,0],[1,1],[2,0]], bikes = [[1,0],[2,2],[2,1]]
Output: [0,2,1]
Explanation: Worker 0 grabs Bike 0 at first. Worker 1 and Worker 2 share the same distance to
Bike 2, thus Worker 1 is assigned to Bike 2, and Worker 2 will take Bike 1. So the output is
[0,2,1].
```

Constraints:

- n == workers.length
- m == bikes.length
- 1 <= n <= m <= 1000
- workers[i].length == bikes[j].length == 2
- 0 <= xi, yi < 1000
- 0 <= xj, yj < 1000
- All worker and bike locations are unique.

>> Brute Force [Heap] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def assign_bikes(self, workers, bikes):
        # Brute Force O(n log n) - sort repeatedly instead of heap
        data = list(workers)
        result = []
        for _ in range(k):
            data.sort()
            result.append(data.pop(0))
        return result
```

Python

Python


```python
# Python solution

def assign_bikes(workers, bikes):
    # List to hold all possible (distance, worker, bike) tuples
    pairs = []
    for worker_index, (wx, wy) in enumerate(workers):
        for bike_index, (bx, by) in enumerate(bikes):
            # Calculate Manhattan distance between worker and bike
            distance = abs(wx - bx) + abs(wy - by)
            pairs.append((distance, worker_index, bike_index))

    # Sort pairs based on distance, then worker index, then bike index
    pairs.sort()

    n = len(workers) # Number of workers
    assigned_workers = [False] * n # Track if worker is already assigned
    assigned_bikes = [False] * len(bikes) # Track if bike is already assigned
    result = [-1] * n # Result list to store bike assignment for each worker

    # Iterate over the sorted pairs and assign bikes
    for distance, worker_index, bike_index in pairs:
        if not assigned_workers[worker_index] and not assigned_bikes[bike_index]:
            result[worker_index] = bike_index # Assign bike to worker
            assigned_workers[worker_index] = True # Mark worker as assigned
            assigned_bikes[bike_index] = True # Mark bike as assigned

    return result

# Example usage:
workers = [[0,0],[2,1]]
bikes = [[1,2],[3,3]]
print(assign_bikes(workers, bikes)) # Output: [1, 0]
```

Python

```python
class Solution:
    def assignBikes(
        self,
        workers: list[list[int]],
        bikes: list[list[int]],
    ) -> list[int]:
        ans = [-1] * len(workers)
        usedBikes = [False] * len(bikes)
        # buckets[k] := [(i, j), ...] where k = dist(workers[i], bikes[j])
        buckets = [[] for _ in range(2001)]

        def dist(p1: list[int], p2: list[int]) -> int:
          return abs(p1[0] - p2[0]) + abs(p1[1] - p2[1])

        for i, worker in enumerate(workers):
          for j, bike in enumerate(bikes):
            buckets[dist(worker, bike)].append((i, j))

        for k in range(2001):
          for i, j in buckets[k]:
            if ans[i] == -1 and not usedBikes[j]:
              ans[i] = j
              usedBikes[j] = True

        return ans
```

Python

```python
class Solution:
    def assignBikes(
        self, workers: List[List[int]], bikes: List[List[int]]
    ) -> List[int]:
        n, m = len(workers), len(bikes)
        arr = []
        for i, j in product(range(n), range(m)):
            dist = abs(workers[i][0] - bikes[j][0]) + abs(workers[i][1] - bikes[j][1])
            arr.append((dist, i, j))
        arr.sort()
        vis1 = [False] * n
        vis2 = [False] * m
        ans = [0] * n
        for _, i, j in arr:
            if not vis1[i] and not vis2[j]:
                vis1[i] = vis2[j] = True
                ans[i] = j
        return ans
```

Python


```python
class Solution:
    def assignBikes(
        self, workers: List[List[int]], bikes: List[List[int]]
    ) -> List[int]:
        n, m = len(workers), len(bikes)
        arr = []
        for i, j in product(range(n), range(m)):
            dist = abs(workers[i][0] - bikes[j][0]) + abs(workers[i][1] - bikes[j][1])
            arr.append((dist, i, j))
        arr.sort()
        vis1 = [False] * n
        vis2 = [False] * m
        ans = [0] * n
        for _, i, j in arr:
            if not vis1[i] and not vis2[j]:
                vis1[i] = vis2[j] = True
                ans[i] = j
        return ans
```

[*] Heap — Pattern Solution (stored optimal)

Python

```python
# Python solution

def assign_bikes(workers, bikes):
    # List to hold all possible (distance, worker, bike) tuples
    pairs = []
    for worker_index, (wx, wy) in enumerate(workers):
        for bike_index, (bx, by) in enumerate(bikes):
            # Calculate Manhattan distance between worker and bike
            distance = abs(wx - bx) + abs(wy - by)
            pairs.append((distance, worker_index, bike_index))

    # Sort pairs based on distance, then worker index, then bike index
    pairs.sort()

    n = len(workers) # Number of workers
    assigned_workers = [False] * n # Track if worker is already assigned
    assigned_bikes = [False] * len(bikes) # Track if bike is already assigned
    result = [-1] * n # Result list to store bike assignment for each worker

    # Iterate over the sorted pairs and assign bikes
    for distance, worker_index, bike_index in pairs:
        if not assigned_workers[worker_index] and not assigned_bikes[bike_index]:
            result[worker_index] = bike_index # Assign bike to worker
            assigned_workers[worker_index] = True # Mark worker as assigned
            assigned_bikes[bike_index] = True # Mark bike as assigned

    return result

# Example usage:
workers = [[0,0],[2,1]]
bikes = [[1,2],[3,3]]
print(assign_bikes(workers, bikes)) # Output: [1, 0]
```

#1167 MEDIUM

Minimum Cost to Connect Sticks

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Greedy · Heap

Lists: Premium 98

Problem:

You have some number of sticks with positive integer lengths. These lengths are given as an array sticks, where sticks[i] is the length of the ith stick.

You can connect any two sticks of lengths x and y into one stick by paying a cost of x + y. You must connect all the sticks until there is only one stick remaining.

Return the minimum cost of connecting all the given sticks into one stick in this way.

Example 1:

```
Input: sticks = [2,4,3]
Output: 14
Explanation: You start with sticks = [2,4,3].
1. Combine sticks 2 and 3 for a cost of 2 + 3 = 5. Now you have sticks = [5,4].
2. Combine sticks 5 and 4 for a cost of 5 + 4 = 9. Now you have sticks = [9].
There is only one stick left, so you are done. The total cost is 5 + 9 = 14.
```

Example 2:

```
Input: sticks = [1,8,3,5]
Output: 30
Explanation: You start with sticks = [1,8,3,5].
1. Combine sticks 1 and 3 for a cost of 1 + 3 = 4. Now you have sticks = [4,8,5].
2. Combine sticks 4 and 5 for a cost of 4 + 5 = 9. Now you have sticks = [9,8].
3. Combine sticks 9 and 8 for a cost of 9 + 8 = 17. Now you have sticks = [17].
There is only one stick left, so you are done. The total cost is 4 + 9 + 17 = 30.
```

Example 3:

```
Input: sticks = [5]
Output: 0
Explanation: There is only one stick, so you don't need to do anything. The total cost is 0.
```

Constraints:

- 1 <= sticks.length <= 104
- 1 <= sticks[i] <= 104

>> Brute Force [Heap] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def connectSticks(self, sticks):
        # Brute Force O(n log n) - sort repeatedly instead of heap
        data = list(sticks)
        result = []
        for _ in range(k):
            data.sort()
            result.append(data.pop(0))
        return result
```


Python

Python

```python
import heapq

def connectSticks(sticks):
    # Initialize the heap with the provided stick lengths
    heapq.heapify(sticks)

    # Initialize total cost to 0
    total_cost = 0

    # Continue connecting sticks until there is only one left
    while len(sticks) > 1:
        # Extract the two smallest sticks
        stick1 = heapq.heappop(sticks)
        stick2 = heapq.heappop(sticks)
        # Calculate the cost to connect these two sticks
        cost = stick1 + stick2
        # Add the cost to the total cost
        total_cost += cost
        # Push the combined stick back into the heap
        heapq.heappush(sticks, cost)

    return total_cost

# Example usage:
# print(connectSticks([2, 4, 3])) # Output: 14
```

Python

```python
class Solution:
  def connectSticks(self, sticks: list[int]) -> int:
    ans = 0
    heapq.heapify(sticks)

    while len(sticks) > 1:
      x = heapq.heappop(sticks)
      y = heapq.heappop(sticks)
      ans += x + y
      heapq.heappush(sticks, x + y)

    return ans
```

Python

```python
class Solution:
    def connectSticks(self, sticks: List[int]) -> int:
        heapify(sticks)
        ans = 0
        while len(sticks) > 1:
            z = heappop(sticks) + heappop(sticks)
            ans += z
            heappush(sticks, z)
        return ans
```

Python


```python
class Solution:
    def connectSticks(self, sticks: List[int]) -> int:
        heapify(sticks)
        ans = 0
        while len(sticks) > 1:
            z = heappop(sticks) + heappop(sticks)
            ans += z
            heappush(sticks, z)
        return ans
```

[*] Heap — Pattern Solution (matched from community answers)

Python

```python
import heapq

def connectSticks(sticks):
    # Initialize the heap with the provided stick lengths
    heapq.heapify(sticks)

    # Initialize total cost to 0
    total_cost = 0

    # Continue connecting sticks until there is only one left
    while len(sticks) > 1:
        # Extract the two smallest sticks
        stick1 = heapq.heappop(sticks)
        stick2 = heapq.heappop(sticks)
        # Calculate the cost to connect these two sticks
        cost = stick1 + stick2
        # Add the cost to the total cost
        total_cost += cost
        # Push the combined stick back into the heap
        heapq.heappush(sticks, cost)

    return total_cost

# Example usage:
# print(connectSticks([2, 4, 3])) # Output: 14
```

#1424 MEDIUM

Diagonal Traverse II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

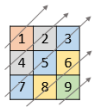
Array · Sorting · Heap

Lists: CodeSignal

Problem:

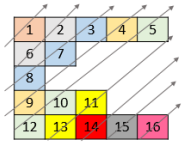
Given a 2D integer array nums, return all elements of nums in diagonal order as shown in the below images.

Example 1:




```
Input: nums = [[1,2,3],[4,5,6],[7,8,9]]
Output: [1,4,2,7,5,3,8,6,9]
```

Example 2:




```
Input: nums = [[1,2,3,4,5],[6,7],[8],[9,10,11],[12,13,14,15,16]]
Output: [1,6,2,8,7,3,9,4,12,10,5,13,11,14,15,16]
```

Constraints:

- 1 <= nums.length <= 105
- 1 <= nums[i].length <= 105
- 1 <= sum(nums[i].length) <= 105
- 1 <= nums[i][j] <= 105

>> Brute Force [Heap] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findDiagonalOrder(self, nums):
        # Brute Force O(n log n) - sort repeatedly instead of heap
        data = list(nums)
        result = []
        for _ in range(k):
            data.sort()
            result.append(data.pop(0))
        return result
```

Python

Python

```python
def findDiagonalOrder(nums):
    # Use a dictionary to collect elements by their diagonal key (i + j)
    from collections import defaultdict
    diagonals = defaultdict(list)

    # Iterate over each row and its elements
    for i, row in enumerate(nums):
        for j, value in enumerate(row):
            # Insert at the beginning to achieve the desired order for the diagonal
            diagonals[i + j].insert(0, value)

    result = []
    # Process the diagonals in order of increasing key
    for key in sorted(diagonals.keys()):
        result.extend(diagonals[key])
    return result

# Example usage:
# nums = [[1,2,3],[4,5,6],[7,8,9]]
# print(findDiagonalOrder(nums)) # Output: [1,4,2,7,5,3,8,6,9]
```

Python

```python
class Solution:
    def findDiagonalOrder(self, nums: List[List[int]]) -> List[int]:
        arr = []
        for i, row in enumerate(nums):
            for j, v in enumerate(row):
                arr.append((i + j, j, v))
        arr.sort()
        return [v[2] for v in arr]
```

Python

```python
class Solution:
    def findDiagonalOrder(self, nums: List[List[int]]) -> List[int]:
        arr = []
        for i, row in enumerate(nums):
            for j, v in enumerate(row):
                arr.append((i + j, j, v))
        arr.sort()
        return [v[2] for v in arr]
```

[*] Heap — Pattern Solution (stored optimal)

Python


```python
def findDiagonalOrder(nums):
    # Use a dictionary to collect elements by their diagonal key (i + j)
    from collections import defaultdict
    diagonals = defaultdict(list)

    # Iterate over each row and its elements
    for i, row in enumerate(nums):
        for j, value in enumerate(row):
            # Insert at the beginning to achieve the desired order for the diagonal
            diagonals[i + j].insert(0, value)

    result = []
    # Process the diagonals in order of increasing key
    for key in sorted(diagonals.keys()):
        result.extend(diagonals[key])
    return result

# Example usage:
# nums = [[1,2,3],[4,5,6],[7,8,9]]
# print(findDiagonalOrder(nums)) # Output: [1,4,2,7,5,3,8,6,9]
```

#23 HARD

Merge k Sorted Lists

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Linked List · Divide and Conquer · Heap · Merge Sort

Lists: Grind 169

Problem:

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order.

Merge all the linked-lists into one sorted linked-list and return it.

Example 1:

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
Explanation: The linked-lists are:
[
1->4->5,
1->3->4,
2->6
]
merging them into one sorted linked list:
1->1->2->3->4->4->5->6
```

Example 2:

```
Input: lists = []
Output: []
```

Example 3:

```
Input: lists = [[]]
Output: []
```

Constraints:

- k == lists.length
- 0 <= k <= 104
- 0 <= lists[i].length <= 500
- -104 <= lists[i][j] <= 104
- lists[i] is sorted in ascending order.
- The sum of lists[i].length will not exceed 104.

>> Brute Force [Heap] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def mergeKLists(self, lists):
        # Brute Force O(N log N) - collect all values, sort, rebuild
        vals = []
        for node in lists:
            while node:
                vals.append(node.val)
                node = node.next
        vals.sort()
        dummy = ListNode(0)
        curr = dummy
        for v in vals:
            curr.next = ListNode(v)
            curr = curr.next
        return dummy.next
```

Python

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

import heapq

class Solution:
    def mergeKLists(self, lists):
        # Create a min-heap
        min_heap = []
        # Heap entry: (node value, unique id, node) to handle duplicate values
        for i, node in enumerate(lists):
            if node:
                heapq.heappush(min_heap, (node.val, i, node))

        # Dummy node to form the start of merged list
        dummy = ListNode(0)
        current = dummy

        # While there are elements in the heap
        while min_heap:
            # Pop the smallest node from the heap
            val, idx, node = heapq.heappop(min_heap)
```



```python
            # Add the smallest node to the merged list
            current.next = node
            current = current.next
            # If there is a next node in the list, push it into the heap
            if node.next:
                heapq.heappush(min_heap, (node.next.val, idx, node.next))

        return dummy.next
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def mergeKLists(self, lists: List[Optional[ListNode]]) -> Optional[ListNode]:
        setattr(ListNode, "__lt__", lambda a, b: a.val < b.val)
        pq = [head for head in lists if head]
        heapify(pq)
        dummy = cur = ListNode()
        while pq:
            node = heappop(pq)
            if node.next:
                heappush(pq, node.next)
            cur.next = node
            cur = cur.next
        return dummy.next
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def mergeKLists(self, lists: List[Optional[ListNode]]) -> Optional[ListNode]:
        setattr(ListNode, "__lt__", lambda a, b: a.val < b.val)
        pq = [head for head in lists if head]
        heapify(pq)
        dummy = cur = ListNode()
        while pq:
            node = heappop(pq)
            if node.next:
                heappush(pq, node.next)
            cur.next = node
            cur = cur.next
        return dummy.next


# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
```



```python
#####


import heapq


'''
__lt__(self, other) for <
__le__(self,other) for <=
__gt__(self, other) for >
__ge__(self, other) for >=
'''

'''
# or create a wrapper class, without modifying existing node class
class NodeWrapper:
    def __init__(self, node):
        self.node = node

    def __lt__(self, other):
        return self.node.val < other.node.val

'''

# overwrite the comparison function, so the node can be comparable
# or else, error, TypeError: '<' not supported between instances of 'ListNode' and 'ListNode'

# define 'gt' will also work, so either lt or gt will do the compare job for ListNode
# ListNode.__gt__ = lambda x, y: (x.val > y.val)

ListNode.__lt__ = lambda x, y: (x.val < y.val)

class Solution:

    def mergeKLists(self, lists: List[Optional[ListNode]]) -> Optional[ListNode]:
        dummy = current = ListNode()
        heap = []
        for i, node in enumerate(lists):
            if node:
                # need to override
                # or else error: TypeError: '<' not supported between instances of 'ListNode' and
'ListNode'
                heapq.heappush(heap, node)

        while heap:
            anode = heapq.heappop(heap)

            current.next = anode
            current = current.next

            if anode.next:
                heapq.heappush(heap, anode.next)

        return dummy.next
```

[*] Heap — Pattern Solution (matched from community answers)

Python


```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

import heapq

class Solution:
    def mergeKLists(self, lists):
        # Create a min-heap
        min_heap = []
        # Heap entry: (node value, unique id, node) to handle duplicate values
        for i, node in enumerate(lists):
            if node:
                heapq.heappush(min_heap, (node.val, i, node))

        # Dummy node to form the start of merged list
        dummy = ListNode(0)
        current = dummy

        # While there are elements in the heap
        while min_heap:
            # Pop the smallest node from the heap
            val, idx, node = heapq.heappop(min_heap)

            # Add the smallest node to the merged list
            current.next = node
            current = current.next
            # If there is a next node in the list, push it into the heap
            if node.next:
                heapq.heappush(min_heap, (node.next.val, idx, node.next))

        return dummy.next
```

#218 HARD

The Skyline Problem

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Divide and Conquer · Binary Indexed Tree · Segment Tree · Line Sweep · Heap · Ordered Set

Lists: Denny Zhang

Problem:

A city's skyline is the outer contour of the silhouette formed by all the buildings in that city when viewed from a distance. Given the locations and heights of all the buildings, return the skyline formed by these buildings collectively.

The geometric information of each building is given in the array buildings where buildings[i] = [lefti, righti, height i]:

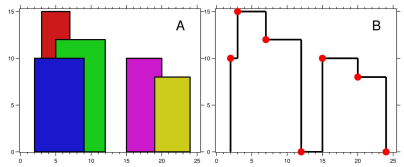
- lefti is the x coordinate of the left edge of the ith building.
- righti is the x coordinate of the right edge of the ith building.
- height i is the height of the ith building.

You may assume all buildings are perfect rectangles grounded on an absolutely flat surface at height 0.

The skyline should be represented as a list of "key points" sorted by their x-coordinate in the form [[x1,y1],[x2,y2],...]. Each key point is the left endpoint of some horizontal segment in the skyline except the last point in the list, which always has a y-coordinate 0 and is used to mark the skyline's termination where the rightmost building ends. Any ground between the leftmost and rightmost buildings should be part of the skyline's contour.

Note: There must be no consecutive horizontal lines of equal height in the output skyline. For instance, [...,[2 3],[4 5],[7 5],[11 5],[12 7],...] is not acceptable; the three lines of height 5 should be merged into one in the final output as such: [...,[2 3],[4 5],[12 7],...]

Example 1:




```
Input: buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
Output: [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]
Explanation:
Figure A shows the buildings of the input.
Figure B shows the skyline formed by those buildings. The red points in figure B represent
the key points in the output list.
```

Example 2:

```
Input: buildings = [[0,2,3],[2,5,3]]
Output: [[0,3],[5,0]]
```

Constraints:

- 1 <= buildings.length <= 104
- 0 <= lefti < righti <= 231 - 1
- 1 <= heighti <= 231 - 1
- buildings is sorted by lefti in non-decreasing order.

>> Brute Force [Heap] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def getSkyline(self, buildings):
        # Brute Force O(n log n) - sort repeatedly instead of heap
        data = list(buildings)
        result = []
        for _ in range(k):
            data.sort()
            result.append(data.pop(0))
        return result
```

Python

Python

```python
import heapq

def getSkyline(buildings):
    # List to store all events: (x, height, right)
    events = []
    # Create events for all buildings: left edge (start) and right edge (end)
    for left, right, height in buildings:
        # For the start point, height is negative for proper sorting (start events come before end events)
        events.append((left, -height, right))
        # For the end point, height is 0 to indicate removal of building from active heap
        events.append((right, 0, 0))

    # Sort events by x coordinate.
    events.sort()

    # Max heap to store active buildings (using negative heights)
    result = []
    # Initialize the heap with a dummy building of height 0 that extends infinitely
    heap = [(0, float('inf'))]

    for x, neg_height, right in events:
        # Remove buildings from the heap that have ended by current x.
        while heap and heap[0][1] <= x:
            heapq.heappop(heap)

        # If event is a start event (neg_height != 0), push it into the heap.
        if neg_height != 0:
            heapq.heappush(heap, (neg_height, right))

        # Current highest building height is at the top of the heap
        current_height = -heap[0][0]
        # If the result is empty or current height differs from last recorded height, add a new key point.
        if not result or current_height != result[-1][1]:
            result.append([x, current_height])

    return result

# Example usage:
buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
print(getSkyline(buildings)) # Expected output: [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]
```

Python


```python
class Solution:
    def getSkyline(self, buildings: list[list[int]]) -> list[list[int]]:
        n = len(buildings)
        if n == 0:
            return []
        if n == 1:
            left, right, height = buildings[0]
            return [[left, height], [right, 0]]

        left = self.getSkyline(buildings[:n // 2])
        right = self.getSkyline(buildings[n // 2:])
        return self._merge(left, right)

    def _merge(self, left: list[list[int]],
               right: list[list[int]]) -> list[list[int]]:
        ans = []
        i = 0 # left's index
        j = 0 # right's index
        leftY = 0
        rightY = 0

        while i < len(left) and j < len(right):
            # Choose the powith smaller x
            if left[i][0] < right[j][0]:
                leftY = left[i][1] # Update the ongoing `leftY`.

                self._addPoint(ans, left[i][0], max(left[i][1], rightY))
                i += 1
            else:
                rightY = right[j][1] # Update the ongoing `rightY`.
                self._addPoint(ans, right[j][0], max(right[j][1], leftY))
                j += 1

        while i < len(left):
            self._addPoint(ans, left[i][0], left[i][1])
            i += 1

        while j < len(right):
            self._addPoint(ans, right[j][0], right[j][1])
            j += 1

        return ans

    def _addPoint(self, ans: list[list[int]], x: int, y: int) -> None:
        if ans and ans[-1][0] == x:
            ans[-1][1] = y
            return
        if ans and ans[-1][1] == y:
            return
        ans.append([x, y])
```

[*] Heap — Pattern Solution (matched from community answers)

Python


```python
import heapq

def getSkyline(buildings):
    # List to store all events: (x, height, right)
    events = []
    # Create events for all buildings: left edge (start) and right edge (end)
    for left, right, height in buildings:
        # For the start point, height is negative for proper sorting (start events come before end events)
        events.append((left, -height, right))
        # For the end point, height is 0 to indicate removal of building from active heap
        events.append((right, 0, 0))

    # Sort events by x coordinate.
    events.sort()

    # Max heap to store active buildings (using negative heights)
    result = []
    # Initialize the heap with a dummy building of height 0 that extends infinitely
    heap = [(0, float('inf'))]

    for x, neg_height, right in events:
        # Remove buildings from the heap that have ended by current x.
        while heap and heap[0][1] <= x:
            heapq.heappop(heap)

        # If event is a start event (neg_height != 0), push it into the heap.
        if neg_height != 0:
            heapq.heappush(heap, (neg_height, right))

        # Current highest building height is at the top of the heap
        current_height = -heap[0][0]
        # If the result is empty or current height differs from last recorded height, add a new key point.
        if not result or current_height != result[-1][1]:
            result.append([x, current_height])

    return result

# Example usage:
buildings = [[2,9,10],[3,7,15],[5,12,12],[15,20,10],[19,24,8]]
print(getSkyline(buildings)) # Expected output: [[2,10],[3,15],[7,12],[12,0],[15,10],[20,8],[24,0]]
```

#272 HARD

Closest Binary Search Tree Value II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Two Pointers · Stack · Tree · DFS · BST · Heap

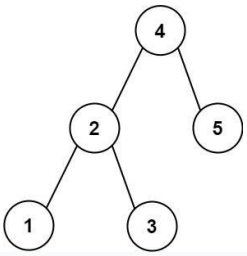
Lists: Premium 98

Problem:

Given the root of a binary search tree, a target value, and an integer k, return the k values in the BST that are closest to the target. You may return the answer in any order.

You are guaranteed to have only one unique set of k values in the BST that are closest to the target.

Example 1:




```
Input: root = [4,2,5,1,3], target = 3.714286, k = 2
Output: [4,3]
```

Example 2:

```
Input: root = [1], target = 0.000000, k = 1
Output: [1]
```

Constraints:

- The number of nodes in the tree is n.
- 1 <= k <= n <= 104.
- 0 <= Node.val <= 109
- -109 <= target <= 109

Follow up: Assume that the BST is balanced. Could you solve it in less than O(n) runtime (where n = total nodes)?

>> Brute Force [Heap] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def closestKValues(self, root: TreeNode, target: float, k: int):
        # Brute Force O(n log n) - sort repeatedly instead of heap
        data = list(root)
        result = []
        for _ in range(k):
            data.sort()
            result.append(data.pop(0))
        return result
```


Python

Python

```python
# Python solution for Closest Binary Search Tree Value II

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def closestKValues(self, root: TreeNode, target: float, k: int):
        # Helper function to initialize predecessors stack
        def initPredecessor(node):
            while node:
                # If node value is less or equal, push to preds and go right
                if node.val <= target:
                    preds.append(node)
                    node = node.right
                else:
                    node = node.left

        # Helper function to initialize successors stack
        def initSuccessor(node):
            while node:
                # If node value is greater than target, push to succs and go left

                if node.val > target:
                    succs.append(node)
                    node = node.left
                else:
                    node = node.right

        # Helper function to get next predecessor value
        def getPredecessor():
            if not preds: return None
            node = preds.pop()
            val = node.val
            # Traverse the left subtree: push nodes from left child then rightmost branch
            node = node.left
            while node:
                preds.append(node)
                node = node.right
            return val

        # Helper function to get next successor value
        def getSuccessor():
            if not succs: return None
            node = succs.pop()
            val = node.val
            # Traverse the right subtree: push nodes from right child then leftmost branch
            node = node.right
```

```
            while node:
                succs.append(node)
                node = node.left
            return val

        preds, succs = [], []
        # Initialize the stacks with correct boundary values
        initPredecessor(root)
        initSuccessor(root)

        result = []
        # If the closest value from predecessor and successor are same, skip one to avoid duplicates.
        if preds and succs and preds[-1].val == succs[-1].val:
            getPredecessor()

        # Merge k closest values
        for _ in range(k):
            if not preds:
                # Only successors left
                result.append(getSuccessor())
            elif not succs:
                # Only predecessors left
                result.append(getPredecessor())
            else:
                # Compare which candidate is closer to the target

                if abs(preds[-1].val - target) <= abs(succs[-1].val - target):
                    result.append(getPredecessor())
                else:
                    result.append(getSuccessor())
        return result
```

Python

```
class Solution:
  def closestKValues(
      self,
      root: TreeNode | None,
      target: float,
      k: int,
  ) -> list[int]:
    dq = collections.deque()

    def inorder(root: TreeNode | None) -> None:
      if not root:
        return

      inorder(root.left)
      dq.append(root.val)
      inorder(root.right)

    inorder(root)

    while len(dq) > k:
      if abs(dq[0] - target) > abs(dq[-1] - target):
        dq.popleft()
      else:
        dq.pop()
```



```
    return list(dq)
```

Python

```
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def closestKValues(self, root: TreeNode, target: float, k: int) -> List[int]:
        def dfs(root):
            if root is None:
                return
            dfs(root.left)
            if len(q) < k:
                q.append(root.val)
            else:
                if abs(root.val - target) >= abs(q[0] - target):
                    return
                q.popleft()
                q.append(root.val)
            dfs(root.right)

        q = deque()
        dfs(root)
        return list(q)
```

Python

```
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def closestKValues(self, root: TreeNode, target: float, k: int) -> List[int]:
        def dfs(root):
            if root is None:
                return
            dfs(root.left)
            if len(q) < k:
                q.append(root.val)
            else:
                if abs(root.val - target) >= abs(q[0] - target):
                    return
                q.popleft()
                q.append(root.val)
            dfs(root.right)

        q = deque()
        dfs(root)
        return list(q)
```



```
############

from collections import deque

# Iterative
class Solution:
    def closestKValues(self, root: TreeNode, target: float, k: int) -> List[int]:
        stack = []
        q = deque()

        # Iterative in-order traversal
        while stack or root:
            while root:
                stack.append(root)
                root = root.left

            root = stack.pop()

            # Process current node
            if len(q) < k:
                q.append(root.val)
            else:
                if abs(root.val - target) < abs(q[0] - target):
                    q.popleft()
                    q.append(root.val)

                else:
                    break
                    # Early stop if the current value is farther than the first value in queue

            root = root.right

        return list(q)
```

[*] Heap — Pattern Solution (stored optimal)

Python


```
# Python solution for Closest Binary Search Tree Value II

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def closestKValues(self, root: TreeNode, target: float, k: int):
        # Helper function to initialize predecessors stack
        def initPredecessor(node):
            while node:
                # If node value is less or equal, push to preds and go right
                if node.val <= target:
                    preds.append(node)
                    node = node.right
                else:
                    node = node.left

        # Helper function to initialize successors stack
        def initSuccessor(node):
            while node:
                # If node value is greater than target, push to succs and go left

                if node.val > target:
                    succs.append(node)
                    node = node.left
                else:
                    node = node.right

        # Helper function to get next predecessor value
        def getPredecessor():
            if not preds: return None
            node = preds.pop()
            val = node.val
            # Traverse the left subtree: push nodes from left child then rightmost branch
            node = node.left
            while node:
                preds.append(node)
                node = node.right
            return val

        # Helper function to get next successor value
        def getSuccessor():
            if not succs: return None
            node = succs.pop()
            val = node.val
            # Traverse the right subtree: push nodes from right child then leftmost branch
            node = node.right
```



```
            while node:
                succs.append(node)
                node = node.left
            return val

        preds, succs = [], []
        # Initialize the stacks with correct boundary values
        initPredecessor(root)
        initSuccessor(root)

        result = []
        # If the closest value from predecessor and successor are same, skip one to avoid duplicates.
        if preds and succs and preds[-1].val == succs[-1].val:
            getPredecessor()

        # Merge k closest values
        for _ in range(k):
            if not preds:
                # Only successors left
                result.append(getSuccessor())
            elif not succs:
                # Only predecessors left
                result.append(getPredecessor())
            else:
                # Compare which candidate is closer to the target

                if abs(preds[-1].val - target) <= abs(succs[-1].val - target):
                    result.append(getPredecessor())
                else:
                    result.append(getSuccessor())
        return result
```

#295 HARD

Find Median from Data Stream

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Two Pointers · Design · Sorting · Heap

Lists: Grind 169

Problem:

The median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values.

- For example, for arr = [2,3,4], the median is 3.
- For example, for arr = [2,3], the median is (2 + 3) / 2 = 2.5.

Implement the MedianFinder class:

- MedianFinder() initializes the MedianFinder object.
- void addNum(int num) adds the integer num from the data stream to the data structure.
- double findMedian() returns the median of all elements so far. Answers within 10-5 of the actual answer will be accepted.

Example 1:


```
Input
["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedian"]
[[], [1], [2], [], [3], []]
Output
[null, null, null, 1.5, null, 2.0]

Explanation
MedianFinder medianFinder = new MedianFinder();
medianFinder.addNum(1); // arr = [1]
medianFinder.addNum(2); // arr = [1, 2]
medianFinder.findMedian(); // return 1.5 (i.e., (1 + 2) / 2)
medianFinder.addNum(3); // arr[1, 2, 3]
medianFinder.findMedian(); // return 2.0
```

Constraints:

- -105 <= num <= 105
- There will be at least one element in the data structure before calling findMedian.
- At most 5 * 104 calls will be made to addNum and findMedian.

Follow up:

- If all integer numbers from the stream are in the range [0, 100], how would you optimize your solution?
- If 99% of all integer numbers from the stream are in the range [0, 100], how would you optimize your solution?

Python

Python

```
import heapq

class MedianFinder:
    def __init__(self):
        # max-heap for the lower half (store negatives to simulate max-heap)
        self.lowerHalf = []
        # min-heap for the upper half
        self.upperHalf = []

    def addNum(self, num: int) -> None:
        # Push to max-heap (simulate by pushing negative)
        heapq.heappush(self.lowerHalf, -num)
        # Ensure every element in lowerHalf is <= every element in upperHalf
        # Move maximum of lowerHalf to upperHalf
        heapq.heappush(self.upperHalf, -heapq.heappop(self.lowerHalf))

        # Balance the heaps: if upperHalf exceeds lowerHalf in size
        if len(self.upperHalf) > len(self.lowerHalf):
            heapq.heappush(self.lowerHalf, -heapq.heappop(self.upperHalf))

    def findMedian(self) -> float:
        # If both heaps are balanced, median is average of both heaps' top elements
        if len(self.lowerHalf) == len(self.upperHalf):
            return (-self.lowerHalf[0] + self.upperHalf[0]) / 2.0
        # Otherwise, median is the top element of the larger heap
```

```python
        return -self.lowerHalf[0]

# Example usage:
# medianFinder = MedianFinder()
# medianFinder.addNum(1)
# medianFinder.addNum(2)
# print(medianFinder.findMedian()) # Outputs 1.5
# medianFinder.addNum(3)
# print(medianFinder.findMedian()) # Outputs 2.0
```

Python

```python
class MedianFinder:
    def __init__(self):
      self.maxHeap = []
      self.minHeap = []

    def addNum(self, num: int) -> None:
      if not self.maxHeap or num <= -self.maxHeap[0]:
        heapq.heappush(self.maxHeap, -num)
      else:
        heapq.heappush(self.minHeap, num)

      # Balance the two heaps s.t.
      # |maxHeap| >= |minHeap| and |maxHeap| - |minHeap| <= 1.
      if len(self.maxHeap) < len(self.minHeap):
        heapq.heappush(self.maxHeap, -heapq.heappop(self.minHeap))
      elif len(self.maxHeap) - len(self.minHeap) > 1:
        heapq.heappush(self.minHeap, -heapq.heappop(self.maxHeap))

    def findMedian(self) -> float:
      if len(self.maxHeap) == len(self.minHeap):
        return (-self.maxHeap[0] + self.minHeap[0]) / 2.0
      return -self.maxHeap[0]
```

Python

```python
class MedianFinder:

    def __init__(self):
        self.minq = []
        self.maxq = []

    def addNum(self, num: int) -> None:
        heappush(self.minq, -heappushpop(self.maxq, -num))
        if len(self.minq) - len(self.maxq) > 1:
            heappush(self.maxq, -heappop(self.minq))

    def findMedian(self) -> float:
        if len(self.minq) == len(self.maxq):
            return (self.minq[0] - self.maxq[0]) / 2
        return self.minq[0]


# Your MedianFinder object will be instantiated and called as such:
# obj = MedianFinder()
# obj.addNum(num)
# param_2 = obj.findMedian()
```

Python


```python
from heapq import heappush, heappop

class MedianFinder:

    def __init__(self):
        # The smaller half of the list, max heap (invert min-heap)
        self.smallerHalf = []
        # The larger half of the list, min heap
        self.largerHalf = []

    def addNum(self, num: int) -> None:

        # Trick for smaller half, use -1*val
        # heaps in python have MAX have comparator like in Java
        heappush(self.smallerHalf, -num)
        heappush(self.largerHalf, -heappop(self.smallerHalf)) # note: not self.smallerHalf.pop()

        if len(self.smallerHalf) < len(self.largerHalf):
            heappush(self.smallerHalf, -heappop(self.largerHalf))

    def findMedian(self) -> float:
        if len(self.smallerHalf) == len(self.largerHalf):
            return (-self.smallerHalf[0] + self.largerHalf[0]) / 2.0
        else:
            return float(-self.smallerHalf[0])


# Your MedianFinder object will be instantiated and called as such:
# obj = MedianFinder()
# obj.addNum(num)
# param_2 = obj.findMedian()


# follow up
class MedianFinder:
    def __init__(self):
        self.counts = [0] * 101
        self.total = 0

    def addNum(self, num: int) -> None:
        self.counts[num] += 1
        self.total += 1

    def findMedian(self) -> float:
        if self.total % 2 == 0:
            # even number of elements
            middle1 = self.total // 2
            middle2 = middle1 + 1
            count = 0
            i1 = i2 = 0
```



```python
            for i in range(101):
                count += self.counts[i]
                if count == middle1 and i1 == 0:
                    i1 = i
                if count >= middle2:
                    i2 = i
                    break
            return (i1 + i2) / 2
        else:
            # odd number of elements
            middle = self.total // 2 + 1
            count = 0
            for i in range(101):
                count += self.counts[i]
                if count >= middle:
                    return i


###########

class MedianFinder:
    def __init__(self):
        """
        Initialize your data structure here.
        """

        self.h1 = []
        self.h2 = []

    def addNum(self, num: int) -> None:
        heappush(self.h1, num)
```

[*] Heap — Pattern Solution (matched from community answers)

Python


```python
import heapq

class MedianFinder:
    def __init__(self):
        # max-heap for the lower half (store negatives to simulate max-heap)
        self.lowerHalf = []
        # min-heap for the upper half
        self.upperHalf = []

    def addNum(self, num: int) -> None:
        # Push to max-heap (simulate by pushing negative)
        heapq.heappush(self.lowerHalf, -num)
        # Ensure every element in lowerHalf is <= every element in upperHalf
        # Move maximum of lowerHalf to upperHalf
        heapq.heappush(self.upperHalf, -heapq.heappop(self.lowerHalf))

        # Balance the heaps: if upperHalf exceeds lowerHalf in size
        if len(self.upperHalf) > len(self.lowerHalf):
            heapq.heappush(self.lowerHalf, -heapq.heappop(self.upperHalf))

    def findMedian(self) -> float:
        # If both heaps are balanced, median is average of both heaps' top elements
        if len(self.lowerHalf) == len(self.upperHalf):
            return (-self.lowerHalf[0] + self.upperHalf[0]) / 2.0
        # Otherwise, median is the top element of the larger heap

        return -self.lowerHalf[0]

# Example usage:
# medianFinder = MedianFinder()
# medianFinder.addNum(1)
# medianFinder.addNum(2)
# print(medianFinder.findMedian()) # Outputs 1.5
# medianFinder.addNum(3)
# print(medianFinder.findMedian()) # Outputs 2.0
```

#358 HARD

Rearrange String k Distance Apart

LeetDoocs · SimplyLeet · WallcCC · LeetCode · Editorial

Hash Table · String · Greedy · Sorting · Heap · Counting

Lists: Premium 98

Problem:

Given a string s and an integer k, rearrange s such that the same characters are at least distance k from each other. If it is not possible to rearrange the string, return an empty string "".

Example 1:

```
Input: s = "aabbcc", k = 3
Output: "abcabc"
Explanation: The same letters are at least a distance of 3 from each other.
```

Example 2:

```
Input: s = "aaabc", k = 3
Output: ""
```



```
Explanation: It is not possible to rearrange the string.
```

Example 3:

```
Input: s = "aaadbbcc", k = 2
Output: "abacabcd"
Explanation: The same letters are at least a distance of 2 from each other.
```

Constraints:

- 1 <= s.length <= 3 * 105
- s consists of only lowercase English letters.
- 0 <= k <= s.length

Python

Python

```python
from collections import Counter, deque
import heapq

def rearrangeString(s: str, k: int) -> str:
    if k == 0:
        return s # No rearrangement needed if k is 0

    # Count the frequency of each character
    char_count = Counter(s)

    # Build a max-heap using negative frequencies
    max_heap = [(-freq, ch) for ch, freq in char_count.items()]
    heapq.heapify(max_heap)

    # Queue to keep track of characters waiting for k distance
    wait_queue = deque()
    result = []

    # Process the heap until it's empty
    while max_heap:
        freq, ch = heapq.heappop(max_heap)
        result.append(ch)

        # Decrease frequency (remember freq is negative)
        freq += 1


        # Add current char to the wait queue
        wait_queue.append((freq, ch))

        # If we've placed k characters, we can re-insert the oldest one if needed
        if len(wait_queue) < k:
            continue

        freq_front, ch_front = wait_queue.popleft()
        if freq_front < 0:
            heapq.heappush(max_heap, (freq_front, ch_front))

    rearranged = ''.join(result)
    return rearranged if len(rearranged) == len(s) else ""
```

Python


```python
class Solution:
  def rearrangeString(self, s: str, k: int) -> str:
    n = len(s)
    ans = []
    count = collections.Counter(s)
    # valid[i] := the leftmost index i can appear
    valid = collections.Counter()

    def getBestLetter(index: int) -> str:
      """Returns the valid letter that has the most count."""
      maxCount = -1
      bestLetter = '*'

      for c in string.ascii_lowercase:
        if count[c] > 0 and count[c] > maxCount and index >= valid[c]:
          bestLetter = c
          maxCount = count[c]

      return bestLetter

    for i in range(n):
      c = getBestLetter(i)
      if c == '*':
        return ''
      ans.append(c)


      count[c] -= 1
      valid[c] = i + k

    return ''.join(ans)
```

Python

```python
class Solution:
    def rearrangeString(self, s: str, k: int) -> str:
        cnt = Counter(s)
        pq = [(-v, c) for c, v in cnt.items()]
        heapify(pq)
        q = deque()
        ans = []
        while pq:
            -v, c = heappop(pq)
            ans.append(c)
            q.append((v - 1, c))
            if len(q) >= k:
                e = q.popleft()
                if e[0]:
                    heappush(pq, e)
        return "" if len(ans) < len(s) else "".join(ans)
```

Python

```python
from heapq import heapify

class Solution:
    def rearrangeString(self, s: str, k: int) -> str:
        h = [(-v, c) for c, v in Counter(s).items()]
        heapify(h)
        q = deque()
        ans = []
        while h:
            v, c = heappop(h)
            v += -1
            ans.append(c)
            q.append((v - 1, c)) # enqueue even if 'v-1==0'
            # not '>='
            # use example k=1, then q size is 2, then q can pop one out
            if len(q) > k: # @note: this is avoid you pick up same char in the same k-segment.
                w, c = q.popleft()
                if w: # w!=0
                    heappush(h, (-w, c))
        return "" if len(ans) != len(s) else "".join(ans)

############

import collections
import heapq


class Solution:
    def rearrangeString(self, s: str, k: int) -> str:
        if k == 0:
            return s

        counts = collections.Counter(s)

        pq = []
        for ch, count in counts.items():
            heapq.heappush(pq, (-count, ch))

        sb = []
        while pq:
            tmp = []
            d = min(k, len(s))
            for i in range(d):
                if not pq:
                    return ""
                count, ch = heapq.heappop(pq)
                sb.append(ch)
                if count < -1: # pushed in is negated, so it's: if actual-positive-count > 1
                    tmp.append((count+1, ch))
                s = s[1:]

            for count, ch in tmp:
                heapq.heappush(pq, (count, ch))

        return "".join(sb)
```

[*] Heap — Pattern Solution (matched from community answers)

Python


```python
from collections import Counter, deque
import heapq

def rearrangeString(s: str, k: int) -> str:
    if k == 0:
        return s # No rearrangement needed if k is 0

    # Count the frequency of each character
    char_count = Counter(s)

    # Build a max-heap using negative frequencies
    max_heap = [(-freq, ch) for ch, freq in char_count.items()]
    heapq.heapify(max_heap)

    # Queue to keep track of characters waiting for k distance
    wait_queue = deque()
    result = []

    # Process the heap until it's empty
    while max_heap:
        freq, ch = heapq.heappop(max_heap)
        result.append(ch)

        # Decrease frequency (remember freq is negative)
        freq += 1


        # Add current char to the wait queue
        wait_queue.append((freq, ch))

        # If we've placed k characters, we can re-insert the oldest one if needed
        if len(wait_queue) < k:
            continue

        freq_front, ch_front = wait_queue.popleft()
        if freq_front < 0:
            heapq.heappush(max_heap, (freq_front, ch_front))

    rearranged = ''.join(result)
    return rearranged if len(rearranged) == len(s) else ""
```

#499 HARD

The Maze III

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · DFS · BFS · Graph · Matrix · Heap · Shortest Path

Lists: Premium 98

Problem:

There is a ball in a maze with empty spaces (represented as 0) and walls (represented as 1). The ball can go through the empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall. When the ball stops, it could choose the next direction (must be different from last chosen direction). There is also a hole in this maze. The ball will drop into the hole if it rolls onto the hole.

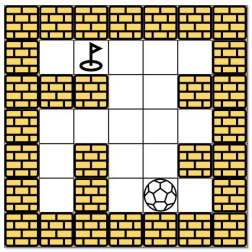
Given the m x n maze, the ball's position ball and the hole's position hole, where ball = [ballrow, ballcol] and hole = [holerow, holecol], return a string instructions of all the instructions that the ball should follow to drop in the hole with the shortest distance possible. If there are multiple valid instructions, return the lexicographically minimum one. If the ball can't drop in the hole, return "impossible".

If there is a way for the ball to drop in the hole, the answer instructions should contain the characters 'u' (i.e., up), 'd' (i.e., down), 'l' (i.e., left), and 'r' (i.e., right).

The distance is the number of empty spaces traveled by the ball from the start position (excluded) to the destination (included).

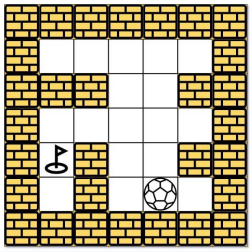
You may assume that the borders of the maze are all walls (see examples).

Example 1:



```
Input: maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]], ball = [4,3],
hole = [0,1]
Output: "lul"
Explanation: There are two shortest ways for the ball to drop into the hole.
The first way is left -> up -> left, represented by "lul".
The second way is up -> left, represented by 'ul'.
Both ways have shortest distance 6, but the first way is lexicographically smaller because
'l' < 'u'. So the output is "lul".
```

Example 2:



```
Input: maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]], ball = [4,3],
hole = [3,0]
Output: "impossible"
Explanation: The ball cannot reach the hole.
```

Example 3:

```
Input: maze = [[0,0,0,0,0,0,0],[0,0,1,0,0,1,0],[0,0,0,0,1,0,0],[0,0,0,0,0,0,0],[0,0,0,0,0,0,1]], ball =
[0,4], hole = [3,5]
Output: "dldr"
```

Constraints:

- m == maze.length
- n == maze[i].length
- 1 <= m, n <= 100
- maze[i][j] is 0 or 1.
- ball.length == 2
- hole.length == 2
- 0 <= ballrow, holerow <= m
- 0 <= ballcol, holecol <= n
- Both the ball and the hole exist in an empty space, and they will not be in the same position initially.
- The maze contains at least 2 empty spaces.

>> Brute Force [Heap] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def findShortestWay(self, maze, ball, hole):
        # Brute Force (O(k log n)) - sort repeatedly instead of heap
        data = list(maze)
        result = []
        for _ in range(k):
            data.sort()
            result.append(data.pop(0))
        return result
```

Python

Python

```python
import heapq

def findShortestWay(maze, ball, hole):
    m, n = len(maze), len(maze[0])
    # Directions: sorted lexicographically as 'd' < 'l' < 'r' < 'u'
    # since ASCII: d(100), l(108), r(114), u(117)
    directions = [('d', 1, 0), ('l', 0, -1), ('r', 0, 1), ('u', -1, 0)]

    # Initialize distance and path record, fill with (infty, "")
    dist = [[float('inf')] * n for _ in range(m)]
    pathRecord = [["""] * n for _ in range(m)]

    # Priority queue stores tuples: (distance, path, row, col)
    heap = [(0, "", ball[0], ball[1])]
    dist[ball[0]][ball[1]] = 0

    while heap:
        curDist, curPath, x, y = heapq.heappop(heap)
        # If we pop a state for the hole then return result.
        if [x, y] == hole:
            return curPath
        # If we have a longer distance than already recorded, continue.
        if curDist > dist[x][y]:
            continue
        # Explore all four directions
```



```python
        for d, dx, dy in directions:
            nx, ny = x, y
            steps = 0
            # Roll the ball until hitting a wall or passing through the hole.
            # Check if ball goes through the hole during rolling.
            while 0 <= nx + dx < m and 0 <= ny + dy < n and maze[nx + dx][ny + dy] == 0:
                nx += dx
                ny += dy
                steps += 1
                if [nx, ny] == hole:
                    break
            newDist = curDist + steps
            newPath = curPath + d
            # If we found a shorter route or equal distance but lexicographically smaller route.
            if newDist < dist[nx][ny] or (newDist == dist[nx][ny] and newPath < pathRecord[nx][ny]):
                dist[nx][ny] = newDist
                pathRecord[nx][ny] = newPath
                heapq.heappush(heap, (newDist, newPath, nx, ny))
    return "impossible"

# Example usage:
maze = [[0,0,0,0,0],
        [1,1,0,0,1],
        [0,0,0,0,0],
        [0,1,0,0,1],


        [0,1,0,0,0]]
ball = [4, 3]
hole = [0, 1]
print(findShortestWay(maze, ball, hole)) # Expected output: "lul"
```

Python

```python
class Solution:
    def findShortestWay(
        self,
        maze: list[list[int]],
        ball: list[int],
        hole: list[int],
    ) -> str:
        ans = 'impossible'
        minSteps = math.inf

        def dfs(i: int, j: int, dx: int, dy: int, steps: int, path: str):
          nonlocal ans
          nonlocal minSteps
          if steps >= minSteps:
            return

          if dx != 0 or dy != 0: # Both are zeros for the initial ball position.
            while (0 <= i + dx < len(maze) and 0 <= j + dy < len(maze[0]) and
                   maze[i + dx][j + dy] != 1):
              i += dx
              j += dy
              steps += 1
              if i == hole[0] and j == hole[1] and steps < minSteps:
                minSteps = steps
                ans = path
```

```
        if maze[i][j] == 0 or steps + 2 < maze[i][j]:
            maze[i][j] = steps + 2 # +2 because maze[i][j] == 0 || 1.
            if dx == 0:
                dfs(i, j, 1, 0, steps, path + 'd')
            if dy == 0:
                dfs(i, j, 0, -1, steps, path + 'l')
            if dy == 0:
                dfs(i, j, 0, 1, steps, path + 'r')
            if dx == 0:
                dfs(i, j, -1, 0, steps, path + 'u')

    dfs(ball[0], ball[1], 0, 0, 0, '')
    return ans
```

Python

```
class Solution:
    def findShortestWay(
        self, maze: List[List[int]], ball: List[int], hole: List[int]
    ) -> str:
        m, n = len(maze), len(maze[0])
        r, c = ball
        rh, ch = hole
        q = deque([(r, c)])
        dist = [[inf] * n for _ in range(m)]
        dist[r][c] = 0
        path = [[None] * n for _ in range(m)]
        path[r][c] = ''
        while q:
            i, j = q.popleft()
            for a, b, d in [(-1, 0, 'u'), (1, 0, 'd'), (0, -1, 'l'), (0, 1, 'r')]:
                x, y, step = i, j, dist[i][j]
                while (
                    0 <= x + a < m
                    and 0 <= y + b < n
                    and maze[x + a][y + b] == 0
                    and (x != rh or y != ch)
                ):
                    x, y = x + a, y + b
                    step += 1
                if dist[x][y] > step or (
                    dist[x][y] == step and path[i][j] + d < path[x][y]
                ):
                    dist[x][y] = step
                    path[x][y] = path[i][j] + d
                    if x != rh or y != ch:
                        q.append((x, y))
        return path[rh][ch] or 'impossible'
```

[*] Heap — Pattern Solution (matched from community answers)

Python


```
import heapq

def findShortestWay(maze, ball, hole):
    m, n = len(maze), len(maze[0])
    # Directions: sorted lexicographically as 'd' < 'l' < 'r' < 'u'
    # since ASCII: d(100), l(108), r(114), u(117)
    directions = [('d', 1, 0), ('l', 0, -1), ('r', 0, 1), ('u', -1, 0)]

    # Initialize distance and path record, fill with (infty, "")
    dist = [[float('inf')] * n for _ in range(m)]
    pathRecord = [["""] * n for _ in range(m)]

    # Priority queue stores tuples: (distance, path, row, col)
    heap = [(0, "", ball[0], ball[1])]
    dist[ball[0][ball[1] = 0

    while heap:
        curDist, curPath, x, y = heapq.heappop(heap)
        # If we pop a state for the hole then return result.
        if [x, y] == hole:
            return curPath
        # If we have a larger distance than already recorded, continue.
        if curDist > dist[x][y]:
            continue
        # Explore all four directions

        for d, dx, dy in directions:
            nx, ny = x, y
            steps = 0
            # Roll the ball until hitting a wall or passing through the hole.
            # Check if ball goes through the hole during rolling.
            while 0 <= nx + dx < m and 0 <= ny + dy < n and maze[nx + dx][ny + dy] == 0:
                nx += dx
                ny += dy
                steps += 1
                if [nx, ny] == hole:
                    break
            newDist = curDist + steps
            newPath = curPath + d
            # If we found a shorter route or equal distance but lexicographically smaller route.
            if newDist < dist[nx][ny] or (newDist == dist[nx][ny] and newPath < pathRecord[nx][ny]):
                dist[nx][ny] = newDist
                pathRecord[nx][ny] = newPath
                heapq.heappush(heap, (newDist, newPath, nx, ny))
    return "impossible"

# Example usage:
maze = [[0,0,0,0,0],
        [1,1,0,0,1],
        [0,0,0,0,0],
        [0,1,0,0,1],

        [0,1,0,0,0]]
ball = [4, 3]
hole = [0, 1]
print(findShortestWay(maze, ball, hole)) # Expected output: "lul"
```


#632 HARD

Smallest Range Covering Elements from K Lists

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Greedy · Heap · Sliding Window · Sorting

Lists: Grind 169

Problem:

You have k lists of sorted integers in non-decreasing order. Find the smallest range that includes at least one number from each of the k lists.

We define the range [a, b] is smaller than range [c, d] if b - a < d - c or a < c if b - a == d - c.

Example 1:

```
Input: nums = [[4,10,15,24,26],[0,9,12,20],[5,18,22,30]]
Output: [20,24]
Explanation:
List 1: [4, 10, 15, 24,26], 24 is in range [20,24].
List 2: [0, 9, 12, 20], 20 is in range [20,24].
List 3: [5, 18, 22, 30], 22 is in range [20,24].
```

Example 2:

```
Input: nums = [[1,2,3],[1,2,3],[1,2,3]]
Output: [1,1]
```

Constraints:

- nums.length == k
- 1 <= k <= 3500
- 1 <= nums[i].length <= 50
- -105 <= nums[i][j] <= 105
- nums[i] is sorted in non-decreasing order.

>> Brute Force [Heap] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def smallestRange(self, nums):
        # Brute Force O(n log n) - sort repeatedly instead of heap
        data = list(nums)
        result = []
        for _ in range(k):
            data.sort()
            result.append(data.pop(0))
        return result
```

Python

Python


```
import heapq

def smallestRange(nums):
    # Initialize the min-heap and current maximum value
    heap = []
    current_max = float('-inf')

    # Add the first element of each list to the heap
    for i, lst in enumerate(nums):
        value = lst[0]
        heapq.heappush(heap, (value, i, 0))
        current_max = max(current_max, value)

    best_range = [float('-inf'), float('inf')]

    # Process until one list is exhausted
    while heap:
        current_min, list_idx, elem_idx = heapq.heappop(heap)

        # Update the best range if the current range is smaller
        if current_max - current_min < best_range[1] - best_range[0] or \
           (current_max - current_min == best_range[1] - best_range[0] and current_min < best_range[0]):
            best_range = [current_min, current_max]

        # If there is no next element in the same list, break out

        if elem_idx + 1 == len(nums[list_idx]):
            break
        next_value = nums[list_idx][elem_idx + 1]
        heapq.heappush(heap, (next_value, list_idx, elem_idx + 1))
        current_max = max(current_max, next_value)

    return best_range

# Example Usage:
nums = [[4,10,15,24,26],[0,9,12,20],[5,18,22,30]]
print(smallestRange(nums))
```

Python

```
class Solution:
    def smallestRange(self, nums: list[list[int]]) -> list[int]:
        minHeap = [(row[0], i, 0) for i, row in enumerate(nums)]
        heapq.heapify(minHeap)

        maxRange = max(row[0] for row in nums)
        minRange = heapq.nsmallest(1, minHeap)[0][0]
        ans = [minRange, maxRange]

        while len(minHeap) == len(nums):
          num, r, c = heapq.heappop(minHeap)
          if c + 1 < len(nums[r]):
            heapq.heappush(minHeap, (nums[r][c + 1], r, c + 1))
            maxRange = max(maxRange, nums[r][c + 1])
            minRange = heapq.nsmallest(1, minHeap)[0][0]
            if maxRange - minRange < ans[1] - ans[0]:
              ans[0], ans[1] = minRange, maxRange

        return ans
```


Python

```
class Solution:
    def smallestRange(self, nums: List[List[int]]) -> List[int]:
        t = [(x, i) for i, v in enumerate(nums) for x in v]
        t.sort()
        cnt = Counter()
        ans = [-inf, inf]
        j = 0
        for i, (b, w) in enumerate(t):
            cnt[w] += 1
            while len(cnt) == len(nums):
                a = t[j][0]
                x = b - a - (ans[1] - ans[0])
                if x < 0 or (x == 0 and a < ans[0]):
                    ans = [a, b]
                w = t[j][1]
                cnt[w] -= 1
                if cnt[w] == 0:
                    cnt.pop(w)
                j += 1
        return ans
```

Python

```
class Solution:
    def smallestRange(self, nums: List[List[int]]) -> List[int]:
        t = [(x, i) for i, v in enumerate(nums) for x in v]
        t.sort()
        cnt = Counter()
        ans = [-inf, inf]
        j = 0
        for b, v in t:
            cnt[v] += 1
            while len(cnt) == len(nums):
                a = t[j][0]
                x = b - a - (ans[1] - ans[0])
                if x < 0 or (x == 0 and a < ans[0]):
                    ans = [a, b]
                w = t[j][1]
                cnt[w] -= 1
                if cnt[w] == 0:
                    cnt.pop(w)
                j += 1
        return ans
```

[*] Heap — Pattern Solution (matched from community answers)

Python


```
import heapq

def smallestRange(nums):
    # Initialize the min-heap and current maximum value
    heap = []
    current_max = float('-inf')

    # Add the first element of each list to the heap
    for i, lst in enumerate(nums):
        value = lst[0]
        heapq.heappush(heap, (value, i, 0))
        current_max = max(current_max, value)

    best_range = [float('-inf'), float('inf')]

    # Process until one list is exhausted
    while heap:
        current_min, list_idx, elem_idx = heapq.heappop(heap)

        # Update the best range if the current range is smaller
        if current_max - current_min < best_range[1] - best_range[0] or \
           (current_max - current_min == best_range[1] - best_range[0] and current_min < best_range[0]):
            best_range = [current_min, current_max]

        # If there is no next element in the same list, break out

        if elem_idx + 1 == len(nums[list_idx]):
            break
        next_value = nums[list_idx][elem_idx + 1]
        heapq.heappush(heap, (next_value, list_idx, elem_idx + 1))
        current_max = max(current_max, next_value)

    return best_range

# Example Usage:
nums = [[4,10,15,24,26],[0,9,12,20],[5,18,22,30]]
print(smallestRange(nums))
```

#759 HARD

Employee Free Time

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Sorting · Heap

Lists: Grind 169

Problem:

We are given a list schedule of employees, which represents the working time for each employee.

Each employee has a list of non-overlapping Intervals, and these intervals are in sorted order.

Return the list of finite intervals representing common, positive-length free time for all employees, also in sorted order.

(Even though we are representing Intervals in the form [x, y], the objects inside are Intervals, not lists or arrays. For example, schedule[0][0].start = 1, schedule[0][0].end = 2, and schedule[0][0][0] is not defined). Also, we wouldn't include intervals like [5, 5] in our answer, as they have zero length.

Example 1:

```
Input: schedule = [[[1,2],[5,6]],[[1,3]],[[4,10]]]
Output: [[3,4]]
Explanation: There are a total of three employees, and all common
free time intervals would be [-inf, 1], [3, 4], [10, inf].
We discard any intervals that contain inf as they aren't finite.
```

Example 2:

```
Input: schedule = [[[1,3],[6,7]],[[2,4]],[[2,5],[9,12]]]
Output: [[5,6],[7,9]]
```

Constraints:

- 1 <= schedule.length , schedule[i].length <= 50
- 0 <= schedule[i].start < schedule[i].end <= 10^8

Python

Python

```python
# Class definition for Interval
class Interval:
    def __init__(self, start, end):
        self.start = start # Start time of the interval
        self.end = end # End time of the interval

def employeeFreeTime(schedule):
    # Flatten the schedule: extract all intervals from each employee
    allIntervals = []
    for employee in schedule:
        for interval in employee:
            allIntervals.append(interval)

    # Sort intervals by start time
    allIntervals.sort(key=lambda interval: interval.start)

    # Merge overlapping intervals
    merged = []
    prev = allIntervals[0]
    for interval in allIntervals[1:]:
        if interval.start <= prev.end:
            # Expand the previous interval if there is an overlap
            prev.end = max(prev.end, interval.end)
        else:
            merged.append(prev)
```



```
            prev = interval
    merged.append(prev)

    # Collect free times from the gaps between merged intervals
    freeTimes = []
    for i in range(1, len(merged)):
        # Only intervals with positive length are added
        if merged[i].start > merged[i-1].end:
            freeTimes.append(Interval(merged[i-1].end, merged[i].start))

    return freeTimes

# Example usage:
# schedule = [
# [Interval(1,2), Interval(5,6)],
# [Interval(1,3)],
# [Interval(4,10)]
# ]
# freeTimes = employeeFreeTime(schedule)
# for interval in freeTimes:
# print(f"[{interval.start}, {interval.end}]")
```

Python

```python
class Solution:
  def employeeFreeTime(self, schedule: '[[Interval]]') -> '[Interval]':
    ans = []
    intervals = []

    for s in schedule:
      intervals.extend(s)

    intervals.sort(key=lambda x: x.start)

    prevEnd = intervals[0].end

    for interval in intervals:
      if interval.start > prevEnd:
        ans.append(Interval(prevEnd, interval.start))
      prevEnd = max(prevEnd, interval.end)

    return ans
```

Python


```
"""
# Definition for an Interval.
class Interval:
    def __init__(self, start: int = None, end: int = None):
        self.start = start
        self.end = end
"""


class Solution:
    def employeeFreeTime(self, schedule: "[[Interval]]") -> "[Interval]":
        intervals = []
        for e in schedule:
            intervals.extend(e)
        intervals.sort(key=lambda x: (x.start, x.end))
        merged = [intervals[0]]
        for x in intervals[1:]:
            if merged[-1].end < x.start:
                merged.append(x)
            else:
                merged[-1].end = max(merged[-1].end, x.end)
        ans = []
        for a, b in pairwise(merged):
            ans.append(Interval(a.end, b.start))
        return ans
```

Python

```
"""
# Definition for an Interval.
class Interval:
    def __init__(self, start: int = None, end: int = None):
        self.start = start
        self.end = end
"""


class Solution:
    def employeeFreeTime(self, schedule: "[[Interval]]") -> "[Interval]":
        intervals = []
        for e in schedule:
            intervals.extend(e)
        intervals.sort(key=lambda x: (x.start, x.end))
        merged = [intervals[0]]
        for x in intervals[1:]:
            if merged[-1].end < x.start:
                merged.append(x)
            else:
                merged[-1].end = max(merged[-1].end, x.end)
        ans = []
        for a, b in pairwise(merged):
            ans.append(Interval(a.end, b.start))
        return ans
```

[*] Heap — Pattern Solution (stored optimal)

Python


```python
# Class definition for Interval
class Interval:
    def __init__(self, start, end):
        self.start = start # Start time of the interval
        self.end = end # End time of the interval

def employeeFreeTime(schedule):
    # Flatten the schedule: extract all intervals from each employee
    allIntervals = []
    for employee in schedule:
        for interval in employee:
            allIntervals.append(interval)

    # Sort intervals by start time
    allIntervals.sort(key=lambda interval: interval.start)

    # Merge overlapping intervals
    merged = []
    prev = allIntervals[0]
    for interval in allIntervals[1:]:
        if interval.start <= prev.end:
            # Expand the previous interval if there is an overlap
            prev.end = max(prev.end, interval.end)
        else:
            merged.append(prev)
```


```
            prev = interval
    merged.append(prev)

    # Collect free times from the gaps between merged intervals
    freeTimes = []
    for i in range(1, len(merged)):
        # Only intervals with positive length are added
        if merged[i].start > merged[i-1].end:
            freeTimes.append(Interval(merged[i-1].end, merged[i].start))

    return freeTimes

# Example usage:
# schedule = [
# [Interval(1,2), Interval(5,6)],
# [Interval(1,3)],
# [Interval(4,10)]
# ]
# freeTimes = employeeFreeTime(schedule)
# for interval in freeTimes:
# print(f"[{interval.start}, {interval.end}]")
```

#1425 HARD

Constrained Subsequence Sum

LeetDoocs · SimplyLeet · WallCC · LeetCode · Editorial

Array · Dynamic Programming · Queue · Sliding Window · Heap · Monotonic Queue

Lists: Denny Zhang

Problem:

Given an integer array nums and an integer k, return the maximum sum of a non-empty subsequence of that array such that for every two consecutive integers in the subsequence, nums[i] and nums[j], where i < j, the condition j - i <= k is satisfied.

A subsequence of an array is obtained by deleting some number of elements (can be zero) from the array, leaving the remaining elements in their original order.

Example 1:

```
Input: nums = [10,2,-10,5,20], k = 2
Output: 37
Explanation: The subsequence is [10, 2, 5, 20].
```

Example 2:

```
Input: nums = [-1,-2,-3], k = 1
Output: -1
Explanation: The subsequence must be non-empty, so we choose the largest number.
```

Example 3:

```
Input: nums = [10,-2,-10,-5,20], k = 2
Output: 23
Explanation: The subsequence is [10, -2, -5, 20].
```

Constraints:

- 1 <= k <= nums.length <= 105
- -104 <= nums[i] <= 104

>> Brute Force [Heap] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def constrained_subsequence_sum(self, nums, k):
        # Brute force O(n log n) - sort repeatedly instead of heap
        data = list(nums)
        result = []
        for _ in range(k):
            data.sort()
            result.append(data.pop(0))
        return result
```

Python

Python


```python
# Python solution using a monotonic deque for efficient window maximum computation.
from collections import deque

def constrained_subsequence_sum(nums, k):
    n = len(nums)
    dp = [0] * n # dp[i] stores the maximum subsequence sum ending at index i
    dp[0] = nums[0]
    result = dp[0]

    # Deque to store indices in a decreasing order of dp values
    deq = deque([0])

    for i in range(1, n):
        # Remove indices from the front which are out of the window
        while deq and i - deq[0] > k:
            deq.popleft()

        # The maximum dp in the window is at the front of the deque
        max_in_window = dp[deq[0]] if dp[deq[0]] > 0 else 0

        # Calculate dp[i] by either starting a new subsequence or extending the best one from the window
        dp[i] = nums[i] + max_in_window

        # Update overall answer
        result = max(result, dp[i])
```

```
        # Maintain the deque monotonically so that dp indices are in decreasing order
        while deq and dp[i] >= dp[deq[-1]]:
            deq.pop()
        deq.append(i)

    return result

# Example usage:
# print(constrained_subsequence_sum([10,2,-10,5,20], 2))
```

Python

```python
class Solution:
  def constrainedSubsetSum(self, nums: list[int], k: int) -> int:
    # dp[i] := the maximum the sum of non-empty subsequences in nums[0..i]
    dp = [0] * len(nums)
    # dq stores dp[i - k], dp[i - k + 1], ..., dp[i - 1] whose values are > 0
    # in decreasing order.
    dq = collections.deque()

    for i, num in enumerate(nums):
      if dq:
        dp[i] = max(dq[0], 0) + num
      else:
        dp[i] = num
      while dq and dq[-1] < dp[i]:
        dq.pop()
      dq.append(dp[i])
      if i >= k and dp[i - k] == dq[0]:
        dq.popleft()

    return max(dp)
```

Python

```python
class Solution:
    def constrainedSubsetSum(self, nums: List[int], k: int) -> int:
        q = deque([0])
        n = len(nums)
        f = [0] * n
        ans = -inf
        for i, x in enumerate(nums):
            while i - q[0] > k:
                q.popleft()
            f[i] = max(0, f[q[0]]) + x
            ans = max(ans, f[i])
            while q and f[q[-1]] <= f[i]:
                q.pop()
            q.append(i)
        return ans
```

Python

```python
class Solution:
    def constrainedSubsetSum(self, nums: List[int], k: int) -> int:
        n = len(nums)
        dp = [0] * n
        ans = -inf
        q = deque()
        for i, v in enumerate(nums):
            if q and i - q[0] > k:
                q.popleft()
            dp[i] = max(0, 0 if not q else dp[q[0]]) + v
            while q and dp[q[-1]] <= dp[i]:
                q.pop()
            q.append(i)
            ans = max(ans, dp[i])
        return ans
```

[*] Heap — Pattern Solution (stored optimal)

Python


```python
# Python solution using a monotonic deque for efficient window maximum computation.
from collections import deque

def constrained_subsequence_sum(nums, k):
    n = len(nums)
    dp = [0] * n # dp[i] stores the maximum subsequence sum ending at index i
    dp[0] = nums[0]
    result = dp[0]

    # Deque to store indices in a decreasing order of dp values
    deq = deque([0])

    for i in range(1, n):
        # Remove indices from the front which are out of the window
        while deq and i - deq[0] > k:
            deq.popleft()

        # The maximum dp in the window is at the front of the deque
        max_in_window = dp[deq[0]] if dp[deq[0]] > 0 else 0

        # Calculate dp[i] by either starting a new subsequence or extending the best one from the window
        dp[i] = nums[i] + max_in_window

        # Update overall answer
        result = max(result, dp[i])


        # Maintain the deque monotonically so that dp indices are in decreasing order
        while deq and dp[i] >= dp[deq[-1]]:
            deq.pop()
        deq.append(i)

    return result

# Example usage:
# print(constrained_subsequence_sum([10,2,-10,5,20], 2))
```


Quick Review — Heap 20 questions · insights · complexity · solution

#215 Kth Largest Element in an Array [Medium]

Key Insights

- kth largest element can be converted to finding the element at index n-k if the array were sorted.
- A min-heap of size k is a practical method to maintain the k largest elements seen so far.
- The Quickselect algorithm can find the kth largest element in average O(n) time with in-place partitioning.
- Choose an appropriate method based on performance needs and worst-case scenarios.

Space & Time

Time Complexity:
- Min-Heap: O(n log k) by maintaining a heap of size k.
- Quickselect: Average O(n) but worst-case O(n^2) with poor pivot selection.

Space Complexity:
- Min-Heap: O(k)
- Quickselect: O(1) additional space with in-place partitioning.

Solution

We can solve the problem using either a min-heap or the Quickselect algorithm.
- Min-Heap Approach: Initialize a min-heap with the first k elements. Iterate through the remaining elements; if an element is greater than the heap's root (the smallest in the heap), replace the root. After processing all elements, the root of the min-heap is the kth largest element. - Quickselect Approach: Choose a pivot and partition the array so that elements larger than the pivot come before it. Determine the position of the pivot; if it matches the index corresponding to the kth largest element, return it. Recurse into the appropriate part of the array.

Key points to consider include maintaining heap size for the first approach and proper pivot selection for the Quickselect approach.

#253 Meeting Rooms II [Medium]

Key Insights

- Sort the intervals based on start times to process meetings in order.
- Use a min-heap (priority queue) to keep track of meeting end times.
- When processing a new meeting, check if its start time is greater than or equal to the smallest end time from the heap.
- If so, it means a meeting room has been freed (the meeting with the smallest end time has ended) and can be reused.
- If not, allocate a new room by pushing the current meeting's end time into the heap.
- The size of the heap at any moment represents the number of rooms currently in use.

Space & Time

Time Complexity: O(n log n) due to sorting and each heap operation (push/pop) taking O(log n).

Space Complexity: O(n) in the worst-case scenario when all meetings overlap, requiring all end times in the heap.

Solution

The solution involves first sorting the meeting intervals by their start times. Then, we utilize a min-heap (priority queue) to keep track of the end times of ongoing meetings. For each meeting, we compare its start time with the smallest end time in the heap:
- If the meeting can start after the meeting at the top of the heap has ended (i.e., its start time is greater than or equal to the smallest end), we remove that end time from the heap (freeing up a room). - Then, we add the current

meeting's end time to the heap. By the end of processing all meetings, the size of the heap represents the minimum number of conference rooms required.

#347 Top K Frequent Elements [Medium]

Key Insights

- Count the frequency of each element using a hash table.
- Use bucket sort to achieve linear time complexity by grouping numbers by their frequency.
- Traverse the frequency buckets from highest to lowest to collect the k most frequent elements.
- Alternative approaches include using heaps or Quickselect, but bucket sort meets the follow-up requirement of better than O(n log n) time complexity.

Space & Time

Time Complexity: O(n) where n is the length of the array (bucket sort and frequency counting are linear).

Space Complexity: O(n) for storing the counts and buckets.

Solution

The solution starts by counting the frequency of each element with a hash table (or dictionary). Then, we create an array of buckets where the index represents the frequency. Each bucket holds the numbers that appear with that frequency. Finally, we iterate over the buckets in reverse order (from high frequency to low) and collect elements until we have k of them. This bucket sort approach avoids the higher complexity of sorting a list of frequencies and guarantees linear time performance.

#505 The Maze II [Medium]

Key Insights

- The ball rolls until it hits a wall, so each move is not one step but a continuous travel in a direction.
- The problem can be modeled as finding the shortest path in a weighted graph where nodes are positions in the maze and edge weights are the distance traveled in that roll.
- A priority queue (min-heap) based algorithm, such as Dijkstra's algorithm, is ideal since each roll covers variable distances.
- Always check if the new computed distance for a cell is less than a previously recorded one before pushing it to the heap.

Space & Time

Time Complexity: O(m * n * log(m * n)), where m and n are the maze dimensions (each cell may get processed and queued with a logarithmic cost).

Space Complexity: O(m * n) for maintaining the distance matrix and the priority queue.

Solution

We solve the problem using a modified version of Dijkstra's algorithm. The maze grid is treated as a graph where each cell is a node and an edge exists between the current cell and the cell where the ball stops after rolling in one direction. For every cell popped from the priority queue, we roll the ball in all four directions until it hits a wall, counting the number of steps taken. If the distance calculated from this roll is shorter than a previously stored distance for the stopping cell, we update the distance and add it to the priority queue. This process continues until we either reach the destination (and return the distance) or exhaust all possibilities (returning -1).

#621 Task Scheduler [Medium]

Key Insights

- Count the frequency of each task.
- The task with the highest frequency determines the base structure.
- Identify how many tasks share the maximum frequency.
- Calculate the total time based on the formula: max(total tasks, (max frequency - 1) * (n + 1) + count of tasks with maximum frequency).

- This approach avoids simulating the entire scheduling by using a greedy strategy.

Space & Time

Time Complexity: O(N) where N is the number of tasks (counting frequency, then iterating over constant 26 letters).

Space Complexity: O(1) since the frequency count uses an array or hash map with fixed size (26 letters).

Solution

We first count the frequency of each task. The task(s) with the maximum frequency establish the minimum structure required. Compute the ideal idle slots based on the count of these maximum tasks and the cooling period n. Finally, compare this computed length with the total number of tasks to get the final answer. This greedy strategy avoids explicit simulation and leverages mathematical formulation.

#658 Find K Closest Elements [Medium]

Key Insights

- The input array is already sorted.
- A binary search can be used to efficiently narrow down the potential starting position.
- A two-pointers or sliding window approach is effective for selecting k consecutive elements.
- The comparison of absolute differences (with a tie-breaker on values) is crucial.
- Maintaining ascending order simplifies the final output.

Space & Time

Time Complexity: O(log n + k), where O(log n) for binary search and O(k) for collecting results.

Space Complexity: O(k), for storing the output.

Solution

We can solve this problem by first using binary search to identify the left-most window where the k closest elements might start. The binary search operates over the possible starting indices for windows of size k (from 0 to n-k). For each middle index mid, we compare the distances between x and arr[mid] and arr[mid + k] to decide whether to move the window to the right or left. Once the correct starting position is found, return the k elements starting from that index. This approach leverages the sorted property of the array to efficiently converge on the optimal window.

#787 Cheapest Flights Within K Stops [Medium]

Key Insights

- Build the graph as an adjacency list of flights and prices.
- Use a min-heap so the cheapest partial route is always processed first.
- Track states as `(city, flightsUsed)` instead of only `city`, because the stop count matters.
- For each popped state, try all outgoing flights and push the updated cost and flight count.
- Do not expand a state once it has already used more than `k + 1` flights.
- Keep the best known cost for each `(city, flightsUsed)` state so worse duplicate states can be skipped.

Space & Time

Time Complexity: O((V + E) * log(V * (k + 1))) for the heap-based search over `(city, stopsUsed)` states.

Space Complexity: O(E + V * (k + 1)) for the graph plus the best-cost tracking table.

Solution

We solve the problem using a Dijkstra-style min-heap search, but the state must include both the city and how many flights have been used so far. We start from `(0, src, 0)` and always pop the cheapest state from the heap. For each state, if we reach `dst`, that cost is the answer. Otherwise, if we still have flights available within the `k + 1` limit, we explore all outgoing edges and push the next states into the heap. To avoid useless work, we store the best cost seen for each `(city, flightsUsed)` pair and only continue when the new state improves that record. This guarantees the cheapest valid route under the stop constraint.

#973 K Closest Points to Origin [Medium]

Key Insights

- Calculate the squared Euclidean distance ($x^2 + y^2$) to avoid the overhead of square root computation.
- Sorting the points based on their distance gives an O(n log n) solution.
- Alternatively, using a max-heap to keep track of the k closest points results in O(n log k) time, which is effective if k is much smaller than n.
- Using Quickselect can achieve an average time complexity of O(n), though the worst-case is $O(n^2)$.
- The problem does not require the output to be sorted in any specific order.

Space & Time

Time Complexity: O(n log n) with a sorting approach, O(n log k) with a heap approach, or average O(n) with a Quickselect approach.

Space Complexity: O(n) if extra space is used for sorting, or O(k) for the heap approach.

Solution

We can solve this problem by first computing the squared Euclidean distance for each point to avoid unnecessary square root operations. Then, we can sort the points based on their computed distance and select the first k points. Alternatively, a max-heap or Quickselect algorithm can be used:

- Max-Heap: Maintain a heap of size k while iterating through the points, ensuring that only the k points with the smallest distances are kept. - Quickselect: Partition the array like in QuickSort to position the kth closest point in its correct position. Once partitioned, all points to the left of that position will be the k closest. The sorting solution is straightforward and efficient enough given the constraints.

#1057 Campus Bikes [Medium]

Key Insights

- Compute the Manhattan distance between every worker and bike pair.
- To achieve the tie-breaking rules, sort all pairs by distance, then by worker index and bike index.
- Once a worker is assigned a bike, skip any other pairs involving that worker or the already assigned bike.
- The problem can also be approached using bucket sort because the Manhattan distance range is limited, but sorting is intuitive given the constraints.

Space & Time

Time Complexity: O(n * m * log(n * m)) where n is the number of workers and m is the number of bikes.

Space Complexity: O(n * m) for storing all worker-bike pairs.

Solution

Create all possible (worker, bike) pairs along with their Manhattan distances. Sort this list of pairs by:
- Manhattan distance, - Worker index (if distances are equal), - Bike index (if both distance and worker index are equal).

Then, iterate over the sorted list and assign bikes to workers if neither the worker nor the bike has already been assigned. This guarantees that at each step, the optimal available pairing (according to the problem's requirements) is made. This method naturally enforces the tie-breaking rules specified in the problem.

#1167 Minimum Cost to Connect Sticks [Medium]

Key Insights

- Always connect the two smallest sticks available to minimize the incremental cost.
- A min-heap (or priority queue) is an ideal data structure to efficiently retrieve the smallest sticks.
- The process continues until only one stick remains.

Space & Time

Time Complexity: O(n log n), where n is the number of sticks (due to heap insertion and removal operations).

Space Complexity: O(n), for storing the sticks in the heap.

Solution

The solution uses a greedy algorithm with a min-heap:

- Insert all stick lengths into a min-heap. - While there are at least two sticks in the heap: Extract the two smallest sticks. Compute the cost of connecting them (sum the two lengths). Add this cost to the total result. Push the combined stick (with length equal to their sum) back into the heap. - Once the heap is reduced to one element, return the total cost.

This approach ensures that at every connection, the smallest possible sticks are merged, which leads to the minimum overall cost.

#1424 Diagonal Traverse II [Medium]

Key Insights

- The diagonal key for each element can be defined as (row_index + column_index).
- Elements in the same diagonal (with equal diagonal key) must be output in an order such that the element from the lower row comes before the one from a higher row.
- Because the input grid may be jagged (rows with varying lengths), you must carefully iterate only over valid indices.
- A hash table (or map) can be used to collect elements grouped by their diagonal key.
- Inserting each element at the beginning of the list for that diagonal builds the list in the desired order without needing an extra reverse step later.
- Finally, traverse the keys in increasing order (from 0 to maximum key) to produce the final result.

Space & Time

Time Complexity: O(N), where N is the total number of elements (each element is processed once).

Space Complexity: O(N), for storing the elements in the map and the output array.

Solution

The solution iterates over every element in nums while computing its diagonal key as (i + j). For each element, the value is inserted at the beginning of the list corresponding to that diagonal key. This ensures that when the diagonal groups are later concatenated in order of increasing key, the internal ordering within each diagonal is correct (i.e., elements appear from the bottom of the diagonal to the top). The final result is built by iterating over the keys in sorted order and appending each list of diagonal elements.

#23 Merge k Sorted Lists [Hard]

Key Insights

- Use a min-heap (priority queue) to efficiently extract the smallest value among the current heads of the lists.
- By pushing the head of each non-empty list into the heap, we can pull the smallest node, then push that node's next element into the heap.
- Alternatively, a divide and conquer approach can be used by recursively merging pairs of lists.
- Careful management of pointers (or references) is key to constructing the new linked list.

Space & Time

Time Complexity: O(N log k) where N is the total number of nodes across all lists and k is the number of lists.

Space Complexity: O(k) due to the heap containing at most one node from each list.

Solution

We use a min-heap to always retrieve the smallest node among the current nodes in the linked lists. First, initialize the heap with the head of each list. Then, continuously extract the smallest node from the heap, attach it to the merged list, and if the extracted node has a next node, insert that into the heap. The process repeats until the heap is empty.

#218 The Skyline Problem [Hard]

Key Insights

- Use a line sweep approach to process building start and end events in sorted order.
- Differentiate between start events (which add to the skyline) and end events (which remove from the skyline).

- Employ a max heap (or a priority queue) to efficiently track the current highest building at each sweep line position.
- When the current maximum height changes, record the corresponding x-coordinate and new height as a key point.
- Take care to remove expired building heights from the heap as the sweep line moves past the building's end.

Space & Time

Time Complexity: O(n log n) due to sorting events and heap operations for each building event.

Space Complexity: O(n) for the event list and the heap used for processing active buildings.

Solution

The solution uses the line sweep algorithm with a max heap (or priority queue) to maintain the heights of active buildings. Each building contributes two events: one when the building starts (adding its height) and one when it ends (removing its height). These events are sorted primarily by their x-coordinate. For start events, we use negative height values to ensure they are processed before end events at the same x-coordinate. During the sweep, the heap is updated by adding new building heights and removing buildings that have ended. Whenever there is a change in the top of the heap (i.e., the current maximum height), a key point is added to the result. This key point signifies a vertical change in the skyline.

#272 Closest Binary Search Tree Value II [Hard]

Key Insights

- The BST property allows efficient narrowing down of candidate nodes.
- Using two stacks to track predecessors (values <= target) and successors (values > target) can efficiently yield the closest values.
- A modified in-order traversal helps in retrieving the next closest candidate from either side.
- Merging two sorted lists of candidate values based on absolute difference with the target is the key step.

Space & Time

Time Complexity: O(k + log n) - O(log n) to initialize the two stacks and O(k) to collect k elements.

Space Complexity: O(log n) - due to the stack space in a balanced BST.

Solution

We can solve the problem by splitting it into two parts:

- Build two stacks: The predecessor stack for nodes with values less than or equal to target. The successor stack for nodes with values greater than target. To build these stacks, traverse the BST starting from the root and push nodes along the path: if the node's value is less than or equal to target, push it to the predecessor stack and go right; if greater, push it to the successor stack and go left. - Merge the two stacks: At each step, compare the top element of the predecessor stack (if available) and the successor stack (if available) based on their distance from the target. Pop from the stack with the closer candidate and add that value to the result list. For the popped node, update the given stack by moving to the next appropriate node in the in-order traversal (for predecessor, go to left child then rightmost descendant; for successor, go to right child then leftmost descendant). Repeat until k values have been collected.

Key Data Structures and Techniques:

- Two stacks to simulate reverse and forward in-order traversals. - Merge technique to choose the closest candidate at each iteration.

#295 Find Median from Data Stream [Hard]

Key Insights

- Use two heaps: a max-heap for the lower half and a min-heap for the upper half of numbers.
- Balancing the heaps ensures the median can be computed in O(1) time once numbers are added.
- Rebalance the heaps after each insertion to ensure size properties hold (difference in sizes is at most 1).

Space & Time

Time Complexity:

- addNum: O(log n) per insertion (heap operations).
- findMedian: O(1) (accessing the top elements).

Space Complexity:

- O(n), where n is the number of elements added in the data stream, as they are stored in the two heaps.

Solution

We maintain two heaps:

- A max-heap (lowerHalf) for the smaller half of the numbers. - A min-heap (upperHalf) for the larger half of the numbers. On inserting a new number, decide the heap membership based on the current median. After each insertion, ensure that the heaps have balanced sizes (the size difference is at most one). When computing the median: - If both heaps have the same number of elements, the median is the average of the max of lowerHalf and the min of upperHalf. - If one heap has more elements, the median is the top element of that heap.

This method makes it efficient to dynamically update and retrieve the median as new numbers are inserted.

#358 Rearrange String k Distance Apart [Hard]

Key Insights

- The frequency of each character dictates the feasibility and placement within the rearrangement.
- A greedy approach picks the most frequent available character to maximize the chance of successful placement.
- A max-heap (priority queue) efficiently selects the highest frequency character each time.
- A waiting queue ensures that once a character is used, it can't be reused until k positions later.
- Special case: When k is 0, no rearrangement is required because all orders satisfy the condition.

Space & Time

Time Complexity: O(n log C) where n is the length of s and C is the number of unique characters.

Space Complexity: O(n) due to the storage used for the heap and wait queue.

Solution

We use a greedy algorithm with a max-heap to always select the character with the highest remaining frequency that is currently available. The algorithm first counts the frequency of each character and pushes these into a max-heap (using negative frequencies for languages like Python where heapq is a min-heap). Each time a character is selected, it is appended to the result string and its frequency is decreased. This character is then placed in a waiting queue to ensure it isn't reused until k positions later. Once the waiting period completes, if the character still has remaining occurrences, it is reinserted into the heap. If at any point the heap is empty and the waiting queue still contains characters that haven't been reinserted while the result string is incomplete, we conclude that a valid rearrangement is impossible.

#499 The Maze III [Hard]

Key Insights

- The ball rolls continuously until hitting a wall or falling into the hole.
- A shortest path search is needed based on rolling distance (each step counts when passing an empty cell).
- Dijkstra's algorithm (or a modified BFS with a priority queue) is appropriate since we need to consider distance and lexicographic order.
- When expanding moves, simulate the roll until the ball stops — if a hole is passed en route, use that stopping point.
- Track the best (distance, path) for each cell to avoid redundant work.

Space & Time

Time Complexity: O(mn log(mn)) where m and n are the maze dimensions

Space Complexity: O(mn)

Solution

We use a Dijkstra-like approach with a priority queue containing tuples of (distance, path, row, col). The priority queue is ordered by distance first and then by the instruction string lexicographically. For each cell, simulate rolling

in each possible direction (up, down, left, right; iterated in alphabetical order so that the lexicographical property is maintained) until the ball stops. If the ball encounters the hole during rolling, treat that as a valid stop immediately. For every new stopping cell, if a shorter distance or lexicographically smaller path is found, update the record and add the new state to the priority queue. If you reach the hole, return the corresponding path string. If no solution is found when the queue is empty, return "impossible".

#632 Smallest Range Covering Elements from K Lists [Hard]

Key Insights

- Use a min heap to maintain the smallest current element among the selected elements from each list.
- Track the current maximum among the chosen elements to easily compute the range.
- Iteratively replace the minimum element with the next element from the same list and update the current maximum.
- Stop when one of the lists is exhausted, as the range must include at least one element from each list.

Space & Time

Time Complexity: O(N log k), where N is the total number of elements across all lists.

Space Complexity: O(k) for the heap, plus additional space for storing the range.

Solution

The solution uses a min heap (priority queue) that stores a tuple containing the current element, the index of the list it comes from, and its index in that list. Initially, the first element from each list is inserted into the heap and the current maximum among them is tracked. In each iteration, the smallest element is removed (heap root) which provides the current minimum, and the range [current_min, current_max] is assessed. If the list from which the minimum element was extracted has a next element, that element is inserted into the heap and the current maximum is updated if necessary. This process continues until one of the lists is exhausted.

#759 Employee Free Time [Hard]

Key Insights

- Flatten the schedule to combine all employees' intervals.
- Sort the flattened intervals by start time.
- Merge overlapping intervals to form a consolidated view of busy times.
- Identify gaps between merged intervals as the common free time.
- Ensure free intervals have strictly positive length.

Space & Time

Time Complexity: O(N log N), where N is the total number of intervals (due to sorting).

Space Complexity: O(N), used for the flattened list and merged intervals.

Solution

The solution leverages interval merging. First, all intervals from every employee are collected and sorted by start time. By iterating through the sorted intervals, overlapping intervals are merged into one consolidated busy period. Finally, the gaps between consecutive merged intervals are identified as free periods available to all employees. This approach efficiently handles the intervals with a simple sort and a linear scan.

#1425 Constrained Subsequence Sum [Hard]

Key Insights

- Use dynamic programming where dp[i] represents the maximum sum of a valid subsequence ending at index i.
- For each index i, consider taking nums[i] alone or appending it to a valid subsequence ending at an index j (where i - j <= k) that maximizes the sum.
- To efficiently obtain the maximum dp[j] in a sliding window of the last k indices, use a monotonic (decreasing) deque.
- The idea is to maintain the window maximum and update it as we proceed through the array.

Space & Time

Time Complexity: O(n)

Space Complexity: O(n) (O(k) extra space for the deque, with dp array using O(n))

Solution

We solve the problem by defining a dp array where dp[i] = nums[i] + max(0, maximum dp value in the window [i-k, i-1]). The trick is to include max(0, ...) to allow starting a new subsequence when previous sums are negative. To access the maximum dp in the window in constant time, a monotonic deque is maintained that stores indices with their dp values in decreasing order. At each index, we:

- Remove elements from the front of the deque if they are out of the current window (i - index > k). - Use the front of the deque (if non-empty) as the maximum dp value from the previous k indices. - Compute dp[i] and update the global maximum answer. - Remove from the back of the deque all indices with dp values less than the current dp[i] since they are not useful. - Insert the current index into the deque.

#15 Matrix — 20 questions · #15 of 21

#422 EASY

Valid Word Square

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Matrix

Lists: Premium 98

Problem:

Given an array of strings words, return true if it forms a valid word square.

A sequence of strings forms a valid word square if the kth row and column read the same string, where 0 <= k < max(numRows, numColumns).

Example 1:

a	b	c	d
b	n	r	t
c	r	m	y
d	t	y	e

```
Input: words = ["abcd","bnrt","crmy","dtye"]
Output: true
Explanation:
The 1st row and 1st column both read "abcd".
The 2nd row and 2nd column both read "bnrt".
The 3rd row and 3rd column both read "crmy".
The 4th row and 4th column both read "dtye".
Therefore, it is a valid word square.
```

Example 2:

a	b	c	d
b	n	r	t
c	r	m	
d	t		

```
Input: words = ["abcd","bnrt","crm","dt"]
Output: true
Explanation:
The 1st row and 1st column both read "abcd".
The 2nd row and 2nd column both read "bnrt".
The 3rd row and 3rd column both read "crm".
The 4th row and 4th column both read "dt".
Therefore, it is a valid word square.
```

Example 3:

b	a	l	l
a	r	e	a
r	e	a	d
l	a	d	y

```
Input: words = ["ball","area","read","lady"]
Output: false
Explanation:
The 3rd row reads "read" while the 3rd column reads "lead".
Therefore, it is NOT a valid word square.
```

Constraints:

- 1 <= words.length <= 500
- 1 <= words[i].length <= 500
- words[i] consists of only lowercase English letters.

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def validWordSquare(self, words):
        # Brute Force Start->O(n^2) - Flood fill from every cell
        m, n = len(words), len(words[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```


Python

Python

```python
# Function to check if the provided words form a valid word square
def validWordSquare(words):
    # Iterate over each row with its index
    for i in range(len(words)):
        # Iterate over each character in the current row
        for j in range(len(words[i])):
            # Check if the corresponding column exists
            if j >= len(words) or i >= len(words[j]) or words[i][j] != words[j][i]:
                return False
    return True

# Example test case
words = ["abcd","bnrt","crmy","dtye"]
print(validWordSquare(words)) # Expected output: True
```

Python

```python
class Solution:
    def validWordSquare(self, words: list[str]) -> bool:
        for i, word in enumerate(words):
            for j, c in enumerate(word):
                if len(words) <= j or len(words[j]) <= i: # out-of-bounds
                    return False
                if c != words[j][i]:
                    return False
        return True
```

Python

```python
class Solution:
    def validWordSquare(self, words: List[str]) -> bool:
        m = len(words)
        for i, w in enumerate(words):
            for j, c in enumerate(w):
                if j >= m or i >= len(words[j]) or c != words[j][i]:
                    return False
        return True
```

Python

```python
class Solution:
    def validWordSquare(self, words: List[str]) -> bool:
        m = len(words)
        for i, w in enumerate(words):
            for j, c in enumerate(w):
                if j >= m or i >= len(words[j]) or c != words[j][i]:
                    return False
        return True
```

[*] Matrix — Pattern Solution (stored optimal)

Python


```python
# Function to check if the provided words form a valid word square
def validWordSquare(words):
    # Iterate over each row with its index
    for i in range(len(words)):
        # Iterate over each character in the current row
        for j in range(len(words[i])):
            # Check if the corresponding column exists
            if j >= len(words) or i >= len(words[j]) or words[i][j] != words[j][i]:
                return False
    return True

# Example test case
words = ["abcd","bnrt","crmy","dtye"]
print(validWordSquare(words)) # Expected output: True
```

#566 EASY

Reshape the Matrix

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Matrix · Simulation

Lists: CodeSignal

Problem:

In MATLAB, there is a handy function called reshape which can reshape an m x n matrix into a new one with a different size r x c keeping its original data.

You are given an m x n matrix mat and two integers r and c representing the number of rows and the number of columns of the wanted reshaped matrix.

The reshaped matrix should be filled with all the elements of the original matrix in the same row-traversing order as they were.

If the reshape operation with given parameters is possible and legal, output the new reshaped matrix; Otherwise, output the original matrix.

Example 1:

1	2
3	4

⇒

1	2	3	4

```
Input: mat = [[1,2],[3,4]], r = 1, c = 4
Output: [[1,2,3,4]]
```

Example 2:

1	2
3	4

⇒

1	2
3	4

```
Input: mat = [[1,2],[3,4]], r = 2, c = 4
Output: [[1,2],[3,4]]
```

Constraints:

- m == mat.length
- n == mat[i].length
- 1 <= m, n <= 100
- -1000 <= mat[i][j] <= 1000
- 1 <= r, c <= 300

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def matrixReshape(self, mat, r, c):
        # Brute Force Start->O(n^2) - Flood fill from every cell
        m, n = len(mat), len(mat[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python


```python
# Function to reshape the matrix
def matrixReshape(mat, r, c):
    # Calculate the dimensions of the original matrix
    m = len(mat)
    n = len(mat[0])

    # Check if reshape is possible
    if m * n != r * c:
        return mat

    # Flatten the original matrix into a list
    flat_list = []
    for row in mat:
        for num in row:
            flat_list.append(num)

    # Create the new matrix by slicing the flat list
    new_matrix = []
    for i in range(0, len(flat_list), c):
        new_matrix.append(flat_list[i:i+c])

    return new_matrix

# Example call:
# print(matrixReshape([[1,2],[3,4]], 1, 4))
```

Python

```python
class Solution:
    def matrixReshape(self, nums: list[list[int]],
                      r: int, c: int) -> list[list[int]]:
        if nums == [] or r * c != len(nums) * len(nums[0]):
            return nums

        ans = [[0 for j in range(c)] for i in range(r)]
        k = 0

        for row in nums:
            for num in row:
                ans[k // c][k % c] = num
                k += 1

        return ans
```

Python

```python
class Solution:
    def matrixReshape(self, mat: List[List[int]], r: int, c: int) -> List[List[int]]:
        m, n = len(mat), len(mat[0])
        if m * n != r * c:
            return mat
        ans = [[0] * c for _ in range(r)]
        for i in range(m * n):
            ans[i // c][i % c] = mat[i // n][i % n]
        return ans
```

Python

```python
class Solution:
    def matrixReshape(self, mat: List[List[int]], r: int, c: int) -> List[List[int]]:
        m, n = len(mat), len(mat[0])
        if m * n != r * c:
            return mat
        ans = [[0] * c for _ in range(r)]
        for i in range(m * n):
            ans[i // c][i % c] = mat[i // n][i % n]
        return ans
```

[*] Matrix — Pattern Solution (stored optimal)

Python

```python
# Function to reshape the matrix
def matrixReshape(mat, r, c):
    # Calculate the dimensions of the original matrix
    m = len(mat)
    n = len(mat[0])

    # Check if reshape is possible
    if m * n != r * c:
        return mat

    # Flatten the original matrix into a list
    flat_list = []
    for row in mat:
        for num in row:
            flat_list.append(num)

    # Create the new matrix by slicing the flat list
    new_matrix = []
    for i in range(0, len(flat_list), c):
        new_matrix.append(flat_list[i:i+c])

    return new_matrix

# Example call:
# print(matrixReshape([[1,2],[3,4]], 1, 4))
```

#766 EASY

Toeplitz Matrix

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Matrix

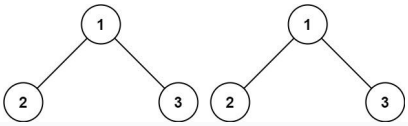
Lists: CodeSignal

Problem:

Given an m x n matrix, return true if the matrix is Toeplitz. Otherwise, return false.

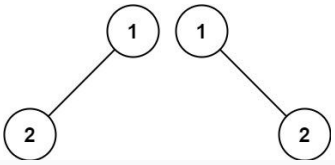
A matrix is Toeplitz if every diagonal from top-left to bottom-right has the same elements.

Example 1:




```
Input: matrix = [[1,2,3,4],[5,1,2,3],[9,5,1,2]]
Output: true
Explanation:
In the above grid, the diagonals are:
"[9]", "[5, 5]", "[1, 1, 1]", "[2, 2, 2]", "[3, 3]", "[4]".
In each diagonal all elements are the same, so the answer is True.
```

Example 2:




```
Input: matrix = [[1,2],[2,2]]
Output: false
Explanation:
The diagonal "[1, 2]" has different elements.
```

Constraints:

- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 20
- 0 <= matrix[i][j] <= 99

Follow up:

- What if the matrix is stored on disk, and the memory is limited such that you can only load at most one row of the matrix into the memory at once?
- What if the matrix is so large that you can only load up a partial row into the memory at once?

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def isToeplitzMatrix(self, matrix):
        # Brute Force O(m^2·n^2) - flood fill from every cell
        m, n = len(matrix), len(matrix[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python

```python
# Function to determine if the matrix is Toeplitz
def isToeplitzMatrix(matrix):
    # Get the number of rows and columns in the matrix
    rows = len(matrix)
    cols = len(matrix[0])

    # Iterate starting from row 1 and column 1 because first row and column are trivially Toeplitz
    for i in range(1, rows):
        for j in range(1, cols):
            # Compare current element with its top-left neighbor
            if matrix[i][j] != matrix[i - 1][j - 1]:
                return False
    return True

# Example usage:
matrix1 = [[1,2,3,4],[5,1,2,3],[9,5,1,2]]
print(isToeplitzMatrix(matrix1)) # Expected output: True
```

Python

```python
class Solution:
   def isToeplitzMatrix(self, matrix: list[list[int]]) -> bool:
      for i in range(len(matrix) - 1):
        for j in range(len(matrix[0]) - 1):
          if matrix[i][j] != matrix[i + 1][j + 1]:
            return False

      return True
```

Python


```python
class Solution:
    def isToeplitzMatrix(self, matrix: List[List[int]]) -> bool:
        m, n = len(matrix), len(matrix[0])
        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][j] != matrix[i - 1][j - 1]:
                    return False
        return True
```

Python

```python
class Solution:
    def isToeplitzMatrix(self, matrix: List[List[int]]) -> bool:
        m, n = len(matrix), len(matrix[0])
        return all(
            matrix[i][j] == matrix[i - 1][j - 1]
            for i in range(1, m)
            for j in range(1, n)
        )
```

[*] Matrix — Pattern Solution (stored optimal)

Python

```python
# Function to determine if the matrix is Toeplitz
def isToeplitzMatrix(matrix):
    # Get the number of rows and columns in the matrix
    rows = len(matrix)
    cols = len(matrix[0])

    # Iterate starting from row 1 and column 1 because first row and column are trivially Toeplitz
    for i in range(1, rows):
        for j in range(1, cols):
            # Compare current element with its top-left neighbor
            if matrix[i][j] != matrix[i - 1][j - 1]:
                return False
    return True

# Example usage:
matrix1 = [[1,2,3,4],[5,1,2,3],[9,5,1,2]]
print(isToeplitzMatrix(matrix1)) # Expected output: True
```

#867 EASY

Transpose Matrix

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

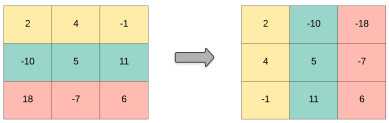
Array · Matrix · Simulation

Lists: CodeSignal

Problem:

Given a 2D integer array matrix, return the transpose of matrix.

The transpose of a matrix is the matrix flipped over its main diagonal, switching the matrix's row and column indices.



Example 1:

```
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [[1,4,7],[2,5,8],[3,6,9]]
```

Example 2:

```
Input: matrix = [[1,2,3],[4,5,6]]
Output: [[1,4],[2,5],[3,6]]
```

Constraints:

- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 1000
- 1 <= m * n <= 105
- -109 <= matrix[i][j] <= 109

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def transpose(self, matrix):
        # Brute Force O(m^2·n^2) - flood fill from every cell
        m, n = len(matrix), len(matrix[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python


```python
# Python solution with clear comments

def transpose(matrix):
    # Number of rows in the original matrix
    num_rows = len(matrix)
    # Number of columns in the original matrix
    num_cols = len(matrix[0])

    # Create a new matrix with dimensions swapped: num_cols x num_rows
    transposed = [[0] * num_rows for _ in range(num_cols)]

    # Iterate through each element in the original matrix
    for i in range(num_rows):
        for j in range(num_cols):
            # Place the element at the swapped index in the transposed matrix
            transposed[j][i] = matrix[i][j]

    return transposed

# Example usage:
matrix = [[1, 2, 3], [4, 5, 6]]
print(transpose(matrix))
```

Python

```python
class Solution:
   def transpose(self, A: list[list[int]]) -> list[list[int]]:
      ans = [[0] * len(A) for _ in range(len(A[0]))]

      for i in range(len(A)):
        for j in range(len(A[0])):
          ans[j][i] = A[i][j]

      return ans
```

Python

```python
class Solution:
    def transpose(self, matrix: List[List[int]]) -> List[List[int]]:
        return list(zip(*matrix))
```

Python

```python
class Solution:
    def transpose(self, matrix: List[List[int]]) -> List[List[int]]:
        return list(zip(*matrix))
```

[*] Matrix — Pattern Solution (stored optimal)

Python

```python
# Python solution with clear comments

def transpose(matrix):
    # Number of rows in the original matrix
    num_rows = len(matrix)
    # Number of columns in the original matrix
    num_cols = len(matrix[0])

    # Create a new matrix with dimensions swapped: num_cols x num_rows
    transposed = [[0] * num_rows for _ in range(num_cols)]

    # Iterate through each element in the original matrix
    for i in range(num_rows):
        for j in range(num_cols):
            # Place the element at the swapped index in the transposed matrix
            transposed[j][i] = matrix[i][j]

    return transposed

# Example usage:
matrix = [[1, 2, 3], [4, 5, 6]]
print(transpose(matrix))
```

#2373 EASY

Largest Local Values in a Matrix

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Matrix

Lists: CodeSignal

Problem:

You are given an n x n integer matrix grid.

Generate an integer matrix maxLocal of size (n − 2) x (n − 2) such that:

- maxLocal[i][j] is equal to the largest value of the 3 x 3 matrix in grid centered around row i + 1 and column j + 1.

In other words, we want to find the largest value in every contiguous 3 x 3 matrix in grid.

Return the generated matrix.

Example 1:

9	9	8	1
5	6	2	6
8	2	6	4
6	2	2	2

9	9
8	6

```
Input: grid = [[9,9,8,1],[5,6,2,6],[8,2,6,4],[6,2,2,2]]
Output: [[9,9],[8,6]]
Explanation: The diagram above shows the original matrix and the generated matrix.
Notice that each value in the generated matrix corresponds to the largest value of a
contiguous 3 x 3 matrix in grid.
```

Example 2:

1	1	1	1	1
1	1	1	1	1
1	1	2	1	1
1	1	1	1	1
1	1	1	1	1

2	2	2
2	2	2
2	2	2

```
Input: grid = [[1,1,1,1,1],[1,1,1,1,1],[1,1,2,1,1],[1,1,1,1,1],[1,1,1,1,1]]
Output: [[2,2,2],[2,2,2],[2,2,2]]
Explanation: Notice that the 2 is contained within every contiguous 3 x 3 matrix in grid.
```


Constraints:

- n == grid.length == grid[i].length
- 3 <= n <= 100
- 1 <= grid[i][j] <= 100

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def largestLocal(self, grid):
        # Brute force: check every cell - flood fill from every cell
        m, n = len(grid), len(grid[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python

```python
# Python solution for finding the largest local values in a matrix

def largestLocal(grid):
    n = len(grid) # dimension of the grid
    # Initialize the result matrix with zeros, of size (n-2) x (n-2)
    result = [[0] * (n - 2) for _ in range(n - 2)]

    # Loop through each possible top-left index of a 3x3 submatrix
    for i in range(n - 2):
        for j in range(n - 2):
            # Compute the maximum in the current 3x3 submatrix
            local_max = 0 # since grid elements are >= 1, starting with 0 is safe
            for k in range(3):
                for l in range(3):
                    local_max = max(local_max, grid[i + k][j + l])
            # Store the local maximum in the result matrix
            result[i][j] = local_max
    return result

# Example usage:
grid = [[9,9,8,1],[5,6,2,6],[8,2,6,4],[6,2,2,2]]
print(largestLocal(grid))
```

Python


```python
class Solution:
    def largestLocal(self, grid: list[list[int]]) -> list[list[int]]:
        n = len(grid)
        ans = [[0] * (n - 2) for _ in range(n - 2)]

        for i in range(n - 2):
            for j in range(n - 2):
                for x in range(i, i + 3):
                    for y in range(j, j + 3):
                        ans[i][j] = max(ans[i][j], grid[x][y])

        return ans
```

Python

```python
class Solution:
    def largestLocal(self, grid: List[List[int]]) -> List[List[int]]:
        n = len(grid)
        ans = [[0] * (n - 2) for _ in range(n - 2)]
        for i in range(n - 2):
            for j in range(n - 2):
                ans[i][j] = max(
                    grid[x][y] for x in range(i, i + 3) for y in range(j, j + 3)
                )
        return ans
```

Python

```python
class Solution:
    def largestLocal(self, grid: List[List[int]]) -> List[List[int]]:
        n = len(grid)
        ans = [[0] * (n - 2) for _ in range(n - 2)]
        for i in range(n - 2):
            for j in range(n - 2):
                ans[i][j] = max(
                    grid[x][y] for x in range(i, i + 3) for y in range(j, j + 3)
                )
        return ans
```

[*] Matrix — Pattern Solution (stored optimal)

Python


```python
# Python solution for finding the largest local values in a matrix

def largestLocal(grid):
    n = len(grid) # dimension of the grid
    # Initialize the result matrix with zeros, of size (n-2) x (n-2)
    result = [[0] * (n - 2) for _ in range(n - 2)]

    # Loop through each possible top-left index of a 3x3 submatrix
    for i in range(n - 2):
        for j in range(n - 2):
            # Compute the maximum in the current 3x3 submatrix
            local_max = 0 # since grid elements are >= 1, starting with 0 is safe
            for k in range(3):
                for l in range(3):
                    local_max = max(local_max, grid[i + k][j + l])
            # Store the local maximum in the result matrix
            result[i][j] = local_max
    return result

# Example usage:
grid = [[9,9,8,1],[5,6,2,6],[8,2,6,4],[6,2,2,2]]
print(largestLocal(grid))
```

#36 MEDIUM

Valid Sudoku

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Matrix

Lists: Grind 169

Problem:

Determine if a 9 x 9 Sudoku board is valid. Only the filled cells need to be validated according to the following rules:

1. Each row must contain the digits 1–9 without repetition.
2. Each column must contain the digits 1–9 without repetition.
3. Each of the nine 3 x 3 sub-boxes of the grid must contain the digits 1–9 without repetition.

Note:

- A Sudoku board (partially filled) could be valid but is not necessarily solvable.
- Only the filled cells need to be validated according to the mentioned rules.

Example 1:

5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

```
Input: board =
[["5","3",".",".","7",".",".",".","."]
,["6",".",".","1","9","5",".",".","."]
,[".","9","8",".",".",".",".","6","."]
,["8",".",".",".","6",".",".",".","3"]
,["4",".",".","8",".","3",".",".","1"]
,["7",".",".",".","2",".",".",".","6"]
,[".","6",".",".",".",".","2","8",".."]
,[".",".",".","4","1","9",".",".","5"]
,[".",".",".",".","8",".",".","7","9"]]
Output: true
```

Example 2:

```
Input: board =
[["8","3",".",".","7",".",".",".",."]
,["6",".",".","1","9","5",".",".",."]
,[".","9","8",".",".",".",".","6",."]
,["8",".",".",".","6",".",".",".","3"]
,["4",".",".","8",".","3",".",".","1"]
,["7",".",".",".","2",".",".",".","6"]
,[".","6",".",".",".",".","2","8",".."]
,[".",".",".","4","1","9",".",".","5"]
,[".",".",".",".","8",".",".","7","9"]]
Output: false
Explanation: Same as Example 1, except with the 5 in the top left corner being modified to 8.
Since there are two 8's in the top left 3x3 sub-box, it is invalid.
```

Constraints:

- board.length == 9
- board[i].length == 9
- board[i][j] is a digit 1–9 or '.'.

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def isValidSudoku(self, board):
        # Brute Force O(81) - check each row, col, box for duplicates
        rows = [set() for _ in range(9)]
        cols = [set() for _ in range(9)]
        boxes = [set() for _ in range(9)]
        for r in range(9):
            for c in range(9):
                val = board[r][c]
                if val == '.': continue
                box_idx = (r // 3) * 3 + c // 3
                if val in rows[r] or val in cols[c] or val in boxes[box_idx]:
                    return False
                rows[r].add(val); cols[c].add(val); boxes[box_idx].add(val)
        return True
```

Python

Python

```python
# Python solution with inline comments
def isValidSudoku(board):
    # Initialize sets for rows, columns, and boxes
    rows = [set() for _ in range(9)]
    cols = [set() for _ in range(9)]
    boxes = [set() for _ in range(9)]

    # Iterate over each cell in the board
    for r in range(9):
        for c in range(9):
            num = board[r][c]
            # Skip empty cells
            if num == '.':
                continue
            # Compute box index using integer division
            box_index = (r // 3) * 3 + (c // 3)
            # Check if the number is already present in the row, column, or box
            if num in rows[r] or num in cols[c] or num in boxes[box_index]:
                return False
            # Add number to row, column, and box sets
            rows[r].add(num)
            cols[c].add(num)
            boxes[box_index].add(num)

    return True
```



```python
# Example usage:
if __name__ == "__main__":
    board = [
        ["5","3",".",".","7",".",".",".","."],
        ["6",".",".","1","9","5",".",".","."],
        [".","9","8",".",".",".",".","6","."],
        ["8",".",".",".","6",".",".",".","3"],
        ["4",".",".","8",".","3",".",".","1"],
        ["7",".",".",".","2",".",".",".","6"],
        [".","6",".",".",".",".","2","8","."],
        [".",".",".","4","1","9",".",".","5"],
        [".",".",".",".","8",".",".","7","9"]
    ]
    print(isValidSudoku(board)) # Expected output: True
```

Python

```python
class Solution:
    def isValidSudoku(self, board: list[list[str]]) -> bool:
        seen = set()

        for i in range(9):
            for j in range(9):
                c = board[i][j]
                if c == '.':
                    continue
                if (c + 'grow ' + str(i) in seen or
                    c + 'gcol ' + str(j) in seen or
                    c + 'gbox ' + str(i // 3) + str(j // 3) in seen):
                    return False
                seen.add(c + 'grow ' + str(i))
                seen.add(c + 'gcol ' + str(j))
                seen.add(c + 'gbox ' + str(i // 3) + str(j // 3))

        return True
```

Python

```python
class Solution:
    def isValidSudoku(self, board: List[List[str]]) -> bool:
        row = [[False] * 9 for _ in range(9)]
        col = [[False] * 9 for _ in range(9)]
        sub = [[False] * 9 for _ in range(9)]
        for i in range(9):
            for j in range(9):
                c = board[i][j]
                if c == '.':
                    continue
                num = int(c) - 1
                k = i // 3 * 3 + j // 3
                if row[i][num] or col[j][num] or sub[k][num]:
                    return False
                row[i][num] = True
                col[j][num] = True
                sub[k][num] = True
        return True
```

Python


```python
class Solution:
    def isValidSudoku(self, board: List[List[str]]) -> bool:
        row = [[False] * 9 for _ in range(9)]
        col = [[False] * 9 for _ in range(9)]
        sub = [[False] * 9 for _ in range(9)]
        for i in range(9):
            for j in range(9):
                c = board[i][j]
                if c == '.':
                    continue
                num = int(c) - 1
                k = i // 3 * 3 + j // 3
                if row[i][num] or col[j][num] or sub[k][num]:
                    return False
                row[i][num] = True
                col[j][num] = True
                sub[k][num] = True
        return True
```

[*] Matrix — Pattern Solution (stored optimal)

Python

```python
# Python solution with inline comments
def isValidSudoku(board):
    # Initialize sets for rows, columns, and boxes
    rows = [set() for _ in range(9)]
    cols = [set() for _ in range(9)]
    boxes = [set() for _ in range(9)]

    # Iterate over each cell in the board
    for r in range(9):
        for c in range(9):
            num = board[r][c]
            # Skip empty cells
            if num == '.':
                continue
            # Compute box index using integer division
            box_index = (r // 3) * 3 + (c // 3)
            # Check if the number is already present in the row, column, or box
            if num in rows[r] or num in cols[c] or num in boxes[box_idx]:
                return False
            # Add number to row, column, and box sets
            rows[r].add(num)
            cols[c].add(num)
            boxes[box_index].add(num)

    return True
```



```python
# Example usage:
if __name__ == "__main__":
    board = [
        ["5","3",".",".","7",".",".",".","."],
        ["6",".",".","1","9","5",".",".","."],
        [".","9","8",".",".",".",".","6","."],
        ["8",".",".",".","6",".",".",".","3"],
        ["4",".",".","8",".","3",".",".","1"],
        ["7",".",".",".","2",".",".",".","6"],
        [".","6",".",".",".",".","2","8","."],
        [".",".",".","4","1","9",".",".","5"],
        [".",".",".",".","8",".",".","7","9"]
    ]
    print(isValidSudoku(board)) # Expected output: True
```

#48 MEDIUM

Rotate Image

LeetCodeCC · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Math · Matrix

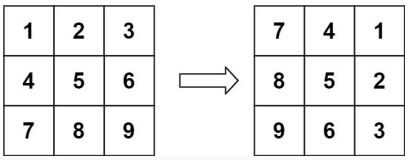
Lists: Grind 169 | CodeSignal

Problem:

You are given an n x n 2D matrix representing an image, rotate the image by 90 degrees (clockwise).

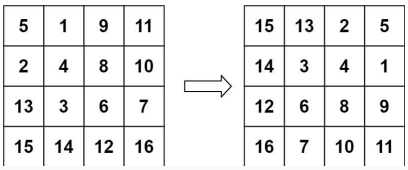
You have to rotate the image in-place, which means you have to modify the input 2D matrix directly. DO NOT allocate another 2D matrix and do the rotation.

Example 1:




```
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [[7,4,1],[8,5,2],[9,6,3]]
```

Example 2:




```
Input: matrix = [[5,1,9,11],[2,4,8,10],[13,3,6,7],[15,14,12,16]]
Output: [[15,13,2,5],[14,3,4,1],[12,6,8,9],[16,7,10,11]]
```

Constraints:

- n == matrix.length == matrix[i].length
- 1 <= n <= 20
- -1000 <= matrix[i][j] <= 1000

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def rotate(self, matrix):
        # Brute Force O(n^2) space - copy to new matrix with rotated indices
        n = len(matrix)
        copy = [row[:] for row in matrix]
        for r in range(n):
            for c in range(n):
                matrix[c][n-1-r] = copy[r][c]
```

Python

Python

```python
def rotate(matrix):
    n = len(matrix)
    # Transpose the matrix by swapping elements across the diagonal
    for i in range(n):
        for j in range(i, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
    # Reverse each row to complete the clockwise rotation
    for i in range(n):
        matrix[i].reverse()

# Example usage:
matrix = [[1,2,3],[4,5,6],[7,8,9]]
rotate(matrix)
print(matrix) # Outputs: [[7,4,1],[8,5,2],[9,6,3]]
```

Python


```python
class Solution:
    def rotate(self, matrix: list[list[int]]) -> None:
        matrix.reverse()
        for i, j in itertools.combinations(range(len(matrix)), 2):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
```

Python

```python
class Solution:
    def rotate(self, matrix: list[list[int]]) -> None:
        for mn in range(len(matrix) // 2):
            mx = len(matrix) - mn - 1
            for i in range(mn, mx):
                offset = i - mn
                top = matrix[mn][i]
                matrix[mn][i] = matrix[mx - offset][mn]
                matrix[mx - offset][mn] = matrix[mx][mx - offset]
                matrix[mx][mx - offset] = matrix[i][mx]
                matrix[i][mx] = top
```

Python

```python
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        n = len(matrix)
        for i in range(n >> 1):
            for j in range(n):
                matrix[i][j], matrix[n - i - 1][j] = matrix[n - i - 1][j], matrix[i][j]
        for i in range(n):
            for j in range(i):
                matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
```

Python

```python
class Solution(object):
    def rotate(self, matrix):
        """
        :type matrix: List[List[int]]
        :rtype: void Do not return anything, modify matrix in-place instead.
        """
        if len(matrix) == 0:
            return
        h = len(matrix)
        w = len(matrix[0])
        for i in range(0, h): # mirror
            for j in range(0, w / 2):
                matrix[i][j], matrix[i][w - j - 1] = matrix[i][w - j - 1], matrix[i][j]

        for i in range(0, h): # transpos
            for j in range(0, w - 1 - i):
                matrix[i][j], matrix[w - 1 - j][h - 1 - i] = matrix[w - 1 - j][h - 1 - i], matrix[i][j]


###########

class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        n = len(matrix)
        for i in range(n >> 1):
```

```
        for j in range(n):
            matrix[i][j], matrix[n - i - 1][j] = matrix[n - i - 1][j], matrix[i][j]
    for i in range(n):
        for j in range(i):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
```

[*] Matrix — Pattern Solution (stored optimal)

Python

```python
def rotate(matrix):
    n = len(matrix)
    # Transpose the matrix by swapping elements across the diagonal
    for i in range(n):
        for j in range(i, n):
            matrix[i][j], matrix[j][i] = matrix[j][i], matrix[i][j]
    # Reverse each row to complete the clockwise rotation
    for i in range(n):
        matrix[i].reverse()

# Example usage:
matrix = [[1,2,3],[4,5,6],[7,8,9]]
rotate(matrix)
print(matrix) # Outputs: [[7,4,1],[8,5,2],[9,6,3]]
```

#54 MEDIUM

Spiral Matrix

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

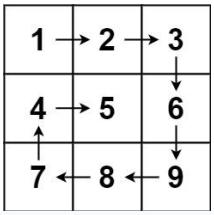
Array · Matrix · Simulation

Lists: Grind 169 | CodeSignal

Problem:

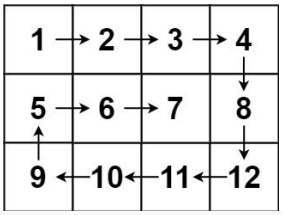
Given an m x n matrix, return all elements of the matrix in spiral order.

Example 1:




```
Input: matrix = [[1,2,3],[4,5,6],[7,8,9]]
Output: [1,2,3,6,9,8,7,4,5]
```

Example 2:




```
Input: matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]
Output: [1,2,3,4,8,12,11,10,9,5,6,7]
```

Constraints:

- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 10
- -100 <= matrix[i][j] <= 100

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def spiralOrder(self, matrix):
        # Brute Force O(m*n) - track visited cells, simulate turning
        if not matrix: return []
        m, n = len(matrix), len(matrix[0])
        directions = [(0,1),(1,0),(0,-1),(-1,0)] # right down left up
        visited = [[False]*n for _ in range(m)]
        r=c=d=0; result=[]
        for _ in range(m*n):
            result.append(matrix[r][c])
            visited[r][c]=True
            dr,dc=directions[d]
            nr,nc=r+dr,c+dc
            if 0<=nr<m and 0<=nc<m and not visited[nr][nc]:
                r,c=nr,nc
            else:
                d=(d+1)%4; dr,dc=directions[d]; r+=dr; c+=dc
        return result
```

Python

Python

```python
def spiralOrder(matrix):
    # Initialize the boundaries of the matrix
    top, bottom = 0, len(matrix) - 1
    left, right = 0, len(matrix[0]) - 1
    result = [] # This will store the spiral order

    # Loop until the pointers cross each other
    while top <= bottom and left <= right:
        # Traverse from left to right on the current top row
        for col in range(left, right + 1):
            result.append(matrix[top][col])
        top += 1 # Move the top boundary down

        # Traverse downwards on the current right column
        for row in range(top, bottom + 1):
            result.append(matrix[row][right])
        right -= 1 # Move the right boundary to the left

        # Traverse from right to left on the current bottom row, if valid
        if top <= bottom:
            for col in range(right, left - 1, -1):
                result.append(matrix[bottom][col])
            bottom -= 1 # Move the bottom boundary up

        # Traverse upwards on the current left column, if valid
```



```
        if left <= right:
            for row in range(bottom, top - 1, -1):
                result.append(matrix[row][left])
            left += 1 # Move the left boundary right

    return result

# Example usage:
# print(spiralOrder([[1,2,3],[4,5,6],[7,8,9]]))
```

Python

```python
class Solution:
  def spiralOrder(self, matrix: list[list[int]]) -> list[int]:
    if not matrix:
      return []

    m = len(matrix)
    n = len(matrix[0])
    ans = []
    r1 = 0
    c1 = 0
    r2 = m - 1
    c2 = n - 1

    # Repeatedly add matrix[r1..r2][c1..c2] to `ans`.
    while len(ans) < m * n:
      j = c1
      while j <= c2 and len(ans) < m * n:
        ans.append(matrix[r1][j])
        j += 1
      i = r1 + 1
      while i <= r2 - 1 and len(ans) < m * n:
        ans.append(matrix[i][c2])
        i += 1
      j = c2
      while j >= c1 and len(ans) < m * n:
        ans.append(matrix[r2][j])
        j -= 1
      i = r2 - 1
      while i >= r1 + 1 and len(ans) < m * n:
        ans.append(matrix[i][c1])
        i -= 1
      r1 += 1
      c1 += 1
      r2 -= 1
      c2 -= 1

    return ans
```

Python


```
class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        m, n = len(matrix), len(matrix[0])
        dirs = [0, 1, 0, -1, 0]
        vis = [[False] * n for _ in range(m)]
        i = j = k = 0
        ans = []
        for _ in range(m * n):
            ans.append(matrix[i][j])
            vis[i][j] = True
            x, y = i + dirs[k], j + dirs[k + 1]
            if x < 0 or x >= m or y < 0 or y >= n or vis[x][y]:
                k = (k + 1) % 4
            i += dirs[k]
            j += dirs[k + 1]
        return ans
```

Python

```
class Solution:
    def spiralOrder(self, matrix: List[List[int]]) -> List[int]:
        m, n = len(matrix), len(matrix[0])
        dirs = (0, 1, 0, -1, 0)
        i = j = k = 0
        ans = []
        vis = set()
        for _ in range(m * n):
            ans.append(matrix[i][j])
            vis.add((i, j))
            x, y = i + dirs[k], j + dirs[k + 1]
            if not 0 <= x < m or not 0 <= y < n or (x, y) in vis:
                k = (k + 1) % 4
            i = i + dirs[k]
            j = j + dirs[k + 1]
        return ans
```

[*] Matrix — Pattern Solution (stored optimal)

Python


```
def spiralOrder(matrix):
    # Initialize the boundaries of the matrix
    top, bottom = 0, len(matrix) - 1
    left, right = 0, len(matrix[0]) - 1
    result = [] # This will store the spiral order

    # Loop until the pointers cross each other
    while top <= bottom and left <= right:
        # Traverse from left to right on the current top row
        for col in range(left, right + 1):
            result.append(matrix[top][col])
        top += 1 # Move the top boundary down

        # Traverse downwards on the current right column
        for row in range(top, bottom + 1):
            result.append(matrix[row][right])
        right -= 1 # Move the right boundary to the left

        # Traverse from right to left on the current bottom row, if valid
        if top <= bottom:
            for col in range(right, left - 1, -1):
                result.append(matrix[bottom][col])
            bottom -= 1 # Move the bottom boundary up

        # Traverse upwards on the current left column, if valid
```

```
        if left <= right:
            for row in range(bottom, top - 1, -1):
                result.append(matrix[row][left])
            left += 1 # Move the left boundary right

    return result

# Example usage:
# print(spiralOrder([[1,2,3],[4,5,6],[7,8,9]]))
```

#59 MEDIUM

Spiral Matrix II

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

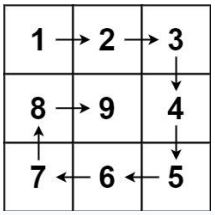
Array · Matrix · Simulation

Lists: CodeSignal

Problem:

Given a positive integer n, generate an n x n matrix filled with elements from 1 to n2 in spiral order.

Example 1:




```
Input: n = 3
Output: [[1,2,3],[8,9,4],[7,6,5]]
```

Example 2:

```
Input: n = 1
Output: [[1]]
```

Constraints:

- 1 <= n <= 20

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def generateMatrix(self, n):
        # Brute Force O(n^2·n^2)  flood fill from every cell
        m, n = len(n), len(n[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python


```python
# Function to generate spiral matrix of size n x n
def generateMatrix(n):
    # Initialize an n x n matrix with zeros
    matrix = [[0] * n for _ in range(n)]
    # Initialize the starting number to fill
    num = 1
    # Set the initial boundaries for rows and columns
    top, bottom = 0, n - 1
    left, right = 0, n - 1

    # Continue the loop until the boundaries are valid
    while top <= bottom and left <= right:
        # Traverse from left to right along the top row
        for j in range(left, right + 1):
            matrix[top][j] = num
            num += 1
        top += 1 # Move the top boundary down

        # Traverse from top to bottom along the right column
        for i in range(top, bottom + 1):
            matrix[i][right] = num
            num += 1
        right -= 1 # Move the right boundary left

        # Traverse from right to left along the bottom row if still in valid bounds

        if top <= bottom:
            for j in range(right, left - 1, -1):
                matrix[bottom][j] = num
                num += 1
            bottom -= 1 # Move the bottom boundary up

        # Traverse from bottom to top along the left column if still in valid bounds
        if left <= right:
            for i in range(bottom, top - 1, -1):
                matrix[i][left] = num
                num += 1
            left += 1 # Move the left boundary right

    return matrix

# Example usage:
print(generateMatrix(3))
```

Python


```python
class Solution:
    def generateMatrix(self, n: int) -> list[list[int]]:
        ans = [[0] * n for _ in range(n)]
        count = 1

        for mn in range(n // 2):
            mx = n - mn - 1
            for i in range(mn, mx):
                ans[mn][i] = count
                count += 1
            for i in range(mn, mx):
                ans[i][mx] = count
                count += 1
            for i in range(mx, mn, -1):
                ans[mx][i] = count
                count += 1
            for i in range(mx, mn, -1):
                ans[i][mn] = count
                count += 1

        if n % 2 == 1:
            ans[n // 2][n // 2] = count

        return ans
```

Python

```python
class Solution:
    def generateMatrix(self, n: int) -> List[List[int]]:
        ans = [[0] * n for _ in range(n)]
        dirs = (0, 1, 0, -1, 0)
        i = j = k = 0
        for v in range(1, n * n + 1):
            ans[i][j] = v
            x, y = i + dirs[k], j + dirs[k + 1]
            if x < 0 or x >= n or y < 0 or y >= n or ans[x][y]:
                k = (k + 1) % 4
            i, j = i + dirs[k], j + dirs[k + 1]
        return ans
```

```python
class Solution:
    def generateMatrix(self, n: int) -> List[List[int]]:
        ans = [[0] * n for _ in range(n)]
        dirs = ((0, 1), (1, 0), (0, -1), (-1, 0))
        i = j = k = 0
        for v in range(1, n * n + 1):
            ans[i][j] = v
            x, y = i + dirs[k][0], j + dirs[k][1]
            if x < 0 or y < 0 or x >= n or y >= n or ans[x][y]:
                k = (k + 1) % 4
                x, y = i + dirs[k][0], j + dirs[k][1]
            i, j = x, y
        return ans
```

[*] Matrix — Pattern Solution (stored optimal)

Python


```python
# Function to generate spiral matrix of size n x n
def generateMatrix(n):
    # Initialize an n x n matrix with zeros
    matrix = [[0] * n for _ in range(n)]
    # Initialize the starting number to fill
    num = 1
    # Set the initial boundaries for rows and columns
    top, bottom = 0, n - 1
    left, right = 0, n - 1

    # Continue the loop until the boundaries are valid
    while top <= bottom and left <= right:
        # Traverse from left to right along the top row
        for j in range(left, right + 1):
            matrix[top][j] = num
            num += 1
        top += 1 # Move the top boundary down

        # Traverse from top to bottom along the right column
        for i in range(top, bottom + 1):
            matrix[i][right] = num
            num += 1
        right -= 1 # Move the right boundary left

        # Traverse from right to left along the bottom row if still in valid bounds

        if top <= bottom:
            for j in range(right, left - 1, -1):
                matrix[bottom][j] = num
                num += 1
            bottom -= 1 # Move the bottom boundary up

        # Traverse from bottom to top along the left column if still in valid bounds
        if left <= right:
            for i in range(bottom, top - 1, -1):
                matrix[i][left] = num
                num += 1
            left += 1 # Move the left boundary right

    return matrix

# Example usage:
print(generateMatrix(3))
```

#64 MEDIUM

Minimum Path Sum

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming · Matrix

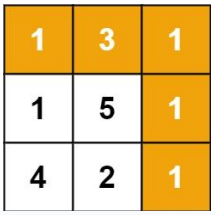
Lists: Denny Zhang

Problem:

Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example 1:




```
Input: grid = [[1,3,1],[1,5,1],[4,2,1]]
Output: 7
Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.
```

Example 2:

```
Input: grid = [[1,2,3],[4,5,6]]
Output: 12
```

Constraints:

- m == grid.length
- n == grid[j].length
- 1 <= m, n <= 200
- 0 <= grid[i][j] <= 200

>> Brute Force [Matrix] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def minPathSum(self, grid):
        # Brute Force O(m^2·n^2) - flood fill from every cell
        m, n = len(grid), len(grid[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python

```python
# Python solution using dynamic programming
def minPathSum(grid):
    # Number of rows and columns in the grid
    rows = len(grid)
    cols = len(grid[0])

    # Initialize the DP table directly in grid to save space
    # Fill the first row by accumulating the sum from the left
    for j in range(1, cols):
        grid[0][j] += grid[0][j-1]

    # Fill the first column by accumulating the sum above
    for i in range(1, rows):
        grid[i][0] += grid[i-1][0]

    # Compute the minimum path sum for the rest of the cells
    for i in range(1, rows):
        for j in range(1, cols):
            # The cell value can be added to the minimum of the top or left cell
            grid[i][j] += min(grid[i-1][j], grid[i][j-1])

    # The bottom-right cell contains the answer
    return grid[rows-1][cols-1]

# Example usage

example_grid = [[1,3,1],[1,5,1],[4,2,1]]
print(minPathSum(example_grid)) # Output should be 7
```

Python

```python
class Solution:
    def minPathSum(self, grid: list[list[int]]) -> int:
        m = len(grid)
        n = len(grid[0])

        for i in range(m):
            for j in range(n):
                if i > 0 and j > 0:
                    grid[i][j] += min(grid[i - 1][j], grid[i][j - 1])
                elif i > 0:
                    grid[i][0] += grid[i - 1][0]
                elif j > 0:
                    grid[0][j] += grid[0][j - 1]

        return grid[m - 1][n - 1]
```

Python

```python
class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        f = [[0] * n for _ in range(m)]
        f[0][0] = grid[0][0]
        for i in range(1, m):
            f[i][0] = f[i - 1][0] + grid[i][0]
        for j in range(1, n):
            f[0][j] = f[0][j - 1] + grid[0][j]
        for i in range(1, m):
            for j in range(1, n):
                f[i][j] = min(f[i - 1][j], f[i][j - 1]) + grid[i][j]
        return f[-1][-1]
```

Python

```python
class Solution:
    def minPathSum(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        f = [[0] * n for _ in range(m)]
        f[0][0] = grid[0][0]
        for i in range(1, m):
            f[i][0] = f[i - 1][0] + grid[i][0]
        for j in range(1, n):
            f[0][j] = f[0][j - 1] + grid[0][j]
        for i in range(1, m):
            for j in range(1, n):
                f[i][j] = min(f[i - 1][j], f[i][j - 1]) + grid[i][j]
        return f[-1][-1]
```

[*] Matrix — Pattern Solution (stored optimal)

Python


```python
# Python solution using dynamic programming
def minPathSum(grid):
    # Number of rows and columns in the grid
    rows = len(grid)
    cols = len(grid[0])

    # Initialize the DP table directly in grid to save space
    # Fill the first row by accumulating the sum from the left
    for j in range(1, cols):
        grid[0][j] += grid[0][j-1]

    # Fill the first column by accumulating the sum above
    for i in range(1, rows):
        grid[i][0] += grid[i-1][0]

    # Compute the minimum path sum for the rest of the cells
    for i in range(1, rows):
        for j in range(1, cols):
            # The cell value is added to the minimum of the top or left cell
            grid[i][j] += min(grid[i-1][j], grid[i][j-1])

    # The bottom-right cell contains the answer
    return grid[rows-1][cols-1]

# Example usage

example_grid = [[1,3,1],[1,5,1],[4,2,1]]
print(minPathSum(example_grid)) # Output should be 7
```

#73 MEDIUM

Set Matrix Zeroes

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Matrix

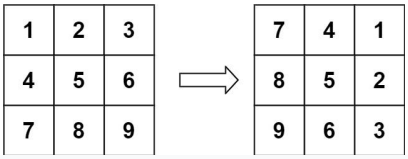
Lists: Grind 169

Problem:

Given an m x n integer matrix matrix, if an element is 0, set its entire row and column to 0's.

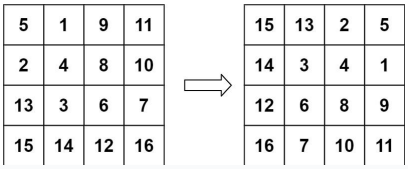
You must do it in place.

Example 1:




```
Input: matrix = [[1,1,1],[1,0,1],[1,1,1]]
Output: [[1,0,1],[0,0,0],[1,0,1]]
```

Example 2:




```
Input: matrix = [[0,1,2,0],[3,4,5,2],[1,3,1,5]]
Output: [[0,0,0,0],[0,4,5,0],[0,3,1,0]]
```

Constraints:

- m == matrix.length
- n == matrix[0].length
- 1 <= m, n <= 200
- -231 <= matrix[i][j] <= 231 - 1

Follow up:

- A straightforward solution using O(mn) space is probably a bad idea.
- A simple improvement uses O(m + n) space, but still not the best solution.
- Could you devise a constant space solution?

>> Brute Force [Matrix] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def setZeroes(self, matrix):
        # Brute force O(m*n (m+n)) - record zero positions, then zero out rows/cols
        zeros = [(r,c) for r in range(len(matrix)) for c in range(len(matrix[0]))
                 if matrix[r][c]==0]
        for r,c in zeros:
            for col in range(len(matrix[0])): matrix[r][col]=0
            for row in range(len(matrix)): matrix[row][c]=0
```

Python

Python

```python
# Python solution for the Set Matrix Zeroes problem

def setZeroes(matrix):
    # Step 1: Determine the dimensions of matrix
    m = len(matrix)
    n = len(matrix[0])

    # Determine if first row or first column should be zeroed
    first_row_zero = any(matrix[0][j] == 0 for j in range(n))
    first_col_zero = any(matrix[i][0] == 0 for i in range(m))

    # Step 2: Use first row and column as markers
    for i in range(1, m):
        for j in range(1, n):
            if matrix[i][j] == 0:
                matrix[i][0] = 0 # mark the first element of the row
                matrix[0][j] = 0 # mark the first element of the column

    # Step 3: Zero out cells based on the markers in first row and column
    for i in range(1, m):
        for j in range(1, n):
            if matrix[i][0] == 0 or matrix[0][j] == 0:
                matrix[i][j] = 0

    # Step 4: Zero out the first row if needed

    if first_row_zero:
        for j in range(n):
            matrix[0][j] = 0

    # Step 5: Zero out the first column if needed
    if first_col_zero:
        for i in range(m):
            matrix[i][0] = 0

# Example usage:
# matrix = [[1,1,1],[1,0,1],[1,1,1]]
# setZeroes(matrix)
# print(matrix)
```

Python


```python
class Solution:
    def setZeroes(self, matrix: list[list[int]]) -> None:
        m = len(matrix)
        n = len(matrix[0])
        shouldFillFirstRow = 0 in matrix[0]
        shouldFillFirstCol = 0 in list(zip(*matrix))[0]

        # Store the information in the first row and the first column.
        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][j] == 0:
                    matrix[i][0] = 0
                    matrix[0][j] = 0

        # Fill 0s for the matrix except the first row and the first column.
        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][0] == 0 or matrix[0][j] == 0:
                    matrix[i][j] = 0

        # Fill 0s for the first row if needed.
        if shouldFillFirstRow:
            matrix[0] = [0] * n

        # Fill 0s for the first column if needed.

        if shouldFillFirstCol:
            for row in matrix:
                row[0] = 0
```

Python

```python
class Solution:
    def setZeroes(self, matrix: List[List[int]]) -> None:
        m, n = len(matrix), len(matrix[0])
        row = [False] * m
        col = [False] * n
        for i in range(m):
            for j in range(n):
                if matrix[i][j] == 0:
                    row[i] = col[j] = True
        for i in range(m):
            for j in range(n):
                if row[i] or col[j]:
                    matrix[i][j] = 0
```

Python


```python
class Solution:
    def setZeroes(self, matrix: List|List[int]]) -> None:
        m, n = len(matrix), len(matrix[0])
        i0 = any(v == 0 for v in matrix[0])
        j0 = any(matrix[i][0] == 0 for i in range(m))
        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][j] == 0:
                    matrix[i][0] = matrix[0][j] = 0
        for i in range(1, m):
            for j in range(1, n):
                if matrix[i][0] == 0 or matrix[0][j] == 0:
                    matrix[i][j] = 0
        if i0:
            for j in range(m):
                matrix[0][j] = 0
        if j0:
            for i in range(m):
                matrix[i][0] = 0
```

[*] Matrix — Pattern Solution (stored optimal)

Python

```python
# Python solution for the Set Matrix Zeroes problem

def setZeroes(matrix):
    # Step 1: Determine the dimensions of matrix
    m = len(matrix)
    n = len(matrix[0])

    # Determine if first row or first column should be zeroed
    first_row_zero = any(matrix[0][j] == 0 for j in range(n))
    first_col_zero = any(matrix[i][0] == 0 for i in range(m))

    # Step 2: Use first row and column as markers
    for i in range(1, m):
        for j in range(1, n):
            if matrix[i][j] == 0:
                matrix[i][0] = 0 # mark the first element of the row
                matrix[0][j] = 0 # mark the first element of the column

    # Step 3: Zero out cells based on the markers in first row and column
    for i in range(1, m):
        for j in range(1, n):
            if matrix[i][0] == 0 or matrix[0][j] == 0:
                matrix[i][j] = 0

    # Step 4: Zero out the first row if needed
```

```
    if first_row_zero:
        for j in range(n):
            matrix[0][j] = 0

    # Step 5: Zero out the first column if needed
    if first_col_zero:
        for i in range(m):
            matrix[i][0] = 0

# Example usage:
# matrix = [[1,1,1],[1,0,1],[1,1,1]]
# setZeroes(matrix)
# print(matrix)
```

#74 MEDIUM

Search a 2D Matrix

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search · Matrix

Lists: Grind 169

Problem:

You are given an m x n integer matrix matrix with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Given an integer target, return true if target is in matrix or false otherwise.

You must write a solution in O(log(m * n)) time complexity.

Example 1:

1	3	5	7
10	11	16	20
23	30	34	60

```
Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3
Output: true
```


Example 2:

5	1	9	11
2	4	8	10
13	3	6	7
15	14	12	16

⇒

15	13	2	5
14	3	4	1
12	6	8	9
16	7	10	11

```
Input: matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 13
Output: false
```

Constraints:

- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 100
- -104 <= matrix[i][j], target <= 104

>> Brute Force [Matrix] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def searchMatrix(self, matrix, target):
        # Brute force O(m*n) - scan every cell
        for row in matrix:
            for val in row:
                if val == target:
                    return True
        return False
```

Python

Python


```
def searchMatrix(matrix, target):
    # Get number of rows and columns
    if not matrix or not matrix[0]:
        return False

    rows, cols = len(matrix), len(matrix[0])
    left, right = 0, rows * cols - 1

    # Perform binary search over the virtual flattened array
    while left <= right:
        # Find the middle index
        mid = (left + right) // 2
        # Map the middle index to row and column in the matrix
        row = mid // cols
        col = mid % cols
        mid_value = matrix[row][col]

        # Check if the mid_value equals the target
        if mid_value == target:
            return True
        elif mid_value < target:
            left = mid + 1 # Move to the right subarray
        else:
            right = mid - 1 # Move to the left subarray

    # Target not found
    return False

# Example usage:
# matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]
# target = 3
# print(searchMatrix(matrix, target))
```

Python

```
class Solution:
    def searchMatrix(self, matrix: list[list[int]], target: int) -> bool:
        if not matrix:
            return False

        m = len(matrix)
        n = len(matrix[0])
        l = 0
        r = m * n

        while l < r:
            mid = (l + r) // 2
            i = mid // n
            j = mid % n
            if matrix[i][j] == target:
                return True
            if matrix[i][j] < target:
                l = mid + 1
            else:
                r = mid

        return False
```

Python


```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        m, n = len(matrix), len(matrix[0])
        left, right = 0, m * n - 1
        while left < right:
            mid = (left + right) >> 1
            x, y = divmod(mid, n)
            if matrix[x][y] >= target:
                right = mid
            else:
                left = mid + 1
        return matrix[left // n][left % n] == target
```

Python

```
class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        m, n = len(matrix), len(matrix[0])
        left, right = 0, m * n - 1
        while left <= right: # must be <=, not <, for matrix=[[1]],target=1
            mid = (left + right) >> 1
            x, y = divmod(mid, n) # note: divide column count
            if matrix[x][y] == target:
                return True
            elif matrix[x][y] > target:
                right = mid - 1
            else:
                left = mid + 1
        return False

##########

class Solution:
    def searchMatrix(self, matrix: List[List[int]], target: int) -> bool:
        m, n = len(matrix), len(matrix[0])
        left, right = 0, m * n - 1
        while left < right:
            mid = (left + right) >> 1
            x, y = divmod(mid, n)
            if matrix[x][y] >= target:

                right = mid
            else:
                left = mid + 1
        return matrix[left // n][left % n] == target
```

[*] Matrix — Pattern Solution (matched from community answers)

Python


```
def searchMatrix(matrix, target):
    # Get number of rows and columns
    if not matrix or not matrix[0]:
        return False

    rows, cols = len(matrix), len(matrix[0])
    left, right = 0, rows * cols - 1

    # Perform binary search over the virtual flattened array
    while left <= right:
        # Find the middle index
        mid = (left + right) // 2
        # Map the middle index to row and column in the matrix
        row = mid // cols
        col = mid % cols
        mid_value = matrix[row][col]

        # Check if the mid_value equals the target
        if mid_value == target:
            return True
        elif mid_value < target:
            left = mid + 1 # Move to the right subarray
        else:
            right = mid - 1 # Move to the left subarray

    # Target not found
    return False

# Example usage:
# matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]
# target = 3
# print(searchMatrix(matrix, target))
```

#221 MEDIUM

Maximal Square

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming · Matrix

Lists: Grind 169

Problem:

Given an m x n binary matrix filled with 0's and 1's, find the largest square containing only 1's and return its area.

Example 1:

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

```
Input: matrix =
[["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]
Output: 4
```

Example 2:

0	1
1	0

```
Input: matrix = [["0","1"],["1","0"]]
Output: 1
```

Example 3:

```
Input: matrix = [["0"]]
Output: 0
```

Constraints:

- m == matrix.length
- n == matrix[i].length

- 1 <= m, n <= 300
- matrix[i][j] is '0' or '1'.

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def maximalSquare(self, matrix):
        # Brute Force O(m^3·n^2) - flood fill from every cell
        m, n = len(matrix), len(matrix[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python

```python
# Python implementation of the Maximal Square problem

def maximalSquare(matrix):
    # Check if matrix is empty
    if not matrix or not matrix[0]:
        return 0
    rows = len(matrix)
    cols = len(matrix[0])
    # DP table with extra row and col for ease of boundary conditions
    dp = [[0] * (cols + 1) for _ in range(rows + 1)]
    max_side = 0
    # Traverse through each cell in the matrix
    for i in range(1, rows + 1):
        for j in range(1, cols + 1):
            # If the current cell is '1'
            if matrix[i - 1][j - 1] == '1':
                # Compute the minimum value among the top, left, and top-left neighbors and add 1
                dp[i][j] = min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]) + 1
                max_side = max(max_side, dp[i][j])
    # The area of the square is the square of the side length
    return max_side * max_side

# Example usage:
matrix = [
    ["1", "0", "1", "0", "0"],
```

```python
    ["1", "0", "1", "1", "1"],
    ["1", "1", "1", "1", "1"],
    ["1", "0", "0", "1", "0"]
]
print(maximalSquare(matrix))
```


Python

```python
class Solution:
    def maximalSquare(self, matrix: list[list[str]]) -> int:
        m = len(matrix)
        n = len(matrix[0])
        dp = [[0] * n for _ in range(m)]
        maxLength = 0

        for i in range(m):
          for j in range(n):
            if i == 0 or j == 0 or matrix[i][j] == '0':
              dp[i][j] = 1 if matrix[i][j] == '1' else 0
            else:
              dp[i][j] = min(dp[i - 1][j - 1], dp[i - 1]
                             [j], dp[i][j - 1]) + 1
            maxLength = max(maxLength, dp[i][j])

        return maxLength * maxLength
```

Python

```python
class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        m, n = len(matrix), len(matrix[0])
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        mx = 0
        for i in range(m):
            for j in range(n):
                if matrix[i][j] == '1':
                    dp[i + 1][j + 1] = min(dp[i][j + 1], dp[i + 1][j], dp[i][j]) + 1
                    mx = max(mx, dp[i + 1][j + 1])
        return mx * mx
```

Python

```
'''
eg. a 10*10 square with full of 1s
    this square move 1 line down
    this square move 1 line right

    so there needs an extra single 1 at bottom right, to make it a larger full square
'''
```

```python
class Solution:
    def maximalSquare(self, matrix: List[List[str]]) -> int:
        m, n = len(matrix), len(matrix[0])
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        mx = 0
        for i in range(m):
            for j in range(n):
                if matrix[i][j] == '1':
                    dp[i + 1][j + 1] = min(dp[i][j + 1], dp[i + 1][j], dp[i][j]) + 1
                    mx = max(mx, dp[i + 1][j + 1])
        return mx * mx
```

[*] Matrix — Pattern Solution (stored optimal)

Python


```python
# Python implementation of the Maximal Square problem

def maximalSquare(matrix):
    # Check if matrix is empty
    if not matrix or not matrix[0]:
        return 0
    rows = len(matrix)
    cols = len(matrix[0])
    # DP table with extra row and col for ease of boundary conditions
    dp = [[0] * (cols + 1) for _ in range(rows + 1)]
    max_side = 0
    # Traverse through each cell in the matrix
    for i in range(1, rows + 1):
        for j in range(1, cols + 1):
            # If the current cell is '1'
            if matrix[i - 1][j - 1] == '1':
                # Compute the minimum value among the top, left, and top-left neighbors and add 1
                dp[i][j] = min(dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]) + 1
                max_side = max(max_side, dp[i][j])
    # The area of the square is the square of the side length
    return max_side * max_side

# Example usage:
matrix = [
    ["1", "0", "1", "0", "0"],
```

```python
    ["1", "0", "1", "1", "1"],
    ["1", "1", "1", "1", "1"],
    ["1", "0", "0", "1", "0"]
]
print(maximalSquare(matrix))
```


#311 MEDIUM

Sparse Matrix Multiplication

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Matrix

Lists: Premium 98

Problem:

Given two sparse matrices mat1 of size m x k and mat2 of size k x n, return the result of mat1 x mat2.

You may assume that multiplication is always possible.

Example 1:

1	0	0
-1	0	3

x

7	0	0
0	0	0
0	0	1

=

7	0	0
-7	0	3

```
Input: mat1 = [[1,0,0],[-1,0,3]], mat2 = [[7,0,0],[0,0,0],[0,0,1]]
Output: [[7,0,0],[-7,0,3]]
```

Example 2:

```
Input: mat1 = [[0]], mat2 = [[0]]
Output: [[0]]
```

Constraints:

- m == mat1.length
- k == mat1[i].length == mat2.length
- n == mat2[i].length
- 1 <= m, n, k <= 100
- -100 <= mat1[i][j], mat2[i][j] <= 100

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def sparseMatrixMultiplication(self, mat1, mat2):
        # Brute Force O(m^2·n^2) - flood fill from every cell
        m, n = len(mat1), len(mat1[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python


```python
def sparseMatrixMultiplication(mat1, mat2):
    # Get dimensions of mat1 and mat2
    m, k = len(mat1), len(mat1[0])
    n = len(mat2[0])
    # Initialize result matrix with zeros
    result = [[0] * n for _ in range(m)]

    # Iterate through each row in mat1
    for i in range(m):
        # For each element in row i of mat1
        for p in range(k):
            if mat1[i][p] != 0: # Only consider non-zero element in mat1
                # Multiply with corresponding elements in the row of mat2
                for j in range(n):
                    if mat2[p][j] != 0: # Only consider non-zero element in mat2
                        result[i][j] += mat1[i][p] * mat2[p][j]

    return result

# Example usage:
print(sparseMatrixMultiplication([[1,0,0],[-1,0,3]], [[7,0,0],[0,0,0],[0,0,1]]))
```

Python

```python
class Solution:
    def multiply(self, mat1: list[list[int]],
                 mat2: list[list[int]]) -> list[list[int]]:

        m = len(mat1)
        n = len(mat2)
        l = len(mat2[0])
        ans = [[0] * l for _ in range(m)]

        for i in range(m):
          for j in range(l):
            for k in range(n):
              ans[i][j] += mat1[i][k] * mat2[k][j]

        return ans
```

Python

```python
class Solution:
    def multiply(self, mat1: List[List[int]], mat2: List[List[int]]) -> List[List[int]]:
        m, n = len(mat1), len(mat2[0])
        ans = [[0] * n for _ in range(m)]
        for i in range(m):
            for j in range(n):
                for k in range(len(mat2)):
                    ans[i][j] += mat1[i][k] * mat2[k][j]
        return ans
```

Python


```python
class Solution:
    def multiply(self, mat1: List[List[int]], mat2: List[List[int]]) -> List[List[int]]:
        def f(mat: List[List[int]]) -> List[List[int]]:
            g = [[] for _ in range(len(mat))]
            for i, row in enumerate(mat):
                for j, x in enumerate(row):
                    if x:
                        g[i].append((j, x))
            return g

        g1 = f(mat1)
        g2 = f(mat2)
        m, n = len(mat1), len(mat2[0])
        ans = [[0] * n for _ in range(m)]
        for i in range(m):
            for k, x in g1[i]:
                for j, y in g2[k]:
                    ans[i][j] += x * y
        return ans
```

[*] Matrix — Pattern Solution (stored optimal)

Python


```python
def sparseMatrixMultiplication(mat1, mat2):
    # Get dimensions of mat1 and mat2
    m, k = len(mat1), len(mat1[0])
    n = len(mat2[0])
    # Initialize result matrix with zeros
    result = [[0] * n for _ in range(m)]

    # Iterate through each row in mat1
    for i in range(m):
        # For each element in row i of mat1
        for p in range(k):
            if mat1[i][p] != 0: # Only consider non-zero element in mat1
                # Multiply with corresponding elements in the row of mat2
                for j in range(n):
                    if mat2[p][j] != 0: # Only consider non-zero element in mat2
                        result[i][j] += mat1[i][p] * mat2[p][j]

    return result

# Example usage:
print(sparseMatrixMultiplication([[1,0,0],[-1,0,3]], [[7,0,0],[0,0,0],[0,0,1]]))
```


#498 MEDIUM

Diagonal Traverse

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

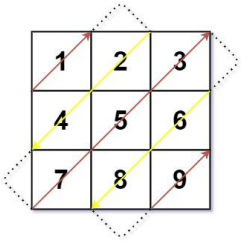
Array · Matrix · Simulation

Lists: CodeSignal

Problem:

Given an m x n matrix mat, return an array of all the elements of the array in a diagonal order.

Example 1:




```
Input: mat = [[1,2,3],[4,5,6],[7,8,9]]
Output: [1,2,4,7,5,3,6,8,9]
```

Example 2:

```
Input: mat = [[1,2],[3,4]]
Output: [1,2,3,4]
```

Constraints:

- m == mat.length
- n == mat[i].length
- 1 <= m, n <= 104
- 1 <= m * n <= 104
- -105 <= mat[i][j] <= 105

>> Brute Force [Matrix] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def findDiagonalOrder(self, mat):
        # Brute Force O(m^2·n^2) - flood fill from every cell
        m, n = len(mat), len(mat[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python

```python
def findDiagonalOrder(mat):
    if not mat or not mat[0]:
        return []

    m, n = len(mat), len(mat[0])
    result = []
    row, col = 0, 0
    direction = 1 # 1 means upward, -1 means downward

    while len(result) < m * n:
        result.append(mat[row][col])
        if direction == 1: # moving upward
            if col == n - 1: # Hit the right boundary
                row += 1
                direction = -1
            elif row == 0: # Hit the top boundary
                col += 1
                direction = -1
            else:
                row -= 1
                col += 1
        else: # moving downward
            if row == m - 1: # Hit the bottom boundary
                col += 1
                direction = 1

            elif col == 0: # Hit the left boundary
                row += 1
                direction = 1
            else:
                row += 1
                col -= 1
    return result

# Example usage:
print(findDiagonalOrder([[1,2,3],[4,5,6],[7,8,9]]))
```

Python


```python
class Solution:
    def findDiagonalOrder(self, mat: List[List[int]]) -> List[int]:
        m, n = len(mat), len(mat[0])
        ans = []
        for k in range(m + n - 1):
            t = []
            i = 0 if k < n else k - n + 1
            j = k if k < n else n - 1
            while i < m and j >= 0:
                t.append(mat[i][j])
                i += 1
                j -= 1
            if k % 2 == 0:
                t = t[::-1]
            ans.extend(t)
        return ans
```

Python

```python
class Solution:
    def findDiagonalOrder(self, mat: List[List[int]]) -> List[int]:
        m, n = len(mat), len(mat[0])
        ans = []
        for k in range(m + n - 1):
            t = []
            i = 0 if k < n else k - n + 1
            j = k if k < n else n - 1
            while i < m and j >= 0:
                t.append(mat[i][j])
                i += 1
                j -= 1
            if k % 2 == 0:
                t = t[::-1]
            ans.extend(t)
        return ans
```

[*] Matrix — Pattern Solution (stored optimal)

Python


```python
def findDiagonalOrder(mat):
    if not mat or not mat[0]:
        return []

    m, n = len(mat), len(mat[0])
    result = []
    row, col = 0, 0
    direction = 1 # 1 means upward, -1 means downward

    while len(result) < m * n:
        result.append(mat[row][col])
        if direction == 1: # moving upward
            if col == n - 1: # Hit the right boundary
                row += 1
                direction = -1
            elif row == 0: # Hit the top boundary
                col += 1
                direction = -1
            else:
                row -= 1
                col += 1
        else: # moving downward
            if row == m - 1: # Hit the bottom boundary
                col += 1
                direction = 1

            elif col == 0: # Hit the left boundary
                row += 1
                direction = 1
            else:
                row += 1
                col -= 1
    return result

# Example usage:
print(findDiagonalOrder([[1,2,3],[4,5,6],[7,8,9]]))
```

#531 MEDIUM

Lonely Pixel I

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Matrix

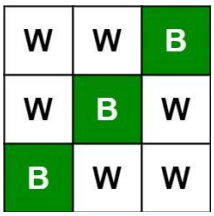
Lists: Premium 98

Problem:

Given an m x n picture consisting of black 'B' and white 'W' pixels, return the number of black lonely pixels.

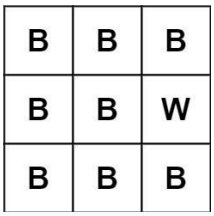
A black lonely pixel is a character 'B' that located at a specific position where the same row and same column don't have any other black pixels.

Example 1:




```
Input: picture = [["W","W","B"],["W","B","W"],["B","W","W"]]
Output: 3
Explanation: All the three 'B's are black lonely pixels.
```

Example 2:




```
Input: picture = [["B","B","B"],["B","B","W"],["B","B","B"]]
Output: 0
```

Constraints:

- m == picture.length
- n == picture[i].length
- 1 <= m, n <= 500

- picture[i][j] is 'W' or 'B'.

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findLonelyPixel(self, picture):
        # Brute Force O(m^2·n^2) - flood fill from every cell
        m, n = len(picture), len(picture[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python

```python
# Function to find the number of lonely black pixels in a picture.
def findLonelyPixel(picture):
    # Get dimensions of the picture
    rows = len(picture)
    cols = len(picture[0]) if rows > 0 else 0

    # Initialize row and column counts with zeros
    rowCount = [0] * rows
    colCount = [0] * cols

    # First pass: Count the number of 'B's in each row and column.
    for i in range(rows):
        for j in range(cols):
            if picture[i][j] == 'B':
                rowCount[i] += 1
                colCount[j] += 1

    # Second pass: Check for lonely black pixels.
    lonelyPixels = 0
    for i in range(rows):
        for j in range(cols):
            if picture[i][j] == 'B' and rowCount[i] == 1 and colCount[j] == 1:
                lonelyPixels += 1

    return lonelyPixels


# Example test
picture = [["W","W","B"], ["W","B","W"], ["B","W","W"]]
print(findLonelyPixel(picture)) # Output: 3
```

Python

```python
class Solution:
    def findLonelyPixel(self, picture: list[list[str]]) -> int:
        m = len(picture)
        n = len(picture[0])
        ans = 0
        rows = [0] * m # rows[i] := the number of B's in rows i
        cols = [0] * n # cols[i] := the number of B's in cols i

        for i in range(m):
          for j in range(n):
            if picture[i][j] == 'B':
              rows[i] += 1
              cols[j] += 1

        for i in range(m):
          if rows[i] == 1: # Only have to examine the rows if rows[i] == 1.
            for j in range(n):
              # After meeting a 'B' in this rows, break and search the next row.
              if picture[i][j] == 'B':
                if cols[j] == 1:
                  ans += 1
                break

        return ans
```

Python

```python
class Solution:
    def findLonelyPixel(self, picture: List[List[str]]) -> int:
        rows = [0] * len(picture)
        cols = [0] * len(picture[0])
        for i, row in enumerate(picture):
            for j, x in enumerate(row):
                if x == "B":
                    rows[i] += 1
                    cols[j] += 1
        ans = 0
        for i, row in enumerate(picture):
            for j, x in enumerate(row):
                if x == "B" and rows[i] == 1 and cols[j] == 1:
                    ans += 1
        return ans
```

Python


```python
class Solution:
    def findLonelyPixel(self, picture: List[List[str]]) -> int:
        m, n = len(picture), len(picture[0])
        rows, cols = [0] * m, [0] * n
        for i in range(m):
            for j in range(n):
                if picture[i][j] == 'B':
                    rows[i] += 1
                    cols[j] += 1
        res = 0
        for i in range(m):
            if rows[i] == 1:
                for j in range(n):
                    if picture[i][j] == 'B' and cols[j] == 1:
                        res += 1
                        break
        return res
```

[*] Matrix — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def findLonelyPixel(self, picture: List[List[str]]) -> int:
        m, n = len(picture), len(picture[0])
        rows, cols = [0] * m, [0] * n
        for i in range(m):
            for j in range(n):
                if picture[i][j] == 'B':
                    rows[i] += 1
                    cols[j] += 1
        res = 0
        for i in range(m):
            if rows[i] == 1:
                for j in range(n):
                    if picture[i][j] == 'B' and cols[j] == 1:
                        res += 1
                        break
        return res
```

#723 MEDIUM

Candy Crush

LeetCodeCx · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers · Matrix · Simulation

Lists: Premium 98

Problem:

This question is about implementing a basic elimination algorithm for Candy Crush.

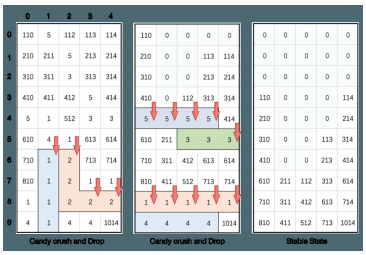
Given an m x n integer array board representing the grid of candy where board[i][j] represents the type of candy. A value of board[i][j] == 0 represents that the cell is empty.

The given board represents the state of the game following the player's move. Now, you need to restore the board to a stable state by crushing candies according to the following rules:

- If three or more candies of the same type are adjacent vertically or horizontally, crush them all at the same time – these positions become empty.
- After crushing all candies simultaneously, if an empty space on the board has candies on top of itself, then these candies will drop until they hit a candy or bottom at the same time. No new candies will drop outside the top boundary.
- After the above steps, there may exist more candies that can be crushed. If so, you need to repeat the above steps.
- If there does not exist more candies that can be crushed (i.e., the board is stable), then return the current board.

You need to perform the above rules until the board becomes stable, then return the stable board.

Example 1:




```
Input: board = [[110,5,112,113,114],[210,211,5,213,214],[310,311,3,313,314],[410,411,412,5,414],[5,1,512,3,3],[610,4,1,613,614],[710,1,2,713,714],[810,1,2,1,1],[1,1,2,2,2],[4,1,4,4,1014]]
Output: [[0,0,0,0,0],[0,0,0,0,0],[0,0,0,0,0],[110,0,0,0,114],[210,0,0,0,214],[310,0,0,113,314],[410,0,0,213,414],[610,211,112,313,614],[710,311,412,613,714],[810,411,512,713,1014]]
```

Example 2:

```
Input: board = [[1,3,5,5,2],[3,4,3,3,1],[3,2,4,5,2],[2,4,4,5,5],[1,4,4,1,1]]
Output: [[1,3,0,0,0],[3,4,0,5,2],[3,2,0,3,1],[2,4,0,5,2],[1,4,3,1,1]]
```

Constraints:

- m == board.length
- n == board[i].length
- 3 <= m, n <= 50
- 1 <= board[i][j] <= 2000

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def candyCrush(self, board):
        # Brute force start: O(n^2) - flood fill from every cell
        m, n = len(board), len(board[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python

```python
def candyCrush(board):
    # Retrieve board dimensions
    m, n = len(board), len(board[0])
    # Loop until no crushable candies remain
    while True:
        # Set to hold all positions that need to be crushed
        crush_set = set()

        # Check horizontally for crushable candies
        for i in range(m):
            for j in range(n - 2):
                # Check if candy exists and three consecutive candies are the same
                if board[i][j] != 0 and abs(board[i][j]) == abs(board[i][j+1]) == abs(board[i][j+2]):
                    crush_set |= {(i, j), (i, j+1), (i, j+2)}

        # Check vertically for crushable candies
        for i in range(m - 2):
            for j in range(n):
                if board[i][j] != 0 and abs(board[i][j]) == abs(board[i+1][j]) == abs(board[i+2][j]):
                    crush_set |= {(i, j), (i+1, j), (i+2, j)}

        # If no candies need to be crushed, the board is stable; break out of the loop.
        if not crush_set:
            break
```



```python
        # Set all marked positions to 0 (simulate crushing)
        for i, j in crush_set:
            board[i][j] = 0

        # Apply gravity: For each column, move uncrushed candies downwards
        for j in range(n):
            # Pointer for placing the next candy from bottom-up
            write_index = m - 1
            # Traverse each column from bottom to top
            for i in range(m - 1, -1, -1):
                if board[i][j] != 0:
                    board[write_index][j] = board[i][j]
                    write_index -= 1
            # Fill remaining positions with 0
            for i in range(write_index, -1, -1):
                board[i][j] = 0

    return board

# Example usage:
board = [[110,5,112,113,114],[210,211,5,213,214],[310,311,3,313,314],
         [410,411,412,5,414],[5,1,512,3,3],[610,4,1,613,614],
         [710,1,2,713,714],[810,1,2,1,1],[1,1,2,2,2],[4,1,4,4,1014]]
print(candyCrush(board))
```

Python

```python
class Solution:
    def candyCrush(self, board: List[List[int]]) -> List[List[int]]:
        m, n = len(board), len(board[0])
        run = True
        while run:
            run = False
            for i in range(m):
                for j in range(n - 2):
                    if (
                        board[i][j] != 0
                        and abs(board[i][j]) == abs(board[i][j + 1])
                        and abs(board[i][j]) == abs(board[i][j + 2])
                    ):
                        run = True
                        board[i][j] = board[i][j + 1] = board[i][j + 2] = -abs(
                            board[i][j]
                        )
            for j in range(n):
                for i in range(m - 2):
                    if (
                        board[i][j] != 0
                        and abs(board[i][j]) == abs(board[i + 1][j])
                        and abs(board[i][j]) == abs(board[i + 2][j])
                    ):
                        run = True
```



```python
                        board[i][j] = board[i + 1][j] = board[i + 2][j] = -abs(
                            board[i][j]
                        )
            if run:
                for j in range(n):
                    curr = m - 1
                    for i in range(m - 1, -1, -1):
                        if board[i][j] > 0:
                            board[curr][j] = board[i][j]
                            curr -= 1
                    while curr > -1:
                        board[curr][j] = 0
                        curr -= 1
        return board
```

[*] Matrix — Pattern Solution (stored optimal)

Python

```python
def candyCrush(board):
    # Retrieve board dimensions
    m, n = len(board), len(board[0])
    # Loop until no crushable candies remain
    while True:
        # Set to hold all positions that need to be crushed
        crush_set = set()

        # Check horizontally for crushable candies
        for i in range(m):
            for j in range(n - 2):
                # Check if candy exists and three consecutive candies are the same
                if board[i][j] != 0 and abs(board[i][j]) == abs(board[i][j+1]) == abs(board[i][j+2]):
                    crush_set |= {(i, j), (i, j+1), (i, j+2)}

        # Check vertically for crushable candies
        for i in range(m - 2):
            for j in range(n):
                if board[i][j] != 0 and abs(board[i][j]) == abs(board[i+1][j]) == abs(board[i+2][j]):
                    crush_set |= {(i, j), (i+1, j), (i+2, j)}

        # If no candies need to be crushed, the board is stable; break out of the loop.
        if not crush_set:
            break
```

```python
        # Set all marked positions to 0 (simulate crushing)
        for i, j in crush_set:
            board[i][j] = 0

        # Apply gravity: For each column, move uncrushed candies downwards
        for j in range(n):
            # Pointer for placing the next candy from bottom-up
            write_index = m - 1
            # Traverse each column from bottom to top
            for i in range(m - 1, -1, -1):
                if board[i][j] != 0:
                    board[write_index][j] = board[i][j]
                    write_index -= 1
            # Fill remaining positions with 0
            for i in range(write_index, -1, -1):
                board[i][j] = 0

    return board

# Example usage:
board = [[110,5,112,113,114],[210,211,5,213,214],[310,311,3,313,314],
         [410,411,412,5,414],[5,1,512,3,3],[610,4,1,613,614],
         [710,1,2,713,714],[810,1,2,1,1],[1,1,2,2,2],[4,1,4,4,1014]]
print(candyCrush(board))
```

#835 MEDIUM

Image Overlap

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Matrix

Lists: CodeSignal

Problem:

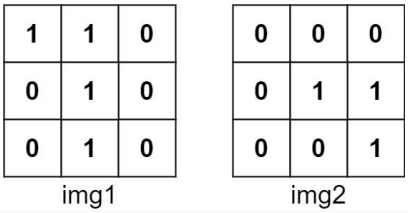
You are given two images, img1 and img2, represented as binary, square matrices of size n x n. A binary matrix has only 0s and 1s as values.

We translate one image however we choose by sliding all the 1 bits left, right, up, and/or down any number of units. We then place it on top of the other image. We can then calculate the overlap by counting the number of positions that have a 1 in both images.

Note also that a translation does not include any kind of rotation. Any 1 bits that are translated outside of the matrix borders are erased.

Return the largest possible overlap.

Example 1:




```
Input: img1 = [[1,1,0],[0,1,0],[0,1,0]], img2 = [[0,0,0],[0,1,1],[0,0,1]]
Output: 3
Explanation: We translate img1 to right by 1 unit and down by 1 unit.

The number of positions that have a 1 in both images is 3 (shown in red).
```

Example 2:

```
Input: img1 = [[1]], img2 = [[1]]
Output: 1
```

Example 3:

```
Input: img1 = [[0]], img2 = [[0]]
Output: 0
```

Constraints:

- n == img1.length == img1[i].length
- n == img2.length == img2[i].length
- 1 <= n <= 30
- img1[i][j] is either 0 or 1.
- img2[i][j] is either 0 or 1.

>> Brute Force [Matrix] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def largestOverlap(self, img1, img2):
        # Brute Force O(n^2)-n^2)) - flood fill from every cell
        m, n = len(img1), len(img1[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python

```python
def largestOverlap(img1, img2):
    # Get the dimension of the square images
    n = len(img1)

    # Gather positions (coordinates) where there is a 1 in img1 and img2
    ones_img1 = []
    ones_img2 = []
    for i in range(n):
        for j in range(n):
            if img1[i][j] == 1:
                ones_img1.append((i, j))
            if img2[i][j] == 1:
                ones_img2.append((i, j))

    # Dictionary to count translation shifts that align 1's of img1 with img2
    translation_count = {}
    max_overlap = 0

    # Iterate over every pair of 1's from both images
    for (x1, y1) in ones_img1:
        for (x2, y2) in ones_img2:
            # Calculate the translation vector needed to align img1's 1 with img2's 1
            delta = (x2 - x1, y2 - y1)
            # Update the count for this translation vector
            translation_count[delta] = translation_count.get(delta, 0) + 1

            # Update max_overlap if we find a higher count
            max_overlap = max(max_overlap, translation_count[delta])

    return max_overlap

# Example usage:
img1 = [[1,1,0],[0,1,0],[0,1,0]]
img2 = [[0,0,0],[0,1,1],[0,0,1]]
print(largestOverlap(img1, img2)) # Expected output: 3
```

Python


```python
class Solution:
    def largestOverlap(self, img1: list[list[int]], img2: list[list[int]]) -> int:
        MAGIC = 100
        ones1 = {(i, j)
                 for i, row in enumerate(img1)
                 for j, num in enumerate(row)
                 if num == 1}
        ones2 = {(i, j)
                 for i, row in enumerate(img2)
                 for j, num in enumerate(row)
                 if num == 1}
        offsetCount = collections.Counter()

        for ax, ay in ones1:
          for bx, by in ones2:
            offsetCount[(ax - bx) * MAGIC + (ay - by)] += 1

        return max(offsetCount.values()) if offsetCount else 0
```

Python

```python
class Solution:
    def largestOverlap(self, img1: List[List[int]], img2: List[List[int]]) -> int:
        n = len(img1)
        cnt = Counter()
        for i in range(n):
            for j in range(n):
                if img1[i][j]:
                    for h in range(n):
                        for k in range(n):
                            if img2[h][k]:
                                cnt[(i - h, j - k)] += 1
        return max(cnt.values()) if cnt else 0
```

Python

```python
class Solution:
    def largestOverlap(self, img1: List[List[int]], img2: List[List[int]]) -> int:
        n = len(img1)
        cnt = Counter()
        for i in range(n):
            for j in range(n):
                if img1[i][j]:
                    for h in range(n):
                        for k in range(n):
                            if img2[h][k]:
                                cnt[(i - h, j - k)] += 1
        return max(cnt.values()) if cnt else 0
```

[*] Matrix — Pattern Solution (stored optimal)

Python


```python
def largestOverlap(img1, img2):
    # Get the dimension of the square images
    n = len(img1)

    # Gather positions (coordinates) where there is a 1 in img1 and img2
    ones_img1 = []
    ones_img2 = []
    for i in range(n):
        for j in range(n):
            if img1[i][j] == 1:
                ones_img1.append((i, j))
            if img2[i][j] == 1:
                ones_img2.append((i, j))

    # Dictionary to count translation shifts that align 1's of img1 with img2
    translation_count = {}
    max_overlap = 0

    # Iterate over every pair of 1's from both images
    for (x1, y1) in ones_img1:
        for (x2, y2) in ones_img2:
            # Calculate the translation vector needed to align img1's 1 with img2's 1
            delta = (x2 - x1, y2 - y1)
            # Update the count for this translation vector
            translation_count[delta] = translation_count.get(delta, 0) + 1

            # Update max_overlap if we find a higher count
            max_overlap = max(max_overlap, translation_count[delta])

    return max_overlap

# Example usage:
img1 = [[1,1,0],[0,1,0],[0,1,0]]
img2 = [[0,0,0],[0,1,1],[0,0,1]]
print(largestOverlap(img1, img2)) # Expected output: 3
```

#1198 MEDIUM

Find Smallest Common Element in All Rows

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Binary Search · Matrix · Counting

Lists: Premium 98

Problem:

Given an m x n matrix mat where every row is sorted in strictly increasing order, return the smallest common element in all rows.

If there is no common element, return -1.

Example 1:

```
Input: mat = [[1,2,3,4,5],[2,4,5,8,10],[3,5,7,9,11],[1,3,5,7,9]]
Output: 5
```

Example 2:


```
Input: mat = [[1,2,3],[2,3,4],[2,3,5]]
Output: 2
```

Constraints:

- m == mat.length
- n == mat[i].length
- 1 <= m, n <= 500
- 1 <= mat[i][j] <= 104
- mat[i] is sorted in strictly increasing order.

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def smallestCommonElement(self, mat):
        # Brute Force O(m^2 n^2 )- flood fill from every cell
        m, n = len(mat), len(mat[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python

```python
def smallestCommonElement(mat):
    # Get number of rows and columns
    m = len(mat)
    n = len(mat[0])

    # Helper function: binary search for target in a row
    def binary_search(row, target):
        left, right = 0, n - 1
        while left <= right:
            mid = (left + right) // 2
            if row[mid] == target:
                return True
            elif row[mid] < target:
                left = mid + 1
            else:
                right = mid - 1
        return False

    # Iterate over each candidate in the first row
    for candidate in mat[0]:
        is_common = True
        # Check each subsequent row for the candidate using binary search
        for i in range(1, m):
            if not binary_search(mat[i], candidate):
                is_common = False

                break
        if is_common:
            return candidate
    return -1

# Example usage:
print(smallestCommonElement([[1,2,3,4,5],[2,4,5,8,10],[3,5,7,9,11],[1,3,5,7,9]]))
```

Python

```python
class Solution:
   def smallestCommonElement(self, mat: list[list[int]]) -> int:
    MAX = 10000
    count = [0] * (MAX + 1)

    for row in mat:
      for a in row:
        count[a] += 1
        if count[a] == len(mat):
          return a

    return -1
```

Python


```python
class Solution:
    def smallestCommonElement(self, mat: List[List[int]]) -> int:
        cnt = Counter()
        for row in mat:
            for x in row:
                cnt[x] += 1
                if cnt[x] == len(mat):
                    return x
        return -1
```

Python

```python
class Solution:
    def smallestCommonElement(self, mat: List[List[int]]) -> int:
        cnt = Counter()
        for row in mat:
            for x in row:
                cnt[x] += 1
                if cnt[x] == len(mat):
                    return x
        return -1
```

[*] Matrix — Pattern Solution (stored optimal)

Python

```python
def smallestCommonElement(mat):
    # Get number of rows and columns
    m = len(mat)
    n = len(mat[0])

    # Helper function: binary search for target in a row
    def binary_search(row, target):
        left, right = 0, n - 1
        while left <= right:
            mid = (left + right) // 2
            if row[mid] == target:
                return True
            elif row[mid] < target:
                left = mid + 1
            else:
                right = mid - 1
        return False

    # Iterate over each candidate in the first row
    for candidate in mat[0]:
        is_common = True
        # Check each subsequent row for the candidate using binary search
        for i in range(1, m):
            if not binary_search(mat[i], candidate):
                is_common = False

                break
        if is_common:
            return candidate
    return -1

# Example usage:
print(smallestCommonElement([[1,2,3,4,5],[2,4,5,8,10],[3,5,7,9,11],[1,3,5,7,9]]))
```


#1074 HARD

Number of Submatrices That Sum to Target

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Matrix · Prefix Sum

Lists: Denny Zhang

Problem:

Given a matrix and a target, return the number of non-empty submatrices that sum to target.

A submatrix x1, y1, x2, y2 is the set of all cells matrix[x][y] with x1 <= x <= x2 and y1 <= y <= y2.

Two submatrices (x1, y1, x2, y2) and (x1', y1', x2', y2') are different if they have some coordinate that is different: for example, if x1 != x1'.

Example 1:

0	1	0
1	1	1
0	1	0

```
Input: matrix = [[0,1,0],[1,1,1],[0,1,0]], target = 0
Output: 4
Explanation: The four 1x1 submatrices that only contain 0.
```

Example 2:

```
Input: matrix = [[1,-1],[-1,1]], target = 0
Output: 5
Explanation: The two 1x2 submatrices, plus the two 2x1 submatrices, plus the 2x2 submatrix.
```

Example 3:

```
Input: matrix = [[904]], target = 0
Output: 0
```

Constraints:

- 1 <= matrix.length <= 100
- 1 <= matrix[0].length <= 100

- -1000 <= matrix[i][j] <= 1000
- -10^8 <= target <= 10^8

>> Brute Force [Matrix] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def numSubmatrixSumTarget(self, matrix, target):
        # Brute force start-stop - flood fill from every cell
        m, n = len(matrix), len(matrix[0])
        result = 0
        for r in range(m):
            for c in range(n):
                visited = set()
                def flood(r, c):
                    if (r,c) in visited or not (0<=r<m and 0<=c<n): return
                    visited.add((r,c))
                    for dr,dc in [(0,1),(0,-1),(1,0),(-1,0)]: flood(r+dr,c+dc)
                flood(r, c)
                result = max(result, len(visited))
        return result
```

Python

Python

```python
# Python solution for counting number of submatrices that sum to target

def numSubmatrixSumTarget(matrix, target):
    # Dimensions of the matrix
    n = len(matrix)
    m = len(matrix[0])
    result = 0

    # Loop over all possible starting rows
    for top in range(n):
        # Initialize a list to store the column sums between row 'top' and 'bottom'
        col_sums = [0] * m

        # Extend the submatrix from row 'top' downwards to 'bottom'
        for bottom in range(top, n):
            # Update column sums for the current bottom row
            for col in range(m):
                col_sums[col] += matrix[bottom][col]

            # Use a dictionary to count cumulative sums in the 1D array (col_sums)
            sum_count = {0: 1}
            curr_sum = 0
            # Iterate over each column in col_sums to find the number of subarrays with sum equal to target
            for value in col_sums:
                curr_sum += value
```



```python
                if (curr_sum - target) in sum_count:
                    result += sum_count[curr_sum - target]
                # Update the dictionary with the current sum count
                sum_count[curr_sum] = sum_count.get(curr_sum, 0) + 1

    return result

# Example usage:
if __name__ == "__main__":
    matrix = [[0, 1, 0], [1, 1, 1], [0, 1, 0]]
    target = 0
    print(numSubmatrixSumTarget(matrix, target)) # Expected Output: 4
```

Python

```python
class Solution:
   def numSubmatrixSumTarget(self, matrix: list[list[int]], target: int) -> int:
    m = len(matrix)
    n = len(matrix[0])
    ans = 0

    # Transfer each row in the matrix to the prefix sum.
    for row in matrix:
      for i in range(1, n):
        row[i] += row[i - 1]

    for baseCol in range(n):
      for j in range(baseCol, n):
        prefixCount = collections.Counter({0: 1})
        summ = 0
        for i in range(m):
          if baseCol > 0:
            summ -= matrix[i][baseCol - 1]
          summ += matrix[i][j]
          ans += prefixCount[summ - target]
          prefixCount[summ] += 1

    return ans
```

Python


```python
class Solution:
    def numSubmatrixSumTarget(self, matrix: List[List[int]], target: int) -> int:
        def f(nums: List[int]) -> int:
            d = defaultdict[int]
            d[0] = 1
            cnt = s = 0
            for x in nums:
                s += x
                cnt += d[s - target]
                d[s] += 1
            return cnt

        m, n = len(matrix), len(matrix[0])
        ans = 0
        for i in range(m):
            col = [0] * n
            for j in range(i, m):
                for k in range(n):
                    col[k] += matrix[j][k]
                ans += f(col)
        return ans
```

Python

```python
class Solution:
    def numSubmatrixSumTarget(self, matrix: List[List[int]], target: int) -> int:
        def f(nums: List[int]) -> int:
            d = defaultdict[int]
            d[0] = 1
            cnt = s = 0
            for x in nums:
                s += x
                cnt += d[s - target]
                d[s] += 1
            return cnt

        m, n = len(matrix), len(matrix[0])
        ans = 0
        for i in range(m):
            col = [0] * n
            for j in range(i, m):
                for k in range(n):
                    col[k] += matrix[j][k]
                ans += f(col)
        return ans
```

[*] Matrix — Pattern Solution (stored optimal)

Python

```python
# Python solution for counting number of submatrices that sum to target

def numSubmatrixSumTarget(matrix, target):
    # Dimensions of the matrix
    n = len(matrix)
    m = len(matrix[0])
    result = 0

    # Loop over all possible starting rows
    for top in range(n):
        # Initialize a list to store the column sums between row 'top' and 'bottom'
        col_sums = [0] * m

        # Extend the submatrix from row 'top' downwards to 'bottom'
        for bottom in range(top, n):
            # Update column sums for the current bottom row
            for col in range(m):
                col_sums[col] += matrix[bottom][col]

            # Use a dictionary to count cumulative sums in the 1D array (col_sums)
            sum_count = {0: 1}
            curr_sum = 0
            # Iterate over the column in col_sums to find the number of subarrays with sum equal to target
            for value in col_sums:
                curr_sum += value

                if (curr_sum - target) in sum_count:
                    result += sum_count[curr_sum - target]
                # Update the dictionary with the current sum count
                sum_count[curr_sum] = sum_count.get(curr_sum, 0) + 1

    return result

# Example usage:
if __name__ == "__main__":
    matrix = [[0, 1, 0], [1, 1, 1], [0, 1, 0]]
    target = 0
    print(numSubmatrixSumTarget(matrix, target)) # Expected Output: 4
```


Quick Review — Matrix 20 questions · insights · complexity · solution

#422 Valid Word Square [Easy]

Key Insights

- The k-th row should be equal to the k-th column.
- Not every word is the same length; checks must be performed within bounds.
- Iterating over each character and cross-verifying with the corresponding column character is sufficient.

Space & Time

Time Complexity: O(N * L) where N is the number of words and L is the average length of the words.

Space Complexity: O(1) additional space is used for the checking procedure.

Solution

We iterate through each word (row) and each character (column) in the word. For every character at position (i, j), we confirm that there is a corresponding word at index j and that its length is greater than i. Then we check that the character at position (j, i) in that word is the same as the current character. If any of these checks fail, we return false. If all checks pass, the words form a valid word square.

#566 Reshape the Matrix [Easy]

Key Insights

- Check if the reshape operation is possible by comparing the total elements (m * n with r * c).
- Flatten the original matrix while preserving the row-traversing order.
- Use the flattened list to build the new matrix row by row.
- If the total number of elements does not match, simply return the original matrix.

Space & Time

Time Complexity: O(m

n) as we iterate through all elements of the matrix once.

Space Complexity: O(m

n) for the additional list used to store flattened elements (or the new matrix).

Solution

The solution involves first checking whether the reshape is possible by ensuring that the product of the rows and columns of the original matrix equals that of the desired matrix (r * c). If not, the original matrix is returned. Otherwise, the matrix is flattened into a one-dimensional list, preserving the row-order traversal. Finally, the flattened list is segmented by the new column size c to form the reshaped matrix. The primary data structures used include a list (or array) for flattening the matrix and then another list (or vector/array) to hold the final reshaped matrix.

#766 Toeplitz Matrix [Easy]

Key Insights

- Each element (except those in the first row and first column) must be equal to its top-left neighbor.
- You can validate the matrix by iterating from the second row/column and comparing each element with matrix[i-1][j-1].
- This simple check allows you to determine if every diagonal has the same elements without extra space.
- For streaming data (loading one row at a time), maintain the previous row for comparison.

Space & Time

Time Complexity: O(m * n)

Space Complexity: O(1)

Solution

The problem is solved by iterating through the matrix starting from the element at position (1, 1). For every element at matrix[i][j], compare it with the element at matrix[i-1][j-1]. If any two elements in this pair do not match, the matrix is not Toeplitz. This check ensures that every diagonal from the top-left to bottom-right has identical elements. In terms of auxiliary space, only constant extra space is used, which makes this approach efficient both in time and space.

#867 Transpose Matrix [Easy]

Key Insights

- The value at position (i, j) in the original matrix moves to (j, i) in the transposed matrix.
- The transposed matrix dimensions are the reverse of the original: if the original is m x n, then the transposed is n x m.
- A simple double loop is sufficient to swap the indices, ensuring O(m*n) time complexity.
- This problem handles both square and rectangular matrices.

Space & Time

Time Complexity: O(m

n)

Space Complexity: O(m

n) for the new matrix holding the transposed values

Solution

The solution involves iterating over each element of the original matrix and assigning it to the corresponding position in the new matrix where the indexes are swapped. We create the transposed matrix with dimensions based on the number of columns and rows of the original matrix, respectively. The approach leverages basic array manipulation with nested loops to fill in the new matrix.

#2373 Largest Local Values in a Matrix [Easy]

Key Insights

- The size of the output matrix is (n - 2) x (n - 2).
- For each valid index (i, j) in the output, examine the 3 x 3 block in grid starting at grid[i][j] to grid[i+2][j+2].
- Since each 3 x 3 block contains exactly 9 elements, finding the maximum in a block takes constant time.
- Use nested loops to slide over the grid and efficiently compute each local maximum.

Space & Time

Time Complexity: O(n^2) - We iterate over (n-2) x (n-2) positions, and for each position, we look at 9 elements.

Space Complexity: O(n^2) - The additional space is required for the output matrix.

Solution

The solution involves iterating over each possible 3 x 3 submatrix in grid. For each (i, j) starting index of the submatrix:

- Examine the nine elements from grid[i][j] to grid[i+2][j+2]. - Compute the maximum value from these elements. - Store the maximum in the corresponding position of the result matrix. Data structures used include the result matrix for storing the maximum values. The algorithmic approach involves two nested loops (one for rows and one for columns) to traverse each possible starting position of a 3 x 3 window. The simplicity of the solution comes from the fixed 3x3 block size, which enables constant-time maximum computation for each block.

#36 Valid Sudoku [Medium]

Key Insights

- Validate each row by checking for duplicate numbers while ignoring empty cells.
- Validate each column in the same way.
- Divide the board into nine 3x3 sub-boxes and check each for duplicate numbers.

- Use hash sets (or equivalent data structures) to keep track of seen digits for rows, columns, and boxes.
- The board size is fixed (9x9), so the overall time complexity is constant with respect to input size.

Space & Time

Time Complexity: O(1) — Since the board size is fixed at 9x9.

Space Complexity: O(1) — The extra space used by the sets is bounded by the fixed board size.

Solution

The solution uses three main data structures: an array of sets for rows, an array of sets for columns, and an array of sets for 3x3 sub-boxes. For each cell, if it is not empty, check if the number is already present in the corresponding row, column, or box. If found, return false immediately. Otherwise, add the number to the respective sets. Compute the index for the 3x3 sub-box using the formula: (row_index // 3) * 3 + (col_index // 3). If no duplicates are found after traversing the entire board, the board is considered valid.

#48 Rotate Image [Medium]

Key Insights

- The rotation can be achieved in two steps: first transpose the matrix and then reverse each row.
- Transposing the matrix swaps matrix[i][j] with matrix[j][i], converting rows into columns.
- Reversing each row post-transposition yields the desired clockwise rotation.
- This approach works in-place with O(1) extra space.

Space & Time

Time Complexity: O(n^2) since every element is processed during the transpose and row reversal.

Space Complexity: O(1) as the operations are performed in-place without extra data structures.

Solution

The solution leverages a two-step in-place transformation:

- Transpose the matrix: Iterate through the matrix and swap each element (i, j) with element (j, i) for j >= i. - Reverse each row: After the transpose, each row is reversed to rotate the matrix 90 degrees clockwise. The key trick is to realize that a transpose followed by a reverse of each row achieves the rotation without needing extra space.

#54 Spiral Matrix [Medium]

Key Insights

- Use four boundaries: top, bottom, left, and right to track the current layer.
- Traverse right across the top row, then down the right column, then left on the bottom row, and finally up the left column.
- After each direction, adjust the corresponding boundary to move inward.
- Continue the traversal until the boundaries cross.

Space & Time

Time Complexity: O(m * n) since each element is visited exactly once.

Space Complexity: O(1) extra space (excluding the space required for the output list).

Solution

The solution uses a simulation approach with boundary pointers (top, bottom, left, right). At each step, the boundaries indicate the current submatrix to traverse. You iterate as follows:

- Traverse from left to right along the top boundary. - Traverse downward along the right boundary. - If the top boundary has not passed the bottom, traverse from right to left along the bottom boundary. - If the left boundary has not passed the right, traverse upward along the left boundary. After each traversal, the boundaries are adjusted inward. This continues until all elements have been visited.

#59 Spiral Matrix II [Medium]

Key Insights

- Use four pointers (top, bottom, left, right) to keep track of the matrix boundaries.

- Traverse the matrix in a spiral: move right along the top row, then down the right column, left along the bottom row, and finally up the left column.
- Update the boundary pointers after each complete traversal of one side.
- Continue until all numbers from 1 to n^2 are filled into the matrix.

Space & Time

Time Complexity: $O(n^2)$

Space Complexity: $O(n^2)$ (required for the output matrix)

Solution

We simulate the spiral movement by initializing four pointers: top, bottom, left, and right, which denote the current boundaries of the matrix that have not yet been filled. For each iteration, we:

- Traverse from left to right along the top boundary, then increment the top pointer. - Traverse from top to bottom along the right boundary, then decrement the right pointer. - If there are rows remaining, traverse from right to left along the bottom boundary, then decrement the bottom pointer. - If there are columns remaining, traverse from bottom to top along the left boundary, then increment the left pointer. This algorithm ensures that the matrix is filled in a spiral order until all numbers are placed.

#64 Minimum Path Sum [Medium]

Key Insights

- The problem can be solved using Dynamic Programming.
- The optimal solution for each cell depends on the minimum path sum of the cell above it and the cell to its left.
- Since you can only move right or down, only two adjacent cells affect the current cell's result.
- Edge cases include the first row and first column, which have only one direction of movement.

Space & Time

Time Complexity: O(m * n), where m and n are the number of rows and columns.

Space Complexity: O(m * n) for the DP table (this can be optimized to O(n) with in-place computations).

Solution

We use a Dynamic Programming (DP) approach where we create a DP table (or use the grid itself) that stores the minimum path sum to each cell. For each cell (i, j):

- If i = 0 (first row), the only possible path is from the left. - If j = 0 (first column), the only possible path is from above. - For other cells, the minimum path sum is the cell value plus the minimum of the two possible previous cells (above and left). This ensures that by the time we reach the bottom-right cell, we have accumulated the minimum sum required to reach that cell.

#73 Set Matrix Zeroes [Medium]

Key Insights

- The challenge is to mark rows and columns that need to be zeroed without using extra space.
- A common trick is to use the first row and first column as markers.
- Special care is needed for the first row and first column, as they are also used as markers.
- The algorithm involves two passes: one for marking and one for updating the matrix based on the markers.

Space & Time

Time Complexity: O(m * n)

Space Complexity: O(1) (in-place, using only constant extra space)

Solution

We can solve this problem by using the first row and first column of the matrix as markers. First, determine if the first row or first column needs to be zeroed, then traverse the rest of the matrix and mark the corresponding first row and column positions if a zero is encountered. Finally, update the matrix backward using the markers, and finally update the first row and column if needed.

The algorithm works in three main steps:

- Check if the first row or first column should be zeroed (store this in two variables). - Iterate through the rest of the matrix: for any element that is zero, mark its row and column by setting the corresponding element in the first row and first column to zero. - Iterate through the matrix again (excluding the first row and column) and set elements to zero if their row or column marker is zero. - Finally, zero out the first row and/or first column if needed.

This approach avoids extra space usage apart from the two boolean flags while ensuring that the matrix is updated correctly.

#74 Search a 2D Matrix [Medium]

Key Insights

- The matrix is essentially a flattened sorted array due to its properties.
- Use binary search to efficiently determine if the target exists.
- Map the 1D binary search index to the corresponding 2D matrix coordinates using division and modulus operations.
- Achieve O(log(m*n)) time complexity by navigating the matrix as if it were a single sorted list.

Space & Time

Time Complexity: O(log(m * n))

Space Complexity: O(1)

Solution

We can treat the matrix as a flattened sorted list without physically copying the elements. Given the total number of elements = m * n, perform binary search on indices ranging from 0 to (m*n - 1). For any index mid, calculate its corresponding row as mid // n and column as mid % n. Compare the element at matrix[row][col] with the target. If it equals target, return true; if it is less than target, search the right half; otherwise, search the left half.

#221 Maximal Square [Medium]

Key Insights

- Use dynamic programming to build a solution where each cell in the DP table represents the side length of the largest square ending at that cell.
- If the current cell is '1', its DP value is the minimum of the top, left, and top-left neighbors plus one.
- Update a global maximum side length during the process to determine the area.
- The square's area is the square of its maximum side length.

Space & Time

Time Complexity: O(m * n), where m is the number of rows and n is the number of columns.

Space Complexity: O(m * n) for the DP table (this can be optimized to O(n) with space reduction techniques).

Solution

We use a two-dimensional dynamic programming approach. We define a DP matrix with one extra row and column to simplify boundary conditions. For each cell (i, j) in the input matrix, if the value is '1', then we calculate the DP value as the minimum of its top, left, and top-left neighboring DP values plus one. This value represents the side length of the square ending at (i, j). We also keep track of the maximum side length found and finally compute the area of the maximal square by squaring the maximum side length.

#311 Sparse Matrix Multiplication [Medium]

Key Insights

- Exploit sparsity by iterating only over non-zero elements.
- For each non-zero element in mat1, multiply it with corresponding non-zero elements in mat2.
- Accumulate the product into the result matrix.
- This approach reduces unnecessary multiplication operations when many elements are zero.

Space & Time

Time Complexity: O(m * k * n) in the worst-case, but significantly lower when many elements are zero.

Space Complexity: O(m * n) for the resulting matrix.

Solution

We loop through each row of mat1 and for every non-zero element, we traverse the corresponding row in mat2. By checking if an element in both matrices is non-zero before multiplying, we avoid redundant calculations. This optimization leverages the sparsity of the matrices. We use standard nested loops along with conditional checks to implement this efficient multiplication.

#498 Diagonal Traverse [Medium]

Key Insights

- Traverse the matrix diagonally with alternating directions: upward (top-right) and downward (bottom-left).
- When the traversal reaches a boundary (top, bottom, left, or right), adjust the position and change the direction.
- Use simulation to process each element exactly once while managing the indices based on the current direction.

Space & Time

Time Complexity: O(m * n) where m is the number of rows and n is the number of columns, as every element is processed once.

Space Complexity: O(m * n) for the output array (excluding the input matrix), while additional space usage remains constant.

Solution

The solution simulates the diagonal traversal using a direction flag (1 for upward and -1 for downward).

- Begin at the matrix's top-left corner. - Append the current element to the result. - Depending on the current direction: If moving upward, decrease the row and increase the column. If moving downward, increase the row and decrease the column. - Adjust for boundary hits: When moving upward: If at the last column, increment the row and switch direction. If at the first row, increment the column and switch direction. When moving downward: If at the last row, increment the column and switch direction. If at the first column, increment the row and switch direction. This process continues until all elements are added to the result array.

#531 Lonely Pixel I [Medium]

Key Insights

- Count the number of 'B' pixels in each row.
- Count the number of 'B' pixels in each column.
- A pixel is considered lonely if the count in its corresponding row and column is exactly 1.
- Two passes through the matrix can efficiently solve the problem: one for counting and one for checking conditions.

Space & Time

Time Complexity: O(m * n), where m and n are the dimensions of the picture.

Space Complexity: O(m + n), used to store the row and column counts.

Solution

The solution works by first scanning the entire matrix to compute the number of black pixels in each row and each column using two arrays/lists. Once these counts are available, a second pass through the matrix determines for each black pixel if it is lonely by checking if its corresponding row and column count equals 1. The approach uses simple array data structures for tallying counts and ensures that each element is processed only a constant number of times.

#723 Candy Crush [Medium]

Key Insights

- Use a simulation approach: repeatedly detect crushable candies and simulate their removal.
- Traverse horizontally and vertically to mark candies that are in groups of three or more.

- Instead of removing candies immediately, mark them for removal and update the board once for the entire pass.
- After crushing, simulate gravity by shifting non-zero candies downward in each column.
- Continue the process until no further candy crushes can be detected.

Space & Time

Time Complexity: O((m

n)^2) in the worst case since each iteration includes scanning the m x n grid and then applying gravity.

Space Complexity: O(m

n) for maintaining additional markers or flags (or using the board itself with in-place modifications).

Solution

The approach involves iteratively scanning the board to identify candy groups that should be crushed using two passes:

- Horizontal Check - For each row, check segments of 3 consecutive candies. - Vertical Check - For each column, check segments of 3 consecutive candies. If any candy qualifies, mark its position for crushing. After marking, update the board by turning all marked positions into 0. Then, for each column, drop down non-zero candies such that all empty cells (zeros) are at the top. Repeat this process until no group of 3 or more adjacent candies is found. Key data structures include the board array and a temporary marker (can be implicit by using signs or a separate boolean matrix) for tracking candies to crush. The algorithm ensures a stable board before returning it.

#835 Image Overlap [Medium]

Key Insights

- You only need to consider the positions of 1's in each matrix.
- The overlap count for a given translation can be computed by comparing the relative differences between the positions of 1's in the two images.
- Using a hash map (or dictionary) to count how many times each translation vector appears lets you compute the maximum overlap efficiently.
- The translation operation is equivalent to shifting the positions, so the best possible translation is the one that aligns the most pairs of 1's.

Space & Time

Time Complexity: O(n^4) in the worst-case when using a nested iteration over positions for both matrices, though the average situation is typically better since not all cells are 1.

Space Complexity: O(n^2) for storing the positions of 1's and the counts for translation vectors.

Solution

The solution involves the following steps:

- Traverse both images to record the coordinates of all 1's. - For every pair of 1's (one from the first image, one from the second), compute the translation vector (offset) that aligns them. - Use a hash map to count how many times each offset appears. - The maximum count recorded in the hash map is the largest overlap achievable by any translation. This approach avoids simulating every possible slide over the matrix and focuses only on relative alignments of 1's.

#1198 Find Smallest Common Element in All Rows [Medium]

Key Insights

- Since every row is strictly increasing, binary search can be used efficiently to check for the presence of an element in a row.
- Iterating over the first row's elements and verifying their presence in all other rows is an effective approach.
- Early termination is possible if an element is not found in any one row.
- Time complexity is manageable given the constraints, even using binary search for each candidate across rows.

Space & Time

Time Complexity: O(m * n * log(n)) where m is the number of rows and n is the number of columns (binary search in each row for each candidate from the first row).

Space Complexity: O(1) (only a few auxiliary variables are used).

Solution

The solution iterates over each element (candidate) in the first row of the matrix. For each candidate, it checks if the candidate is present in every other row using binary search, taking advantage of the sorted order. If a candidate is found in every row, return it immediately as it will be the smallest such common element because the first row is processed in order. If no candidate is common across all rows, return -1.

#1074 Number of Submatrices That Sum to Target [Hard]

Key Insights

- Transform the 2D submatrix sum problem into multiple 1D subarray sum problems.
- Precompute cumulative sums by fixing two row boundaries and then summing columns between these rows.
- Use a hash table (or dictionary) to count the number of subarrays with a given sum, reducing the problem to the classic "subarray sum equals k" question.
- Iterate over all possible row pairs and for each, treat the column sums as an array where the target sum is sought.

Space & Time

Time Complexity: $O(n^2 * m)$, where n is the number of rows and m is the number of columns.

Space Complexity: O(m), used by the hash table for storing cumulative sums along the columns.

Solution

The idea is to iterate over all pairs of rows (top and bottom) and, for each pair, compute the cumulative column sums between these rows. This converts the problem into finding the number of subarrays in a one-dimensional array (the column sums) that add up to the target. For each such 1D problem, use a hash table to track the frequency of cumulative sums seen so far and count how many times the difference (current sum - target) has been seen. This count gives the number of subarrays ending at the current index which sum to target. Combining counts from all possible row pairs yields the final result.

#16 Linked List — 23 questions · #16 of 21

#21 EASY

Merge Two Sorted Lists

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Linked List · Recursion

Lists: Grind 169

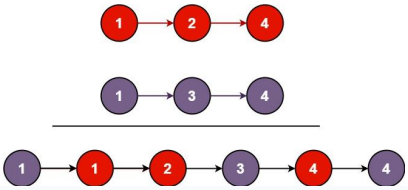
Problem:

You are given the heads of two sorted linked lists list1 and list2.

Merge the two lists into one sorted list. The list should be made by splicing together the nodes of the first two lists.

Return the head of the merged linked list.

Example 1:




```
Input: list1 = [1,2,4], list2 = [1,3,4]
Output: [1,1,2,3,4,4]
```

Example 2:

```
Input: list1 = [], list2 = []
Output: []
```

Example 3:

```
Input: list1 = [], list2 = [0]
Output: [0]
```

Constraints:

- The number of nodes in both lists is in the range [0, 50].
- -100 <= Node.val <= 100
- Both list1 and list2 are sorted in non-decreasing order.

Python

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def mergeTwoLists(list1: ListNode, list2: ListNode) -> ListNode:
    # Create a dummy node to serve as the start of the merged list
    dummy = ListNode(-1)
    current = dummy

    # Iterate while both lists have nodes
    while list1 and list2:
        # Compare the values of the current nodes of both lists
        if list1.val <= list2.val:
            current.next = list1 # Link the smaller node to the merged list
            list1 = list1.next # Move the pointer in list1
        else:
            current.next = list2
            list2 = list2.next
        current = current.next # Move the pointer in the merged list

    # Once one list is exhausted, attach the remaining nodes from the other list
    if list1:
        current.next = list1

    else:
        current.next = list2

    # The merged list head is at dummy.next
    return dummy.next
```

Python

```python
class Solution:
    def mergeTwoLists(
        self,
        list1: ListNode | None,
        list2: ListNode | None,
    ) -> ListNode | None:
        if not list1 or not list2:
            return list1 if list1 else list2
        if list1.val > list2.val:
            list1, list2 = list2, list1
        list1.next = self.mergeTwoLists(list1.next, list2)
        return list1
```

Python


```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def mergeTwoLists(
        self, list1: Optional[ListNode], list2: Optional[ListNode]
    ) -> Optional[ListNode]:
        if list1 is None or list2 is None:
            return list1 or list2
        if list1.val <= list2.val:
            list1.next = self.mergeTwoLists(list1.next, list2)
            return list1
        else:
            list2.next = self.mergeTwoLists(list1, list2.next)
            return list2
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
# without ops after while loop
class Solution:
    def mergeTwoLists(self, list1: Optional[ListNode], list2: Optional[ListNode]) -> Optional[ListNode]:
        dummy = ListNode()
        current = dummy
        while list1 or list2:
            v1 = list1.val if list1 else float('inf')
            v2 = list2.val if list2 else float('inf')

            if v1 < v2:
                current.next = list1
                list1 = list1.next
            else:
                current.next = list2
                list2 = list2.next
            current = current.next

        return dummy.next

############
```

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def mergeTwoLists(
        self, list1: Optional[ListNode], list2: Optional[ListNode]
    ) -> Optional[ListNode]:
        dummy = ListNode()
        curr = dummy
        while list1 and list2:
            if list1.val <= list2.val:
                curr.next = list1
                list1 = list1.next
            else:
                curr.next = list2
                list2 = list2.next
            curr = curr.next
        curr.next = list1 or list2
        return dummy.next
"""
        curr.next = list1 or list2
```

```
better than:

        if list1:
            current.next = list1
        if list2:
            current.next = list2

"""

###########

# recursion
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def mergeTwoLists(self, l1: ListNode, l2: ListNode) -> ListNode:
        if not l1: # If l1 is empty, return l2
            return l2
        if not l2: # If l2 is empty, return l1
            return l1

        if l1.val < l2.val:
            l1.next = self.mergeTwoLists(l1.next, l2)
            return l1
        else:
```


[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def mergeTwoLists(list1: ListNode, list2: ListNode) -> ListNode:
    # Create a dummy node to serve as the start of the merged list
    dummy = ListNode(-1)
    current = dummy

    # Iterate while both lists have nodes
    while list1 and list2:
        # Compare the values of the current nodes of both lists
        if list1.val <= list2.val:
            current.next = list1 # Link the smaller node to the merged list
            list1 = list1.next # Move the pointer in list1
        else:
            current.next = list2
            list2 = list2.next
        current = current.next # Move the pointer in the merged list

    # Once one list is exhausted, attach the remaining nodes from the other list
    if list1:
        current.next = list1

    else:
        current.next = list2

    # The merged list head is at dummy.next
    return dummy.next
```

#141 EASY

Linked List Cycle

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · Linked List · Two Pointers

Lists: Grind 169

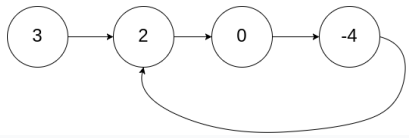
Problem:

Given head, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. Note that pos is not passed as a parameter.

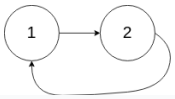
Return true if there is a cycle in the linked list. Otherwise, return false.

Example 1:




```
Input: head = [3,2,0,-4], pos = 1
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 1st node
(0-indexed).
```

Example 2:




```
Input: head = [1,2], pos = 0
Output: true
Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.
```

Example 3:

```
Input: head = [1], pos = -1
Output: false
Explanation: There is no cycle in the linked list.
```

Constraints:

- The number of the nodes in the list is in the range [0, 104].
- -105 <= Node.val <= 105
- pos is -1 or a valid index in the linked-list.

Follow up: Can you solve it using O(1) (i.e. constant) memory?

Python

Python


```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next

class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        # Initialize two pointers, slow and fast, starting from the head.
        slow = head
        fast = head

        # Traverse the list until fast pointer reaches the end.
        while fast and fast.next:
            slow = slow.next # Move slow pointer one step.
            fast = fast.next.next # Move fast pointer two steps.

            # If both pointers meet, there is a cycle.
            if slow == fast:
                return True

        # If loop terminates, there is no cycle.
        return False
```

Python

```python
class Solution:
  def hasCycle(self, head: ListNode) -> bool:
    slow = head
    fast = head

    while fast and fast.next:
      slow = slow.next
      fast = fast.next.next
      if slow == fast:
        return True

    return False
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, x):
# self.val = x
# self.next = None


class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        s = set()
        while head:
            if head in s:
                return True
            s.add(head)
            head = head.next
        return False
```

Python


```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next

class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        # Initialize two pointers, slow and fast, starting from the head.
        slow = head
        fast = head

        # Traverse the list until fast pointer reaches the end.
        while fast and fast.next:
            slow = slow.next # Move slow pointer one step.
            fast = fast.next.next # Move fast pointer two steps.

            # If both pointers meet, there is a cycle.
            if slow == fast:
                return True

        # If loop terminates, there is no cycle.
        return False
```

[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next

class Solution:
    def hasCycle(self, head: Optional[ListNode]) -> bool:
        # Initialize two pointers, slow and fast, starting from the head.
        slow = head
        fast = head

        # Traverse the list until fast pointer reaches the end.
        while fast and fast.next:
            slow = slow.next # Move slow pointer one step.
            fast = fast.next.next # Move fast pointer two steps.

            # If both pointers meet, there is a cycle.
            if slow == fast:
                return True

        # If loop terminates, there is no cycle.
        return False
```

#206 EASY

Reverse Linked List

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

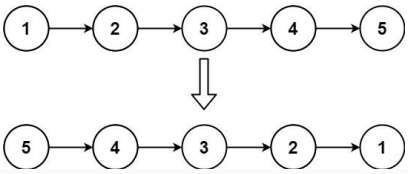
Linked List · Recursion

Lists: Grind 169

Problem:

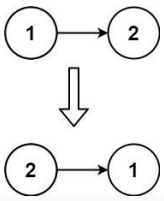
Given the head of a singly linked list, reverse the list, and return the reversed list.

Example 1:




```
Input: head = [1,2,3,4,5]
Output: [5,4,3,2,1]
```

Example 2:




```
Input: head = [1,2]
Output: [2,1]
```

Example 3:

```
Input: head = []
Output: []
```

Constraints:

- The number of nodes in the list is the range [0, 5000].
- -5000 <= Node.val <= 5000

Follow up: A linked list can be reversed either iteratively or recursively. Could you implement both?

>> Brute Force [Linked List] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def reverseList(self, head):
        # Brute Force O(n) space - collect values, rebuild in reverse
        vals = []
        curr = head
        while curr:
            vals.append(curr.val)
            curr = curr.next
        dummy = ListNode(0)
        curr = dummy
        for v in reversed(vals):
            curr.next = ListNode(v)
            curr = curr.next
        return dummy.next
```

Python

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val # Store the node value
        self.next = next # Pointer to the next node

# Iterative approach to reverse the linked list.
def reverseList(head):
    prev = None # Initially, there is no previous node
    current = head # Start with the head of the list
    while current:
        next_node = current.next # Temporarily store the next node
        current.next = prev # Reverse the current node's pointer
        prev = current # Move previous pointer one step forward
        current = next_node # Move to the next node in the list
    return prev # The new head of the reversed list

# Recursive approach to reverse the linked list.
def reverseListRecursive(head):
    # Base case: empty list or single node
    if not head or not head.next:
        return head
    new_head = reverseListRecursive(head.next) # Reverse the rest of the list
    head.next.next = head # Reverse the pointer for the current node
    head.next = None # Set the next pointer of the current node to None

    return new_head # Return the new head of the reversed list
```

Python

```python
class Solution:
  def reverseList(self, head: ListNode | None) -> ListNode | None:
    if not head or not head.next:
      return head

    newHead = self.reverseList(head.next)
    head.next.next = head
    head.next = None
    return newHead
```


Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        dummy = ListNode()
        curr = head
        while curr:
            next = curr.next
            curr.next = dummy.next
            dummy.next = curr
            curr = next
        return dummy.next
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def reverseList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        pre, p = None, head
        while p:
            q = p.next
            p.next = pre
            pre = p
            p = q
        return pre


class Solution(object): # recursive
  def reverseList(self, root):
    if not root or not root.next:
      return root

    ret = self.reverseList(root.next)
    root.next.next = root # root.next is end of the newly reversed list ret
    root.next = None
    return ret


class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        dummy = ListNode()
        curr = head
        while curr:
            next = curr.next
            curr.next = dummy.next
            dummy.next = curr
            curr = next
        return dummy.next
```


[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def reverseList(self, head: ListNode) -> ListNode:
        dummy = ListNode()
        curr = head
        while curr:
            next = curr.next
            curr.next = dummy.next
            dummy.next = curr
            curr = next
        return dummy.next
```

#234 EASY

Palindrome Linked List

LeetCodes · SimplyLeet · WalkCC · LeetCode · Editorial

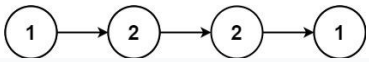
Linked List · Two Pointers · Recursion

Lists: Grind 169

Problem:

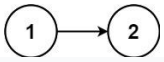
Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

Example 1:




```
Input: head = [1,2,2,1]
Output: true
```

Example 2:




```
Input: head = [1,2]
Output: false
```

Constraints:

- The number of nodes in the list is in the range [1, 105].
- 0 <= Node.val <= 9

Follow up: Could you do it in O(n) time and O(1) space?

Python

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        # Find the midpoint using slow and fast pointers
        slow, fast = head, head
        while fast and fast.next:
            slow = slow.next # Move slow pointer by 1
            fast = fast.next.next # Move fast pointer by 2

        # Reverse the second half of the list
        prev = None
        while slow:
            next_node = slow.next # Store the next node
            slow.next = prev # Reverse the current node's pointer
            prev = slow # Move prev to current node
            slow = next_node # Move to next node

        # Compare the first half and the reversed second half node by node
        left, right = head, prev
        while right: # Only need to compare till the end of second half

            if left.val != right.val:
                return False # Mismatch found, not a palindrome
            left = left.next # Move left pointer
            right = right.next # Move right pointer

        return True # All nodes matched, it's a palindrome
```

Python


```python
class Solution:
  def isPalindrome(self, head: ListNode) -> bool:
    def reverseList(head: ListNode) -> ListNode:
      prev = None
      curr = head
      while curr:
        next = curr.next
        curr.next = prev
        prev = curr
        curr = next
      return prev

    slow = head
    fast = head

    while fast and fast.next:
      slow = slow.next
      fast = fast.next.next

    if fast:
      slow = slow.next
    slow = reverseList(slow)

    while slow:
      if slow.val != head.val:

        return False
      slow = slow.next
      head = head.next

    return True
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        slow, fast = head, head.next
        while fast and fast.next:
            slow, fast = slow.next, fast.next.next
        pre, cur = None, slow.next
        while cur:
            t = cur.next
            cur.next = pre
            pre, cur = cur, t
        while pre:
            if pre.val != head.val:
                return False
            pre, head = pre.next, head.next
        return True
```

Python


```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def isPalindrome(self, head: Optional[ListNode]) -> bool:
        slow, fast = head, head.next
        while fast and fast.next:
            slow, fast = slow.next, fast.next.next
        pre, cur = None, slow.next
        while cur:
            t = cur.next
            cur.next = pre
            pre, cur = cur, t
        while pre:
            if pre.val != head.val:
                return False
            pre, head = pre.next, head.next
        return True
```

[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def isPalindrome(self, head: ListNode) -> bool:
        # Find the midpoint using slow and fast pointers
        slow, fast = head, head
        while fast and fast.next:
            slow = slow.next # Move slow pointer by 1
            fast = fast.next.next # Move fast pointer by 2

        # Reverse the second half of the list
        prev = None
        while slow:
            next_node = slow.next # Store the next node
            slow.next = prev # Reverse the current node's pointer
            prev = slow # Move prev to current node
            slow = next_node # Move to next node

        # Compare the first half and the reversed second half node by node
        left, right = head, prev
        while right: # Only need to compare till the end of second half

            if left.val != right.val:
                return False # Mismatch found, not a palindrome
            left = left.next # Move left pointer
            right = right.next # Move right pointer

        return True # All nodes matched, it's a palindrome
```

#706 EASY

Design HashMap

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Linked List · Design · Hash Function

Lists: Denny Zhang

Problem:

Design a HashMap without using any built-in hash table libraries.

Implement the MyHashMap class:

- MyHashMap() initializes the object with an empty map.
- void put(int key, int value) inserts a (key, value) pair into the HashMap. If the key already exists in the map, update the corresponding value.
- int get(int key) returns the value to which the specified key is mapped, or -1 if this map contains no mapping for the key.
- void remove(key) removes the key and its corresponding value if the map contains the mapping for the key.

Example 1:

```
Input
["MyHashMap", "put", "put", "get", "get", "put", "get", "remove", "get"]
[[], [1, 1], [2, 2], [1], [3], [2, 1], [2], [2], [2]]
Output
[null, null, null, 1, -1, null, 1, null, -1]

Explanation
MyHashMap myHashMap = new MyHashMap();
myHashMap.put(1, 1); // The map is now [[1,1]]
myHashMap.put(2, 2); // The map is now [[1,1], [2,2]]
myHashMap.get(1); // return 1, The map is now [[1,1], [2,2]]
myHashMap.get(3); // return -1 (i.e., not found), The map is now [[1,1], [2,2]]
myHashMap.put(2, 1); // The map is now [[1,1], [2,1]] (i.e., update the existing value)
myHashMap.get(2); // return 1, The map is now [[1,1], [2,1]]
myHashMap.remove(2); // remove the mapping for 2, The map is now [[1,1]]
myHashMap.get(2); // return -1 (i.e., not found), The map is now [[1,1]]
```

Constraints:

- 0 <= key, value <= 106
- At most 104 calls will be made to put, get, and remove.

Python

Python


```python
# ListNode class for handling collisions in each bucket
class ListNode:
    def __init__(self, key, value, next=None):
        self.key = key # Store key
        self.value = value # Store value
        self.next = next # Pointer to the next node

class MyHashMap:
    def __init__(self):
        self.size = 1000 # Define the bucket size
        self.buckets = [None] * self.size # Initialize buckets as an array of None

    def hash(self, key):
        # Simple modulo based hash function to map keys to bucket indices
        return key % self.size

    def put(self, key: int, value: int) -> None:
        index = self.hash(key)
        if self.buckets[index] is None:
            # If the bucket is empty, create a new node and place it here
            self.buckets[index] = ListNode(key, value)
        else:
            current = self.buckets[index]
            while True:
                if current.key == key:
```



```python
                    # If key exists, update its value
                    current.value = value
                    return
                if current.next is None:
                    break
                current = current.next
            # Append a new node at the end of the list if key not found
            current.next = ListNode(key, value)

    def get(self, key: int) -> int:
        index = self.hash(key)
        current = self.buckets[index]
        while current:
            if current.key == key:
                # Key found; return its value
                return current.value
            current = current.next
        # Key not found; return -1
        return -1

    def remove(self, key: int) -> None:
        index = self.hash(key)
        current = self.buckets[index]
        if current is None:
            return
```

```python
        if current.key == key:
            # Remove head node if it contains the key
            self.buckets[index] = current.next
        else:
            prev = current
            current = current.next
            while current:
                if current.key == key:
                    # Bypass the node to remove it from the list
                    prev.next = current.next
                    return
                prev, current = current, current.next
```

Python

```python
class MyHashMap:
    def __init__(self):
        self.data = [-1] * 1000001

    def put(self, key: int, value: int) -> None:
        self.data[key] = value

    def get(self, key: int) -> int:
        return self.data[key]

    def remove(self, key: int) -> None:
        self.data[key] = -1


# Your MyHashMap object will be instantiated and called as such:
# obj = MyHashMap()
# obj.put(key,value)
# param_2 = obj.get(key)
# obj.remove(key)
```

Python


```python
class MyHashMap:
    def __init__(self):
        self.data = [-1] * 1000001

    def put(self, key: int, value: int) -> None:
        self.data[key] = value

    def get(self, key: int) -> int:
        return self.data[key]

    def remove(self, key: int) -> None:
        self.data[key] = -1


# Your MyHashMap object will be instantiated and called as such:
# obj = MyHashMap()
# obj.put(key,value)
# param_2 = obj.get(key)
# obj.remove(key)

#############

class MyHashMap: # hash to bucket
    def __init__(self):
        """
```

```python
        Initialize your data structure here.
        """
        self.size = 1000
        self.buckets = [[] for _ in range(self.size)]

    def _hash(self, key: int) -> int:
        """
        Generate a hash for a given key.
        """
        return key % self.size

    def put(self, key: int, value: int) -> None:
        """
        Value will always be non-negative.
        """
        hash_key = self._hash(key)
        for i, (k, v) in enumerate(self.buckets[hash_key]):
            if k == key:
                self.buckets[hash_key][i] = (key, value)
                return
        self.buckets[hash_key].append((key, value))

    def get(self, key: int) -> int:
        """
        Returns the value to which the specified key is mapped, or -1 if this map contains no mapping for
the key.
```



```python
        """
        hash_key = self._hash(key)
        for k, v in self.buckets[hash_key]:
            if k == key:
                return v
        return -1

    def remove(self, key: int) -> None:
        """
        Removes the mapping of the specified value key if this map contains a mapping for the key.
        """
        hash_key = self._hash(key)
        for i, (k, v) in enumerate(self.buckets[hash_key]):
            if k == key:
                del self.buckets[hash_key][i]
                return

##############

class MyHashMap: # open addressing with linear probing
    def __init__(self):
        self.capacity = 10000
        self.data = [None] * self.capacity
        self.DELETED = (None, None) # Marker for deleted entries

    def _hash(self, key):
        return key % self.capacity

    def put(self, key: int, value: int) -> None:
        idx = self._hash(key)
```

[*] Linked List — Pattern Solution (matched from community answers)

Python


```python
# ListNode class for handling collisions in each bucket
class ListNode:
    def __init__(self, key, value, next=None):
        self.key = key # Store key
        self.value = value # Store value
        self.next = next # Pointer to the next node

class MyHashMap:
    def __init__(self):
        self.size = 1000 # Define the bucket size
        self.buckets = [None] * self.size # Initialize buckets as an array of None

    def hash(self, key):
        # Simple modulo based hash function to map keys to bucket indices
        return key % self.size

    def put(self, key: int, value: int) -> None:
        index = self.hash(key)
        if self.buckets[index] is None:
            # If the bucket is empty, create a new node and place it here
            self.buckets[index] = ListNode(key, value)
        else:
            current = self.buckets[index]
            while True:
                if current.key == key:
```



```python
                    # If key exists, update its value
                    current.value = value
                    return
                if current.next is None:
                    break
                current = current.next
            # Append a new node at the end of the list if key not found
            current.next = ListNode(key, value)

    def get(self, key: int) -> int:
        index = self.hash(key)
        current = self.buckets[index]
        while current:
            if current.key == key:
                # Key found; return its value
                return current.value
            current = current.next
        # Key not found; return -1
        return -1

    def remove(self, key: int) -> None:
        index = self.hash(key)
        current = self.buckets[index]
        if current is None:
            return
```

```
        if current.key == key:
            # Remove head node if it contains the key
            self.buckets[index] = current.next
        else:
            prev = current
            current = current.next
            while current:
                if current.key == key:
                    # Bypass the node to remove it from the list
                    prev.next = current.next
                    return
                prev, current = current, current.next
```

#876 EASY

Middle of the Linked List

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Linked List · Two Pointers

Lists: Grind 169

Problem:

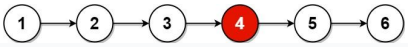
Given the head of a singly linked list, return the middle node of the linked list.

If there are two middle nodes, return the second middle node.

Example 1:

```
Input: head = [1,2,3,4,5]
Output: [3,4,5]
Explanation: The middle node of the list is node 3.
```

Example 2:




```
Input: head = [1,2,3,4,5,6]
Output: [4,5,6]
Explanation: Since the list has two middle nodes with values 3 and 4, we return the second
one.
```

Constraints:

- The number of nodes in the list is in the range [1, 100].
- 1 <= Node.val <= 100

Python

Python


```
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next

class Solution:
    def middleNode(self, head: ListNode) -> ListNode:
        # Initialize slow and fast pointers at the head of the list
        slow = head
        fast = head
        # Loop until fast reaches the end of the list
        while fast and fast.next:
            # Move slow pointer one step
            slow = slow.next
            # Move fast pointer two steps
            fast = fast.next.next
        # When loop ends, slow pointer points to the middle node
        return slow
```

Python

```
class Solution:
  def middleNode(self, head: ListNode) -> ListNode:
    slow = head
    fast = head

    while fast and fast.next:
      slow = slow.next
      fast = fast.next.next

    return slow
```

Python

```
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def middleNode(self, head: ListNode) -> ListNode:
        slow = fast = head
        while fast and fast.next:
            slow, fast = slow.next, fast.next.next
        return slow
```

Python

```
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def middleNode(self, head: ListNode) -> ListNode:
        slow = fast = head
        while fast and fast.next:
            slow, fast = slow.next, fast.next.next
        return slow
```


[*] Linked List — Pattern Solution (matched from community answers)

Python

```
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next

class Solution:
    def middleNode(self, head: ListNode) -> ListNode:
        # Initialize slow and fast pointers at the head of the list
        slow = head
        fast = head
        # Loop until fast reaches the end of the list
        while fast and fast.next:
            # Move slow pointer one step
            slow = slow.next
            # Move fast pointer two steps
            fast = fast.next.next
        # When loop ends, slow pointer points to the middle node
        return slow
```

#1474 EASY

Delete N Nodes After M Nodes of Linked List

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Linked List

Lists: Premium 98

Problem:

You are given the head of a linked list and two integers m and n.

Traverse the linked list and remove some nodes in the following way:

- Start with the head as the current node.
- Keep the first m nodes starting with the current node.
- Remove the next n nodes
- Keep repeating steps 2 and 3 until you reach the end of the list.

Return the head of the modified list after removing the mentioned nodes.

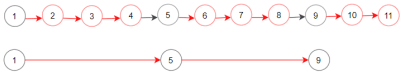
Example 1:

```
Input: head = [1,2,3,4,5,6,7,8,9,10,11,12,13], m = 2, n = 3
Output: [1,2,6,7,11,12]
Explanation: Keep the first (m = 2) nodes starting from the head of the linked List (1 ->2)
show in black nodes.
```



```
Delete the next (n = 3) nodes (3 -> 4 -> 5) show in read nodes.
Continue with the same procedure until reaching the tail of the Linked List.
Head of the linked list after removing nodes is returned.
```

Example 2:




```
Input: head = [1,2,3,4,5,6,7,8,9,10,11], m = 1, n = 3
Output: [1,5,9]
Explanation: Head of linked list after removing nodes is returned.
```

Constraints:

- The number of nodes in the list is in the range [1, 104].
- 1 <= Node.val <= 106
- 1 <= m, n <= 1000

Follow up: Could you solve this problem by modifying the list in-place?

Python

Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def deleteNodes(self, head: ListNode, m: int, n: int) -> ListNode:
        # Initialize the current pointer at the head of the list.
        current = head
        while current:
            # Keep m nodes: traverse m-1 next pointers
            for i in range(1, m):
                if current is None: # if list ends, break
                    break
                current = current.next
            # If we have reached the end of the list, break the loop.
            if current is None:
                break
            # Start deletion from the next node after the m-th node.
            temp = current.next
            # Delete next n nodes.
            for j in range(n):
                if temp is None:
                    break
```



```
            temp = temp.next
        # Connect the m-th node to the node after the n deleted nodes.
        current.next = temp
        # Move current pointer to continue from the next kept node.
        current = temp
    return head
```

Python

```
class Solution:
  def deleteNodes(
      self,
      head: ListNode | None,
      m: int,
      n: int,
  ) -> ListNode | None:
    curr = head
    prev = None # prev.next == curr

    while curr:
      # Set the m-th node as `prev`.
      for _ in range(m):
        if not curr:
          break
        prev = curr
        curr = curr.next
      # Set the (m + n + 1)-th node as `curr`.
      for _ in range(n):
        if not curr:
          break
        curr = curr.next
      # Delete the nodes [m + 1..n - 1].
      prev.next = curr

    return head
```

Python

```
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def deleteNodes(self, head: ListNode, m: int, n: int) -> ListNode:
        pre = head
        while pre:
            for _ in range(m - 1):
                if pre:
                    pre = pre.next
            if pre is None:
                return head
            cur = pre
            for _ in range(n):
                if cur:
                    cur = cur.next
            pre.next = None if cur is None else cur.next
            pre = pre.next
        return head
```


Python

```
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def deleteNodes(self, head: ListNode, m: int, n: int) -> ListNode:
        pre = head
        while pre:
            for _ in range(m - 1):
                if pre:
                    pre = pre.next
            if pre is None:
                return head
            cur = pre
            for _ in range(n):
                if cur:
                    cur = cur.next
            pre.next = None if cur is None else cur.next
            pre = pre.next
        return head
```

[*] Linked List — Pattern Solution (matched from community answers)

Python

```
class Solution:
  def deleteNodes(
      self,
      head: ListNode | None,
      m: int,
      n: int,
  ) -> ListNode | None:
    curr = head
    prev = None # prev.next == curr

    while curr:
      # Set the m-th node as `prev`.
      for _ in range(m):
        if not curr:
          break
        prev = curr
        curr = curr.next
      # Set the (m + n + 1)-th node as `curr`.
      for _ in range(n):
        if not curr:
          break
        curr = curr.next
      # Delete the nodes [m + 1..n - 1].
      prev.next = curr

    return head
```

#2 MEDIUM

Add Two Numbers

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Linked List · Math · Recursion

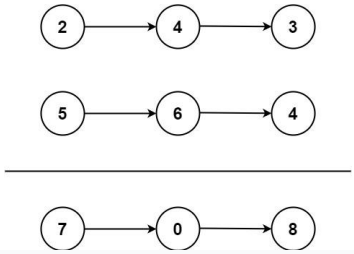
Lists: Grind 169

Problem:

You are given two non-empty linked lists representing two non-negative integers. The digits are stored in reverse order, and each of their nodes contains a single digit. Add the two numbers and return the sum as a linked list.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

Example 1:




```
Input: l1 = [2,4,3], l2 = [5,6,4]
Output: [7,0,8]
Explanation: 342 + 465 = 807.
```

Example 2:

```
Input: l1 = [0], l2 = [0]
Output: [0]
```

Example 3:

```
Input: l1 = [9,9,9,9,9,9,9], l2 = [9,9,9,9]
Output: [8,9,9,9,0,0,0,1]
```

Constraints:

- The number of nodes in each linked list is in the range [1, 100].
- 0 <= Node.val <= 9
- It is guaranteed that the list represents a number that does not have leading zeros.

>> Brute Force [Linked List] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def addTwoNumbers(self, l1, l2):
        # Brute Force - convert both lists to integers, add, convert back
        def to_int(node):
            num, mul = 0, 1
            while node:
                num += node.val * mul
                mul *= 10
                node = node.next
            return num
        total = to_int(l1) + to_int(l2)
        dummy = curr = ListNode(0)
        if total == 0: return ListNode(0)
        while total:
            curr.next = ListNode(total % 10)
            curr = curr.next
            total //= 10
        return dummy.next
```

Python

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def addTwoNumbers(l1, l2):
    # Initialize a dummy head node to simplify the result list construction.
    dummy = ListNode()
    current = dummy
    carry = 0

    # Traverse through both linked lists until all nodes and any carry are processed.
    while l1 or l2 or carry:
        # Get the current values; if a node is missing, use 0.
        val1 = l1.val if l1 else 0
        val2 = l2.val if l2 else 0

        # Calculate the sum of current digits and carry.
        total = val1 + val2 + carry
        carry = total // 10 # Update carry for next iteration.
        digit = total % 10 # Compute current digit.

        # Append the digit as a new node in the result list.
        current.next = ListNode(digit)
```



```
        current = current.next

        # Move to the next nodes if available.
        if l1:
            l1 = l1.next
        if l2:
            l2 = l2.next

    # Return the head of the newly formed linked list.
    return dummy.next
```

Python

```python
class Solution:
  def addTwoNumbers(self, l1: ListNode, l2: ListNode) -> ListNode:
    dummy = ListNode(0)
    curr = dummy
    carry = 0

    while carry or l1 or l2:
      if l1:
        carry += l1.val
        l1 = l1.next
      if l2:
        carry += l2.val
        l2 = l2.next
      curr.next = ListNode(carry % 10)
      carry //= 10
      curr = curr.next

    return dummy.next
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def addTwoNumbers(
        self, l1: Optional[ListNode], l2: Optional[ListNode]
    ) -> Optional[ListNode]:
        dummy = ListNode()
        carry, curr = 0, dummy
        while l1 or l2 or carry:
            s = (l1.val if l1 else 0) + (l2.val if l2 else 0) + carry
            carry, val = divmod(s, 10)
            curr.next = ListNode(val)
            curr = curr.next
            l1 = l1.next if l1 else None
            l2 = l2.next if l2 else None
        return dummy.next
```

Python


```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def addTwoNumbers(
        self, l1: Optional[ListNode], l2: Optional[ListNode]
    ) -> Optional[ListNode]:
        dummy = ListNode()
        carry, curr = 0, dummy
        while l1 or l2 or carry:
            s = (l1.val if l1 else 0) + (l2.val if l2 else 0) + carry
            carry, val = divmod(s, 10)
            curr.next = ListNode(val)
            curr = curr.next
            l1 = l1.next if l1 else None
            l2 = l2.next if l2 else None
        return dummy.next
```

[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

def addTwoNumbers(l1, l2):
    # Initialize a dummy head node to simplify the result list construction.
    dummy = ListNode()
    current = dummy
    carry = 0

    # Traverse through both linked lists until all nodes and any carry are processed.
    while l1 or l2 or carry:
        # Get the current values; if a node is missing, use 0.
        val1 = l1.val if l1 else 0
        val2 = l2.val if l2 else 0

        # Calculate the sum of current digits and carry.
        total = val1 + val2 + carry
        carry = total // 10 # Update carry for next iteration.
        digit = total % 10 # Compute current digit.

        # Append the digit as a new node in the result list.
        current.next = ListNode(digit)
```



```
        current = current.next

        # Move to the next nodes if available.
        if l1:
            l1 = l1.next
        if l2:
            l2 = l2.next

    # Return the head of the newly formed linked list.
    return dummy.next
```

#19 MEDIUM

Remove Nth Node From End of List

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

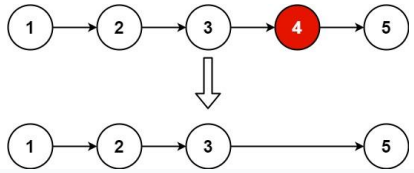
Linked List · Two Pointers

Lists: Grind 169

Problem:

Given the head of a linked list, remove the nth node from the end of the list and return its head.

Example 1:




```
Input: head = [1,2,3,4,5], n = 2
Output: [1,2,3,5]
```

Example 2:

```
Input: head = [1], n = 1
Output: []
```

Example 3:

```
Input: head = [1,2], n = 1
Output: [1]
```

Constraints:

- The number of nodes in the list is sz.
- 1 <= sz <= 30

- 0 <= Node.val <= 100
- 1 <= n <= sz

Follow up: Could you do this in one pass?

Python

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val # Holds the value of the node
        self.next = next # Points to the next node in the list

class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        # Create a dummy node to simplify edge cases (like removing the head)
        dummy = ListNode(0)
        dummy.next = head

        # Initialize two pointers starting at the dummy node
        first = dummy
        second = dummy

        # Advance first pointer n+1 steps ahead to maintain a gap of n nodes
        for _ in range(n + 1):
            first = first.next

        # Move both pointers until first reaches the end of the list
        while first:
            first = first.next
            second = second.next


        # Skip the node to remove it from the list
        second.next = second.next.next

        # Return the adjusted list starting from dummy.next
        return dummy.next
```

Python

```python
class Solution:
  def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
    slow = head
    fast = head

    for _ in range(n):
      fast = fast.next
    if not fast:
      return head.next

    while fast.next:
      slow = slow.next
      fast = fast.next
    slow.next = slow.next.next

    return head
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
        dummy = ListNode(next=head)
        fast = slow = dummy
        for _ in range(n):
            fast = fast.next
        while fast.next:
            slow, fast = slow.next, fast.next
        slow.next = slow.next.next
        return dummy.next
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def removeNthFromEnd(self, head: Optional[ListNode], n: int) -> Optional[ListNode]:
        dummy = ListNode(next=head)
        fast = slow = dummy
        for _ in range(n):
            fast = fast.next
        while fast.next:
            slow, fast = slow.next, fast.next
        slow.next = slow.next.next
        return dummy.next
```

[*] Linked List — Pattern Solution (matched from community answers)

Python


```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val # Holds the value of the node
        self.next = next # Points to the next node in the list

class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        # Create a dummy node to simplify edge cases (like removing the head)
        dummy = ListNode(0)
        dummy.next = head

        # Initialize two pointers starting at the dummy node
        first = dummy
        second = dummy

        # Advance first pointer n+1 steps ahead to maintain a gap of n nodes
        for _ in range(n + 1):
            first = first.next

        # Move both pointers until first reaches the end of the list
        while first:
            first = first.next
            second = second.next


        # Skip the node to remove it from the list
        second.next = second.next.next

        # Return the adjusted list starting from dummy.next
        return dummy.next
```

#24 MEDIUM

Swap Nodes in Pairs

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Linked List · Recursion

Lists: Grind 169

Problem:

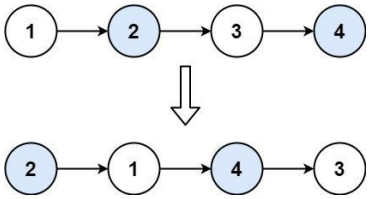
Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

Example 1:

Input: head = [1,2,3,4]

Output: [2,1,4,3]

Explanation:



Example 2:

Input: head = []

Output: []

Example 3:

Input: head = [1]

Output: [1]

Example 4:

Input: head = [1,2,3]

Output: [2,1,3]

Constraints:

- The number of nodes in the list is in the range [0, 100].
- 0 <= Node.val <= 100

Python

Python


```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        # Create a dummy node and point it to the head of the list.
        dummy = ListNode(0)
        dummy.next = head
        # Initialize the pointer 'prev' to dummy.
        prev = dummy

        # Traverse the list while at least two nodes remain.
        while head and head.next:
            # Identify the two nodes to swap.
            first_node = head
            second_node = head.next

            # Swapping the nodes by reassigning pointers.
            prev.next = second_node # Link previous node to second node.
            first_node.next = second_node.next # Link first node to node after second.
            second_node.next = first_node # Link second node to first to complete swap.


            # Update the pointers for next pair processing.
            prev = first_node # Move 'prev' pointer to the end of the swapped pair.
            head = first_node.next # Move 'head' pointer to the next pair.

        # Return the new head of the list.
        return dummy.next
```

Python

```python
class Solution:
  def swapPairs(self, head: ListNode) -> ListNode:
    def getLength(head: ListNode) -> int:
      length = 0
      while head:
        length += 1
        head = head.next
      return length

    length = getLength(head)
    dummy = ListNode(0, head)
    prev = dummy
    curr = head

    for _ in range(length // 2):
      next = curr.next
      curr.next = next.next
      next.next = prev.next
      prev.next = next
      prev = curr
      curr = curr.next

    return dummy.next
```

Python


```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if head is None or head.next is None:
            return head
        t = self.swapPairs(head.next.next)
        p = head.next
        p.next = head
        head.next = t
        return p
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def swapPairs(self, head: Optional[ListNode]) -> Optional[ListNode]:
        dummy = ListNode(next=head)
        pre, cur = dummy, head
        while cur and cur.next:
            t = cur.next
            cur.next = t.next
            t.next = cur
            pre.next = t
            pre, cur = cur, cur.next
        return dummy.next
```

[*] Linked List — Pattern Solution (matched from community answers)

Python


```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def swapPairs(self, head: ListNode) -> ListNode:
        # Create a dummy node and point it to the head of the list.
        dummy = ListNode(0)
        dummy.next = head
        # Initialize the pointer 'prev' to dummy.
        prev = dummy

        # Traverse the list while at least two nodes remain.
        while head and head.next:
            # Identify the two nodes to swap.
            first_node = head
            second_node = head.next

            # Swapping the nodes by reassigning pointers.
            prev.next = second_node # Link previous node to second node.
            first_node.next = second_node.next # Link first node to node after second.
            second_node.next = first_node # Link second node to first to complete swap.


            # Update the pointers for next pair processing.
            prev = first_node # Move 'prev' pointer to the end of the swapped pair.
            head = first_node.next # Move 'head' pointer to the next pair.

        # Return the new head of the list.
        return dummy.next
```

#61 MEDIUM

Rotate List

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

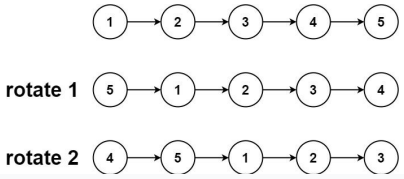
Linked List · Two Pointers

Lists: Grind 169

Problem:

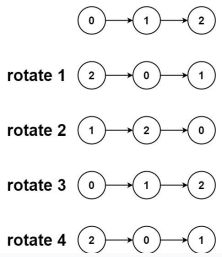
Given the head of a linked list, rotate the list to the right by k places.

Example 1:




```
Input: head = [1,2,3,4,5], k = 2
Output: [4,5,1,2,3]
```

Example 2:




```
Input: head = [0,1,2], k = 4
Output: [2,0,1]
```

Constraints:

- The number of nodes in the list is in the range [0, 500].
- -100 <= Node.val <= 100
- 0 <= k <= 2 * 109

Python

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, x):
# self.val = x
# self.next = None

class Solution:
    def rotateRight(self, head: ListNode, k: int) -> ListNode:
        if not head or not head.next or k == 0: # Check if rotation is necessary
            return head

        # Compute the length of the list and locate the tail node
        length = 1
        tail = head
        while tail.next:
            tail = tail.next
            length += 1

        # Calculate the effective number of rotations
        k = k % length
        if k == 0:
            return head

        # Find the new tail: (length - k - 1) steps from the head
        stepsToNewTail = length - k - 1

        newTail = head
        for _ in range(stepsToNewTail):
            newTail = newTail.next

        # Set the new head and break the list
        newHead = newTail.next
        newTail.next = None

        # Connect the old tail to the original head
        tail.next = head

        return newHead
```

Python


```python
class Solution:
   def rotateRight(self, head: ListNode, k: int) -> ListNode:
     if not head or not head.next or k == 0:
       return head

     tail = head
     length = 1
     while tail.next:
       tail = tail.next
       length += 1
     tail.next = head # Circle the list.

     t = length - k % length
     for _ in range(t):
       tail = tail.next
     newHead = tail.next
     tail.next = None

     return newHead
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
        if head is None or head.next is None:
            return head
        cur, n = head, 0
        while cur:
            n += 1
            cur = cur.next
        k %= n
        if k == 0:
            return head
        fast = slow = head
        for _ in range(k):
            fast = fast.next
        while fast.next:
            fast, slow = fast.next, slow.next

        ans = slow.next
        slow.next = None
        fast.next = head

        return ans
```

Python


```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def rotateRight(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
        if head is None or head.next is None:
            return head
        cur, n = head, 0
        while cur:
            n += 1
            cur = cur.next
        k %= n
        if k == 0:
            return head
        fast = slow = head
        for _ in range(k):
            fast = fast.next
        while fast.next:
            fast, slow = fast.next, slow.next

        ans = slow.next
        slow.next = None
        fast.next = head

        return ans
```

[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, x):
# self.val = x
# self.next = None

class Solution:
    def rotateRight(self, head: ListNode, k: int) -> ListNode:
        if not head or not head.next or k == 0: # Check if rotation is necessary
            return head

        # Compute the length of the list and locate the tail node
        length = 1
        tail = head
        while tail.next:
            tail = tail.next
            length += 1

        # Calculate the effective number of rotations
        k = k % length
        if k == 0:
            return head

        # Find the new tail: (length - k - 1) steps from the head
        stepsToNewTail = length - k - 1
```



```python
        newTail = head
        for _ in range(stepsToNewTail):
            newTail = newTail.next

        # Set the new head and break the list
        newHead = newTail.next
        newTail.next = None

        # Connect the old tail to the original head
        tail.next = head

        return newHead
```

#146 MEDIUM

LRU Cache

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · Linked List · Design

Lists: Grind 169

Problem:

Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.

Implement the LRUCache class:

- LRUCache(int capacity) Initialize the LRU cache with positive size capacity.
- int get(int key) Return the value of the key if the key exists, otherwise return -1.
- void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, evict the least recently used key.

The functions get and put must each run in O(1) average time complexity.

Example 1:

```
Input
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, null, -1, 3, 4]

Explanation
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // cache is {1=1}
lRUCache.put(2, 2); // cache is {1=1, 2=2}
lRUCache.get(1); // return 1
lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}
lRUCache.get(2); // returns -1 (not found)
lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}
lRUCache.get(1); // return -1 (not found)
lRUCache.get(3); // return 3
lRUCache.get(4); // return 4
```

Constraints:

- 1 <= capacity <= 3000
- 0 <= key <= 104
- 0 <= value <= 105
- At most 2 * 105 calls will be made to get and put.

Python

Python

```python
# Define a node for the doubly linked list.
class Node:
    def __init__(self, key, value):
        self.key = key # key of the node
        self.value = value # value of the node
        self.prev = None # pointer to previous node
        self.next = None # pointer to next node

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity # maximum capacity of the cache
        self.cache = {} # dictionary mapping key -> node
        # Create dummy head and tail nodes.
        self.head = Node(0, 0)
        self.tail = Node(0, 0)
        self.head.next = self.tail
        self.tail.prev = self.head

    # Helper function to remove a node from the linked list.
    def _remove(self, node):
        prev_node = node.prev
        next_node = node.next
        prev_node.next = next_node
        next_node.prev = prev_node

    # Helper function to add a node right after the head.
    def _add(self, node):
        node.prev = self.head
        node.next = self.head.next
        self.head.next.prev = node
        self.head.next = node

    # Returns the value of the key if it exists; otherwise, returns -1.
    def get(self, key):
        if key in self.cache:
            node = self.cache[key]
            # Move the accessed node to the head (marking it as recently used)
            self._remove(node)
            self._add(node)
            return node.value
        return -1

    # Updates the value of the key if it exists; otherwise, adds the key-value pair.
    # Removes the least recently used key if the cache exceeds its capacity.
    def put(self, key, value):
        if key in self.cache:
            # Remove the old node.
            self._remove(self.cache[key])
        node = Node(key, value)
        self._add(node)
```

```
        self.cache[key] = node
        if len(self.cache) > self.capacity:
            # Remove the least recently used element.
            lru = self.tail.prev
            self._remove(lru)
            del self.cache[lru.key]
```

Python

```
class Node:
    def __init__(self, key: int, value: int):
      self.key = key
      self.value = value
      self.prev = None
      self.next = None


class LRUCache:
    def __init__(self, capacity: int):
      self.capacity = capacity
      self.keyToNode = {}
      self.head = Node(-1, -1)
      self.tail = Node(-1, -2)
      self.join(self.head, self.tail)

    def get(self, key: int) -> int:
      if key not in self.keyToNode:
        return -1

      node = self.keyToNode[key]
      self.remove(node)
      self.moveToHead(node)
      return node.value
```

```
    def put(self, key: int, value: int) -> None:
      if key in self.keyToNode:
        node = self.keyToNode[key]
        node.value = value
        self.remove(node)
        self.moveToHead(node)
        return

      if len(self.keyToNode) == self.capacity:
        lastNode = self.tail.prev
        del self.keyToNode[lastNode.key]
        self.remove(lastNode)

      self.moveToHead(Node(key, value))
      self.keyToNode[key] = self.head.next

    def join(self, node1: Node, node2: Node):
      node1.next = node2
      node2.prev = node1

    def moveToHead(self, node: Node):
      self.join(node, self.head.next)
      self.join(self.head, node)

    def remove(self, node: Node):
```



```
    self.join(node.prev, node.next)
```

Python

```
class Node:
    def __init__(self, key=0, val=0):
        self.key = key
        self.val = val
        self.prev = None
        self.next = None


class LRUCache:
    def __init__(self, capacity: int):
        self.cache = {} # key -> Node(val)
        self.head = Node() # dummy node
        self.tail = Node() # dummy node
        self.capacity = capacity
        self.size = 0
        self.head.next = self.tail # note: key setup
        self.tail.prev = self.head

    def get(self, key: int) -> int:
        if key not in self.cache:
            return -1
        node = self.cache[key]
        self.move_to_head(node)
        return node.val
```

```
    def put(self, key: int, value: int) -> None:
        if key in self.cache:
            node = self.cache[key]
            node.val = value
            self.move_to_head(node)
        else:
            node = Node(key, value)
            self.cache[key] = node
            self.add_to_head(node)
            self.size += 1
            if self.size > self.capacity:
                tail = self.remove_tail()
                self.cache.pop(tail.key)
                self.size -= 1

    def move_to_head(self, node):
        self.remove_node(node)
        self.add_to_head(node)

    def remove_node(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def add_to_head(self, node):
        node.next = self.head.next
```



```
        node.prev = self.head
        self.head.next = node
        node.next.prev = node

    def remove_tail(self):
        node = self.tail.prev
        self.remove_node(node)
        return node


# Your LRUCache object will be instantiated and called as such:
# obj = LRUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)

###########

'''
example:

>>> od = collections.OrderedDict()
>>>
>>> od[1]=1
>>> od[2]=2
>>> od[3]=3
```

```
>>>
>>> od
OrderedDict([(1, 1), (2, 2), (3, 3)])
>>> od.move_to_end(1)
>>> od
```

[*] Linked List — Pattern Solution (matched from community answers)

Python


```
class Node:
  def __init__(self, key: int, value: int):
    self.key = key
    self.value = value
    self.prev = None
    self.next = None


class LRUCache:
  def __init__(self, capacity: int):
    self.capacity = capacity
    self.keyToNode = {}
    self.head = Node(-1, -1)
    self.tail = Node(-1, -1)
    self.join(self.head, self.tail)

  def get(self, key: int) -> int:
    if key not in self.keyToNode:
      return -1

    node = self.keyToNode[key]
    self.remove(node)
    self.moveToHead(node)
    return node.value
```

```
  def put(self, key: int, value: int) -> None:
    if key in self.keyToNode:
      node = self.keyToNode[key]
      node.value = value
      self.remove(node)
      self.moveToHead(node)
      return

    if len(self.keyToNode) == self.capacity:
      lastNode = self.tail.prev
      del self.keyToNode[lastNode.key]
      self.remove(lastNode)

    self.moveToHead(Node(key, value))
    self.keyToNode[key] = self.head.next

  def join(self, node1: Node, node2: Node):
    node1.next = node2
    node2.prev = node1

  def moveToHead(self, node: Node):
    self.join(node, self.head.next)
    self.join(self.head, node)

  def remove(self, node: Node):
```

```
    self.join(node.prev, node.next)
```

#148 MEDIUM

Sort List

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

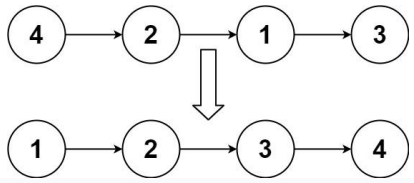
Linked List · Two Pointers · Divide and Conquer · Sorting · Merge Sort

Lists: Grind 169

Problem:

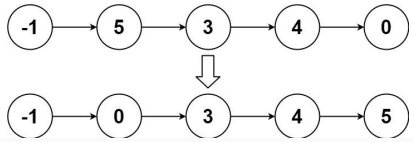
Given the head of a linked list, return the list after sorting it in ascending order.

Example 1:




```
Input: head = [4,2,1,3]
Output: [1,2,3,4]
```

Example 2:




```
Input: head = [-1,5,3,4,0]
Output: [-1,0,3,4,5]
```

Example 3:

```
Input: head = []
Output: []
```

Constraints:

- The number of nodes in the list is in the range [0, 5 * 104].
- -105 <= Node.val <= 105

Follow up: Can you sort the linked list in O(n logn) time and O(1) memory (i.e. constant space)?

Python

Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def sortList(self, head: ListNode) -> ListNode:
        # Base case: if list is empty or contains a single element.
        if not head or not head.next:
            return head

        # Function to split the list into two halves
        def getMid(head):
            slow, fast = head, head.next
            # Move fast twice as fast as slow to locate middle
            while fast and fast.next:
                slow = slow.next
                fast = fast.next.next
            return slow

        # Split the linked list into two halves
        mid = getMid(head)
        right_head = mid.next
        mid.next = None # cut the list into two parts
```

```
        # Recursive calls to sort each half
        left = self.sortList(head)
        right = self.sortList(right_head)

        # Merge two sorted lists
        return merge(left, right)

def merge(l1, l2):
    dummy = ListNode(0)
    tail = dummy
    # Merge the lists by comparing nodes
    while l1 and l2:
        if l1.val <= l2.val:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next
    # Append the remaining nodes
    tail.next = l1 if l1 else l2
    return dummy.next
```

Python

```python
class Solution:
    def sortList(self, head: ListNode) -> ListNode:
      def split(head: ListNode, k: int) -> ListNode:
        while k > 1 and head:
          head = head.next
          k -= 1
        rest = head.next if head else None
        if head:
          head.next = None
        return rest

      def merge(l1: ListNode, l2: ListNode) -> tuple:
        dummy = ListNode(0)
        tail = dummy

        while l1 and l2:
          if l1.val > l2.val:
            l1, l2 = l2, l1
          tail.next = l1
          l1 = l1.next
          tail = tail.next
        tail.next = l1 if l1 else l2
        while tail.next:
          tail = tail.next

        return dummy.next, tail

      length = 0
      curr = head
      while curr:
        length += 1
        curr = curr.next

      dummy = ListNode(0, head)

      k = 1
      while k < length:
        curr = dummy.next
        tail = dummy
        while curr:
          l = curr
          r = split(l, k)
          curr = split(r, k)
          mergedHead, mergedTail = merge(l, r)
          tail.next = mergedHead
          tail = mergedTail
        k *= 2

      return dummy.next
```

Python


```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def sortList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if head is None or head.next is None:
            return head
        slow, fast = head, head.next
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next
        l1, l2 = head, slow.next
        slow.next = None
        l1, l2 = self.sortList(l1), self.sortList(l2)
        dummy = ListNode()
        tail = dummy
        while l1 and l2:
            if l1.val <= l2.val:
                tail.next = l1
                l1 = l1.next
            else:
                tail.next = l2
                l2 = l2.next

            tail = tail.next
        tail.next = l1 or l2
        return dummy.next
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def sortList(self, head: ListNode) -> ListNode:
        if head is None or head.next is None:
            return head
        slow, fast = head, head.next
        while fast and fast.next:
            slow, fast = slow.next, fast.next.next
        t = slow.next
        slow.next = None
        l1, l2 = self.sortList(head), self.sortList(t)
        dummy = ListNode()
        cur = dummy
        while l1 and l2:
            if l1.val <= l2.val:
                cur.next = l1
                l1 = l1.next
            else:
                cur.next = l2
                l2 = l2.next
            cur = cur.next
```



```python
        cur.next = l1 or l2 # add the rest
        return dummy.next
```

[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def sortList(self, head: ListNode) -> ListNode:
        # Base case: if list is empty or contains a single element.
        if not head or not head.next:
            return head

        # Function to split the list into two halves
        def getMid(head):
            slow, fast = head, head.next
            # Move fast twice as fast as slow to locate middle
            while fast and fast.next:
                slow = slow.next
                fast = fast.next.next
            return slow

        # Split the linked list into two halves
        mid = getMid(head)
        right_head = mid.next
        mid.next = None # cut the list into two parts

        # Recursive calls to sort each half
        left = self.sortList(head)
        right = self.sortList(right_head)

        # Merge two sorted lists
        return merge(left, right)

def merge(l1, l2):
    dummy = ListNode(0)
    tail = dummy
    # Merge the lists by comparing nodes
    while l1 and l2:
        if l1.val < l2.val:
            tail.next = l1
            l1 = l1.next
        else:
            tail.next = l2
            l2 = l2.next
        tail = tail.next
    # Append the remaining nodes
    tail.next = l1 if l1 else l2
    return dummy.next
```


#328 MEDIUM

Odd Even Linked List

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Linked List

Lists: Grind 169

Problem:

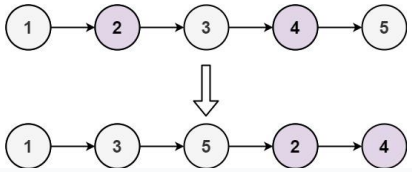
Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return the reordered list.

The first node is considered odd, and the second node is even, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

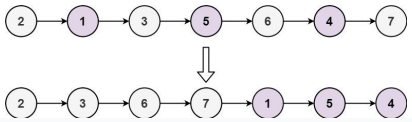
You must solve the problem in O(1) extra space complexity and O(n) time complexity.

Example 1:




```
Input: head = [1,2,3,4,5]
Output: [1,3,5,2,4]
```

Example 2:




```
Input: head = [2,1,3,5,6,4,7]
Output: [2,3,6,7,1,5,4]
```

Constraints:

- The number of nodes in the linked list is in the range [0, 104].

- -106 <= Node.val <= 106

Python

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def oddEvenList(self, head: ListNode) -> ListNode:
        # Check for empty list or list with one node.
        if not head or not head.next:
            return head

        odd = head # Point to the first (odd indexed) node.
        even = head.next # Point to the second (even indexed) node.
        even_head = even # Save the head of the even list.

        # Reorder the list by rearranging next pointers.
        while even and even.next:
            odd.next = even.next # Connect current odd to next odd.
            odd = odd.next # Move odd pointer.

            even.next = odd.next # Connect current even to next even.
            even = even.next # Move even pointer.

        # Append the even list after the odd list.

        odd.next = even_head
        return head
```

Python

```python
class Solution:
    def oddEvenList(self, head: ListNode) -> ListNode:
        oddHead = ListNode(0)
        evenHead = ListNode(0)
        odd = oddHead
        even = evenHead
        isOdd = True

        while head:
          if isOdd:
            odd.next = head
            odd = head
          else:
            even.next = head
            even = head
          head = head.next
          isOdd = not isOdd

        even.next = None
        odd.next = evenHead.next
        return oddHead.next
```

Python


```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def oddEvenList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if head is None:
            return None
        a = head
        b = c = head.next
        while b and b.next:
            a.next = b.next
            a = a.next
            b.next = a.next
            b = b.next
        a.next = c
        return head
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def oddEvenList(self, head: Optional[ListNode]) -> Optional[ListNode]:
        if head is None:
            return None
        a = head
        b = c = head.next
        while b and b.next:
            a.next = b.next
            a = a.next
            b.next = a.next
            b = b.next
        a.next = c
        return head
```

[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def oddEvenList(self, head: ListNode) -> ListNode:
        # Check for empty list or list with one node.
        if not head or not head.next:
            return head

        odd = head # Point to the first (odd indexed) node.
        even = head.next # Point to the second (even indexed) node.
        even_head = even # Save the head of the even list.

        # Rearrange the list by rearranging next pointers.
        while even and even.next:
            odd.next = even.next # Connect current odd to next odd.
            odd = odd.next # Move odd pointer.

            even.next = odd.next # Connect current even to next even.
            even = even.next # Move even pointer.

        # Append the even list after the odd list.

        odd.next = even_head
        return head
```

#369 MEDIUM

Plus One Linked List

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Linked List · Math · Recursion

Lists: Premium 98

Problem:

Given a non-negative integer represented as a linked list of digits, plus one to the integer.

The digits are stored such that the most significant digit is at the head of the list.

Example 1:

```
Input: head = [1,2,3]
Output: [1,2,4]
```

Example 2:

```
Input: head = [0]
Output: [1]
```

Constraints:

- The number of nodes in the linked list is in the range [1, 100].
- 0 <= Node.val <= 9
- The number represented by the linked list does not contain leading zeros except for the zero itself.

>> Brute Force [Linked List] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def reverseList(self, head):
        # Brute Force - collect into list, manipulate, rebuild
        vals = []
        curr = head
        while curr:
            vals.append(curr.val)
            curr = curr.next
        dummy = ListNode(0)
        curr = dummy
        for v in vals:
            curr.next = ListNode(v)
            curr = curr.next
        return dummy.next
```

Python

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

# Helper function to reverse the linked list
def reverseList(head):
    prev = None
    curr = head
    while curr:
        nxt = curr.next # store next node
        curr.next = prev # reverse pointer
        prev = curr # move prev forward
        curr = nxt # move to next node
    return prev

def plusOne(head: ListNode) -> ListNode:
    # Reverse the linked list to start addition from the least significant digit
    head = reverseList(head)

    current = head
    carry = 1 # initialize carry as the plus one
    while current and carry:
        current.val += carry # add carry to the current node

        carry = current.val // 10 # calculate new carry
        current.val %= 10 # update the node's value
        # If we're at the end and there's still a carry, create a new node.
        if not current.next and carry:
            current.next = ListNode(0)
        current = current.next

    # Reverse the list back to its original order before returning.
    return reverseList(head)
```

Python


```python
class Solution:
    def plusOne(self, head: ListNode) -> ListNode:
        if not head:
            return ListNode(1)
        if self._addOne(head) == 1:
            return ListNode(1, head)
        return head

    def _addOne(self, node: ListNode) -> int:
        carry = self._addOne(node.next) if node.next else 1
        sum = node.val + carry
        node.val = sum % 10
        return sum // 10
```

Python

```python
class Solution:
    def plusOne(self, head: ListNode) -> ListNode:
        dummy = ListNode(0)
        curr = dummy
        dummy.next = head

        while head:
            if head.val != 9:
                curr = head
            head = head.next
        # `curr` now points to the rightmost non-9 node.

        curr.val += 1
        while curr.next:
            curr.next.val = 0
            curr = curr.next

        return dummy.next if dummy.val == 0 else dummy
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def plusOne(self, head: Optional[ListNode]) -> Optional[ListNode]:
        dummy = ListNode(0, head)
        target = dummy
        while head:
            if head.val != 9:
                target = head
            head = head.next
        target.val += 1
        target = target.next
        while target:
            target.val = 0
            target = target.next
        return dummy if dummy.val else dummy.next
```

Python


```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def plusOne(self, head: ListNode) -> ListNode:
        dummy = ListNode(0, head)
        target = dummy
        while head:
            if head.val != 9:
                target = head
            head = head.next
        target.val += 1
        target = target.next
        while target:
            target.val = 0
            target = target.next
        return dummy if dummy.val else dummy.next
```

[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def plusOne(self, head: ListNode) -> ListNode:
        dummy = ListNode(0)
        curr = dummy
        dummy.next = head

        while head:
          if head.val != 9:
            curr = head
          head = head.next
        # `curr` now points to the rightmost non-9 node.

        curr.val += 1
        while curr.next:
          curr.next.val = 0
          curr = curr.next

        return dummy.next if dummy.val == 0 else dummy
```

#430 MEDIUM

Flatten a Multilevel Doubly Linked List

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Linked List · DFS · Doubly Linked List

Lists: Premium 98

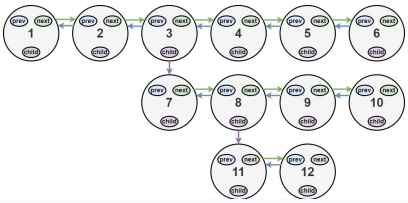
Problem:

You are given a doubly linked list, which contains nodes that have a next pointer, a previous pointer, and an additional child pointer. This child pointer may or may not point to a separate doubly linked list, also containing these special nodes. These child lists may have one or more children of their own, and so on, to produce a multilevel data structure as shown in the example below.

Given the head of the first level of the list, flatten the list so that all the nodes appear in a single-level, doubly linked list. Let curr be a node with a child list. The nodes in the child list should appear after curr and before curr.next in the flattened list.

Return the head of the flattened list. The nodes in the list must have all of their child pointers set to null.

Example 1:




```
Input: head = [1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]
Output: [1,2,3,7,8,11,12,9,10,4,5,6]
Explanation: The multilevel linked list in the input is shown.
After flattening the multilevel linked list it becomes:
```

Example 2:

```
Input: head = [1,2,null,3]
Output: [1,3,2]
Explanation: The multilevel linked list in the input is shown.
After flattening the multilevel linked list it becomes:
```

Example 3:

```
Input: head = []
Output: []
Explanation: There could be empty list in the input.
```

Constraints:

- The number of Nodes will not exceed 1000.
- 1 <= Node.val <= 105

How the multilevel linked list is represented in test cases:

We use the multilevel linked list from Example 1 above:

```
1---2---3---4---5---6--NULL
|
7---8---9---10--NULL
|
11--12--NULL
```


The serialization of each level is as follows:

```
[1,2,3,4,5,6,null]
[7,8,9,10,null]
[11,12,null]
```

To serialize all levels together, we will add nulls in each level to signify no node connects to the upper node of the previous level. The serialization becomes:

```
[1, 2, 3, 4, 5, 6, null]
|
[null, null, 7, 8, 9, 10, null]
|
[ null, 11, 12, null]
```

Merging the serialization of each level and removing trailing nulls we obtain:

```
[1,2,3,4,5,6,null,null,null,7,8,9,10,null,null,11,12]
```

>> Brute Force [Linked List] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def flatten(self, head):
        # Brute Force - collect into list, manipulate, rebuild
        vals = []
        curr = head
        while curr:
            vals.append(curr.val)
            curr = curr.next
        dummy = ListNode(0)
        curr = dummy
        for v in vals:
            curr.next = ListNode(v)
            curr = curr.next
        return dummy.next
```

Python

Python

```python
# Definition for a Node.
class Node:
    def __init__(self, val, prev=None, next=None, child=None):
        self.val = val
        self.prev = prev
        self.next = next
        self.child = child

def flatten(head):
    # If the list is empty, simply return None.
    if not head:
        return head

    # Helper function to perform DFS and return the tail of the flattened list.
    def dfs(node):
        current = node
        last = None
        while current:
            next_node = current.next # Store current.next before modifications
            # If current node has a child, we flatten the child list recursively.
            if current.child:
                # Recursively flatten the child list.
                child_tail = dfs(current.child)
                # Connect current node with the start of the child list.
                current.next = current.child

                current.child.prev = current
                # If there was a next node, attach it after the child list.
                if next_node:
                    child_tail.next = next_node
                    next_node.prev = child_tail
                # Set child pointer to None as required.
                current.child = None
                # Update the tail pointer.
                last = child_tail
            else:
                last = current
            # Move to the next node (which might have been updated after flattening a child).
            current = next_node
        return last

    dfs(head) # Begin DFS traversal from the head.
    return head

# Example to illustrate usage:
# head = Node(1)
# head.next = Node(2)
# head.child = Node(3)
# flatten(head)
```

Python


```python
class Solution:
    def flatten(self, head: 'Node') -> 'Node':
        def flatten(head: 'Node', rest: 'Node') -> 'Node':
            if not head:
                return rest

            head.next = flatten(head.child, flatten(head.next, rest))
            if head.next:
                head.next.prev = head
            head.child = None
            return head

        return flatten(head, None)
```

Python

```python
class Solution:
    def flatten(self, head: 'Node') -> 'Node':
        curr = head

        while curr:
            if curr.child:
                cachedNext = curr.next
                curr.next = curr.child
                curr.child.prev = curr
                curr.child = None
                tail = curr.next
                while tail.next:
                    tail = tail.next
                tail.next = cachedNext
                if cachedNext:
                    cachedNext.prev = tail
            curr = curr.next

        return head
```

Python


```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val, prev, next, child):
        self.val = val
        self.prev = prev
        self.next = next
        self.child = child
"""


class Solution:
    def flatten(self, head: 'Node') -> 'Node':
        def preorder(pre, cur):
            if cur is None:
                return pre
            cur.prev = pre
            pre.next = cur

            t = cur.next
            tail = preorder(cur, cur.child)
            cur.child = None
            return preorder(tail, t)

        if head is None:

            return None
        dummy = Node(0, None, head, None)
        preorder(dummy, head)
        dummy.next.prev = None
        return dummy.next
```

Python

```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val, prev, next, child):
        self.val = val
        self.prev = prev
        self.next = next
        self.child = child
"""


class Solution:
    def flatten(self, head: 'Node') -> 'Node':
        def preorder(pre, cur):
            if cur is None:
                return pre
            cur.prev = pre
            pre.next = cur

            t = cur.next
            tail = preorder(cur, cur.child)
            cur.child = None
            return preorder(tail, t)

        if head is None:
```



```python
            return None
        dummy = Node(0, None, head, None)
        preorder(dummy, head)
        dummy.next.prev = None
        return dummy.next
```

[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
# Definition for a Node.
class Node:
    def __init__(self, val, prev=None, next=None, child=None):
        self.val = val
        self.prev = prev
        self.next = next
        self.child = child

def flatten(head):
    # If the list is empty, simply return None.
    if not head:
        return head

    # Helper function to perform DFS and return the tail of the flattened list.
    def dfs(node):
        current = node
        last = None
        while current:
            next_node = current.next # Store current.next before modifications
            # If current node has a child, we flatten the child list recursively.
            if current.child:
                # Recursively flatten the child list.
                child_tail = dfs(current.child)
                # Connect current node with the start of the child list.
                current.next = current.child

                current.child.prev = current
                # If there was a next node, attach it after the child list.
                if next_node:
                    child_tail.next = next_node
                    next_node.prev = child_tail
                # Set child pointer to None as required.
                current.child = None
                # Update the tail pointer.
                last = child_tail
            else:
                last = current
            # Move to the next node (which might have been updated after flattening a child).
            current = next_node
        return last

    dfs(head) # Begin DFS traversal from the head.
    return head

# Example to illustrate usage:
# head = Node(1)
# head.next = Node(2)
# head.child = Node(3)
# flatten(head)
```


#622 MEDIUM

Design Circular Queue

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Linked List · Design · Queue

Lists: Denny Zhang

Problem:

Design your implementation of the circular queue. The circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle, and the last position is connected back to the first position to make a circle. It is also called "Ring Buffer".

One of the benefits of the circular queue is that we can make use of the spaces in front of the queue. In a normal queue, once the queue becomes full, we cannot insert the next element even if there is a space in front of the queue. But using the circular queue, we can use the space to store new values.

Implement the MyCircularQueue class:

- MyCircularQueue(k) Initializes the object with the size of the queue to be k.
- int Front() Gets the front item from the queue. If the queue is empty, return -1.
- int Rear() Gets the last item from the queue. If the queue is empty, return -1.
- boolean enQueue(int value) Inserts an element into the circular queue. Return true if the operation is successful.
- boolean deQueue() Deletes an element from the circular queue. Return true if the operation is successful.
- boolean isEmpty() Checks whether the circular queue is empty or not.
- boolean isFull() Checks whether the circular queue is full or not.

You must solve the problem without using the built-in queue data structure in your programming language.

Example 1:

```
Input
["MyCircularQueue", "enQueue", "enQueue", "enQueue", "enQueue", "Rear", "isFull", "deQueue",
"enQueue", "Rear"]
[[3], [1], [2], [3], [4], [], [], [], [4], []]
Output
[null, true, true, true, false, 3, true, true, true, 4]

Explanation
MyCircularQueue myCircularQueue = new MyCircularQueue(3);
myCircularQueue.enQueue(1); // return True
myCircularQueue.enQueue(2); // return True
myCircularQueue.enQueue(3); // return True
myCircularQueue.enQueue(4); // return False
myCircularQueue.Rear(); // return 3
myCircularQueue.isFull(); // return True
myCircularQueue.deQueue(); // return True
myCircularQueue.enQueue(4); // return True
myCircularQueue.Rear(); // return 4
```


Constraints:

- 1 <= k <= 1000
- 0 <= value <= 1000
- At most 3000 calls will be made to enQueue, deQueue, Front, Rear, isEmpty, and isFull.

Python

Python

```python
# Python implementation of MyCircularQueue

class MyCircularQueue:
    def __init__(self, k):
        # Initialize the queue with capacity k
        self.capacity = k
        self.queue = [0] * k # fixed-size array
        self.front = 0 # pointer to the front element
        self.rear = 0 # pointer to the next insertion position
        self.count = 0 # current number of elements in the queue

    def enQueue(self, value):
        # Check if the queue is full
        if self.isFull():
            return False
        # Insert element at rear pointer and update pointer using modulo for circularity
        self.queue[self.rear] = value
        self.rear = (self.rear + 1) % self.capacity
        self.count += 1
        return True

    def deQueue(self):
        # Check if the queue is empty
        if self.isEmpty():
            return False
```

```
        # Move front pointer to the next element using modulo for circularity
        self.front = (self.front + 1) % self.capacity
        self.count -= 1
        return True

    def Front(self):
        # Return -1 if the queue is empty, else return the front element
        if self.isEmpty():
            return -1
        return self.queue[self.front]

    def Rear(self):
        # Return -1 if the queue is empty
        if self.isEmpty():
            return -1
        # Since rear points to the next insertion position, get the last element index
        return self.queue[(self.rear - 1 + self.capacity) % self.capacity]

    def isEmpty(self):
        # Check if count is 0
        return self.count == 0

    def isFull(self):
        # Check if the queue has reached its capacity
        return self.count == self.capacity
```

Python

```
class MyCircularQueue:

    def __init__(self, k: int):
        self.q = [0] * k
        self.size = 0
        self.capacity = k
        self.front = 0

    def enQueue(self, value: int) -> bool:
        if self.isFull():
            return False
        self.q[(self.front + self.size) % self.capacity] = value
        self.size += 1
        return True

    def deQueue(self) -> bool:
        if self.isEmpty():
            return False
        self.front = (self.front + 1) % self.capacity
        self.size -= 1
        return True

    def Front(self) -> int:
        return -1 if self.isEmpty() else self.q[self.front]
```



```
    def Rear(self) -> int:
        if self.isEmpty():
            return -1
        return self.q[(self.front + self.size - 1) % self.capacity]

    def isEmpty(self) -> bool:
        return self.size == 0

    def isFull(self) -> bool:
        return self.size == self.capacity


# Your MyCircularQueue object will be instantiated and called as such:
# obj = MyCircularQueue(k)
# param_1 = obj.enQueue(value)
# param_2 = obj.deQueue()
# param_3 = obj.Front()
# param_4 = obj.Rear()
# param_5 = obj.isEmpty()
# param_6 = obj.isFull()
```

Python

```
class MyCircularQueue:
    def __init__(self, k: int):
        self.q = [0] * k
        self.front = 0
        self.size = 0
        self.capacity = k

    def enQueue(self, value: int) -> bool:
        if self.isFull():
            return False
        idx = (self.front + self.size) % self.capacity
        self.q[idx] = value
        self.size += 1
        return True

    def deQueue(self) -> bool:
        if self.isEmpty():
            return False
        self.front = (self.front + 1) % self.capacity
        self.size -= 1
        return True

    def Front(self) -> int:
        return -1 if self.isEmpty() else self.q[self.front]
```



```
    def Rear(self) -> int:
        if self.isEmpty():
            return -1
        idx = (self.front + self.size - 1) % self.capacity
        return self.q[idx]

    def isEmpty(self) -> bool:
        return self.size == 0

    def isFull(self) -> bool:
        return self.size == self.capacity


# Your MyCircularQueue object will be instantiated and called as such:
# obj = MyCircularQueue(k)
# param_1 = obj.enQueue(value)
# param_2 = obj.deQueue()
# param_3 = obj.Front()
# param_4 = obj.Rear()
# param_5 = obj.isEmpty()
# param_6 = obj.isFull()

###########

class MyCircularQueue(object):


    def __init__(self, k):
        """
        Initialize your data structure here. Set the size of the queue to be k.
        :type k: int
        """
        self.queue = []
        self.size = k
        self.front = 0
        self.rear = 0

    def enQueue(self, value):
        """
        Insert an element into the circular queue. Return true if the operation is successful.
        :type value: int
        :rtype: bool
        """
        if self.rear - self.front < self.size:
            self.queue.append(value)
            self.rear += 1
            return True
        else:
            return False

    def deQueue(self):


        """
        Delete an element from the circular queue. Return true if the operation is successful.
        :rtype: bool
        """
        if self.rear - self.front > 0:
```


[*] Linked List — Pattern Solution (stored optimal)

Python

```
# Python implementation of MyCircularQueue

class MyCircularQueue:
    def __init__(self, k):
        # Initialize the queue with capacity k
        self.capacity = k
        self.queue = [0] * k # Fixed-size array
        self.front = 0 # pointer to the front element
        self.rear = 0 # pointer to the next insertion position
        self.count = 0 # current number of elements in the queue

    def enQueue(self, value):
        # Check if the queue is full
        if self.isFull():
            return False
        # Insert element at rear pointer and update pointer using modulo for circularity
        self.queue[self.rear] = value
        self.rear = (self.rear + 1) % self.capacity
        self.count += 1
        return True

    def deQueue(self):
        # Check if the queue is empty
        if self.isEmpty():
            return False


        # Move front pointer to the next element using modulo for circularity
        self.front = (self.front + 1) % self.capacity
        self.count -= 1
        return True

    def Front(self):
        # Return -1 if the queue is empty, else return the front element
        if self.isEmpty():
            return -1
        return self.queue[self.front]

    def Rear(self):
        # Return -1 if the queue is empty
        if self.isEmpty():
            return -1
        # Since rear points to the next insertion position, get the last element index
        return self.queue[(self.rear - 1 + self.capacity) % self.capacity]

    def isEmpty(self):
        # Check if count is 0
        return self.count == 0

    def isFull(self):
        # Check if the queue has reached its capacity
        return self.count == self.capacity
```

#707 MEDIUM

Design Linked List

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Linked List · Design

Lists: Denny Zhang

Problem:

Design your implementation of the linked list. You can choose to use a singly or doubly linked list. A node in a singly linked list should have two attributes: val and next. val is the value of the current node, and next is a pointer/reference to the next node. If you want to use the doubly linked list, you will need one more attribute prev to indicate the previous node in the linked list. Assume all nodes in the linked list are 0-indexed.

Implement the MyLinkedList class:

- MyLinkedList() Initializes the MyLinkedList object.
- int get(int index) Get the value of the indexth node in the linked list. If the index is invalid, return -1.
- void addAtHead(int val) Add a node of value val before the first element of the linked list. After the insertion, the new node will be the first node of the linked list.
- void addAtTail(int val) Append a node of value val as the last element of the linked list.
- void addAtIndex(int index, int val) Add a node of value val before the indexth node in the linked list. If index equals the length of the linked list, the node will be appended to the end of the linked list. If index is greater than the length, the node will not be inserted.
- void deleteAtIndex(int index) Delete the indexth node in the linked list, if the index is valid.

Example 1:

```
Input
["MyLinkedList", "addAtHead", "addAtTail", "addAtIndex", "get", "deleteAtIndex", "get"]
[[], [1], [3], [1, 2], [1], [1], [1]]
Output
[null, null, null, null, 2, null, 3]

Explanation
MyLinkedList myLinkedList = new MyLinkedList();
myLinkedList.addAtHead(1);
myLinkedList.addAtTail(3);
myLinkedList.addAtIndex(1, 2); // linked list becomes 1->2->3
myLinkedList.get(1); // return 2
myLinkedList.deleteAtIndex(1); // now the linked list is 1->3
myLinkedList.get(1); // return 3
```

Constraints:

- 0 <= index, val <= 1000
- Please do not use the built-in LinkedList library.
- At most 2000 calls will be made to get, addAtHead, addAtTail, addAtIndex and deleteAtIndex.

Python

Python


```
# Define a node for the singly linked list
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val # The value of the node
        self.next = next # Pointer to the next node

class MyLinkedList:
    def __init__(self):
        # Initialize size and a dummy head node to simplify operations
        self.size = 0
        self.head = ListNode(0) # dummy head

    def get(self, index: int) -> int:
        # If index is out of bounds, return -1
        if index < 0 or index >= self.size:
            return -1
        current = self.head.next # start from the actual head
        for _ in range(index):
            current = current.next
        return current.val

    def addAtHead(self, val: int) -> None:
        self.addAtIndex(0, val) # add at index 0

    def addAtTail(self, val: int) -> None:


        self.addAtIndex(self.size, val) # add at the end

    def addAtIndex(self, index: int, val: int) -> None:
        # Normalize negative index to 0 (insert at head)
        if index < 0:
            index = 0
        # Only insert if index is valid (<= size)
        if index > self.size:
            return
        self.size += 1
        # Start from the dummy head node
        pred = self.head
        for _ in range(index):
            pred = pred.next
        # Insert new node
        new_node = ListNode(val)
        new_node.next = pred.next
        pred.next = new_node

    def deleteAtIndex(self, index: int) -> None:
        # If index is out of bounds, do nothing
        if index < 0 or index >= self.size:
            return
        self.size -= 1
        pred = self.head


        for _ in range(index):
            pred = pred.next
        # Remove the node by skipping it
        pred.next = pred.next.next
```

Python

```python
from dataclasses import dataclass


@dataclass
class ListNode:
  val: int
  next: ListNode | None = None


class MyLinkedList:
  def __init__(self):
    self.length = 0
    self.dummy = ListNode(0)

  def get(self, index: int) -> int:
    if index < 0 or index >= self.length:
      return -1
    curr = self.dummy.next
    for _ in range(index):
      curr = curr.next
    return curr.val

  def addAtHead(self, val: int) -> None:
    curr = self.dummy.next
    self.dummy.next = ListNode(val)

    self.dummy.next.next = curr
    self.length += 1

  def addAtTail(self, val: int) -> None:
    curr = self.dummy
    while curr.next:
      curr = curr.next
    curr.next = ListNode(val)
    self.length += 1

  def addAtIndex(self, index: int, val: int) -> None:
    if index > self.length:
      return
    curr = self.dummy
    for _ in range(index):
      curr = curr.next
    temp = curr.next
    curr.next = ListNode(val)
    curr.next.next = temp
    self.length += 1

  def deleteAtIndex(self, index: int) -> None:
    if index < 0 or index >= self.length:
      return
    curr = self.dummy

    for _ in range(index):
      curr = curr.next
    temp = curr.next
    curr.next = temp.next
    self.length -= 1
```

Python


```python
class MyLinkedList:
    def __init__(self):
        self.dummy = ListNode()
        self.cnt = 0

    def get(self, index: int) -> int:
        if index < 0 or index >= self.cnt:
            return -1
        cur = self.dummy.next
        for _ in range(index):
            cur = cur.next
        return cur.val

    def addAtHead(self, val: int) -> None:
        self.addAtIndex(0, val)

    def addAtTail(self, val: int) -> None:
        self.addAtIndex(self.cnt, val)

    def addAtIndex(self, index: int, val: int) -> None:
        if index > self.cnt:
            return
        pre = self.dummy
        for _ in range(index):
            pre = pre.next

        pre.next = ListNode(val, pre.next)
        self.cnt += 1

    def deleteAtIndex(self, index: int) -> None:
        if index >= self.cnt:
            return
        pre = self.dummy
        for _ in range(index):
            pre = pre.next
        t = pre.next
        pre.next = t.next
        t.next = None
        self.cnt -= 1


# Your MyLinkedList object will be instantiated and called as such:
# obj = MyLinkedList()
# param_1 = obj.get(index)
# obj.addAtHead(val)
# obj.addATTail(val)
# obj.addAtIndex(index,val)
# obj.deleteAtIndex(index)
```

Python


```python
# doubly linked node
class ListNode:
    def __init__(self, value=0, prev=None, next=None):
        self.value = value
        self.prev = prev
        self.next = next

class MyLinkedList:
    def __init__(self):
        self.head = ListNode(0) # Sentinel node as pseudo-head
        self.tail = ListNode(0) # Sentinel node as pseudo-tail
        self.head.next = self.tail
        self.tail.prev = self.head
        self.size = 0

    def get(self, index: int) -> int:
        if index < 0 or index >= self.size:
            return -1
        if index + 1 < self.size - index: # decide start from head or tail
            curr = self.head
            for _ in range(index + 1):
                curr = curr.next
        else:
            curr = self.tail
            for _ in range(self.size - index):

                curr = curr.prev
        return curr.value

    def addAtHead(self, val: int) -> None:
        pred, succ = self.head, self.head.next
        self.size += 1
        to_add = ListNode(val, pred, succ)
        pred.next = to_add
        succ.prev = to_add

    def addAtTail(self, val: int) -> None:
        succ, pred = self.tail, self.tail.prev
        self.size += 1
        to_add = ListNode(val, pred, succ)
        pred.next = to_add
        succ.prev = to_add

    def addAtIndex(self, index: int, val: int) -> None:
        if index > self.size:
            return
        if index < 0:
            index = 0
        if index < self.size - index: # decide start from head or tail
            pred = self.head
            for _ in range(index):
```



```python
                pred = pred.next
            succ = pred.next
        else:
            succ = self.tail
            for _ in range(self.size - index):
                succ = succ.prev
            pred = succ.prev
        self.size += 1
        to_add = ListNode(val, pred, succ)
        pred.next = to_add
        succ.prev = to_add

    def deleteAtIndex(self, index: int) -> None:
        if index < 0 or index >= self.size:
            return
        if index < self.size - index:
            pred = self.head
            for _ in range(index):
                pred = pred.next
            succ = pred.next.next
        else:
            succ = self.tail
            for _ in range(self.size - index - 1):
                succ = succ.prev
            pred = succ.prev.prev

        self.size -= 1
        pred.next = succ
        succ.prev = pred

# Your MyLinkedList object will be instantiated and called as such:
```

[*] Linked List — Pattern Solution (matched from community answers)

Python


```python
# doubly linked node
class ListNode:
    def __init__(self, value=0, prev=None, next=None):
        self.value = value
        self.prev = prev
        self.next = next

class MyLinkedList:
    def __init__(self):
        self.head = ListNode(0) # Sentinel node as pseudo-head
        self.tail = ListNode(0) # Sentinel node as pseudo-tail
        self.head.next = self.tail
        self.tail.prev = self.head
        self.size = 0

    def get(self, index: int) -> int:
        if index < 0 or index >= self.size:
            return -1
        if index + 1 < self.size - index: # decide start from head or tail
            curr = self.head
            for _ in range(index + 1):
                curr = curr.next
        else:
            curr = self.tail
            for _ in range(self.size - index):

                curr = curr.prev
        return curr.value

    def addAtHead(self, val: int) -> None:
        pred, succ = self.head, self.head.next
        self.size += 1
        to_add = ListNode(val, pred, succ)
        pred.next = to_add
        succ.prev = to_add

    def addAtTail(self, val: int) -> None:
        succ, pred = self.tail, self.tail.prev
        self.size += 1
        to_add = ListNode(val, pred, succ)
        pred.next = to_add
        succ.prev = to_add

    def addAtIndex(self, index: int, val: int) -> None:
        if index > self.size:
            return
        if index < 0:
            index = 0
        if index < self.size - index: # decide start from head or tail
            pred = self.head
            for _ in range(index):
```



```python
                pred = pred.next
            succ = pred.next
        else:
            succ = self.tail
            for _ in range(self.size - index):
                succ = succ.prev
            pred = succ.prev
        self.size += 1
        to_add = ListNode(val, pred, succ)
        pred.next = to_add
        succ.prev = to_add

    def deleteAtIndex(self, index: int) -> None:
        if index < 0 or index >= self.size:
            return
        if index < self.size - index:
            pred = self.head
            for _ in range(index):
                pred = pred.next
            succ = pred.next.next
        else:
            succ = self.tail
            for _ in range(self.size - index - 1):
                succ = succ.prev
            pred = succ.prev.prev

        self.size -= 1
        pred.next = succ
        succ.prev = pred

# Your MyLinkedList object will be instantiated and called as such:
```

#708 MEDIUM

Insert into a Sorted Circular Linked List

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Linked List

Lists: Premium 98

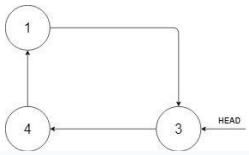
Problem:

Given a Circular Linked List node, which is sorted in non-descending order, write a function to insert a value insertVal into the list such that it remains a sorted circular list. The given node can be a reference to any single node in the list and may not necessarily be the smallest value in the circular list.

If there are multiple suitable places for insertion, you may choose any place to insert the new value. After the insertion, the circular list should remain sorted.

If the list is empty (i.e., the given node is null), you should create a new single circular list and return the reference to that single node. Otherwise, you should return the originally given node.

Example 1:




```
Input: head = [3,4,1], insertVal = 2
Output: [3,4,1,2]
Explanation: In the figure above, there is a sorted circular list of three elements. You are
given a reference to the node with value 3, and we need to insert 2 into the list. The new
node should be inserted between node 1 and node 3. After the insertion, the list should look
like this, and we should still return node 3.
```

Example 2:

```
Input: head = [], insertVal = 1
Output: [1]
Explanation: The list is empty (given head is null). We create a new single circular list and
return the reference to that single node.
```

Example 3:

```
Input: head = [1], insertVal = 0
Output: [1,0]
```

Constraints:

- The number of nodes in the list is in the range [0, 5 * 104].
- -106 <= Node.val, insertVal <= 106

Python

Python


```python
# Definition for a Node.
class Node:
    def __init__(self, val, next=None):
        self.val = val
        self.next = next

class Solution:
    def insert(self, head: 'Node', insertVal: int) -> 'Node':
        # If the list is empty, create a new node pointing to itself and return it.
        if not head:
            newNode = Node(insertVal)
            newNode.next = newNode
            return newNode

        curr = head
        inserted = False

        while True:
            # Case 1: insert between two nodes where insertVal is between curr.val and curr.next.val.
            if curr.val <= insertVal <= curr.next.val:
                inserted = True
            # Case 2: curr.val > curr.next.val, i.e., we are at the turning point.
            # In this case, insert if insertVal is greater than curr.val (largest)
            # or insertVal is less than curr.next.val (smallest).
            elif curr.val > curr.next.val:

                if insertVal >= curr.val or insertVal <= curr.next.val:
                    inserted = True

            if inserted:
                newNode = Node(insertVal, curr.next)
                curr.next = newNode
                return head

            curr = curr.next
            # If we have traversed the entire circle and came back to head, break.
            if curr == head:
                break

        # Case 3: All values are the same or no valid index was found, insert arbitrarily.
        newNode = Node(insertVal, curr.next)
        curr.next = newNode
        return head
```

Python


```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val=None, next=None):
        self.val = val
        self.next = next
"""


class Solution:
    def insert(self, head: 'Optional[Node]', insertVal: int) -> 'Node':
        node = Node(insertVal)
        if head is None:
            node.next = node
            return node
        prev, curr = head, head.next
        while curr != head:
            if prev.val <= insertVal <= curr.val or (
                prev.val > curr.val and (insertVal >= prev.val or insertVal <= curr.val)
            ):
                break
            prev, curr = curr, curr.next
        prev.next = node
        node.next = curr
        return head
```

Python

```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val=None, next=None):
        self.val = val
        self.next = next
"""


class Solution:
    def insert(self, head: 'Optional[Node]', insertVal: int) -> 'Node':
        node = Node(insertVal)
        if head is None:
            node.next = node
            return node
        prev, curr = head, head.next
        while curr != head:
            if prev.val <= insertVal <= curr.val or (
                prev.val > curr.val and (insertVal >= prev.val or insertVal <= curr.val)
            ):
                break
            prev, curr = curr, curr.next
        prev.next = node
        node.next = curr
        return head
```

[*] Linked List — Pattern Solution (matched from community answers)

Python


```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val=None, next=None):
        self.val = val
        self.next = next
"""


class Solution:
    def insert(self, head: 'Optional[Node]', insertVal: int) -> 'Node':
        node = Node(insertVal)
        if head is None:
            node.next = node
            return node
        prev, curr = head, head.next
        while curr != head:
            if prev.val <= insertVal <= curr.val or (
                prev.val > curr.val and (insertVal >= prev.val or insertVal <= curr.val)
            ):
                break
            prev, curr = curr, curr.next
        prev.next = node
        node.next = curr
        return head
```

Python

#1171 MEDIUM

Remove Zero Sum Consecutive Nodes from Linked List

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · Linked List

Lists: Denny Zhang

Problem:

Given the head of a linked list, we repeatedly delete consecutive sequences of nodes that sum to 0 until there are no such sequences.

After doing so, return the head of the final linked list. You may return any such answer.

(Note that in the examples below, all sequences are serializations of ListNode objects.)

Example 1:

```
Input: head = [1,2,-3,3,1]
Output: [3,1]
Note: The answer [1,2,1] would also be accepted.
```

Example 2:

```
Input: head = [1,2,3,-3,4]
Output: [1,2,4]
```

Example 3:

```
Input: head = [1,2,3,-3,-2]
Output: [1]
```

Constraints:

- The given linked list will contain between 1 and 1000 nodes.
- Each node in the linked list has -1000 <= node.val <= 1000.

Python

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def removeZeroSumSublists(self, head: ListNode) -> ListNode:
        # Create a dummy node to simplify cases where removal starts at the head.
        dummy = ListNode(0)
        dummy.next = head

        # Dictionary to store prefix sum to node mapping.
        sum_to_node = {}
        prefix_sum = 0
        current = dummy

        # First pass: record the last occurrence of each prefix sum.
        while current:
            prefix_sum += current.val
            sum_to_node[prefix_sum] = current
            current = current.next

        # Second pass: remove nodes that are part of zero-sum sequences.
        prefix_sum = 0

        current = dummy
        while current:
            prefix_sum += current.val
            # Link current node's next pointer to skip the zero-sum subsequence.
            current.next = sum_to_node[prefix_sum].next
            current = current.next

        return dummy.next
```

Python


```python
class Solution:
    def removeZeroSumSublists(self, head: ListNode) -> ListNode:
        dummy = ListNode(0, head)
        prefix = 0
        prefixToNode = {0: dummy}

        while head:
          prefix += head.val
          prefixToNode[prefix] = head
          head = head.next

        prefix = 0
        head = dummy

        while head:
          prefix += head.val
          head.next = prefixToNode[prefix].next
          head = head.next

        return dummy.next
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def removeZeroSumSublists(self, head: Optional[ListNode]) -> Optional[ListNode]:
        dummy = ListNode(next=head)
        last = {}
        s, cur = 0, dummy
        while cur:
            s += cur.val
            last[s] = cur
            cur = cur.next
        s, cur = 0, dummy
        while cur:
            s += cur.val
            cur.next = last[s].next
            cur = cur.next
        return dummy.next
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def removeZeroSumSublists(self, head: Optional[ListNode]) -> Optional[ListNode]:
        dummy = ListNode(next=head)
        last = {}
        s, cur = 0, dummy
        while cur:
            s += cur.val
            last[s] = cur
            cur = cur.next
        s, cur = 0, dummy
        while cur:
            s += cur.val
            cur.next = last[s].next
            cur = cur.next
        return dummy.next
```

[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def removeZeroSumSublists(self, head: ListNode) -> ListNode:
        # Create a dummy node to simplify cases where removal starts at the head.
        dummy = ListNode(0)
        dummy.next = head

        # Dictionary to store prefix sum to node mapping.
        sum_to_node = {}
        prefix_sum = 0
        current = dummy

        # First pass: record the last occurrence of each prefix sum.
        while current:
            prefix_sum += current.val
            sum_to_node[prefix_sum] = current
            current = current.next

        # Second pass: remove nodes that are part of zero-sum sequences.
        prefix_sum = 0

        current = dummy
        while current:
            prefix_sum += current.val
            # Link current node's next pointer to skip the zero-sum subsequence.
            current.next = sum_to_node[prefix_sum].next
            current = current.next

        return dummy.next
```


#25 HARD

Reverse Nodes in k-Group

LeetDoocs · SimplyLeet · WallCC · LeetCode · Editorial

Linked List · Recursion

Lists: Grind 169

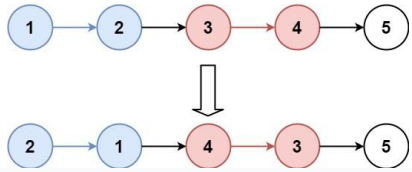
Problem:

Given the head of a linked list, reverse the nodes of the list k at a time, and return the modified list.

k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.

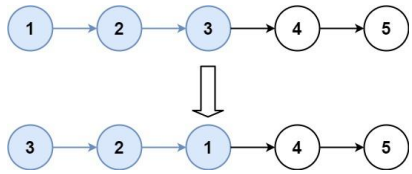
You may not alter the values in the list's nodes, only nodes themselves may be changed.

Example 1:




```
Input: head = [1,2,3,4,5], k = 2
Output: [2,1,4,3,5]
```

Example 2:




```
Input: head = [1,2,3,4,5], k = 3
Output: [3,2,1,4,5]
```

Constraints:

- The number of nodes in the list is n.
- 1 <= k <= n <= 5000
- 0 <= Node.val <= 1000

Follow-up: Can you solve the problem in O(1) extra memory space?

>> Brute Force [Linked List] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def getKthNode(self, curr, k):
        # Brute force - collect into list, manipulate, rebuild
        vals = []
        curr = curr
        while curr:
            vals.append(curr.val)
            curr = curr.next
        dummy = ListNode(0)
        curr = dummy
        for v in vals:
            curr.next = ListNode(v)
            curr = curr.next
        return dummy.next
```

Python

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        # Create a dummy node to simplify edge cases
        dummy = ListNode(0)
        dummy.next = head
        group_prev = dummy

        while True:
            # Find the kth node from group_prev
            kth = self.getKthNode(group_prev, k)
            if not kth:
                break
            group_next = kth.next

            # Reverse the group starting at group_prev.next up to kth
            prev = kth.next
            curr = group_prev.next
            while curr != group_next:
                temp = curr.next # Store next node
```



```
            curr.next = prev # Reverse pointer
            prev = curr # Move prev forward
            curr = temp # Move curr forward

        # Reconnect the reversed group with the previous part
        temp = group_prev.next # temp is the start of the original group, becomes the new tail
        group_prev.next = kth # Connect previous part to the kth node (new head of reversed group)
        group_prev = temp # Move group_prev to the end of the reversed group

    return dummy.next

# Helper function to get the kth node from the current node
def getKthNode(self, curr, k):
    while curr and k > 0:
        curr = curr.next
        k -= 1
    return curr
```

Python

```python
class Solution:
  def reverseKGroup(self, head: ListNode | None, k: int) -> ListNode | None:
    if not head:
      return None

    tail = head

    for _ in range(k):
      # There are less than k nodes in the list, do nothing.
      if not tail:
        return head
      tail = tail.next

    newHead = self._reverse(head, tail)
    head.next = self.reverseKGroup(tail, k)
    return newHead

  def _reverse(
      self,
      head: ListNode | None,
      tail: ListNode | None,
  ) -> ListNode | None:
    """Reverses [head, tail)."""
    prev = None
    curr = head

    while curr != tail:
      next = curr.next
      curr.next = prev
      prev = curr
      curr = next
    return prev
```

Python


```python
class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        if not head or k == 1:
            return head

        def getLength(head: ListNode) -> int:
            length = 0
            while head:
                length += 1
                head = head.next
            return length

        length = getLength(head)
        dummy = ListNode(0, head)
        prev = dummy
        curr = head

        for _ in range(length // k):
            for _ in range(k - 1):
                next = curr.next
                curr.next = next.next
                next.next = prev.next
                prev.next = next
            prev = curr
            curr = curr.next


        return dummy.next
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def reverseKGroup(self, head: Optional[ListNode], k: int) -> Optional[ListNode]:
        def reverse(head: Optional[ListNode]) -> Optional[ListNode]:
            dummy = ListNode()
            cur = head
            while cur:
                nxt = cur.next
                cur.next = dummy.next
                dummy.next = cur
                cur = nxt
            return dummy.next

        dummy = pre = ListNode(next=head)
        while pre:
            cur = pre
            for _ in range(k):
                cur = cur.next
                if cur is None:
                    return dummy.next
            node = pre.next
```



```
            nxt = cur.next
            cur.next = None
            pre.next = reverse(node)
            node.next = nxt
            pre = node
        return dummy.next
```

Python

```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        '''
            for reverse 1->2->3->4->5, process is like
                1->None, 2->3->4->5
                2->1->None, 3->4->5
                3->2->1->None, 4->5
                4->3->2->1->None, 5
                5->4->3->2->1->None, None
        '''
        def reverseList(head):
            pre, p = None, head
            while p:
                pnext = p.next
                p.next = pre
                pre = p
                p = pnext
            return pre

        dummy = ListNode(next=head)
```

```
        pre = cur = dummy
        while cur.next:
            for _ in range(k):
                cur = cur.next
                if cur is None:
                    return dummy.next
            t = cur.next
            cur.next = None # cut from next k-group, so to reverseList() for current k-group
            start = pre.next
            pre.next = reverseList(start)
            start.next = t # so now 'start' is the last node of k-group
            pre = cur = start # same as the reset dummy before while loop
        return dummy.next

###########

# Definition for singly-linked list.
# class ListNode(object):
# def __init__(self, x):
# self.val = x
# self.next = None

class Solution(object):
    def reverseKGroup(self, head, k):
        """

        :type head: ListNode
        :type k: int
        :rtype: ListNode
        """

        def reverseList(head, k):
          pre = None
          cur = head
          while cur and k > 0:
            tmp = cur.next
            cur.next = pre
            pre = cur
            cur = tmp
            k -= 1
          head.next = cur
          return cur, pre

        length = 0
        p = head
        while p:
          length += 1
          p = p.next
        if length < k:
          return head
        step = length / k

        ret = None
        pre = None
        p = head
        while p and step:
          next, newHead = reverseList(p, k)
```


[*] Linked List — Pattern Solution (matched from community answers)

Python

```
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def reverseKGroup(self, head: ListNode, k: int) -> ListNode:
        # Create a dummy node to simplify edge cases
        dummy = ListNode(0)
        dummy.next = head
        group_prev = dummy

        while True:
            # Find the kth node from group_prev
            kth = self.getKthNode(group_prev, k)
            if not kth:
                break
            group_next = kth.next

            # Reverse the group starting at group_prev.next up to kth
            prev = kth.next
            curr = group_prev.next
            while curr != group_next:
                temp = curr.next # Store next node

                curr.next = prev # Reverse pointer
                prev = curr # Move prev forward
                curr = temp # Move curr forward

            # Reconnect the reversed group with the previous part
            temp = group_prev.next # temp is the start of the original group, becomes the new tail
            group_prev.next = kth # Connect previous part to the kth node (new head of reversed group)
            group_prev = temp # Move group_prev to the end of the reversed group

        return dummy.next

    # Helper function to get the kth node from the current node
    def getKthNode(self, curr, k):
        while curr and k > 0:
            curr = curr.next
            k -= 1
        return curr
```

#432 HARD

All O'one Data Structure

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · Design · Doubly-Linked List · Linked List

Lists: Denny Zhang

Problem:

Design a data structure to store the strings' count with the ability to return the strings with minimum and maximum counts.

Implement the AllOne class:

- AllOne() Initializes the object of the data structure.
- inc(String key) Increments the count of the string key by 1. If key does not exist in the data structure, insert it with count 1.
- dec(String key) Decrements the count of the string key by 1. If the count of key is 0 after the decrement, remove it from the data structure. It is guaranteed that key exists in the data structure before the decrement.
- getMaxKey() Returns one of the keys with the maximal count. If no element exists, return an empty string "".
- getMinKey() Returns one of the keys with the minimum count. If no element exists, return an empty string "".

Note that each function must run in O(1) average time complexity.

Example 1:

```
Input
["AllOne", "inc", "inc", "getMaxKey", "getMinKey", "inc", "getMaxKey", "getMinKey"]
[[], ["hello"], ["hello"], [], [], ["leet"], [], []]
Output
[null, null, null, "hello", "hello", null, "hello", "leet"]

Explanation
AllOne allOne = new AllOne();
allOne.inc("hello");
allOne.inc("hello");
allOne.getMaxKey(); // return "hello"
allOne.getMinKey(); // return "hello"
allOne.inc("leet");
allOne.getMaxKey(); // return "hello"
allOne.getMinKey(); // return "leet"
```

Constraints:

- 1 <= key.length <= 10
- key consists of lowercase English letters.
- It is guaranteed that for each call to dec, key is existing in the data structure.
- At most 5 * 104 calls will be made to inc, dec, getMaxKey, and getMinKey.

Python

Python


```
# Node class to represent a frequency bucket in the doubly linked list
class Node:
    def __init__(self, count):
        self.count = count # frequency count for keys in this node
        self.keys = set() # set of keys with this frequency
        self.prev = None # previous node in the linked list
        self.next = None # next node in the linked list

class AllOne:
    def __init__(self):
        # Use sentinel nodes for easier handling of edge cases.
        self.head = Node(float('-inf'))
        self.tail = Node(float('inf'))
        self.head.next = self.tail
        self.tail.prev = self.head
        # Map each key to its corresponding node.
        self.keyToNode = {}

    # Helper: insert newNode after prevNode in the linked list.
    def _insertNodeAfter(self, prevNode, newNode):
        newNode.prev = prevNode
        newNode.next = prevNode.next
        prevNode.next.prev = newNode
        prevNode.next = newNode

    # Helper: remove a node if it has no keys.
    def _removeNode(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def inc(self, key: str) -> None:
        # If key is new, treat current node as head.
        if key not in self.keyToNode:
            curr = self.head
        else:
            curr = self.keyToNode[key]
            curr.keys.remove(key)
        # Check if the next node has count = current count + 1.
        targetCount = (0 if curr == self.head else curr.count) + 1
        if curr.next == self.tail or curr.next.count != targetCount:
            # Create new node with targetCount.
            newNode = Node(targetCount)
            self._insertNodeAfter(curr, newNode)
        else:
            newNode = curr.next
        newNode.keys.add(key)
        self.keyToNode[key] = newNode
        # Remove the current node if it is not a sentinel and has no keys.
        if curr != self.head and len(curr.keys) == 0:
            self._removeNode(curr)
```



```
    def dec(self, key: str) -> None:
        curr = self.keyToNode[key]
        curr.keys.remove(key)
        # If decrementing causes the count to be 0, remove key completely.
        if curr.count == 1:
            del self.keyToNode[key]
        else:
            targetCount = curr.count - 1
            # Check if the previous node has the target count.
            if curr.prev == self.head or curr.prev.count != targetCount:
                newNode = Node(targetCount)
                # Insert newNode before curr.
                self._insertNodeAfter(curr.prev, newNode)
            else:
                newNode = curr.prev
            newNode.keys.add(key)
            self.keyToNode[key] = newNode
        # Clean up current node if empty.
        if len(curr.keys) == 0:
            self._removeNode(curr)

    def getMaxKey(self) -> str:
        # The node before tail has the maximal count.
        return next(iter(self.tail.prev.keys)) if self.tail.prev != self.head else ""

    def getMinKey(self) -> str:
        # The node after head has the minimal count.
        return next(iter(self.head.next.keys)) if self.head.next != self.tail else ""
```

Python

```
from dataclasses import dataclass


@dataclass
class Node:
    def __init__(self, count: int, key: str | None = None):
        self.count = count
        self.keys: set[str] = {key} if key else set()
        self.prev: Node | None = None
        self.next: Node | None = None

    def __eq__(self, other) -> bool:
        if not isinstance(other, Node):
            return NotImplemented
        return self.count == other.count and self.keys == other.keys


class AllOne:
    def __init__(self):
        self.keyToNode: dict[str, Node] = {}
        self.head = Node(0)
        self.tail = Node(0)
        self.head.next = self.tail
        self.tail.prev = self.head
```



```
    def inc(self, key: str) -> None:
        if key in self.keyToNode:
            self._incrementExistingKey(key)
        else:
            self._addNewKey(key)

    def dec(self, key: str) -> None:
        # It is guaranteed that key exists in the data structure before the
        # decrement.
        self._decrementExistingKey(key)

    def getMaxKey(self) -> str:
        return '' if self.tail.prev == self.head \
            else next(iter(self.tail.prev.keys))

    def getMinKey(self) -> str:
        return '' if self.head.next == self.tail \
            else next(iter(self.head.next.keys))

    def _addNewKey(self, key: str) -> None:
        """Adds a new node with frequency 1."""
        if self.head.next.count == 1:
            self.head.next.keys.add(key)
        else:
            self._insertAfter(self.head, Node(1, key))

        self.keyToNode[key] = self.head.next

    def _incrementExistingKey(self, key: str) -> None:
        """Increments the frequency of the key by 1."""
        node = self.keyToNode[key]
        node.keys.remove(key)
        if node.next == self.tail or node.next.count > node.count + 1:
            self._insertAfter(node, Node(node.count + 1))
        node.next.keys.add(key)
        self.keyToNode[key] = node.next
        if not node.keys:
            self._remove(node)

    def _decrementExistingKey(self, key: str) -> None:
        """Decrements the count of the key by 1."""
        node = self.keyToNode[key]
        node.keys.remove(key)
        if node.count - 1:
            if node.prev == self.head or node.prev.count != node.count - 1:
                self._insertAfter(node.prev, Node(node.count - 1))
            node.prev.keys.add(key)
            self.keyToNode[key] = node.prev
        else:
            del self.keyToNode[key]
            if not node.keys:

                self._remove(node)

    def _insertAfter(self, node: Node, newNode: Node) -> None:
        newNode.prev = node
        newNode.next = node.next
```

Python

```python
class Node:
    def __init__(self, key="", cnt=0):
        self.prev = None
        self.next = None
        self.cnt = cnt
        self.keys = {key}

    def insert(self, node):
        node.prev = self
        node.next = self.next
        node.prev.next = node
        node.next.prev = node
        return node

    def remove(self):
        self.prev.next = self.next
        self.next.prev = self.prev


class AllOne:
    def __init__(self):
        self.root = Node()
        self.root.next = self.root
        self.root.prev = self.root
        self.nodes = {}


    def inc(self, key: str) -> None:
        root, nodes = self.root, self.nodes
        if key not in nodes:
            if root.next == root or root.next.cnt > 1:
                nodes[key] = root.insert(Node(key, 1))
            else:
                root.next.keys.add(key)
                nodes[key] = root.next
        else:
            curr = nodes[key]
            next = curr.next
            if next == root or next.cnt > curr.cnt + 1:
                nodes[key] = curr.insert(Node(key, curr.cnt + 1))
            else:
                next.keys.add(key)
                nodes[key] = next
            curr.keys.discard(key)
            if not curr.keys:
                curr.remove()

    def dec(self, key: str) -> None:
        root, nodes = self.root, self.nodes
        curr = nodes[key]
        if curr.cnt == 1:
```



```python
            nodes.pop(key)
        else:
            prev = curr.prev
            if prev == root or prev.cnt < curr.cnt - 1:
                nodes[key] = prev.insert(Node(key, curr.cnt - 1))
            else:
                prev.keys.add(key)
                nodes[key] = prev
        curr.keys.discard(key)
        if not curr.keys:
            curr.remove()

    def getMaxKey(self) -> str:
        return next(iter(self.root.prev.keys))

    def getMinKey(self) -> str:
        return next(iter(self.root.next.keys))


# Your AllOne object will be instantiated and called as such:
# obj = AllOne()
# obj.inc(key)
# obj.dec(key)
# param_3 = obj.getMaxKey()
# param_4 = obj.getMinKey()
```

[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
# Node class to represent a frequency bucket in the doubly linked list
class Node:
    def __init__(self, count):
        self.count = count # frequency count for keys in this node
        self.keys = set() # set of keys with this frequency
        self.prev = None # previous node in the linked list
        self.next = None # next node in the linked list

class AllOne:
    def __init__(self):
        # Use sentinel nodes for easier handling of edge cases.
        self.head = Node(float('-inf'))
        self.tail = Node(float('inf'))
        self.head.next = self.tail
        self.tail.prev = self.head
        # Map each key to its corresponding node.
        self.keyToNode = {}

    # Helper: insert newNode after prevNode in the linked list.
    def _insertNodeAfter(self, prevNode, newNode):
        newNode.prev = prevNode
        newNode.next = prevNode.next
        prevNode.next.prev = newNode
        prevNode.next = newNode
```



```python
    # Helper: remove a node if it has no keys.
    def _removeNode(self, node):
        node.prev.next = node.next
        node.next.prev = node.prev

    def inc(self, key: str) -> None:
        # If key is new, treat current node as head.
        if key not in self.keyToNode:
            curr = self.head
        else:
            curr = self.keyToNode[key]
            curr.keys.remove(key)
        # Check if the next node has count = current count + 1.
        targetCount = (0 if curr == self.head else curr.count) + 1
        if curr.next == self.tail or curr.next.count != targetCount:
            # Create new node with targetCount.
            newNode = Node(targetCount)
            self._insertNodeAfter(curr, newNode)
        else:
            newNode = curr.next
        newNode.keys.add(key)
        self.keyToNode[key] = newNode
        # Remove the current node if it is not a sentinel and has no keys.
        if curr != self.head and len(curr.keys) == 0:
            self._removeNode(curr)


    def dec(self, key: str) -> None:
        curr = self.keyToNode[key]
        curr.keys.remove(key)
        # If decrementing causes the count to be 0, remove key completely.
        if curr.count == 1:
            del self.keyToNode[key]
        else:
            targetCount = curr.count - 1
            # Check if the previous node has the target count.
            if curr.prev == self.head or curr.prev.count != targetCount:
                newNode = Node(targetCount)
                # Insert newNode before current.
                self._insertNodeAfter(curr.prev, newNode)
            else:
                newNode = curr.prev
            newNode.keys.add(key)
            self.keyToNode[key] = newNode
        # Clean up current node if empty.
        if len(curr.keys) == 0:
            self._removeNode(curr)

    def getMaxKey(self) -> str:
        # The node before tail has the maximal count.
        return next(iter(self.tail.prev.keys)) if self.tail.prev != self.head else ""


    def getMinKey(self) -> str:
        # The node after head has the minimal count.
        return next(iter(self.head.next.keys)) if self.head.next != self.tail else ""
```


#460 HARD

LFU Cache

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · Linked List · Design · Doubly-Linked List

Lists: Denny Zhang

Problem:

Design and implement a data structure for a Least Frequently Used (LFU) cache.

Implement the LFUCache class:

- LFUCache(int capacity) Initializes the object with the capacity of the data structure.
- int get(int key) Gets the value of the key if the key exists in the cache. Otherwise, returns -1.
- void put(int key, int value) Update the value of the key if present, or inserts the key if not already present. When the cache reaches its capacity, it should invalidate and remove the least frequently used key before inserting a new item. For this problem, when there is a tie (i.e., two or more keys with the same frequency), the least recently used key would be invalidated.

To determine the least frequently used key, a use counter is maintained for each key in the cache. The key with the smallest use counter is the least frequently used key.

When a key is first inserted into the cache, its use counter is set to 1 (due to the put operation). The use counter for a key in the cache is incremented either a get or put operation is called on it.

The functions get and put must each run in O(1) average time complexity.

Example 1:

```
Input
["LFUCache", "put", "put", "get", "put", "get", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, 3, null, -1, 3, 4]

Explanation
// cnt(x) = the use counter for key x
// cache=[] will show the last used order for tiebreakers (leftmost element is most recent)
LFUCache lfu = new LFUCache(2);
lfu.put(1, 1); // cache=[1,_], cnt(1)=1
lfu.put(2, 2); // cache=[2,1], cnt(2)=1, cnt(1)=1
lfu.get(1); // return 1
// cache=[1,2], cnt(2)=1, cnt(1)=2
lfu.put(3, 3); // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.
  // cache=[3,1], cnt(3)=1, cnt(1)=2
lfu.get(2); // return -1 (not found)
lfu.get(3); // return 3
// cache=[3,1], cnt(3)=2, cnt(1)=2
lfu.put(4, 4); // Both 1 and 3 have the same cnt, but 1 is LRU, invalidate 1.
// cache=[4,3], cnt(4)=1, cnt(3)=2
lfu.get(1); // return -1 (not found)
lfu.get(3); // return 3
// cache=[3,4], cnt(4)=1, cnt(3)=3
lfu.get(4); // return 4
```



```
// cache=[4,3], cnt(4)=2, cnt(3)=3
```

Constraints:

- 1 <= capacity <= 104
- 0 <= key <= 105
- 0 <= value <= 109
- At most 2 * 105 calls will be made to get and put.

Python

Python

```python
from collections import OrderedDict

class LFUCache:
    def __init__(self, capacity: int):
        # Initialize cache capacity and helper variables.
        self.capacity = capacity
        self.min_freq = 0 # Track the minimum frequency in cache.
        # Map key -> (value, frequency)
        self.key_to_val_freq = {}
        # Map frequency -> OrderedDict of keys for LRU ordering.
        self.freq_to_keys = {}

    def update_freq(self, key):
        # Increase frequency of the key since it was accessed.
        value, freq = self.key_to_val_freq[key]
        # Remove key from its current frequency list.
        self.freq_to_keys[freq].pop(key)
        # If current frequency list is empty, update min_freq if necessary.
        if not self.freq_to_keys[freq]:
            del self.freq_to_keys[freq]
            if self.min_freq == freq:
                self.min_freq += 1
        new_freq = freq + 1
        # Insert key into the list corresponding to the new frequency.
        if new_freq not in self.freq_to_keys:
```



```python
            self.freq_to_keys[new_freq] = OrderedDict()
        self.freq_to_keys[new_freq][key] = None
        # Update the frequency entry for the key.
        self.key_to_val_freq[key] = (value, new_freq)

    def get(self, key: int) -> int:
        # Return -1 if the key is not present.
        if key not in self.key_to_val_freq:
            return -1
        # Update frequency due to access.
        self.update_freq(key)
        return self.key_to_val_freq[key][0]

    def put(self, key: int, value: int) -> None:
        # If capacity is zero, nothing can be added.
        if self.capacity == 0:
            return
        if key in self.key_to_val_freq:
            # Update value and frequency if key exists.
            self.key_to_val_freq[key] = (value, self.key_to_val_freq[key][1])
            self.update_freq(key)
        else:
            # If cache is full, evict the least frequently used (and least recently used) key.
            if len(self.key_to_val_freq) >= self.capacity:
                # Remove the first key from the lowest frequency list.

                key_to_remove, _ = self.freq_to_keys[self.min_freq].popitem(last=False)
                del self.key_to_val_freq[key_to_remove]
                if not self.freq_to_keys[self.min_freq]:
                    del self.freq_to_keys[self.min_freq]
            # Insert the new key with frequency 1.
            self.key_to_val_freq[key] = (value, 1)
            if 1 not in self.freq_to_keys:
                self.freq_to_keys[1] = OrderedDict()
            self.freq_to_keys[1][key] = None
            self.min_freq = 1 # Reset min_freq.
```

Python

```python
class Node:
    def __init__(self, key: int, value: int, freq: int, it):
        self.key = key
        self.value = value
        self.freq = freq
        self.it = it


class LFUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.minFreq = 0
        self.keyToNode = {}
        self.freqToList = {}

    def get(self, key: int) -> int:
        if key not in self.keyToNode:
            return -1

        node = self.keyToNode[key]
        self._touch(node)
        return node.value

    def put(self, key: int, value: int) -> None:
        if self.capacity == 0:
```



```python
            return

        if key in self.keyToNode:
            node = self.keyToNode[key]
            node.value = value
            self._touch(node)
            return

        if len(self.keyToNode) == self.capacity:
            # Evict the LRU key from minFreq list
            keyToEvict = self.freqToList[self.minFreq][-1]
            self.freqToList[self.minFreq].pop()
            del self.keyToNode[keyToEvict]

        self.minFreq = 1
        if 1 not in self.freqToList:
            self.freqToList[1] = []
        self.freqToList[1].insert(0, key)
        self.keyToNode[key] = Node(key, value, 1, 0) # Using index 0 as iterator

    def _touch(self, node: Node) -> None:
        # Update the node's frequency
        prevFreq = node.freq
        node.freq += 1
        newFreq = node.freq

        # Remove the key from `prevFreq`'s list
        self.freqToList[prevFreq].remove(node.key)
        if not self.freqToList[prevFreq]:
            del self.freqToList[prevFreq]
            # Update `minFreq` if needed
            if prevFreq == self.minFreq:
                self.minFreq += 1

        # Insert the key to the front of `newFreq`'s list
        if newFreq not in self.freqToList:
            self.freqToList[newFreq] = []
        self.freqToList[newFreq].insert(0, node.key)
        node.it = 0 # Using index 0 as iterator
```

Python

```python
from collections import defaultdict

class LFUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.key_to_value = {}
        self.key_to_freq = defaultdict(int)
        self.freq_to_keys = defaultdict(OrderedDict)
        self.min_freq = 0

    def get(self, key: int) -> int:
        if key not in self.key_to_value:
            return -1

        # Update the frequency
        self._update_frequency(key)
        return self.key_to_value[key]

    def put(self, key: int, value: int) -> None:
        if self.capacity == 0:
            return

        if key in self.key_to_value:
            # Update the value and frequency
            self.key_to_value[key] = value
```



```python
            self._update_frequency(key)
        else:
            if len(self.key_to_value) >= self.capacity:
                # Evict the least frequently used key
                self._evict()

            # Add the new key-value pair
            self.key_to_value[key] = value
            self.key_to_freq[key] = 1
            self.freq_to_keys[1][key] = None
            self.min_freq = 1

    def _update_frequency(self, key: int) -> None:
        freq = self.key_to_freq[key]
        del self.freq_to_keys[freq][key]

        if not self.freq_to_keys[freq]:
            # If there are no keys with the previous frequency, update min_freq
            if self.min_freq == freq:
                self.min_freq += 1
            del self.freq_to_keys[freq]

        freq += 1
        self.key_to_freq[key] = freq
        self.freq_to_keys[freq][key] = None


    def _evict(self) -> None:
        keys = self.freq_to_keys[self.min_freq]
        evict_key, _ = keys.popitem(last=False)
        del self.key_to_value[evict_key]
        del self.key_to_freq[evict_key]

# Your LFUCache object will be instantiated and called as such:
# obj = LFUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)


#########

class Node:
    def __init__(self, key: int, value: int) -> None:
        self.key = key
        self.value = value
        self.freq = 1
        self.prev = None
        self.next = None


class DoublyLinkedList:

    def __init__(self) -> None:
        self.head = Node(-1, -1)
        self.tail = Node(-1, -1)
        self.head.next = self.tail
        self.tail.prev = self.head
```


[*] Linked List — Pattern Solution (matched from community answers)

Python

```python
from collections import defaultdict

class LFUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.key_to_value = {}
        self.key_to_freq = defaultdict(int)
        self.freq_to_keys = defaultdict(OrderedDict)
        self.min_freq = 0

    def get(self, key: int) -> int:
        if key not in self.key_to_value:
            return -1

        # Update the frequency
        self._update_frequency(key)
        return self.key_to_value[key]

    def put(self, key: int, value: int) -> None:
        if self.capacity == 0:
            return

        if key in self.key_to_value:
            # Update the value and frequency
            self.key_to_value[key] = value
```



```python
            self._update_frequency(key)
        else:
            if len(self.key_to_value) >= self.capacity:
                # Evict the least frequently used key
                self._evict()

            # Add the new key-value pair
            self.key_to_value[key] = value
            self.key_to_freq[key] = 1
            self.freq_to_keys[1][key] = None
            self.min_freq = 1

    def _update_frequency(self, key: int) -> None:
        freq = self.key_to_freq[key]
        del self.freq_to_keys[freq][key]

        if not self.freq_to_keys[freq]:
            # If there are no keys with the previous frequency, update min_freq
            if self.min_freq == freq:
                self.min_freq += 1
            del self.freq_to_keys[freq]

        freq += 1
        self.key_to_freq[key] = freq
        self.freq_to_keys[freq][key] = None
```



```python
    def _evict(self) -> None:
        keys = self.freq_to_keys[self.min_freq]
        evict_key, _ = keys.popitem(last=False)
        del self.key_to_value[evict_key]
        del self.key_to_freq[evict_key]

# Your LFUCache object will be instantiated and called as such:
# obj = LFUCache(capacity)
# param_1 = obj.get(key)
# obj.put(key,value)


#########

class Node:
    def __init__(self, key: int, value: int) -> None:
        self.key = key
        self.value = value
        self.freq = 1
        self.prev = None
        self.next = None


class DoublyLinkedList:

    def __init__(self) -> None:
        self.head = Node(-1, -1)
        self.tail = Node(-1, -1)
        self.head.next = self.tail
        self.tail.prev = self.head
```


Quick Review — Linked List 23 questions · insights · complexity · solution

#21 Merge Two Sorted Lists [Easy]

Key Insights

- Leverage the fact that both lists are sorted.
- Use two pointers (one for each list) to compare nodes sequentially.
- Create a dummy node to simplify edge cases and build the resulting list.
- Iterate until one list is exhausted, then append the remaining nodes from the other list.
- There is also a recursive approach that can simplify the logic but may use more stack space.

Space & Time

Time Complexity: O(n + m), where n and m are the lengths of the two lists.

Space Complexity: O(1) for the iterative solution (O(n + m) stack space for the recursive solution).

Solution

We use an iterative approach with a dummy node to merge two sorted linked lists. Two pointers traverse list1 and list2; at each step, the node with the smaller value is appended to our merged list. This process repeats until one of the lists is exhausted, after which we append the remaining nodes from the other list. This approach works in one pass over both lists, ensuring linear time complexity and constant extra space usage.

#141 Linked List Cycle [Easy]

Key Insights

- Use two pointers (slow and fast) to traverse the list.
- The fast pointer moves two steps at a time and the slow pointer moves one step.
- If the fast pointer ever meets the slow pointer, a cycle exists.
- If the fast pointer reaches the end of the list (null), the list has no cycle.

Space & Time

Time Complexity: O(n) where n is the number of nodes in the linked list.

Space Complexity: O(1) as only two pointers are used.

Solution

We use the Floyd's Cycle Detection algorithm (Tortoise and Hare approach). Two pointers, slow and fast, are initialized at the head of the list. The slow pointer moves one step at a time while the fast pointer moves two steps. If the linked list has a cycle, the fast pointer will eventually meet the slow pointer. If the fast pointer reaches the end (i.e., a null pointer), the linked list does not have a cycle.

#206 Reverse Linked List [Easy]

Key Insights

- The problem can be solved using either an iterative or recursive approach.
- Iterative method uses constant space by updating pointers in-place.
- Recursive method leverages the call stack to reverse the list, which could use O(n) space in the worst case.
- Both approaches traverse each node exactly once, yielding linear time complexity.

Space & Time

Time Complexity: O(n) for both iterative and recursive approaches, where n is the number of nodes.

Space Complexity: O(1) for the iterative approach and O(n) for the recursive approach because of the recursion call stack.

Solution

We can solve the problem by iterating through the list while reversing the next pointers of each node. We maintain two pointers (previous and current) and update them as we traverse the list, so that by the end, the previous pointer is at the new head of the reversed list.

Alternatively, a recursive approach reverses the rest of the list first and then adjusts the pointers so that the original head becomes the tail.

#234 Palindrome Linked List [Easy]

Key Insights

- Use two pointers (slow and fast) to reach the middle of the list.
- Reverse the second half of the list.
- Compare the nodes from the start of the list to the reversed half.
- This approach meets the O(n) time and O(1) space follow-up requirement.

Space & Time

Time Complexity: O(n) - each node is visited a constant number of times.

Space Complexity: O(1) - only constant extra space is used (in-place reversal).

Solution

We first use the two-pointer technique to locate the midpoint of the linked list. The slow pointer moves one step at a time while the fast pointer moves two steps. Once the middle is found, we reverse the second half of the list in-place. Finally, we traverse both halves concurrently to compare the node values. If all corresponding values match, the linked list is a palindrome. A potential gotcha is handling the odd number of nodes; in that case, we ignore the middle element when comparing.

#706 Design HashMap [Easy]

Key Insights

- Use a fixed-size array of buckets where each bucket uses a linked list to handle collisions.
- Choose a simple hash function (e.g., modulo operation) to map a key to its corresponding bucket.
- For each bucket, traverse the linked list to update an existing key or append a new node if the key does not exist.
- Removal requires updating pointers in the linked list to bypass the removed node.

Space & Time

Time Complexity: Average O(1) for put, get, and remove; worst-case O(n) when many keys hash to the same bucket.

Space Complexity: O(n), where n is the number of key-value pairs stored.

Solution

We implement the HashMap using an array of fixed size (e.g., 1000). Each array element is the head of a linked list that manages collisions using chaining. The key is hashed using a modulo operation. For the put operation, if a key already exists in the chain, its value is updated; otherwise, a new node is appended. The get operation traverses the chain at the computed bucket index to find and return the corresponding value, or returns -1 if not found. The remove operation also traverses the list, carefully reconnecting the links to remove the target node while handling edge cases such as removal at the head.

#876 Middle of the Linked List [Easy]

Key Insights

- Use two pointers (slow and fast) to traverse the list.
- The fast pointer moves two steps at a time while the slow pointer moves one step.
- When the fast pointer reaches the end, the slow pointer will be at the middle node.
- This approach naturally handles both odd and even lengths of the list.

Space & Time

Time Complexity: O(n), where n is the number of nodes in the list.

Space Complexity: O(1), constant extra space.

Solution

The solution uses the two-pointer technique:

- Initialize two pointers, slow and fast, at the head. - Traverse the list, moving slow by one step and fast by two steps during each iteration. - When fast reaches the end or has no next node, slow will be at the middle. - Return the slow pointer (the middle node).

This strategy ensures linear time complexity and constant space, efficiently finding the middle node even for lists with an even number of elements.

#1474 Delete N Nodes After M Nodes of Linked List [Easy]

Key Insights

- Traverse the linked list while keeping count.
- Skip m nodes (i.e., keep them) and then proceed to delete the next n nodes.
- Update pointers to bypass the removed nodes.
- This is an in-place modification problem, so no additional list structure is needed.
- Be cautious about edge cases when the list has fewer than m or n nodes remaining.

Space & Time

Time Complexity: O(N), where N is the number of nodes in the list.

Space Complexity: O(1) since the modification is done in-place.

Solution

The solution involves iterating through the linked list with a pointer. For each cycle:

- Traverse m nodes and keep them. - Then, from the (m+1)th node, skip and remove the next n nodes by adjusting the 'next' pointer of the m-th node to point to the node after these n nodes. - Continue until you reach the end of the list. This in-place approach avoids using extra space and maintains O(N) time complexity.

#2 Add Two Numbers [Medium]

Key Insights

- The digits are stored in reverse order, meaning the first node is the least significant digit.
- A carry may need to propagate through subsequent nodes.
- The two linked lists might be of different lengths.
- Continue processing until both lists are fully traversed and any remaining carry is handled.

Space & Time

Time Complexity: O(max(n, m)), where n and m are the lengths of the two linked lists.

Space Complexity: O(max(n, m) + 1) for the result list in the worst case.

Solution

The solution iterates through both linked lists node by node. For each pair of corresponding nodes, their values and any carry from the previous operation are summed. The result is decomposed into a current digit (modulo 10) and an updated carry (total divided by 10). A new node is created for the current digit and appended to the resulting linked list. This process continues until all nodes are processed and any leftover carry is added as a final node.

#19 Remove Nth Node From End of List [Medium]

Key Insights

- Use a dummy node at the start to simplify edge cases, especially when the head is removed.
- Utilize two pointers (fast and slow) and maintain a gap of n+1 nodes between them.
- By moving both pointers simultaneously, the slow pointer will point to the node immediately before the target node when the fast pointer reaches the end.

Space & Time

Time Complexity: O(L), where L is the length of the list

Space Complexity: O(1)

Solution

We use a two-pointer approach to achieve a one-pass solution. A dummy node is created and linked to the head to handle edge cases gracefully (such as needing to remove the head). The fast pointer is advanced n+1 steps ahead of the slow pointer to maintain the required gap. Then, both pointers are moved simultaneously until the fast pointer reaches the end of the list. At that point, the slow pointer is immediately before the node to be deleted. We then adjust the pointers to remove the target node from the list.

#24 Swap Nodes in Pairs [Medium]

Key Insights

- Use a dummy node to simplify edge cases, such as when the list is empty or has only one node.
- Process nodes in pairs and update the pointers accordingly.
- Ensure that the swapping does not lose the connection between the swapped pair and the rest of the list.
- The solution can be implemented iteratively or recursively.

Space & Time

Time Complexity: O(n) - Each node is processed once.

Space Complexity: O(1) - Only constant extra space is used.

Solution

We use an iterative approach with a dummy node. The dummy node points to the head of the list. We maintain a pointer that traverses the list in pairs. For each pair, we adjust the next pointers to swap the two nodes. By the end, the dummy's next pointer will be the new head of the modified list. The main idea is to keep track of the previous node (prev) before the pair, and then adjust pointers accordingly using temporary variables to hold references to the nodes being swapped.

#61 Rotate List [Medium]

Key Insights

- Calculate the length of the list and locate the tail.
- Use k modulo length to handle rotations greater than the list size.
- Find the new tail of the list at position (length - k - 1) and update pointers accordingly.
- Link the original tail to the original head to form a circular list, then break the circle at the new tail.

Space & Time

Time Complexity: O(n), where n is the number of nodes in the list, since we traverse the list a few times.

Space Complexity: O(1), as the rotation is done in place without extra data structures.

Solution

We first traverse the list to determine its length and identify the tail node. The effective rotation is calculated by taking k modulo the length of the list. If the result is zero, the list remains unchanged. Otherwise, we locate the new tail node by moving (length - k - 1) steps from the head. The node after the new tail becomes the new head. Finally, we break the link after the new tail and connect the old tail to the original head to finalize the rotated list.

#146 LRU Cache [Medium]

Key Insights

- Use a Hash Table (dictionary) to allow O(1) access to nodes by key.
- Use a Doubly Linked List to record the usage order. The head represents the most recently used element while the tail represents the least recently used.
- When a key is accessed or updated, move the corresponding node to the front (head) of the list.
- When the cache exceeds capacity, remove the node at the tail end which is the least recently used entry.

Space & Time

Time Complexity: O(1) per get and put operation.

Space Complexity: O(capacity), storing key-node pairs and linked list nodes.

Solution

The solution combines a hash table (or dictionary) and a doubly linked list:

- The hash table stores keys and corresponding node addresses. - The doubly linked list maintains the order of usage with the most recently used items near the head. - For get(key), check the dictionary. If found, move the node to the head and return its value; if not, return -1. - For put(key, value), if the key exists, update its value and move it to the head. If not, create a new node and add it right after the head. If the capacity is reached, remove the tail node (least recently used) and update the dictionary accordingly. - Dummy head and tail nodes are used to simplify node addition and removal procedures without worrying about edge cases.

#148 Sort List [Medium]

Key Insights

- Merge sort is well-suited for linked lists; it naturally relies on splitting the list and merging sorted sublists.
- The slow and fast pointer technique is used to find the middle point of the list.
- A recursive merge sort implementation may use O(log n) space for recursion stack. To achieve O(1) extra space, an iterative bottom-up merge sort approach would be preferred.
- Merging two sorted lists is efficient with linked lists because of their sequential access.

Space & Time

Time Complexity: O(n log n)

Space Complexity: O(1) extra space for an iterative bottom-up approach (O(log n) for recursive calls if using recursion)

Solution

The solution employs the merge sort algorithm tailored for linked lists. The overall steps are:

- Use the slow and fast pointers to find the middle of the list and split it into two halves. - Recursively sort the two halves. - Merge the two sorted halves by comparing node values. - Ensure that during merging, pointers are adjusted properly to build the sorted list.

For an O(1) space solution, an iterative bottom-up merge sort can be implemented. However, the recursive merge sort is easier to understand and implement, though it uses O(log n) space on the call stack.

#328 Odd Even Linked List [Medium]

Key Insights

- Use two pointers: one for odd-indexed nodes and one for even-indexed nodes.
- The first node is considered odd, and the second node is even.
- Traverse the list once, reassigning the next pointers to segregate odd and even nodes.
- Finally, attach the even list at the end of the odd list.
- The solution must use O(1) extra space and O(n) time complexity.

Space & Time

Time Complexity: O(n) since the list is traversed once.

Space Complexity: O(1) as only a few pointers are used.

Solution

Maintain two pointers, odd and even, starting from the head and head.next, respectively. As you iterate, update odd.next to point to the next odd element (skipping an even node) and even.next to point to the next even element. Once the end is reached, attach the even list to the end of the odd list. This in-place rearrangement preserves the initial ordering within odd and even groups while meeting the space and time constraints.

#369 Plus One Linked List [Medium]

Key Insights

- The digits are stored such that the most significant digit comes first, but addition is easier starting from the least significant digit.
- Reversing the list allows simpler management of the carry during addition.
- After processing the addition, reversing the list back restores the original order.
- Consider special cases where all digits are 9, requiring a new head node to accommodate an additional digit.

Space & Time

Time Complexity: O(n), where n is the number of nodes in the list

Space Complexity: O(1) extra space when using an iterative reversal method. (O(n) if recursion is used)

Solution

We solve the problem by first reversing the linked list, which converts it to a format where the least significant digit is processed first. We then iterate over the reversed list, adding one and managing any carry. If we encounter a carry at the end of the list, a new node is appended to handle the overflow. Finally, we reverse the list again to restore the original order. This method efficiently handles the addition with constant extra space during processing.

#430 Flatten a Multilevel Doubly Linked List [Medium]

Key Insights

- Use a depth-first traversal to process child lists before proceeding to the next pointer.
- When a node with a child is found, recursively flatten the child list.
- Splice the flattened child list in between the current node and the next node.
- Maintain previous and next pointers correctly to preserve the doubly linked list properties.
- Ensure that all child pointers are set to null after flattening.

Space & Time

Time Complexity: O(n), where n is the total number of nodes, since each node is processed exactly once.

Space Complexity: O(n) in the worst-case due to recursion stack in case of a highly nested list.

Solution

We can solve this problem using a recursive depth-first search (DFS) approach. The main idea is to traverse the list and, whenever a node with a child pointer is encountered, recursively flatten its child list. Once the child list is flattened, it can be spliced into the main list between the current node and the node originally following it. We then continue processing the next pointer. This ensures that the entire list is flattened in depth-first order. The recursive helper function returns the tail of the flattened segment, which is used to connect the subsequent parts of the list correctly.

#622 Design Circular Queue [Medium]

Key Insights

- Leverage a fixed-size array to store the queue elements.
- Use two pointers: one for the front and one for the rear.
- Maintain a count variable (or use pointer arithmetic) to differentiate between an empty and a full queue.
- Use modulo arithmetic to wrap pointers when they reach the end of the array.
- Ensure all operations (enQueue, deQueue, Front, Rear, isEmpty, isFull) have O(1) time complexity.

Space & Time

Time Complexity: O(1) per operation for enQueue, deQueue, Front, Rear, isEmpty, and isFull.

Space Complexity: O(k), where k is the capacity of the circular queue.

Solution

We implement the circular queue using a fixed-size array alongside two pointers, front and rear, and a counter to keep track of the number of elements in the queue. The front pointer indicates the index of the first element, and the rear pointer indicates the index where the next element will be enqueued. When enqueuing an element, we insert the element at the rear index and then increment the rear pointer modulo the size of the array. Similarly, when

dequeuing, we remove the element at the front index and increment the front pointer modulo the size. The counter helps in quickly checking if the queue is empty or full (count equals 0 or count equals capacity, respectively). This design ensures all operations remain O(1) and correctly wraps around, forming a circular structure.

#707 Design Linked List [Medium]

Key Insights

- Use a dummy head node to simplify insertion and deletion at the head.
- Maintain a size counter to quickly validate index-based operations.
- For every operation, traverse the list up to the required point since random access is not available.
- Ensure robustness by handling edge cases like negative indices and indices larger than the current length.

Space & Time

Time Complexity:

- get, addAtIndex, and deleteAtIndex operations require O(n) time due to the need to traverse the list.

- addAtHead and addAtTail (if tail pointer is not maintained) are O(n) in worst case; however, these can be optimized further if a tail pointer is added.

Space Complexity:

- O(n) space for storing n nodes.

Solution

The solution employs a singly linked list with a dummy head node to ease edge-case handling. A size counter is maintained for quick index validations. Each node holds a value and a pointer to the next node. Operations operate by traversing the list to the correct insertion or deletion point. This design makes it simple to add the functionality of getting values, inserting at specified indices, and removing nodes.

#708 Insert into a Sorted Circular Linked List [Medium]

Key Insights

- Handle the empty list edge-case by creating a new node that points to itself.
- Traverse the list until the proper insertion point is found.
- Consider the "turning point" where the maximum value is followed by the minimum value.
- If no proper place is found in one full rotation, insert the new node arbitrarily (e.g., after the head).

Space & Time

Time Complexity: O(n) in the worst-case when a full loop is required.

Space Complexity: O(1) since we only use a few extra pointers.

Solution

We traverse the circular linked list starting from the arbitrary given node. For each pair of consecutive nodes (curr and next), we check if the new value can be inserted between them by confirming that:

- The new value lies between curr and next (curr.val ≤ insertVal ≤ next.val). - Or if we are at the pivot point where the maximum value meets the minimum value (i.e., curr.val > next.val) and then either the new value is larger than or equal to curr.val (should be appended after the maximum) or less than or equal to next.val (should be inserted before the minimum).

If no proper spot is found after one complete loop, we insert anywhere (all values in the list are the same). A pointer reference is updated accordingly to maintain the circular structure.

#1171 Remove Zero Sum Consecutive Nodes from Linked List [Medium]

Key Insights

- Use a dummy node to handle edge cases such as the removal of nodes at the head.
- Compute a running (prefix) sum while traversing the list.
- Use a hash table (or hashmap) to map each prefix sum to the most recent node where that sum occurred.

- If the same prefix sum is encountered again, the nodes in between sum to 0 and can be removed by adjusting pointers.
- Perform two passes: the first to record prefix sums and their corresponding nodes, and the second to remove zero-sum sequences.

Space & Time

Time Complexity: O(n) - Two passes over the list.

Space Complexity: O(n) - Hash table storing at most n prefix sums.

Solution

The solution uses a two-pass algorithm with a hash table. In the first pass, compute the prefix sum for each node and store the mapping from prefix sum to the node (overwriting with the latest occurrence). This ensures that for any repeated prefix sum, the nodes in between sum to 0. In the second pass, traverse the list again and use the hash table to skip the zero-sum subsequences by redirecting the next pointer of a node to the next pointer of the stored node for that prefix sum. Key data structures are the hash table and a dummy node to simplify pointer adjustments.

#25 Reverse Nodes in k-Group [Hard]

Key Insights

- Use a dummy node to simplify edge cases.
- Identify groups of k nodes using a helper function.
- Reverse the nodes in each group using pointer manipulation.
- If a remaining group has fewer than k nodes, leave it unchanged.
- The reversal is done in-place with O(1) extra space.

Space & Time

Time Complexity: O(n) where n is the number of nodes (each node is processed a constant number of times)

Space Complexity: O(1) (only constant extra space is used)

Solution

The solution involves iterating through the linked list by identifying groups of k consecutive nodes. For each group: - Find the kth node from the current starting point. If it does not exist, the remainder of the list is not reversed. - Reverse the nodes in the identified group in-place by adjusting the next pointers. - Reconnect the reversed group with the previous part of the list and update pointers to move to the next group. The approach leverages a dummy node for easier handling of head modifications and uses a helper function to fetch the kth node. The in-place reversal ensures that the extra memory usage is constant.

#432 All O'one Data Structure [Hard]

Key Insights

- Use a Doubly-Linked List (DLL) where each node holds a unique count value and a set of keys with that count.
- Maintain a Hash Map (keyToNode) to quickly locate the node corresponding to each key.
- Insert new nodes in the DLL when a key's count is incremented or decremented and the adjacent node does not have the target count.
- Remove nodes from the DLL when no keys remain in that count.
- The head of the DLL holds the minimum count and the tail holds the maximum count.

Space & Time

Time Complexity: O(1) average for inc, dec, getMaxKey, and getMinKey.

Space Complexity: O(n) where n is the number of unique keys in the data structure.

Solution

The solution uses a combination of a doubly-linked list and hash maps. Each node in the DLL represents a frequency and maintains a set of keys that have that frequency. A hash map maps keys to their corresponding node. When inc(key) or dec(key) is called, we update the key's frequency and move the key to the appropriate node in the DLL. If the adjacent node with the required frequency doesn't exist, we create a new node and insert it in the correct

position. Removal of a key from a node and deletion of empty nodes ensures that getMinKey() and getMaxKey() can be returned directly from the head and tail of the DLL.

#460 LFU Cache [Hard]

Key Insights

- Maintain a mapping from key to its value and frequency.
- Use an additional mapping from frequency to keys (maintaining the order of insertion or recent use) for quick eviction.
- Keep a global minFreq variable to track the lowest frequency present in the cache.
- When a key is accessed or updated, increase its frequency and relocate it in the frequency mapping.
- Evictions occur by removing the oldest key from the lowest frequency bucket.

Space & Time

Time Complexity: O(1) average for both get and put.

Space Complexity: O(capacity), where capacity is the maximum number of keys stored.

Solution

The solution utilizes two main data structures:

- A hash map (keyToValFreq) that maps keys to a pair (value, frequency). - A second hash map (freqToKeys) mapping frequencies to an ordered list/set of keys. This data structure preserves the order in which keys were added to allow removal of the least recently used key when multiple keys share the same frequency.

A global minFreq variable is maintained to quickly identify the bucket with the least frequency. When a key is accessed or updated, the key is removed from its current frequency list and placed in the next higher frequency list. If the frequency list for the minimal frequency becomes empty, minFreq is updated.

This design guarantees that both get and put operations execute in constant time on average.

#17 DFS — 23 questions · #17 of 21

#463 EASY

Island Perimeter

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · DFS · BFS · Matrix

Lists: Denny Zhang

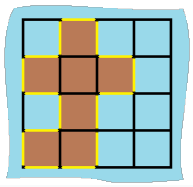
Problem:

You are given row x col grid representing a map where grid[i][j] = 1 represents land and grid[i][j] = 0 represents water.

Grid cells are connected horizontally/vertically (not diagonally). The grid is completely surrounded by water, and there is exactly one island (i.e., one or more connected land cells).

The island doesn't have "lakes", meaning the water inside isn't connected to the water around the island. One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.

Example 1:



```
Input: grid = [[0,1,0,0],[1,1,1,0],[0,1,0,0],[1,1,0,0]]
Output: 16
Explanation: The perimeter is the 16 yellow stripes in the image above.
```

Example 2:

```
Input: grid = [[1]]
Output: 4
```

Example 3:

```
Input: grid = [[1,0]]
Output: 4
```


Constraints:

- row == grid.length
- col == grid[i].length
- 1 <= row, col <= 100
- grid[i][j] is 0 or 1.
- There is exactly one island in grid.

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def islandPerimeter(self, grid):
        # Brute force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python

```python
# Python solution for Island Perimeter
def islandPerimeter(grid):
    # Initialize perimeter to 0
    perimeter = 0

    # Get number of rows and columns
    rows = len(grid)
    cols = len(grid[0])

    # Iterate through each cell in the grid
    for i in range(rows):
        for j in range(cols):
            # Only process if current cell is land
            if grid[i][j] == 1:
                # Each land cell contributes 4 edges initially
                cell_perimeter = 4

                # Check if the upper neighbor is also land
                if i > 0 and grid[i-1][j] == 1:
                    cell_perimeter -= 1
                # Check if the lower neighbor is also land
                if i < rows - 1 and grid[i+1][j] == 1:
                    cell_perimeter -= 1
                # Check if the left neighbor is also land
                if j > 0 and grid[i][j-1] == 1:
```



```python
                    cell_perimeter -= 1
                # Check if the right neighbor is also land
                if j < cols - 1 and grid[i][j+1] == 1:
                    cell_perimeter -= 1

                # Add the effective perimeter of the current cell to the overall perimeter
                perimeter += cell_perimeter

    return perimeter

# Example usage:
# grid = [[0,1,0,0],[1,1,1,0],[0,1,0,0],[1,1,0,0]]
# print(islandPerimeter(grid)) # Output should be 16
```

Python

```python
class Solution:
    def islandPerimeter(self, grid: list[list[int]]) -> int:
        m = len(grid)
        n = len(grid[0])

        islands = 0
        neighbors = 0

        for i in range(m):
            for j in range(n):
                if grid[i][j] == 1:
                    islands += 1
                    if i + 1 < m and grid[i + 1][j] == 1:
                        neighbors += 1
                    if j + 1 < n and grid[i][j + 1] == 1:
                        neighbors += 1

        return islands * 4 - neighbors * 2
```

Python

```python
class Solution:
    def islandPerimeter(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        ans = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 1:
                    ans += 4
                    if i > 0 and grid[i - 1][j] == 1:
                        ans -= 2
                    if j < n - 1 and grid[i][j + 1] == 1:
                        ans -= 2
        return ans
```

Python


```python
class Solution:
    def islandPerimeter(self, grid: List[List[int]]) -> int:
        m, n = len(grid), len(grid[0])
        ans = 0
        for i in range(m):
            for j in range(n):
                if grid[i][j] == 1:
                    ans += 4
                    if i > 0 and grid[i - 1][j] == 1:
                        ans -= 2
                    if j < n - 1 and grid[i][j + 1] == 1:
                        ans -= 2
        return ans
```

[*] DFS — Pattern Solution (stored optimal)

Python

```python
# Python solution for Island Perimeter
def islandPerimeter(grid):
    # Initialize perimeter to 0
    perimeter = 0

    # Get number of rows and columns
    rows = len(grid)
    cols = len(grid[0])

    # Iterate through each cell in the grid
    for i in range(rows):
        for j in range(cols):
            # Only process if current cell is land
            if grid[i][j] == 1:
                # Each land cell contributes 4 edges initially
                cell_perimeter = 4

                # Check if the upper neighbor is also land
                if i > 0 and grid[i-1][j] == 1:
                    cell_perimeter -= 1
                # Check if the lower neighbor is also land
                if i < rows - 1 and grid[i+1][j] == 1:
                    cell_perimeter -= 1
                # Check if the left neighbor is also land
                if j > 0 and grid[i][j-1] == 1:
```

#733 EASY

Flood Fill

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · DFS · BFS · Matrix

Lists: Grind 169

Problem:

You are given an image represented by an m x n grid of integers image, where image[i][j] represents the pixel value of the image. You are also given three integers sr, sc, and color. Your task is to perform a flood fill on the image starting from the pixel image[sr][sc].

To perform a flood fill:

1. Begin with the starting pixel and change its color to color.
2. Perform the same process for each pixel that is directly adjacent (pixels that share a side with the original pixel, either horizontally or vertically) and shares the same color as the starting pixel.
3. Keep repeating this process by checking neighboring pixels of the updated pixels and modifying their color if it matches the original color of the starting pixel.
4. The process stops when there are no more adjacent pixels of the original color to update.

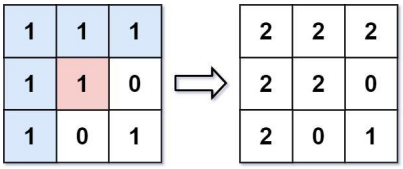
Return the modified image after performing the flood fill.

Example 1:

Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2

Output: [[2,2,2],[2,2,0],[2,0,1]]

Explanation:



From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.

Note the bottom corner is not colored 2, because it is not horizontally or vertically connected to the starting pixel.

Example 2:

Input: image = [[0,0,0],[0,0,0]], sr = 0, sc = 0, color = 0

Output: [[0,0,0],[0,0,0]]

Explanation:

The starting pixel is already colored with 0, which is the same as the target color. Therefore, no changes are made to the image.

Constraints:

- m == image.length
- n == image[i].length
- 1 <= m, n <= 50
- 0 <= image[i][j], color < 216
- 0 <= sr < m
- 0 <= sc < n

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def floodFill(self, image, sr, sc, color):
        # Brute Force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python


```python
def floodFill(image, sr, sc, color):
    # Get the original color of the starting pixel
    orig_color = image[sr][sc]
    # If the new color is the same as the original, return the image as is
    if orig_color == color:
        return image
    rows, cols = len(image), len(image[0])

    # Recursive DFS helper function to fill the image
    def dfs(r, c):
        # Check boundaries and if the current pixel is the same as the original color
        if r < 0 or r >= rows or c < 0 or c >= cols or image[r][c] != orig_color:
            return
        # Update the current pixel's color
        image[r][c] = color
        # Recursively call DFS in all four directions
        dfs(r - 1, c) # up
        dfs(r + 1, c) # down
        dfs(r, c - 1) # left
        dfs(r, c + 1) # right

    dfs(sr, sc)
    return image
```

Python

```python
class Solution:
    def floodFill(self, image: list[list[int]],
                  sr: int, sc: int, newColor: int) -> list[list[int]]:
        startColor = image[sr][sc]
        seen = set()

        def dfs(i: int, j: int) -> None:
            if i < 0 or i >= len(image) or j < 0 or j >= len(image[0]):
                return
            if image[i][j] != startColor or (i, j) in seen:
                return

            image[i][j] = newColor
            seen.add((i, j))

            dfs(i + 1, j)
            dfs(i - 1, j)
            dfs(i, j + 1)
            dfs(i, j - 1)

        dfs(sr, sc)
        return image
```

Python


```python
class Solution:
    def floodFill(
        self, image: List[List[int]], sr: int, sc: int, color: int
    ) -> List[List[int]]:
        def dfs(ii: int, jj: int):
            image[ii][jj] = color
            for a, b in pairwise(dirs):
                x, y = ii + a, jj + b
                if 0 <= x < len(image) and 0 <= y < len(image[0]) and image[x][y] == oc:
                    dfs(x, y)

        oc = image[sr][sc]
        if oc != color:
            dirs = (-1, 0, 1, 0, -1)
            dfs(sr, sc)
        return image
```

Python

```python
class Solution:
    def floodFill(
        self, image: List[List[int]], sr: int, sc: int, color: int
    ) -> List[List[int]]:
        def dfs(i, j):
            if (
                not 0 <= i < m
                or not 0 <= j < n
                or image[i][j] != oc
                or image[i][j] == color
            ):
                return
            image[i][j] = color
            for a, b in pairwise(dirs):
                dfs(i + a, j + b)


        dirs = (-1, 0, 1, 0, -1)
        m, n = len(image), len(image[0])
        oc = image[sr][sc]
        dfs(sr, sc)
        return image
```

[*] DFS — Pattern Solution (matched from community answers)

Python


```python
def floodFill(image, sr, sc, color):
    # Get the original color of the starting pixel
    orig_color = image[sr][sc]
    # If the new color is the same as the original, return the image as is
    if orig_color == color:
        return image
    rows, cols = len(image), len(image[0])

    # Recursive DFS helper function to fill the image
    def dfs(r, c):
        # Check boundaries and if the current pixel is the same as the original color
        if r < 0 or r >= rows or c < 0 or c >= cols or image[r][c] != orig_color:
            return
        # Update the current pixel's color
        image[r][c] = color
        # Recursively call DFS in all four directions
        dfs(r - 1, c) # up
        dfs(r + 1, c) # down
        dfs(r, c - 1) # left
        dfs(r, c + 1) # right

    dfs(sr, sc)
    return image
```

#130 MEDIUM

Surrounded Regions

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · DFS · BFS · Union Find · Matrix

Lists: Denny Zhang

Problem:

You are given an m x n matrix board containing letters 'X' and 'O', capture regions that are surrounded:

- Connect: A cell is connected to adjacent cells horizontally or vertically.
- Region: To form a region connect every 'O' cell.
- Surround: A region is surrounded if none of the 'O' cells in that region are on the edge of the board. Such regions are completely enclosed by 'X' cells.

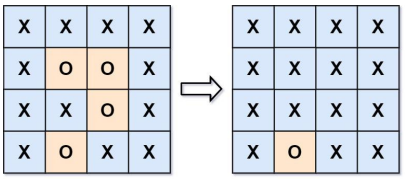
To capture a surrounded region, replace all 'O's with 'X's in-place within the original board. You do not need to return anything.

Example 1:

Input: board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]

Output: [["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]

Explanation:



In the above diagram, the bottom region is not captured because it is on the edge of the board and cannot be surrounded.

Example 2:

Input: board = [["X"]]

Output: [["X"]]

Constraints:

- m == board.length
- n == board[i].length
- 1 <= m, n <= 200
- board[i][j] is 'X' or 'O'.

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def solve(self, board):
        # Brute Force Dim-n) - BFS from border 'O's, mark safe, flip the rest
        from collections import deque
        if not board: return
        m, n = len(board), len(board[0])
        safe = set()
        queue = deque()
        for r in range(m):
            for c in range(n):
                if (r==0 or r==m-1 or c==0 or c==n-1) and board[r][c]=='O':
                    queue.append((r,c))
        while queue:
            r,c = queue.popleft()
            if (r,c) in safe or board[r][c]!='O': continue
            safe.add((r,c))
            for dr,dc in [(1,0),(-1,0),(0,1),(0,-1)]:
                nr,nc = r+dr,c+dc
                if 0<=nr<m and 0<=nc<n: queue.append((nr,nc))
        for r in range(m):
            for c in range(n):
                if board[r][c]=='O' and (r,c) not in safe:
                    board[r][c]='X'
```

Python

Python

```python
def solve(board):
    if not board or not board[0]:
        return

    rows, cols = len(board), len(board[0])

    def dfs(r, c):
        # Return if out of bounds or current cell is not 'O'
        if r < 0 or c < 0 or r >= rows or c >= cols or board[r][c] != 'O':
            return

        # Mark the cell as safe
        board[r][c] = '+'
        # Explore four possible directions
        dfs(r + 1, c)
        dfs(r - 1, c)
        dfs(r, c + 1)
        dfs(r, c - 1)

    # Traverse the board borders
    for i in range(rows):
        if board[i][0] == 'O':
            dfs(i, 0)
        if board[i][cols - 1] == 'O':
            dfs(i, cols - 1)
    for j in range(cols):
```



```python
        if board[0][j] == 'O':
            dfs(0, j)
        if board[rows - 1][j] == 'O':
            dfs(rows - 1, j)

    # Flip surrounded 'O's and restore safe cells
    for i in range(rows):
        for j in range(cols):
            if board[i][j] == 'O':
                board[i][j] = 'X'
            elif board[i][j] == '+':
                board[i][j] = 'O'
```

Python

```python
class Solution:
  def solve(self, board: list[list[str]]) -> None:
    if not board:
      return

    DIRS = ((0, 1), (1, 0), (0, -1), (-1, 0))
    m = len(board)
    n = len(board[0])
    q = collections.deque()

    for i in range(m):
      for j in range(n):
        if i * j == 0 or i == m - 1 or j == n - 1:
          if board[i][j] == 'O':
            q.append((i, j))
            board[i][j] = '+'

    # Mark the grids that stretch from the four sides with '+'.
    while q:
      i, j = q.popleft()
      for dx, dy in DIRS:
        x = i + dx
        y = j + dy
        if x < 0 or x == m or y < 0 or y == n:
          continue

        if board[x][y] != 'O':
          continue
        q.append((x, y))
        board[x][y] = '+'

    for row in board:
      for i, c in enumerate(row):
        row[i] = 'O' if c == '+' else 'X'
```

Python


```python
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        def dfs(ii: int, jj: int):
            if not (0 <= i < m and 0 <= j < n and board[i][j] == "O"):
                return
            board[i][j] = "."
            for a, b in pairwise((-1, 0, 1, 0, -1)):
                dfs(i + a, j + b)

        m, n = len(board), len(board[0])
        for i in range(m):
            dfs(i, 0)
            dfs(i, n - 1)
        for j in range(n):
            dfs(0, j)
            dfs(m - 1, j)
        for i in range(m):
            for j in range(n):
                if board[i][j] == ".":
                    board[i][j] = "O"
                elif board[i][j] == "O":
                    board[i][j] = "X"
```

Python

```python
class Solution:
    def solve(self, board: List[List[str]]) -> None:
        """

        Do not return anything, modify board in-place instead.
        """

        def dfs(i, j):
            board[i][j] = "."
            for a, b in [(0, -1), (0, 1), (1, 0), (-1, 0)]:
                x, y = i + a, j + b
                if 0 <= x < m and 0 <= y < n and board[x][y] == 'O':
                    dfs(x, y)

        m, n = len(board), len(board[0])
        for i in range(m):
            for j in range(n):
                if board[i][j] == 'O' and (
                    i == 0 or i == m - 1 or j == 0 or j == n - 1
                    ):
                    dfs(i, j)
        for i in range(m):
            for j in range(n):
                if board[i][j] == 'O':
                    board[i][j] = 'X'
                elif board[i][j] == '.':

                    board[i][j] = 'O'
```

[*] DFS — Pattern Solution (matched from community answers)

Python


```python
def solve(board):
    if not board or not board[0]:
        return

    rows, cols = len(board), len(board[0])

    def dfs(r, c):
        # Return if out of bounds or current cell is not 'O'
        if r < 0 or c < 0 or r >= rows or c >= cols or board[r][c] != 'O':
            return

        # Mark the cell as safe
        board[r][c] = '+'
        # Explore four possible directions
        dfs(r + 1, c)
        dfs(r - 1, c)
        dfs(r, c + 1)
        dfs(r, c - 1)

    # Traverse the board borders
    for i in range(rows):
        if board[i][0] == 'O':
            dfs(i, 0)
        if board[i][cols - 1] == 'O':
            dfs(i, cols - 1)
    for j in range(cols):

        if board[0][j] == 'O':
            dfs(0, j)
        if board[rows - 1][j] == 'O':
            dfs(rows - 1, j)

    # Flip surrounded 'O's and restore safe cells
    for i in range(rows):
        for j in range(cols):
            if board[i][j] == 'O':
                board[i][j] = 'X'
            elif board[i][j] == '+':
                board[i][j] = 'O'
```

#133 MEDIUM

Clone Graph

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · DFS · BFS · Graph

Lists: Grind 169

Problem:

Given a reference of a node in a connected undirected graph.

Return a deep copy (clone) of the graph.

Each node in the graph contains a value (int) and a list (List[Node]) of its neighbors.

```
class Node {
public int val;
public List<Node> neighbors;
}
```

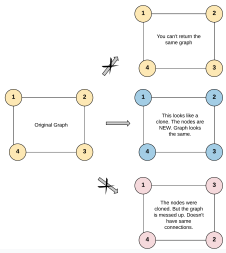

Test case format:

For simplicity, each node's value is the same as the node's index (1-indexed). For example, the first node with val == 1, the second node with val == 2, and so on. The graph is represented in the test case using an adjacency list.

An adjacency list is a collection of unordered lists used to represent a finite graph. Each list describes the set of neighbors of a node in the graph.

The given node will always be the first node with val = 1. You must return the copy of the given node as a reference to the cloned graph.

Example 1:




```
Input: adjList = [[2,4],[1,3],[2,4],[1,3]]
Output: [[2,4],[1,3],[2,4],[1,3]]
Explanation: There are 4 nodes in the graph.
1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).
4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).
```

Example 2:


```
Input: adjList = [[]]
Output: [[]]
Explanation: Note that the input contains one empty list. The graph consists of only one node
with val = 1 and it does not have any neighbors.
```

Example 3:

```
Input: adjList = []
Output: []
Explanation: This an empty graph, it does not have any nodes.
```

Constraints:

- The number of nodes in the graph is in the range [0, 100].
- 1 <= Node.val <= 100
- Node.val is unique for each node.
- There are no repeated edges and no self-loops in the graph.
- The Graph is connected and all nodes can be visited starting from the given node.

Python

Python

```python
# Definition for a Node.
class Node:
    def __init__(self, val, neighbors=None):
        self.val = val
        # Initialize neighbors with empty list if None is provided
        self.neighbors = neighbors if neighbors is not None else []

def cloneGraph(node: 'Node') -> 'Node':
    # Hash map to track original nodes and their corresponding clones
    old_to_new = {}

    def dfs(current):
        # Base case: if the current node is None, return None
        if current is None:
            return None
        # If the node has already been cloned, return the clone to avoid duplication and cycle issues
        if current in old_to_new:
            return old_to_new[current]
        # Create a clone for the current node
        copy = Node(current.val)
        old_to_new[current] = copy
        # Recursively clone all neighbors
        for neighbor in current.neighbors:
            copy.neighbors.append(dfs(neighbor))
        return copy

    return dfs(node)
```

Python

```python
class Solution:
  def cloneGraph(self, node: 'Node') -> 'Node':
    if not node:
      return None

    q = collections.deque([node])
    map = {node: Node(node.val)}

    while q:
      u = q.popleft()
      for v in u.neighbors:
        if v not in map:
          map[v] = Node(v.val)
          q.append(v)
        map[u].neighbors.append(map[v])

    return map[node]
```

Python


```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val = 0, neighbors = None):
        self.val = val
        self.neighbors = neighbors if neighbors is not None else []
"""

from typing import Optional


class Solution:
    def cloneGraph(self, node: Optional["Node"]) -> Optional["Node"]:
        def dfs(node):
            if node is None:
                return None
            if node in g:
                return g[node]
            cloned = Node(node.val)
            g[node] = cloned
            for nxt in node.neighbors:
                cloned.neighbors.append(dfs(nxt))
            return cloned

        g = defaultdict()

        return dfs(node)
```

Python

```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val = 0, neighbors = None):
        self.val = val
        self.neighbors = neighbors if neighbors is not None else []
"""


class Solution:
    def cloneGraph(self, node: 'Node') -> 'Node':
        visited = defaultdict()

        def clone(node):
            if node is None:
                return None
            if node in visited:
                return visited[node]
            c = Node(node.val)
            visited[node] = c
            for e in node.neighbors:
                c.neighbors.append(clone(e))
            return c

        return clone(node)
```

[*] DFS — Pattern Solution (matched from community answers)

Python


```python
# Definition for a Node.
class Node:
    def __init__(self, val, neighbors=None):
        self.val = val
        # Initialize neighbors with empty list if None is provided
        self.neighbors = neighbors if neighbors is not None else []

def cloneGraph(node: 'Node') -> 'Node':
    # Hash map to track original nodes and their corresponding clones
    old_to_new = {}

    def dfs(current):
        # Base case: if the current node is None, return None
        if current is None:
            return None
        # If the node has already been cloned, return the clone to avoid duplication and cycle issues
        if current in old_to_new:
            return old_to_new[current]
        # Create a clone for the current node
        copy = Node(current.val)
        old_to_new[current] = copy
        # Recursively clone all neighbors
        for neighbor in current.neighbors:
            copy.neighbors.append(dfs(neighbor))
        return copy

    return dfs(node)
```

#200 MEDIUM

Number of Islands

LeetCodeCS · SimplyLand · WalkCC · LeetCode · Editorial

Array · DFS · BFS · Union Find · Matrix

Lists: Grind 169

Problem:

Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return the number of islands.

An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

Example 1:

```
Input: grid = [
["1","1","1","1","0"],
["1","1","0","1","0"],
["1","1","0","0","0"],
["0","0","0","0","0"]
]
Output: 1
```

Example 2:


```
Input: grid = [
["1","1","0","0","0"],
["1","1","0","0","0"],
["0","0","1","0","0"],
["0","0","0","1","1"]
]
Output: 3
```

Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 300
- grid[i][j] is '0' or '1'.

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def numIslands(self, grid):
        # Brute force O(m*n*m*n) - DFS from every unvisited land cell
        if not grid: return 0
        m, n = len(grid), len(grid[0])
        visited = set()
        def dfs(r, c):
            if r<0 or r>=m or c<0 or c>=n: return
            if (r,c) in visited or grid[r][c]!='0': return
            visited.add((r,c))
            for dr,dc in [(1,0),(-1,0),(0,1),(0,-1)]:
                dfs(r+dr, c+dc)
        count = 0
        for r in range(m):
            for c in range(n):
                if grid[r][c]=='1' and (r,c) not in visited:
                    dfs(r, c)
                    count += 1
        return count
```

Python

Python


```python
# Function to count the number of islands in the grid
def numIslands(grid):
    # If the grid is empty, return 0
    if not grid:
        return 0
    rows, cols = len(grid), len(grid[0])
    islands = 0

    # Depth-First Search helper to mark the entire island as visited
    def dfs(r, c):
        # Check if the current cell is out of bounds or water
        if r < 0 or r >= rows or c < 0 or c >= cols or grid[r][c] == "0":
            return
        # Mark the cell as visited by setting it to water
        grid[r][c] = "0"
        # Recurse on the four adjacent cells (up, down, left, right)
        dfs(r + 1, c)
        dfs(r - 1, c)
        dfs(r, c + 1)
        dfs(r, c - 1)

    # Iterate over every cell in the grid
    for r in range(rows):
        for c in range(cols):
            # If a land cell is found, conduct DFS to visit the entire island

            if grid[r][c] == "1":
                islands += 1
                dfs(r, c)

    return islands
```

Python

```python
class Solution:
  def numIslands(self, grid: list[list[str]]) -> int:
    DIRS = ((0, 1), (1, 0), (0, -1), (-1, 0))
    m = len(grid)
    n = len(grid[0])

    def bfs(r, c):
      q = collections.deque([(r, c)])
      grid[r][c] = '2' # Mark '2' as visited.
      while q:
        i, j = q.popleft()
        for dx, dy in DIRS:
          x = i + dx
          y = j + dy
          if x < 0 or x >= m or y < 0 or y >= n:
            continue
          if grid[x][y] != '1':
            continue
          q.append((x, y))
          grid[x][y] = '2' # Mark '2' as visited.

    ans = 0

    for i in range(m):
      for j in range(n):
```



```python
        if grid[i][j] == '1':
          bfs(i, j)
          ans += 1

    return ans
```

Python

```python
class Solution:
  def numIslands(self, grid: list[list[str]]) -> int:
    m = len(grid)
    n = len(grid[0])

    def dfs(i: int, j: int) -> None:
      if i < 0 or i >= m or j < 0 or j >= n:
        return
      if grid[i][j] != '1':
        return

      grid[i][j] = '2' # Mark '2' as visited.
      dfs(i + 1, j)
      dfs(i - 1, j)
      dfs(i, j + 1)
      dfs(i, j - 1)

    ans = 0

    for i in range(m):
      for j in range(n):
        if grid[i][j] == '1':
          dfs(i, j)
          ans += 1

    return ans
```

Python

```python
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        def dfs(i, j):
            grid[i][j] = '0'
            for a, b in pairwise(dirs):
                x, y = i + a, j + b
                if 0 <= x < m and 0 <= y < n and grid[x][y] == '1':
                    dfs(x, y)

        ans = 0
        dirs = (-1, 0, 1, 0, -1)
        m, n = len(grid), len(grid[0])
        for i in range(m):
            for j in range(n):
                if grid[i][j] == '1':
                    dfs(i, j)
                    ans += 1
        return ans
```

Python

```python
# dfs
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        def dfs(i, j):
            if not (0 <= i < m and 0 <= j < n and grid[i][j] == '1'):
                return
            grid[i][j] = '0'
            for a, b in pairwise(dirs):
                x, y = i + a, j + b
                dfs(x, y)

        ans = 0
        dirs = (-1, 0, 1, 0, -1)
        m, n = len(grid), len(grid[0])
        for i in range(m):
            for j in range(n):
                if grid[i][j] == '1':
                    dfs(i, j)
                    ans += 1
        return ans

##############

# bfs
from collections import deque


class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid:
            return 0

        m, n = len(grid), len(grid[0])
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        islands = 0

        for i in range(m):
            for j in range(n):
                if grid[i][j] == "1":
                    islands += 1
                    q = deque([(i, j)]) # no need to reset q, q already drained from previous bfs
                    while q:
                        x, y = q.popleft()
                        for dx, dy in directions:
                            nx, ny = x + dx, y + dy
                            if 0 <= nx < m and 0 <= ny < n and grid[nx][ny] == "1":
                                q.append((nx, ny))
                                grid[nx][ny] = "0" # mark as visited
        return islands

##############
```



```python
# union find
# similar to UF in https://leetcode.ca/2016-09-30-305-Number-of-Islands-II/
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        def check(i, j):
            return 0 <= i < m and 0 <= j < n and grid[i][j] == "1" # Check for "1" instead of 1

        def find(x):
            if p[x] != x:
                p[x] = find(p[x])
            return p[x]

        m, n = len(grid), len(grid[0])
        p = list(range(m * n))
        cur = 0 # Initialize cur as 0 instead of n
        for i in range(m):
            for j in range(n):
                if grid[i][j] == "1":
                    cur += 1 # Increment cur when encountering "1"
                    for x, y in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                        if check(i + x, j + y) and find(i + n + j) != find((i + x) + n + j + y):
                            p[find(i + n + j)] = find((i + x) + n + j + y)
                            cur -= 1

        return cur


###########
```

[*] DFS — Pattern Solution (matched from community answers)

Python

```python
# dfs
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        def dfs(i, j):
            if not (0 <= i < m and 0 <= j < n and grid[i][j] == '1'):
                return
            grid[i][j] = '0'
            for a, b in pairwise(dirs):
                x, y = i + a, j + b
                dfs(x, y)

        ans = 0
        dirs = (-1, 0, 1, 0, -1)
        m, n = len(grid), len(grid[0])
        for i in range(m):
            for j in range(n):
                if grid[i][j] == '1':
                    dfs(i, j)
                    ans += 1
        return ans

##############

# bfs
from collections import deque
```



```python
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        if not grid:
            return 0

        m, n = len(grid), len(grid[0])
        directions = [(0, 1), (0, -1), (1, 0), (-1, 0)]
        islands = 0

        for i in range(m):
            for j in range(n):
                if grid[i][j] == "1":
                    islands += 1
                    q = deque([(i, j)]) # no need to reset q, q already drained from previous bfs
                    while q:
                        x, y = q.popleft()
                        for dx, dy in directions:
                            nx, ny = x + dx, y + dy
                            if 0 <= nx < m and 0 <= ny < n and grid[nx][ny] == "1":
                                q.append((nx, ny))
                                grid[nx][ny] = "0" # mark as visited
        return islands

##############


# union find
# similar to UF in https://leetcode.ca/2016-09-30-305-Number-of-Islands-II/
class Solution:
    def numIslands(self, grid: List[List[str]]) -> int:
        def check(i, j):
            return 0 <= i < m and 0 <= j < n and grid[i][j] == "1" # Check for "1" instead of 1

        def find(x):
            if p[x] != x:
                p[x] = find(p[x])
            return p[x]

        m, n = len(grid), len(grid[0])
        p = list(range(m * n))
        cur = 0 # Initialize cur as 0 instead of n
        for i in range(m):
            for j in range(n):
                if grid[i][j] == "1":
                    cur += 1 # Increment cur when encountering "1"
                    for x, y in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
                        if check(i + x, j + y) and find(i + n + j) != find((i + x) + n + j + y):
                            p[find(i + n + j)] = find((i + x) + n + j + y)
                            cur -= 1

        return cur


###########
```

#207 MEDIUM

Course Schedule

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

DFS · BFS · Graph · Topological Sort

Lists: Grind 169

Problem:

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

- For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

Example 1:

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: true
Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0. So it is possible.
```

Example 2:

```
Input: numCourses = 2, prerequisites = [[1,0],[0,1]]
Output: false
Explanation: There are a total of 2 courses to take.
To take course 1 you should have finished course 0, and to take course 0 you should also have
finished course 1. So it is impossible.
```

Constraints:

- 1 <= numCourses <= 2000
- 0 <= prerequisites.length <= 5000
- prerequisites[i].length == 2
- 0 <= ai, bi < numCourses
- All the pairs prerequisites[i] are unique.

>> Brute Force [DFS] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def canFinish(self, numCourses, prerequisites):
        # Brute Force O(V+E) - DFS cycle detection on adjacency list
        from collections import defaultdict
        graph = defaultdict(list)
        for a, b in prerequisites:
            graph[b].append(a)
        # 0=unvisited, 1=in-stack, 2=done
        state = [0] * numCourses
        def has_cycle(node):
            if state[node] == 1: return True
            if state[node] == 2: return False
            state[node] = 1
            for nei in graph[node]:
                if has_cycle(nei): return True
            state[node] = 2
            return False
        return not any(has_cycle(c) for c in range(numCourses))
```

Python

Python

```python
from collections import deque

def canFinish(numCourses, prerequisites):
    # Create a list to hold the number of prerequisites (indegree) for each course.
    indegree = [0] * numCourses
    # Create an adjacency list for each course.
    graph = {i: [] for i in range(numCourses)}

    # Build the graph and update indegree for each course.
    for course, prereq in prerequisites:
        graph[prereq].append(course)
        indegree[course] += 1

    # Initialize a queue with courses that have no prerequisites.
    queue = deque([i for i in range(numCourses) if indegree[i] == 0])

    # Counter for courses that can be completed.
    visited = 0

    while queue:
        course = queue.popleft() # Remove course with no remaining prerequisites.
        visited += 1 # Increment count of completed courses.
        # For each neighbor course, reduce its indegree.
        for neighbor in graph[course]:
            indegree[neighbor] -= 1 # Remove dependency.
```



```python
            # If no more prerequisites, add to queue.
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    # If we processed all courses, it's possible to finish.
    return visited == numCourses

# Example usage:
print(canFinish(2, [[1,0]])) # Expected output: True
print(canFinish(2, [[1,0],[0,1]])) # Expected output: False
```

Python

```python
from enum import Enum


class State(Enum):
    INIT = 0
    VISITING = 1
    VISITED = 2


class Solution:
    def canFinish(self, numCourses: int, prerequisites: list[list[int]]) -> bool:
        graph = [[] for _ in range(numCourses)]
        states = [State.INIT] * numCourses

        for v, u in prerequisites:
            graph[u].append(v)

        def hasCycle(u: int) -> bool:
            if states[u] == State.VISITING:
                return True
            if states[u] == State.VISITED:
                return False
            states[u] = State.VISITING
            if any(hasCycle(v) for v in graph[u]):
                return True

            states[u] = State.VISITED
            return False

        return not any(hasCycle(i) for i in range(numCourses))
```

Python

```python
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        q = [[] for _ in range(numCourses)]
        indeg = [0] * numCourses
        for a, b in prerequisites:
            g[b].append(a)
            indeg[a] += 1
        q = [i for i, x in enumerate(indeg) if x == 0]
        for i in q:
            numCourses -= 1
            for j in g[i]:
                indeg[j] -= 1
                if indeg[j] == 0:
                    q.append(j)
        return numCourses == 0
```

Python

```python
'''
>>> from collections import defaultdict

>>> a = defaultdict(int)
>>> a
defaultdict(<type 'int'>, {})
>>>
>>> a['hehehe']
0
>>> a
defaultdict(<type 'int'>, {'hehehe': 0})


>>> a = defaultdict(list)
>>> a
defaultdict(<type 'list'>, {})
>>> a['hehehe']
[]
>>> a
defaultdict(<type 'list'>, {'hehehe': []})
'''
class Solution:
    def canFinish(self, numCourses: int, prerequisites: List[List[int]]) -> bool:
        q = defaultdict(list)
        indeg = [0] * numCourses

        for a, b in prerequisites:
            g[b].append(a)
            indeg[a] += 1
        cnt = 0
        q = deque([i for i, v in enumerate(indeg) if v == 0])
        while q:
            i = q.popleft()
            cnt += 1
            for j in g[i]:
                indeg[j] -= 1
                if indeg[j] == 0:
                    q.append(j)
        return cnt == numCourses
```


[*] DFS — Pattern Solution (stored optimal)

Python

```python
from collections import deque

def canFinish(numCourses, prerequisites):
    # Create a list to hold the number of prerequisites (indegree) for each course.
    indegree = [0] * numCourses
    # Create an adjacency list for each course.
    graph = {i: [] for i in range(numCourses)}

    # Build the graph and update indegree for each course.
    for course, prereq in prerequisites:
        graph[prereq].append(course)
        indegree[course] += 1

    # Initialize a queue with courses that have no prerequisites.
    queue = deque([i for i in range(numCourses) if indegree[i] == 0])

    # Counter for courses that can be completed.
    visited = 0

    while queue:
        course = queue.popleft() # Remove course with no remaining prerequisites.
        visited += 1 # Increment count of completed courses.
        # For each neighbor course, reduce its indegree.
        for neighbor in graph[course]:
            indegree[neighbor] -= 1 # Remove dependency.

            # If no more prerequisites, add to queue.
            if indegree[neighbor] == 0:
                queue.append(neighbor)

    # If we processed all courses, it's possible to finish.
    return visited == numCourses

# Example usage:
print(canFinish(2, [[1,0]])) # Expected output: True
print(canFinish(2, [[1,0],[0,1]])) # Expected output: False
```

#210 MEDIUM

Course Schedule II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

DFS · BFS · Graph · Topological Sort

List: Grind 169

Problem:

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you must take course bi first if you want to take course ai.

- For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.

Example 1:

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take course 1 you should have
finished course 0. So the correct course order is [0,1].
```

Example 2:

```
Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: [0,2,1,3]
Explanation: There are a total of 4 courses to take. To take course 3 you should have
finished both courses 1 and 2. Both courses 1 and 2 should be taken after you finished course
0.
So one correct course order is [0,1,2,3]. Another correct ordering is [0,2,1,3].
```

Example 3:

```
Input: numCourses = 1, prerequisites = []
Output: [0]
```

Constraints:

- 1 <= numCourses <= 2000
- 0 <= prerequisites.length <= numCourses * (numCourses - 1)
- prerequisites[i].length == 2
- 0 <= ai, bi < numCourses
- ai != bi
- All the pairs [ai, bi] are distinct.

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findOrder(self, numCourses, prerequisites):
        # Brute Force O(V+E) - DFS topological sort
        from collections import defaultdict
        graph = defaultdict(list)
        for a, b in prerequisites:
            graph[b].append(a)
        state = [0] * numCourses
        result = []
        def dfs(node):
            if state[node] == 1: return False # cycle
            if state[node] == 2: return True
            state[node] = 1
            for nei in graph[node]:
                if not dfs(nei): return False
            state[node] = 2
            result.append(node)
            return True
        for c in range(numCourses):
            if not dfs(c): return []
        return result[::-1]
```

Python

Python

```python
# Python solution using BFS (Kahn's Algorithm)
from collections import deque

def findOrder(numCourses, prerequisites):
    # Build the graph and the in-degree array
    in_degree = [0] * numCourses
    graph = {i: [] for i in range(numCourses)}
    for course, pre in prerequisites:
        graph[pre].append(course)
        in_degree[course] += 1

    # Initialize the queue with all courses having zero in-degree
    queue = deque([i for i in range(numCourses) if in_degree[i] == 0])
    order = []

    # Process courses
    while queue:
        course = queue.popleft()
        order.append(course)
        for neighbor in graph[course]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # If the ordering includes all courses, return it; otherwise, return an empty list

    return order if len(order) == numCourses else []

# Example usage:
# print(findOrder(4, [[1,0],[2,0],[3,1],[3,2]]))
```

Python

```python
from enum import Enum


class State(Enum):
    INIT = 0
    VISITING = 1
    VISITED = 2


class Solution:
    def findOrder(
        self,
        numCourses: int,
        prerequisites: list[list[int]],
    ) -> list[int]:
        ans = []
        graph = [[] for _ in range(numCourses)]
        states = [State.INIT] * numCourses

        for v, u in prerequisites:
          graph[u].append(v)

        def hasCycle(u: int) -> bool:
          if states[u] == State.VISITING:
            return True
```



```python
        if states[u] == State.VISITED:
          return False
        states[u] = State.VISITING
        if any(hasCycle(v) for v in graph[u]):
          return True
        states[u] = State.VISITED
        ans.append(u)
        return False

    if any(hasCycle(i) for i in range(numCourses)):
      return []

    return ans[::-1]
```

Python

```python
class Solution:
    def findOrder(
        self,
        numCourses: int,
        prerequisites: list[list[int]],
    ) -> list[int]:
        ans = []
        graph = [[] for _ in range(numCourses)]
        inDegrees = [0] * numCourses
        q = collections.deque()

        # Build the graph.
        for v, u in prerequisites:
          graph[u].append(v)
          inDegrees[v] += 1

        # Perform topological sorting.
        q = collections.deque([i for i, d in enumerate(inDegrees) if d == 0])

        while q:
          u = q.popleft()
          ans.append(u)
          for v in graph[u]:
            inDegrees[v] -= 1
            if inDegrees[v] == 0:

              q.append(v)

        return ans if len(ans) == numCourses else []
```

Python


```python
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:
        g = defaultdict(list)
        indeg = [0] * numCourses
        for a, b in prerequisites:
            g[b].append(a)
            indeg[a] += 1
        ans = []
        q = deque([i for i, x in enumerate(indeg) if x == 0])
        while q:
            i = q.popleft()
            ans.append(i)
            for j in g[i]:
                indeg[j] -= 1
                if indeg[j] == 0:
                    q.append(j)
        return ans if len(ans) == numCourses else []
```

Python

```python
class Solution:
    def findOrder(self, numCourses: int, prerequisites: List[List[int]]) -> List[int]:
        g = defaultdict(list)
        indeg = [0] * numCourses
        for a, b in prerequisites:
            g[b].append(a)
            indeg[a] += 1
        q = deque([i for i, v in enumerate(indeg) if v == 0])
        ans = [] # added from previous question
        while q:
            i = q.popleft()
            ans.append(i)
            # assumption is only one path
            # so in question "You may assume that there are no duplicate edges in the input prerequisites."
            for j in g[i]:
                indeg[j] -= 1
                if indeg[j] == 0:
                    q.append(j)
        return ans if len(ans) == numCourses else []
```

[*] DFS — Pattern Solution (stored optimal)

Python

```python
# Python solution using BFS (Kahn's Algorithm)
from collections import deque

def findOrder(numCourses, prerequisites):
    # Build the graph and the in-degree array
    in_degree = [0] * numCourses
    graph = {i: [] for i in range(numCourses)}
    for course, pre in prerequisites:
        graph[pre].append(course)
        in_degree[course] += 1

    # Initialize the queue with all courses having zero in-degree
    queue = deque([i for i in range(numCourses) if in_degree[i] == 0])
    order = []

    # Process courses
    while queue:
        course = queue.popleft()
        order.append(course)
        for neighbor in graph[course]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # If the ordering includes all courses, return it; otherwise, return an empty list

    return order if len(order) == numCourses else []

# Example usage:
# print(findOrder(4, [[1,0],[2,0],[3,1],[3,2]]))
```

#261 MEDIUM

Graph Valid Tree

LeetCodeCX · SimplyLeet · WalkCC · LeetCode · Editorial

DFS · BFS · Union Find · Graph

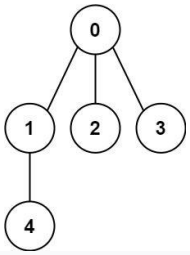
Lists: Grind 169 | Premium 98

Problem:

You have a graph of n nodes labeled from 0 to n - 1. You are given an integer n and a list of edges where edges[i] = [ai, bi] indicates that there is an undirected edge between nodes ai and bi in the graph.

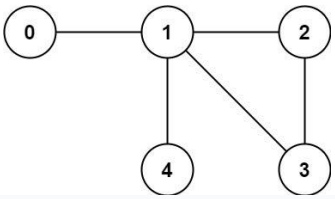
Return true if the edges of the given graph make up a valid tree, and false otherwise.

Example 1:




```
Input: n = 5, edges = [[0,1],[0,2],[0,3],[1,4]]
Output: true
```

Example 2:




```
Input: n = 5, edges = [[0,1],[1,2],[2,3],[1,3],[1,4]]
Output: false
```

Constraints:

- 1 <= n <= 2000
- 0 <= edges.length <= 5000
- edges[i].length == 2

- 0 <= ai, bi < n
- ai != bi
- There are no self-loops or repeated edges.

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def validTree(self, n, edges):
        # Brute Force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python

```python
# Python solution with DFS

def validTree(n, edges):
    # If the number of edges is not exactly n - 1, it cannot be a tree
    if len(edges) != n - 1:
        return False

    # Build the graph as an adjacency list
    graph = {i: [] for i in range(n)}
    for a, b in edges:
        graph[a].append(b)
        graph[b].append(a)

    visited = set()

    # DFS helper function that accepts current node and its parent node
    def dfs(node, parent):
        # Add the current node to visited
        visited.add(node)
        # Traverse all the neighboring nodes
        for neighbor in graph[node]:
            # Skip the neighbor if it is the parent to avoid trivial cycle detection
            if neighbor == parent:
                continue
            # If neighbor is already visited, a cycle is detected
```



```python
            if neighbor in visited:
                return False
            # Recursive backtrack, if cycle is found in recursion, propagate False upward
            if not dfs(neighbor, node):
                return False
        return True

    # Start DFS from node 0, with parent set to -1 (non-existent)
    if not dfs(0, -1):
        return False

    # Check if all nodes were visited (i.e., the graph is connected)
    return len(visited) == n

# Example usage:
print(validTree(5, [[0, 1], [0, 2], [0, 3], [1, 4]])) # Expected output: True
print(validTree(5, [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]])) # Expected output: False
```

Python

```python
class Solution:
  def validTree(self, n: int, edges: list[list[int]]) -> bool:
    if n == 0 or len(edges) != n - 1:
      return False

    graph = [[] for _ in range(n)]
    q = collections.deque([0])
    seen = {0}

    for u, v in edges:
      graph[u].append(v)
      graph[v].append(u)

    while q:
      u = q.popleft()
      for v in graph[u]:
        if v not in seen:
          q.append(v)
          seen.add(v)

    return len(seen) == n
```

Python


```python
class UnionFind:
    def __init__(self, n: int):
        self.count = n
        self.id = list(range(n))
        self.rank = [0] * n

    def unionByRank(self, u: int, v: int) -> None:
        i = self._find(u)
        j = self._find(v)
        if i == j:
            return
        if self.rank[i] < self.rank[j]:
            self.id[i] = j
        elif self.rank[i] > self.rank[j]:
            self.id[j] = i
        else:
            self.id[i] = j
            self.rank[j] += 1
        self.count -= 1

    def _find(self, u: int) -> int:
        if self.id[u] != u:
            self.id[u] = self._find(self.id[u])
        return self.id[u]


class Solution:
    def validTree(self, n: int, edges: list[list[int]]) -> bool:
        if n == 0 or len(edges) != n - 1:
            return False

        uf = UnionFind(n)

        for u, v in edges:
            uf.unionByRank(u, v)

        return uf.count == 1
```

Python

```python
class Solution:
    def validTree(self, n: int, edges: List[List[int]]) -> bool:
        def find(x: int) -> int:
            if p[x] != x:
                p[x] = find(p[x])
            return p[x]

        p = list(range(n))
        for a, b in edges:
            pa, pb = find(a), find(b)
            if pa == pb:
                return False
            p[pa] = pb
            n -= 1
        return n == 1
```

Python


```python
class Solution:
    def validTree(self, n: int, edges: List[List[int]]) -> bool:
        def find(x):
            if p[x] != x:
                p[x] = find(p[x])
            return p[x]

        p = list(range(n))
        for a, b in edges:
            if find(a) == find(b):
                return False
            p[find(a)] = find(b)
            n -= 1
        return n == 1
```

[*] DFS — Pattern Solution (matched from community answers)

Python

```python
# Python solution with DFS

def validTree(n, edges):
    # If the number of edges is not exactly n - 1, it cannot be a tree
    if len(edges) != n - 1:
        return False

    # Build the graph as an adjacency list
    graph = {i: [] for i in range(n)}
    for a, b in edges:
        graph[a].append(b)
        graph[b].append(a)

    visited = set()

    # DFS helper function that accepts current node and its parent node
    def dfs(node, parent):
        # Add the current node to visited
        visited.add(node)
        # Traverse all the neighboring nodes
        for neighbor in graph[node]:
            # Skip the neighbor if it is the parent to avoid trivial cycle detection
            if neighbor == parent:
                continue
            # If neighbor is already visited, a cycle is detected
```

```
            if neighbor in visited:
                return False
            # Recurse deeper; if cycle is found in recursion, propagate False upward
            if not dfs(neighbor, node):
                return False
        return True

    # Start DFS from node 0, with parent set to -1 (non-existent)
    if not dfs(0, -1):
        return False

    # Check if all nodes were visited (i.e., the graph is connected)
    return len(visited) == n

# Example usage:
print(validTree(5, [[0, 1], [0, 2], [0, 3], [1, 4]])) # Expected output: True
print(validTree(5, [[0, 1], [1, 2], [2, 3], [1, 3], [1, 4]])) # Expected output: False
```

#310 MEDIUM

Minimum Height Trees

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

DFS · BFS · Graph · Topological Sort

Lists: Grind 169

Problem:

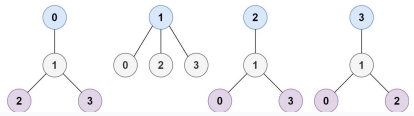
A tree is an undirected graph in which any two vertices are connected by exactly one path. In other words, any connected graph without simple cycles is a tree.

Given a tree of n nodes labelled from 0 to n - 1, and an array of n - 1 edges where edges[i] = [ai, bi] indicates that there is an undirected edge between the two nodes ai and bi in the tree, you can choose any node of the tree as the root. When you select a node x as the root, the result tree has height h. Among all possible rooted trees, those with minimum height (i.e. min(h)) are called minimum height trees (MHTs).

Return a list of all MHTs' root labels. You can return the answer in any order.

The height of a rooted tree is the number of edges on the longest downward path between the root and a leaf.

Example 1:

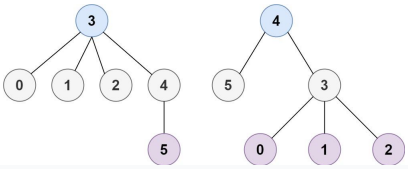



```
Input: n = 4, edges = [[1,0],[1,2],[1,3]]
Output: [1]
Explanation: As shown, the height of the tree is 1 when the root is the node with label 1
```



```
which is the only MHT.
```

Example 2:




```
Input: n = 6, edges = [[3,0],[3,1],[3,2],[3,4],[5,4]]
Output: [3,4]
```

Constraints:

- 1 <= n <= 2 * 104
- edges.length == n - 1
- 0 <= ai, bi < n
- ai != bi
- All the pairs (ai, bi) are distinct.
- The given input is guaranteed to be a tree and there will be no repeated edges.

Python

Python


```
# Python solution with line-by-line comments
class Solution:
    def findMinHeightTrees(self, n: int, edges: List[List[int]]) -> List[int]:
        # Special case: if there's only one node, return it as the only MHT root.
        if n == 1:
            return [0]

        # Build the graph using an adjacency list with sets for neighbors.
        graph = [set() for _ in range(n)]
        for u, v in edges:
            graph[u].add(v)
            graph[v].add(u)

        # Initialize the first layer of leaves (nodes with one neighbor).
        leaves = [i for i in range(n) if len(graph[i]) == 1]

        # Remove leaves layer by layer until at most 2 nodes remain.
        remaining_nodes = n
        while remaining_nodes > 2:
            remaining_nodes -= len(leaves)
            new_leaves = []
            # Remove current leaves.
            for leaf in leaves:
                # Since it's a leaf, there is only one neighbor.
                neighbor = graph[leaf].pop()

                # Remove the connection from the neighbor to the leaf.
                graph[neighbor].remove(leaf)
                # If the neighbor becomes a leaf, add it to the new leaves list.
                if len(graph[neighbor]) == 1:
                    new_leaves.append(neighbor)
            leaves = new_leaves

        # The remaining nodes are the roots of the MHTs.
        return leaves
```

Python


```
class Solution:
    def findMinHeightTrees(self, n: int, edges: list[list[int]]) -> list[int]:
      if n == 1 or not edges:
        return [0]

      ans = []
      graph = collections.defaultdict(set)

      for u, v in edges:
        graph[u].add(v)
        graph[v].add(u)

      for label, children in graph.items():
        if len(children) == 1:
          ans.append(label)

      while n > 2:
        n -= len(ans)
        nextLeaves = []
        for leaf in ans:
          u = next(iter(graph[leaf]))
          graph[u].remove(leaf)
          if len(graph[u]) == 1:
            nextLeaves.append(u)
        ans = nextLeaves

      return ans
```

Python

```
class Solution:
    def findMinHeightTrees(self, n: int, edges: List[List[int]]) -> List[int]:
        if n == 1:
            return [0]
        g = [[] for _ in range(n)]
        degree = [0] * n
        for a, b in edges:
            g[a].append(b)
            g[b].append(a)
            degree[a] += 1
            degree[b] += 1
        q = deque(i for i in range(n) if degree[i] == 1)
        ans = []
        while q:
            ans.clear()
            for _ in range(len(q)):
                a = q.popleft()
                ans.append(a)
                for b in g[a]:
                    degree[b] -= 1
                    if degree[b] == 1:
                        q.append(b)
        return ans
```

Python


```
from typing import List, Set
from collections import defaultdict


class Solution:
    def findMinHeightTrees(self, n: int, edges: List[List[int]]) -> List[int]:
        if n == 1:
            return [0]

        graph = defaultdict(set)
        for a, b in edges:
            graph[a].add(b)
            graph[b].add(a)

        # just check neighbours[] size, not indgree[] for each node
        leaves = [i for i in range(n) if len(graph[i]) == 1]

        while n > 2:
            n -= len(leaves)
            new_leaves = []
            for leaf in leaves:
                # Since it is only one in basket pop(), because it's a leaf node
                neighbor = graph[leaf].pop()
                graph[neighbor].remove(leaf)
                if len(graph[neighbor]) == 1:
                    new_leaves.append(neighbor)
            leaves = new_leaves

        return leaves

##############

class Solution:
    def findMinHeightTrees(self, n: int, edges: List[List[int]]) -> List[int]:
        if n == 1:
            return [0]
        q = defaultdict(list)
        degree = [0] * n
        for a, b in edges:
            q[a].append(b)
            q[b].append(a)
            degree[a] += 1 # not needed, just len(q[a]) is good enough, will update it in next round
updating
            degree[b] += 1
        q = deque()
        for i in range(n):
            if degree[i] == 1:
                q.append(i)
        ans = []
        while q:
            n = len(q)
```



```
            ans.clear()
            for _ in range(n):
                a = q.popleft()
                ans.append(a)
                for b in g[a]:
                    degree[b] -= 1
                    if degree[b] == 1: # final round only 2 left (a,b), then degree[b] here will be 0, and
no more node enqueue
                        q.append(b)
        return ans
```

[*] DFS — Pattern Solution (stored optimal)

Python

```
# Python solution with line-by-line comments
class Solution:
    def findMinHeightTrees(self, n: int, edges: List[List[int]]) -> List[int]:
        # Special case: if there's only one node, return it as the only MHT root.
        if n == 1:
            return [0]

        # Build the graph using an adjacency list with sets for neighbors.
        graph = [set() for _ in range(n)]
        for u, v in edges:
            graph[u].add(v)
            graph[v].add(u)

        # Initialize the first layer of leaves (nodes with one neighbor).
        leaves = [i for i in range(n) if len(graph[i]) == 1]

        # Remove leaves layer by layer until at most 2 nodes remain.
        remaining_nodes = n
        while remaining_nodes > 2:
            remaining_nodes -= len(leaves)
            new_leaves = []
            # Remove current leaves.
            for leaf in leaves:
                # Since it's a leaf, there is only one neighbor.
                neighbor = graph[leaf].pop()

                # Remove the connection from the neighbor to the leaf.
                graph[neighbor].remove(leaf)
                # If the neighbor becomes a leaf, add it to the new leaves list.
                if len(graph[neighbor]) == 1:
                    new_leaves.append(neighbor)
            leaves = new_leaves

        # The remaining nodes are the roots of the MHTs.
        return leaves
```

#323 MEDIUM

Number of Connected Components in an Undirected Graph

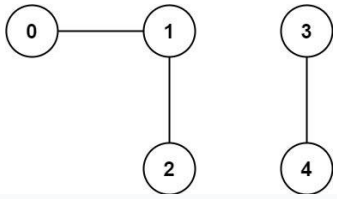
LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

DFS · BFS · Union Find · Graph

Problem:
You have a graph of n nodes. You are given an integer n and an array edges where edges[i] = [ai, bi] indicates that there is an edge between ai and bi in the graph.

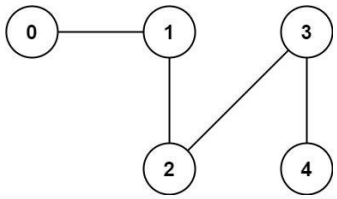
Return the number of connected components in the graph.

Example 1:




```
Input: n = 5, edges = [[0,1],[1,2],[3,4]]
Output: 2
```

Example 2:




```
Input: n = 5, edges = [[0,1],[1,2],[2,3],[3,4]]
Output: 1
```

Constraints:

- 1 <= n <= 2000
- 1 <= edges.length <= 5000
- edges[i].length == 2
- 0 <= ai <= bi < n
- ai != bi
- There are no repeated edges.

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def count_components(self, n, edges):
        # Brute Force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python

```python
# Function to count the number of connected components in an undirected graph.
def count_components(n, edges):
    # Build the adjacency list for the graph.
    graph = {i: [] for i in range(n)}
    for a, b in edges:
        graph[a].append(b)
        graph[b].append(a)

    # Set to keep track of visited nodes.
    visited = set()

    # Helper function to perform DFS traversal starting from a node.
    def dfs(node):
        stack = [node]
        while stack:
            current = stack.pop()
            if current not in visited:
                visited.add(current)
                for neighbor in graph[current]:
                    if neighbor not in visited:
                        stack.append(neighbor)

    # Count connected components by initiating DFS on unvisited nodes.
    count = 0
    for i in range(n):
```



```
        if i not in visited:
            dfs(i)
            count += 1
    return count

# Example usage:
print(count_components(5, [[0,1],[1,2],[3,4]])) # Expected output: 2
```

Python

```python
class Solution:
  def countComponents(self, n: int, edges: list[list[int]]) -> int:
    ans = 0
    graph = [[] for _ in range(n)]
    seen = set()

    for u, v in edges:
      graph[u].append(v)
      graph[v].append(u)

    def bfs(node: int, seen: set[int]) -> None:
      q = collections.deque([node])
      seen.add(node)

      while q:
        u = q.pop()
        for v in graph[u]:
          if v not in seen:
            q.append(v)
            seen.add(v)

    for i in range(n):
      if i not in seen:
        bfs(i, seen)
        ans += 1


    return ans
```

Python


```
class UnionFind:
  def __init__(self, n: int):
    self.count = n
    self.id = list(range(n))
    self.rank = [0] * n

  def unionByRank(self, u: int, v: int) -> None:
    i = self._find(u)
    j = self._find(v)
    if i == j:
      return
    if self.rank[i] < self.rank[j]:
      self.id[i] = j
    elif self.rank[i] > self.rank[j]:
      self.id[j] = i
    else:
      self.id[i] = j
      self.rank[j] += 1
    self.count -= 1

  def _find(self, u: int) -> int:
    if self.id[u] != u:
      self.id[u] = self._find(self.id[u])
    return self.id[u]


class Solution:
  def countComponents(self, n: int, edges: list[list[int]]) -> int:
    uf = UnionFind(n)

    for u, v in edges:
      uf.unionByRank(u, v)

    return uf.count
```

Python

```python
class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        def dfs(i: int) -> int:
            if i in vis:
                return 0
            vis.add(i)
            for j in g[i]:
                dfs(j)
            return 1

        g = [[] for _ in range(n)]
        for a, b in edges:
            g[a].append(b)
            g[b].append(a)
        vis = set()
        return sum(dfs(i) for i in range(n))
```

Python


```
class UnionFind:
    def __init__(self, n):
        self.p = list(range(n))
        self.size = [1] * n

    def find(self, x):
        if self.p[x] != x:
            self.p[x] = self.find(self.p[x])
        return self.p[x]

    def union(self, a, b):
        pa, pb = self.find(a), self.find(b)
        if pa == pb:
            return False
        if self.size[pa] > self.size[pb]:
            self.p[pb] = pa
            self.size[pa] += self.size[pb]
        else:
            self.p[pa] = pb
            self.size[pb] += self.size[pa]
        return True


class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:

        uf = UnionFind(n)
        for a, b in edges:
            s -= uf.union(a, b)
        return n
```

Python

```
'''
example-2: [[0, 1], [1, 2], [2, 3], [3, 4]]

for loop

[0, 1]
    0 parent set to 1
    1 parent set to 1

[1, 2]
    1 parent set to 2
    2 parent set to 2

[2, 3]
    2 parent set to 3
    3 parent set to 3

final:
    0 parent set to 1
    1 parent set to 2
    2 parent set to 3
    3 parent set to 3

===> so filter for 'i == find(i)'
'''
```



```
class Solution:
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        def find(x):
            if p[x] != x:
                p[x] = find(p[x])
            return p[x]

        p = list(range(n))
        for a, b in edges:
            p[find(a)] = find(b)
        return sum(i == find(i) for i in range(n)) # or len(set(parents))

###########

class Solution: # more literal
    def countComponents(self, n: int, edges: List[List[int]]) -> int:
        # Initialize parent array for union-find
        parent = [i for i in range(n)]

        # Perform union-find on each edge
        for u, v in edges:
            self.union(u, v, parent)

        # Count the number of distinct parents (connected components)
        count = len(set(self.find(x, parent) for x in range(n)))

        return count

    def union(self, x: int, y: int, parent: List[int]) -> None:
        rootX = self.find(x, parent)
        rootY = self.find(y, parent)
        if rootX != rootY:
            parent[rootX] = rootY

    def find(self, x: int, parent: List[int]) -> int:
        if parent[x] != x:
            parent[x] = self.find(parent[x], parent)
        return parent[x]
```

[*] DFS — Pattern Solution (matched from community answers)

Python

```python
# Function to count the number of connected components in an undirected graph.
def count_components(n, edges):
    # Build the adjacency list for the graph.
    graph = {i: [] for i in range(n)}
    for a, b in edges:
        graph[a].append(b)
        graph[b].append(a)

    # Set to keep track of visited nodes.
    visited = set()

    # Helper function to perform DFS traversal starting from a node.
    def dfs(node):
        stack = [node]
        while stack:
            current = stack.pop()
            if current not in visited:
                visited.add(current)
                for neighbor in graph[current]:
                    if neighbor not in visited:
                        stack.append(neighbor)

    # Count connected components by initiating DFS on unvisited nodes.
    count = 0
    for i in range(n):

        if i not in visited:
            dfs(i)
            count += 1
    return count

# Example usage:
print(count_components(5, [[0,1],[1,2],[3,4]])) # Expected output: 2
```

#417 MEDIUM

Pacific Atlantic Water Flow

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · DFS · BFS · Matrix

Lists: Grind 169

Problem:

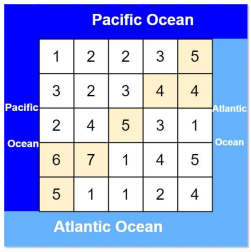
There is an m x n rectangular island that borders both the Pacific Ocean and Atlantic Ocean. The Pacific Ocean touches the island's left and top edges, and the Atlantic Ocean touches the island's right and bottom edges.

The island is partitioned into a grid of square cells. You are given an m x n integer matrix heights where heights[r][c] represents the height above sea level of the cell at coordinate (r, c).

The island receives a lot of rain, and the rain water can flow to neighboring cells directly north, south, east, and west if the neighboring cell's height is less than or equal to the current cell's height. Water can flow from any cell adjacent to an ocean into the ocean.

Return a 2D list of grid coordinates result where result[i] = [ri, ci] denotes that rain water can flow from cell (ri, ci) to both the Pacific and Atlantic oceans.

Example 1:




```
Input: heights = [[1,2,2,3,5],[3,2,3,4,4],[2,4,5,3,1],[6,7,1,4,5],[5,1,1,2,4]]
Output: [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]
Explanation: The following cells can flow to the Pacific and Atlantic oceans, as shown below:
 [0,4]: [0,4] -> Pacific Ocean
   [0,4] -> Atlantic Ocean
 [1,3]: [1,3] -> [0,3] -> Pacific Ocean
   [1,3] -> [1,4] -> Atlantic Ocean
 [1,4]: [1,4] -> [1,3] -> [0,3] -> Pacific Ocean
   [1,4] -> Atlantic Ocean
 [2,2]: [2,2] -> [1,2] -> [0,2] -> Pacific Ocean
   [2,2] -> [2,3] -> [2,4] -> Atlantic Ocean
 [3,0]: [3,0] -> Pacific Ocean
   [3,0] -> [4,0] -> Atlantic Ocean
 [3,1]: [3,1] -> [3,0] -> Pacific Ocean
   [3,1] -> [4,1] -> Atlantic Ocean
 [4,0]: [4,0] -> Pacific Ocean
   [4,0] -> Atlantic Ocean
 Note that there are other possible paths for these cells to flow to the Pacific and Atlantic
 oceans.
```

Example 2:

```
Input: heights = [[1]]
Output: [[0,0]]
Explanation: The water can flow from the only cell to the Pacific and Atlantic oceans.
```

Constraints:

- m == heights.length

- n == heights[r].length
- 1 <= m, n <= 200
- 0 <= heights[r][c] <= 105

Python

Python

```python
# Python solution for Pacific Atlantic Water Flow

class Solution:
    def pacificAtlantic(self, heights: List[List[int]]) -> List[List[int]]:
        if not heights or not heights[0]:
            return []

        rows, cols = len(heights), len(heights[0])
        pacific_reachable = [[False for _ in range(cols)] for _ in range(rows)]
        atlantic_reachable = [[False for _ in range(cols)] for _ in range(rows)]

        # Directions: up, down, left, right
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        # DFS function to mark reachable cells
        def dfs(r, c, reachable):
            reachable[r][c] = True
            for dr, dc in directions:
                new_r, new_c = r + dr, c + dc
                # Check bounds and conditions to flow water
                if (0 <= new_r < rows and 0 <= new_c < cols and
                    not reachable[new_r][new_c] and
                    heights[new_r][new_c] >= heights[r][c]):
                    dfs(new_r, new_c, reachable)

        # DFS from Pacific ocean borders (top row and left column)
        for i in range(rows):
            dfs(i, 0, pacific_reachable)
        for j in range(cols):
            dfs(0, j, pacific_reachable)

        # DFS from Atlantic ocean borders (bottom row and right column)
        for i in range(rows):
            dfs(i, cols - 1, atlantic_reachable)
        for j in range(cols):
            dfs(rows - 1, j, atlantic_reachable)

        # Collect cells that can reach both oceans
        result = []
        for r in range(rows):
            for c in range(cols):
                if pacific_reachable[r][c] and atlantic_reachable[r][c]:
                    result.append([r, c])
        return result
```

Python


```python
class Solution:
  def pacificAtlantic(self, heights: list[list[int]]) -> list[list[int]]:
    m = len(heights)
    n = len(heights[0])
    seenP = [[False] * n for _ in range(m)]
    seenA = [[False] * n for _ in range(m)]

    def dfs(i: int, j: int, h: int, seen: list[list[bool]]) -> None:
      if i < 0 or i == m or j < 0 or j == n:
        return
      if seen[i][j] or heights[i][j] < h:
        return

      seen[i][j] = True
      dfs(i + 1, j, heights[i][j], seen)
      dfs(i - 1, j, heights[i][j], seen)
      dfs(i, j + 1, heights[i][j], seen)
      dfs(i, j - 1, heights[i][j], seen)

    for i in range(m):
      dfs(i, 0, 0, seenP)
      dfs(i, n - 1, 0, seenA)

    for j in range(n):
      dfs(0, j, 0, seenP)

      dfs(m - 1, j, 0, seenA)

    return [[i, j]
            for i in range(m)
            for j in range(n)
            if seenP[i][j] and seenA[i][j]]
```

Python


```python
class Solution:
    def pacificAtlantic(self, heights: List[List[int]]) -> List[List[int]]:
        def bfs(q, vis):
            while q:
                for _ in range(len(q)):
                    i, j = q.popleft()
                    for a, b in [(0, -1), (0, 1), (1, 0), (-1, 0)]:
                        x, y = i + a, j + b
                        if (
                            0 <= x < m
                            and 0 <= y < n
                            and (x, y) not in vis
                            and heights[x][y] >= heights[i][j]
                        ):
                            vis.add((x, y))
                            q.append((x, y))

        m, n = len(heights), len(heights[0])
        vis1, vis2 = set(), set()
        q1 = deque()
        q2 = deque()
        for i in range(m):
            for j in range(n):
                if i == 0 or j == 0:
                    vis1.add((i, j))

                    q1.append((i, j))
                if i == m - 1 or j == n - 1:
                    vis2.add((i, j))
                    q2.append((i, j))
        bfs(q1, vis1)
        bfs(q2, vis2)
        return [
            (i, j)
            for i in range(m)
            for j in range(n)
            if (i, j) in vis1 and (i, j) in vis2
        ]
```

[*] DFS — Pattern Solution (matched from community answers)

Python


```python
# Python solution for Pacific Atlantic Water Flow

class Solution:
    def pacificAtlantic(self, heights: List[List[int]]) -> List[List[int]]:
        if not heights or not heights[0]:
            return []

        rows, cols = len(heights), len(heights[0])
        pacific_reachable = [[False for _ in range(cols)] for _ in range(rows)]
        atlantic_reachable = [[False for _ in range(cols)] for _ in range(rows)]

        # Directions: up, down, left, right
        directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

        # DFS function to mark reachable cells
        def dfs(r, c, reachable):
            reachable[r][c] = True
            for dr, dc in directions:
                new_r, new_c = r + dr, c + dc
                # Check bounds and conditions to flow water
                if (0 <= new_r < rows and 0 <= new_c < cols and
                    not reachable[new_r][new_c] and
                    heights[new_r][new_c] >= heights[r][c]):
                    dfs(new_r, new_c, reachable)

        # DFS from Pacific ocean borders (top row and left column)
        for i in range(rows):
            dfs(i, 0, pacific_reachable)
        for j in range(cols):
            dfs(0, j, pacific_reachable)

        # DFS from Atlantic ocean borders (bottom row and right column)
        for i in range(rows):
            dfs(i, cols - 1, atlantic_reachable)
        for j in range(cols):
            dfs(rows - 1, j, atlantic_reachable)

        # Collect cells that can reach both oceans
        result = []
        for r in range(rows):
            for c in range(cols):
                if pacific_reachable[r][c] and atlantic_reachable[r][c]:
                    result.append([r, c])
        return result
```

#490 MEDIUM

The Maze

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · DFS · BFS · Matrix

Lists: Premium 98

Problem:

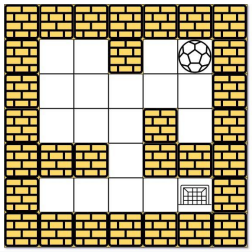
There is a ball in a maze with empty spaces (represented as 0) and walls (represented as 1). The ball can go through the empty spaces by rolling up, down, left or right, but it won't stop rolling until hitting a wall.

When the ball stops, it could choose the next direction.

Given the m x n maze, the ball's start position and the destination, where start = [startrow, startcol] and destination = [destinationrow, destinationcol], return true if the ball can stop at the destination, otherwise return false.

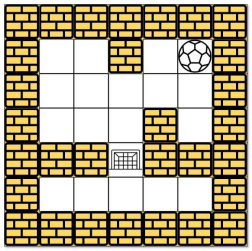
You may assume that the borders of the maze are all walls (see examples).

Example 1:



```
Input: maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4],
destination = [4,4]
Output: true
Explanation: One possible way is : left -> down -> left -> down -> right -> down -> right.
```

Example 2:



```
Input: maze = [[0,0,1,0,0],[0,0,0,0,0],[0,0,0,1,0],[1,1,0,1,1],[0,0,0,0,0]], start = [0,4],
destination = [3,2]
Output: false
Explanation: There is no way for the ball to stop at the destination. Notice that you can
pass through the destination but you cannot stop there.
```

Example 3:

```
Input: maze = [[0,0,0,0,0],[1,1,0,0,1],[0,0,0,0,0],[0,1,0,0,1],[0,1,0,0,0]], start = [4,3],
destination = [0,1]
Output: false
```

Constraints:

- m == maze.length
- n == maze[i].length
- 1 <= m, n <= 100
- maze[i][j] is 0 or 1.
- start.length == 2
- destination.length == 2
- 0 <= startrow, destinationrow < m
- 0 <= startcol, destinationcol < n
- Both the ball and the destination exist in an empty space, and they will not be in the same position initially.
- The maze contains at least 2 empty spaces.

>> Brute Force [DFS] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def hasPath(self, maze, start, destination):
        # Brute Force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python

```python
# Function to determine if there is a valid path in the maze from start to destination
def hasPath(maze, start, destination):
    from collections import deque
    rows, cols = len(maze), len(maze[0])
    # Create a visited 2D array to keep track of processed cells
    visited = [[False] * cols for _ in range(rows)]
    queue = deque([tuple(start)])
    visited[start[0]][start[1]] = True

    # Directions: up, down, left, right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # Process nodes in BFS fashion
    while queue:
        current = queue.popleft()
        # Check if the current position is the destination
        if [current[0], current[1]] == destination:
            return True
        # Explore all four possible directions
        for dx, dy in directions:
            x, y = current
            # Roll the ball until it hits a wall or boundary
            while 0 <= x + dx < rows and 0 <= y + dy < cols and maze[x + dx][y + dy] == 0:
                x += dx
                y += dy
```



```python
            # If this new stop has not been visited, add it to the queue
            if not visited[x][y]:
                visited[x][y] = True
                queue.append((x, y))
    return False

# Example usage:
maze = [[0,0,1,0,0],
        [0,0,0,0,0],
        [0,0,0,1,0],
        [1,1,0,1,1],
        [0,0,0,0,0]]
start = [0,4]
destination = [4,4]
print(hasPath(maze, start, destination)) # Expected output: True
```

Python

```python
class Solution:
    def hasPath(
        self,
        maze: list[list[int]],
        start: list[int],
        destination: list[int],
    ) -> bool:
        DIRS = ((0, 1), (1, 0), (0, -1), (-1, 0))
        m = len(maze)
        n = len(maze[0])
        q = collections.deque([(start[0], start[1])])
        seen = {(start[0], start[1])}

        def isValid(x: int, y: int) -> bool:
            return 0 <= x < m and 0 <= y < n and maze[x][y] == 0

        while q:
            i, j = q.popleft()
            for dx, dy in DIRS:
                x = i
                y = j
                while isValid(x + dx, y + dy):
                    x += dx
                    y += dy
                if [x, y] == destination:
                    return True
                if (x, y) in seen:
                    continue
                q.append((x, y))
                seen.add((x, y))

        return False
```

Python


```python
class Solution:
    def hasPath(
        self, maze: List[List[int]], start: List[int], destination: List[int]
    ) -> bool:
        def dfs(i, j):
            if vis[i][j]:
                return
            vis[i][j] = True
            if [i, j] == destination:
                return
            for a, b in [(0, -1), (0, 1), (1, 0), (-1, 0)]:
                x, y = i, j
                while 0 <= x + a < m and 0 <= y + b < n and maze[x + a][y + b] == 0:
                    x, y = x + a, y + b
                dfs(x, y)

        m, n = len(maze), len(maze[0])
        vis = [[False] * n for _ in range(m)]
        dfs(start[0], start[1])
        return vis[destination[0]][destination[1]]
```

Python

```python
class Solution:
    def hasPath(
        self, maze: List[List[int]], start: List[int], destination: List[int]
    ) -> bool:
        m, n = len(maze), len(maze[0])
        q = deque([(start)])
        rs, cs = start
        vis = {(rs, cs)}
        while q:
            i, j = q.popleft()
            for a, b in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
                x, y = i, j
                while 0 <= x + a < m and 0 <= y + b < n and maze[x + a][y + b] == 0:
                    x, y = x + a, y + b
                if [x, y] == destination:
                    return True
                if (x, y) not in vis:
                    vis.add((x, y))
                    q.append((x, y))
        return False
```

[*] DFS — Pattern Solution (matched from community answers)

Python


```python
class Solution:
    def hasPath(
        self, maze: List[List[int]], start: List[int], destination: List[int]
    ) -> bool:
        def dfs(i, j):
            if vis[i][j]:
                return
            vis[i][j] = True
            if [i, j] == destination:
                return
            for a, b in [(0, -1), (0, 1), (1, 0), (-1, 0)]:
                x, y = i, j
                while 0 <= x + a < m and 0 <= y + b < n and maze[x + a][y + b] == 0:
                    x, y = x + a, y + b
                dfs(x, y)

        m, n = len(maze), len(maze[0])
        vis = [[False] * n for _ in range(m)]
        dfs(start[0], start[1])
        return vis[destination[0]][destination[1]]
```

#694 MEDIUM

Number of Distinct Islands

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · DFS · BFS · Union Find · Hash Function

Lists: Premium 98

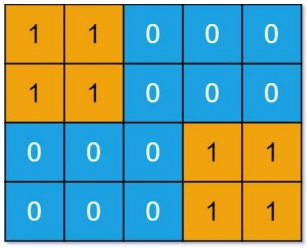
Problem:

You are given an m x n binary matrix grid. An island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

An island is considered to be the same as another if and only if one island can be translated (and not rotated or reflected) to equal the other.

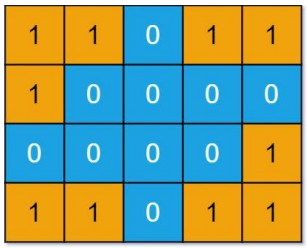
Return the number of distinct islands.

Example 1:




```
Input: grid = [[1,1,0,0,0],[1,1,0,0,0],[0,0,0,1,1],[0,0,0,1,1]]
Output: 1
```

Example 2:




```
Input: grid = [[1,1,0,1,1],[1,0,0,0,0],[0,0,0,0,1],[1,1,0,1,1]]
Output: 3
```


Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 50
- grid[i][j] is either 0 or 1.

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def numDistinctIslands(self, grid):
        # Brute force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python

```python
def numDistinctIslands(grid):
    # Get the number of rows and columns
    rows, cols = len(grid), len(grid[0])
    # Set to store unique island shapes
    distinct_shapes = set()

    # DFS function to explore the island and record its relative shape
    def dfs(r, c, origin_r, origin_c, shape):
        # Check boundaries and if the cell is unvisited land
        if r < 0 or c < 0 or r >= rows or c >= cols or grid[r][c] == 0:
            return
        # Mark the cell as visited by setting it to 0
        grid[r][c] = 0
        # Record the relative position from the origin of this island
        shape.append((r - origin_r, c - origin_c))
        # Explore all 4-directionally adjacent cells
        dfs(r + 1, c, origin_r, origin_c, shape)
        dfs(r - 1, c, origin_r, origin_c, shape)
        dfs(r, c + 1, origin_r, origin_c, shape)
        dfs(r, c - 1, origin_r, origin_c, shape)

    # Iterate over every cell in the grid
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1:
```



```
                shape = []
                dfs(i, j, i, j, shape)
                # Convert the shape list to a tuple of sorted coordinates for canonical form
                distinct_shapes.add(tuple(sorted(shape)))

    # The number of distinct islands is the size of the set
    return len(distinct_shapes)

# Example usage:
grid = [[1,1,0,0,0],
        [1,1,0,0,0],
        [0,0,0,1,1],
        [0,0,0,1,1]]
print(numDistinctIslands(grid)) # Output: 1
```

Python

```python
class Solution:
  def numDistinctIslands(self, grid: list[list[int]]) -> int:
    seen = set()

    def dfs(i: int, j: int, i0: int, j0: int):
      if i < 0 or i == len(grid) or j < 0 or j == len(grid[0]):
        return
      if grid[i][j] == 0 or (i, j) in seen:
        return

      seen.add((i, j))
      island.append((i - i0, j - j0))
      dfs(i + 1, j, i0, j0)
      dfs(i - 1, j, i0, j0)
      dfs(i, j + 1, i0, j0)
      dfs(i, j - 1, i0, j0)

    islands = set() # all the different islands

    for i in range(len(grid)):
      for j in range(len(grid[0])):
        island = []
        dfs(i, j, i, j)
        if island:
          islands.add(frozenset(island))


    return len(islands)
```

Python


```python
class Solution:
    def numDistinctIslands(self, grid: List[List[int]]) -> int:
        def dfs(i: int, j: int, k: int):
            grid[i][j] = 0
            path.append(str(k))
            dirs = (-1, 0, 1, 0, -1)
            for h in range(1, 5):
                x, y = i + dirs[h - 1], j + dirs[h]
                if 0 <= x < m and 0 <= y < n and grid[x][y]:
                    dfs(x, y, h)
            path.append(str(-k))

        paths = set()
        path = []
        m, n = len(grid), len(grid[0])
        for i, row in enumerate(grid):
            for j, x in enumerate(row):
                if x:
                    dfs(i, j, 0)
                    paths.add("".join(path))
                    path.clear()
        return len(paths)
```

Python

```python
class Solution:
    def numDistinctIslands(self, grid: List[List[int]]) -> int:
        def dfs(i: int, j: int, k: int):
            grid[i][j] = 0
            path.append(str(k))
            dirs = (-1, 0, 1, 0, -1)
            for h in range(1, 5):
                x, y = i + dirs[h - 1], j + dirs[h]
                if 0 <= x < m and 0 <= y < n and grid[x][y]:
                    dfs(x, y, h)
            path.append(str(-k))

        paths = set()
        path = []
        m, n = len(grid), len(grid[0])
        for i, row in enumerate(grid):
            for j, x in enumerate(row):
                if x:
                    dfs(i, j, 0)
                    paths.add("".join(path))
                    path.clear()
        return len(paths)
```

[*] DFS — Pattern Solution (matched from community answers)

Python


```python
def numDistinctIslands(grid):
    # Get the number of rows and columns
    rows, cols = len(grid), len(grid[0])
    # Set to store unique island shapes
    distinct_shapes = set()

    # DFS function to explore the island and record its relative shape
    def dfs(r, c, origin_r, origin_c, shape):
        # Check boundaries and if the cell is unvisited land
        if r < 0 or c < 0 or r >= rows or c >= cols or grid[r][c] == 0:
            return
        # Mark the cell as visited by setting it to 0
        grid[r][c] = 0
        # Record the relative position from the origin of this island
        shape.append((r - origin_r, c - origin_c))
        # Explore all 4-directionally adjacent cells
        dfs(r + 1, c, origin_r, origin_c, shape)
        dfs(r - 1, c, origin_r, origin_c, shape)
        dfs(r, c + 1, origin_r, origin_c, shape)
        dfs(r, c - 1, origin_r, origin_c, shape)

    # Iterate over every cell in the grid
    for i in range(rows):
        for j in range(cols):
            if grid[i][j] == 1:
```

```
                shape = []
                dfs(i, j, i, j, shape)
                # Convert the shape list to a tuple of sorted coordinates for canonical form
                distinct_shapes.add(tuple(sorted(shape)))

    # The number of distinct islands is the size of the set
    return len(distinct_shapes)

# Example usage:
grid = [[1,1,0,0,0],
        [1,1,0,0,0],
        [0,0,0,1,1],
        [0,0,0,1,1]]
print(numDistinctIslands(grid)) # Output: 1
```

#695 MEDIUM

Max Area of Island

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · DFS · BFS · Union Find · Matrix

Lists: Denny Zhang

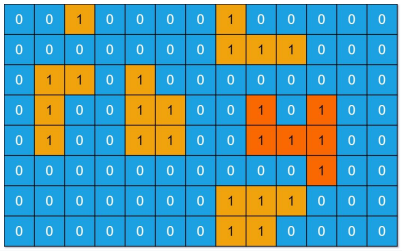
Problem:

You are given an m x n binary matrix grid. An island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The area of an island is the number of cells with a value 1 in the island.

Return the maximum area of an island in grid. If there is no island, return 0.

Example 1:




```
Input: grid = [[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,1,1,0,1,0,0,0,0,0,0,0,0
,0,0],[0,1,0,0,1,1,0,0,1,0,1,0,0],[0,1,0,0,1,1,0,0,1,1,1,0,0],[0,0,0,0,0,0,0,0,0,0,1,0,0],[0,
0,0,0,0,0,0,1,1,1,0,0,0],[0,0,0,0,0,0,0,0,1,1,0,0,0]]
Output: 6
Explanation: The answer is not 11, because the island must be connected 4-directionally.
```

Example 2:

```
Input: grid = [[0,0,0,0,0,0,0,0]]
Output: 0
```

Constraints:

- m == grid.length
- n == grid[i].length
- 1 <= m, n <= 50
- grid[i][j] is either 0 or 1.

>> Brute Force [DFS] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def maxAreaOfIsland(self, grid):
        # Brute Force O(m^2·n^2) - DFS from every unvisited land cell, track size
        m, n = len(grid), len(grid[0])
        visited = set()
        def dfs(r, c):
            if r<0 or r>=m or c<0 or c>=n: return 0
            if (r,c) in visited or grid[r][c]==0: return 0
            visited.add((r,c))
            return 1 + dfs(r+1,c) + dfs(r-1,c) + dfs(r,c+1) + dfs(r,c-1)
        return max(dfs(r,c) for r in range(m) for c in range(n), default=0)
```

Python

Python

```
# Function to compute the maximum area of an island in a grid using DFS
def maxAreaOfIsland(grid):
    # Determine the dimensions of the grid
    rows = len(grid)
    cols = len(grid[0]) if rows else 0

    # Helper function for performing DFS on the grid
    def dfs(r, c):
        # Check if the current cell is out-of-bounds or is water
        if r < 0 or c < 0 or r >= rows or c >= cols or grid[r][c] == 0:
            return 0
        # Mark the current cell as visited by setting it to 0 (water)
        grid[r][c] = 0
        area = 1 # Count the current cell as part of the island
        # Recursively explore all four directions
        area += dfs(r + 1, c)
        area += dfs(r - 1, c)
        area += dfs(r, c + 1)
        area += dfs(r, c - 1)
        return area

    max_area = 0
    # Iterate through each cell in the grid
    for r in range(rows):
        for c in range(cols):

            if grid[r][c] == 1:
                # Update max_area with the area obtained from the DFS
                max_area = max(max_area, dfs(r, c))
    return max_area
```

Python


```
class Solution:
    def maxAreaOfIsland(self, grid: list[list[int]]) -> int:
        def dfs(i: int, j: int) -> int:
            if i < 0 or i >= len(grid) or j < 0 or j >= len(grid[0]):
                return 0
            if grid[i][j] != 1:
                return 0

            grid[i][j] = 2

            return 1 + dfs(i + 1, j) + dfs(i - 1, j) + dfs(i, j + 1) + dfs(i, j - 1)

        return max(dfs(i, j) for i in range(len(grid)) for j in range(len(grid[0])))
```

Python

```
class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        def dfs(i: int, j: int) -> int:
            if grid[i][j] == 0:
                return 0
            ans = 1
            grid[i][j] = 0
            dirs = (-1, 0, 1, 0, -1)
            for a, b in pairwise(dirs):
                x, y = i + a, j + b
                if 0 <= x < m and 0 <= y < n:
                    ans += dfs(x, y)
            return ans

        m, n = len(grid), len(grid[0])
        return max(dfs(i, j) for i in range(m) for j in range(n))
```

Python

```
class Solution:
    def maxAreaOfIsland(self, grid: List[List[int]]) -> int:
        def dfs(i: int, j: int) -> int:
            if grid[i][j] == 0:
                return 0
            ans = 1
            grid[i][j] = 0
            dirs = (-1, 0, 1, 0, -1)
            for a, b in pairwise(dirs):
                x, y = i + a, j + b
                if 0 <= x < m and 0 <= y < n:
                    ans += dfs(x, y)
            return ans

        m, n = len(grid), len(grid[0])
        return max(dfs(i, j) for i in range(m) for j in range(n))
```

[*] DFS — Pattern Solution (matched from community answers)

Python


```
# Function to compute the maximum area of an island in a grid using DFS
def maxAreaOfIsland(grid):
    # Determine the dimensions of the grid
    rows = len(grid)
    cols = len(grid[0]) if rows else 0

    # Helper function for performing DFS on the grid
    def dfs(r, c):
        # Check if the current cell is out-of-bounds or is water
        if r < 0 or c < 0 or r >= rows or c >= cols or grid[r][c] == 0:
            return 0
        # Mark the current cell as visited by setting it to 0 (water)
        grid[r][c] = 0
        area = 1 # Count the current cell as part of the island
        # Recursively explore all four directions
        area += dfs(r + 1, c)
        area += dfs(r - 1, c)
        area += dfs(r, c + 1)
        area += dfs(r, c - 1)
        return area

    max_area = 0
    # Iterate through each cell in the grid
    for r in range(rows):
        for c in range(cols):

            if grid[r][c] == 1:
                # Update max_area with the area obtained from the DFS
                max_area = max(max_area, dfs(r, c))
    return max_area
```

#721 MEDIUM

Accounts Merge

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · String · DFS · BFS · Union Find

Lists: Grind 169

Problem:

Given a list of accounts where each element accounts[i] is a list of strings, where the first element accounts[i][0] is a name, and the rest of the elements are emails representing emails of the account.

Now, we would like to merge these accounts. Two accounts definitely belong to the same person if there is some common email to both accounts. Note that even if two accounts have the same name, they may belong to different people as people could have the same name. A person can have any number of accounts initially, but all of their accounts definitely have the same name.

After merging the accounts, return the accounts in the following format: the first element of each account is the name, and the rest of the elements are emails in sorted order. The accounts themselves can be returned in any order.

Example 1:

```
Input: accounts = [["John","johnsmith@mail.com","john_newyork@mail.com"],["John","johnsmith@m
ail.com","john00@mail.com"],["Mary","mary@mail.com"],["John","johnnybravo@mail.com"]]
```



```
Output: [["John","john00@mail.com","john_newyork@mail.com","johnsmith@mail.com"],["Mary","mar
y@mail.com"],["John","johnnybravo@mail.com"]]
Explanation:
The first and second John's are the same person as they have the common email
"johnsmith@mail.com".
The third John and Mary are different people as none of their email addresses are used by
other accounts.
We could return these lists in any order, for example the answer [['Mary', 'mary@mail.com'],
['John', 'johnnybravo@mail.com'],
['John', 'john00@mail.com', 'john_newyork@mail.com', 'johnsmith@mail.com']] would still be
accepted.
```

Example 2:

```
Input: accounts = [["Gabe","Gabe0@m.co","Gabe3@m.co","Gabe1@m.co"],["Kevin","Kevin3@m.co","Ke
vin5@m.co","Kevin0@m.co"],["Ethan","Ethan5@m.co","Ethan4@m.co","Ethan0@m.co"],["Hanzo","Hanzo
3@m.co","Hanzo1@m.co","Hanzo0@m.co"],["Fern","Fern5@m.co","Fern1@m.co","Fern0@m.co"]]
Output: [["Ethan","Ethan0@m.co","Ethan4@m.co","Ethan5@m.co"],["Gabe","Gabe0@m.co","Gabe1@m.co
","Gabe3@m.co"],["Hanzo","Hanzo0@m.co","Hanzo1@m.co","Hanzo3@m.co"],["Kevin","Kevin0@m.co","K
evin3@m.co","Kevin5@m.co"],["Fern","Fern0@m.co","Fern1@m.co","Fern5@m.co"]]
```

Constraints:

- 1 <= accounts.length <= 1000
- 2 <= accounts[i].length <= 10
- 1 <= accounts[i][j].length <= 30
- accounts[i][0] consists of English letters.
- accounts[i][j] (for j > 0) is a valid email.

>> Brute Force [DFS] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def accountsMerge(self, accounts):
        # Brute Force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python


```
# Python solution for Accounts Merge

# Define the UnionFind class for disjoint set operations
class UnionFind:
    def __init__(self):
        self.parent = {}

    # Find with path compression
    def find(self, x):
        if self.parent.setdefault(x, x) != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    # Union two nodes
    def union(self, x, y):
        self.parent[self.find(x)] = self.find(y)

class Solution:
    def accountsMerge(self, accounts):
        uf = UnionFind()
        email_to_name = {}

        # Iterate over each account and union emails
        for account in accounts:
            name = account[0]

            first_email = account[1]
            for email in account[1:]:
                uf.union(first_email, email)
                email_to_name[email] = name

        # Group emails by their root parent
        groups = {}
        for email in email_to_name:
            parent = uf.find(email)
            groups.setdefault(parent, []).append(email)

        # Return the merged account list with sorted emails
        result = []
        for parent, emails in groups.items():
            result.append([email_to_name[parent]] + sorted(emails))
        return result

# Example usage:
# accounts = [["John","johnsmith@mail.com","john_newyork@mail.com"],
# ["John","johnsmith@mail.com","john00@mail.com"],
# ["Mary","mary@mail.com"],
# ["John","johnnybravo@mail.com"]]
# solution = Solution()
# print(solution.accountsMerge(accounts))
```

Python


```
class Solution:
    def accountsMerge(self, accounts: List[List[str]]) -> List[List[str]]:
        def find(x):
            if p[x] != x:
                p[x] = find(p[x])
            return p[x]

        n = len(accounts)
        p = list(range(n))
        email_id = {}
        for i, account in enumerate(accounts):
            name = account[0]
            for email in account[1:]:
                if email in email_id:
                    p[find(i)] = find(email_id[email])
                else:
                    email_id[email] = i
        mp = defaultdict(set)
        for i, account in enumerate(accounts):
            for email in account[1:]:
                mp[find(i)].add(email)

        ans = []
        for i, emails in mp.items():
            t = [accounts[i][0]]

            t.extend(sorted(emails))
            ans.append(t)
        return ans
```

[*] DFS — Pattern Solution (stored optimal)

Python

```
# Python solution for Accounts Merge

# Define the UnionFind class for disjoint set operations
class UnionFind:
    def __init__(self):
        self.parent = {}

    # Find with path compression
    def find(self, x):
        if self.parent.setdefault(x, x) != x:
            self.parent[x] = self.find(self.parent[x])
        return self.parent[x]

    # Union two nodes
    def union(self, x, y):
        self.parent[self.find(x)] = self.find(y)

class Solution:
    def accountsMerge(self, accounts):
        uf = UnionFind()
        email_to_name = {}

        # Iterate over each account and union emails
        for account in accounts:
            name = account[0]
```

```
            first_email = account[1]
            for email in account[1:]:
                uf.union(first_email, email)
                email_to_name[email] = name

        # Group emails by their root parent
        groups = {}
        for email in email_to_name:
            parent = uf.find(email)
            groups.setdefault(parent, []).append(email)

        # Return the merged account list with sorted emails
        result = []
        for parent, emails in groups.items():
            result.append([email_to_name[parent]] + sorted(emails))
        return result

# Example usage:
# accounts = [["John","johnsmith@mail.com","john_newyork@mail.com"],
# ["John","johnsmith@mail.com","john00@mail.com"],
# ["Mary","mary@mail.com"],
# ["John","johnnybravo@mail.com"]]
# solution = Solution()
# print(solution.accountsMerge(accounts))
```

#785 MEDIUM

Is Graph Bipartite?

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

DFS · BFS · Union Find · Graph

Lists: Denny Zhang

Problem:

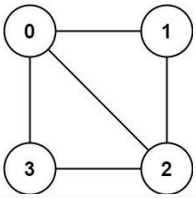
There is an undirected graph with n nodes, where each node is numbered between 0 and n - 1. You are given a 2D array graph, where graph[u] is an array of nodes that node u is adjacent to. More formally, for each v in graph[u], there is an undirected edge between node u and node v. The graph has the following properties:

- There are no self-edges (graph[u] does not contain u).
- There are no parallel edges (graph[u] does not contain duplicate values).
- If v is in graph[u], then u is in graph[v] (the graph is undirected).
- The graph may not be connected, meaning there may be two nodes u and v such that there is no path between them.

A graph is bipartite if the nodes can be partitioned into two independent sets A and B such that every edge in the graph connects a node in set A and a node in set B.

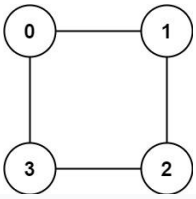
Return true if and only if it is bipartite.

Example 1:




```
Input: graph = [[1,2,3],[0,2],[0,1,3],[0,2]]
Output: false
Explanation: There is no way to partition the nodes into two independent sets such that every
edge connects a node in one and a node in the other.
```

Example 2:




```
Input: graph = [[1,3],[0,2],[1,3],[0,2]]
Output: true
Explanation: We can partition the nodes into two sets: {0, 2} and {1, 3}.
```

Constraints:

- graph.length == n
- 1 <= n <= 100
- 0 <= graph[u].length < n
- 0 <= graph[u][i] <= n - 1

- graph[u] does not contain u.
- All the values of graph[u] are unique.
- If graph[u] contains v, then graph[v] contains u.

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def isBipartite(self, graph):
        # Brute Force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python

```python
# Using Breadth-First Search (BFS)
from collections import deque

def isBipartite(graph):
    n = len(graph)
    # color array: -1 means uncolored, 0 and 1 are the two colors.
    colors = [-1] * n

    # Iterate through each node to handle disconnected components
    for i in range(n):
        if colors[i] == -1:
            # Start BFS from node i with color 0
            queue = deque([i])
            colors[i] = 0
            while queue:
                node = queue.popleft()
                # Check each neighbor of the current node
                for neighbor in graph[node]:
                    if colors[neighbor] == -1:
                        # Assign the opposite color to the neighbor
                        colors[neighbor] = 1 - colors[node]
                        queue.append(neighbor)
                    elif colors[neighbor] == colors[node]:
                        # Found a neighbor with the same color: not bipartite
                        return False

    return True

# Example usage:
if __name__ == '__main__':
    graph_example = [[1,3],[0,2],[1,3],[0,2]]
    print(isBipartite(graph_example)) # Expected output: True
```


Python

```python
from enum import Enum


class Color(Enum):
  WHITE = 0
  RED = 1
  GREEN = 2


class Solution:
  def isBipartite(self, graph: list[list[int]]) -> bool:
    colors = [Color.WHITE] * len(graph)

    for i in range(len(graph)):
      # This node has been colored, so do nothing.
      if colors[i] != Color.WHITE:
        continue
      # Always paint red for a white node.
      colors[i] = Color.RED
      # BFS
      q = collections.deque([i])
      while q:
        for _ in range(len(q)):
          u = q.popleft()
          for v in graph[u]:
            if colors[v] == colors[u]:
              return False
            if colors[v] == Color.WHITE:
              colors[v] = Color.RED if colors[u] == Color.GREEN else Color.GREEN
              q.append(v)

    return True
```

Python

```python
class Solution:
    def isBipartite(self, graph: List[List[int]]) -> bool:
        def dfs(a: int, c: int) -> bool:
            color[a] = c
            for b in graph[a]:
                if color[b] == c or (color[b] == 0 and not dfs(b, -c)):
                    return False
            return True

        n = len(graph)
        color = [0] * n
        for i in range(n):
            if color[i] == 0 and not dfs(i, 1):
                return False
        return True
```

Python


```python
class Solution:
    def isBipartite(self, graph: List[List[int]]) -> bool:
        def dfs(u, c):
            color[u] = c
            for v in graph[u]:
                if not color[v]:
                    if not dfs(v, 3 - c):
                        return False
                elif color[v] == c:
                    return False
            return True

        n = len(graph)
        color = [0] * n
        for i in range(n):
            if not color[i] and not dfs(i, 1):
                return False
        return True
```

[*] DFS — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def isBipartite(self, graph: List[List[int]]) -> bool:
        def dfs(a: int, c: int) -> bool:
            color[a] = c
            for b in graph[a]:
                if color[b] == c or (color[b] == 0 and not dfs(b, -c)):
                    return False
            return True

        n = len(graph)
        color = [0] * n
        for i in range(n):
            if color[i] == 0 and not dfs(i, 1):
                return False
        return True
```

#1236 MEDIUM

Web Crawler

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · DFS · BFS · Interactive

Lists: Premium 98

Problem:

Given a url startUrl and an interface HtmlParser, implement a web crawler to crawl all links that are under the same hostname as startUrl.

Return all urls obtained by your web crawler in any order.

Your crawler should:

- Start from the page: startUrl
- Call HtmlParser.getUrls(url) to get all urls from a webpage of given url.

- Do not crawl the same link twice.
- Explore only the links that are under the same hostname as startUrl.

As shown in the example url above, the hostname is example.org. For simplicity sake, you may assume all urls use http protocol without any port specified. For example, the urls http://leetcode.com/problems and http://leetcode.com/contest are under the same hostname, while urls http://example.org/test and http://example.com/abc are not under the same hostname.

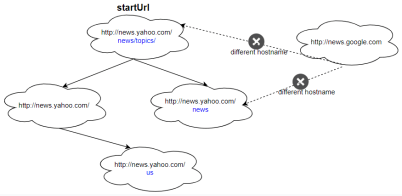
The HtmlParser interface is defined as such:

```
interface HtmlParser {
// Return a list of all urls from a webpage of given url.
public List<String> getUrls(String url);
}
```

Below are two examples explaining the functionality of the problem, for custom testing purposes you'll have three variables urls, edges and startUrl. Notice that you will only have access to startUrl in your code, while urls and edges are not directly accessible to you in code.

Note: Consider the same URL with the trailing slash "/" as a different URL. For example, "http://news.yahoo.com", and "http://news.yahoo.com/" are different urls.

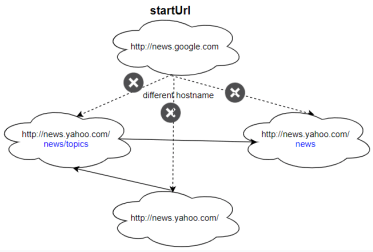
Example 1:




```
Input:
urls = [
```

```
  "http://news.yahoo.com",
  "http://news.yahoo.com/news",
  "http://news.yahoo.com/news/topics/",
  "http://news.google.com",
  "http://news.yahoo.com/us"
]
edges = [[2,0],[2,1],[3,2],[3,1],[0,4]]
startUrl = "http://news.yahoo.com/news/topics/"
Output: [
  "http://news.yahoo.com",
  "http://news.yahoo.com/news",
  "http://news.yahoo.com/news/topics/",
  "http://news.yahoo.com/us"
]
```

Example 2:




```
Input:
urls = [
  "http://news.yahoo.com",
  "http://news.yahoo.com/news",
  "http://news.yahoo.com/news/topics/",
  "http://news.google.com"
]
edges = [[0,2],[2,1],[3,2],[3,1],[3,0]]
startUrl = "http://news.google.com"
Output: ["http://news.google.com"]
Explanation: The startUrl links to all other pages that do not share the same hostname.
```


Constraints:

- 1 <= urls.length <= 1000
- 1 <= urls[i].length <= 300
- startUrl is one of the urls.
- Hostname label must be from 1 to 63 characters long, including the dots, may contain only the ASCII letters from 'a' to 'z', digits from '0' to '9' and the hyphen-minus character ('-').
- The hostname may not start or end with the hyphen-minus character ('-').
- See: https://en.wikipedia.org/wiki/Hostname#Restrictions_on_valid_hostnames
- You may assume there're no duplicates in url library.

Python

Python

```python
# Python Code Solution

class Solution:
    def crawl(self, startUrl: str, htmlParser: 'HtmlParser') -> list[str]:
        from collections import deque

        # Helper function to extract hostname from a url string.
        def get_hostname(url: str) -> str:
            # Remove protocol prefix "http://"
            hostname = url.split("://")[1]
            # The hostname is the part before the first "/" (if exists)
            return hostname.split("/")[0]

        # Extract the hostname from the starting URL.
        start_hostname = get_hostname(startUrl)

        # Initialize a set to keep track of visited urls and a queue for BFS.
        visited = set([startUrl])
        queue = deque([startUrl])

        # Perform BFS traversal of the web links.
        while queue:
            current_url = queue.popleft()
            # Retrieve all URLs found on the current URL.
            for url in htmlParser.getUrls(current_url):

                # Check if URL is under the same hostname and has not been visited.
                if url not in visited and get_hostname(url) == start_hostname:
                    visited.add(url)
                    queue.append(url)

        # Return all visited URLs as a list.
        return list(visited)
```

Python


```python
# """
# This is HtmlParser's API interface.
# You should not implement it, or speculate about its implementation
# """
# Class HtmlParser(object):
# def getUrls(self, url: str) -> list[str]:

class Solution:
  def crawl(self, startUrl: str, htmlParser: 'HtmlParser') -> list[str]:
    q = collections.deque([startUrl])
    seen = {startUrl}
    hostname = startUrl.split('/')[2]

    while q:
      curUrl = q.popleft()
      for url in htmlParser.getUrls(curUrl):
        if url in seen:
          continue
        if hostname in url:
          q.append(url)
          seen.add(url)

    return seen
```

Python

```python
# """
# This is HtmlParser's API interface.
# You should not implement it, or speculate about its implementation
# """
# class HtmlParser(object):
# def getUrls(self, url):
# """
# :type url: str
# :rtype List[str]
# """


class Solution:
    def crawl(self, startUrl: str, htmlParser: 'HtmlParser') -> List[str]:
        def host(url):
            url = url[7:]
            return url.split('/')[0]

        def dfs(url):
            if url in ans:
                return
            ans.add(url)
            for next in htmlParser.getUrls(url):
                if host(url) == host(next):
                    dfs(next)


        ans = set()
        dfs(startUrl)
        return list(ans)
```

Python


```python
# """
# This is HtmlParser's API interface.
# You should not implement it, or speculate about its implementation
# """
# class HtmlParser(object):
# def getUrls(self, url):
# """
# :type url: str
# :rtype List[str]
# """

from collections import deque

class Solution:
    def crawl(self, startUrl: str, htmlParser: 'HtmlParser') -> List[str]:
        visited = set()
        domain = self.get_domain(startUrl)
        queue = deque([startUrl])
        visited.add(startUrl)

        while queue:
            current_url = queue.popleft()
            for url in htmlParser.getUrls(current_url):
                if url not in visited and self.get_domain(url) == domain:
                    visited.add(url)

                    queue.append(url)

        return list(visited)

    def get_domain(self, url: str) -> str:
        return url.split('/')[2]
```

[*] DFS — Pattern Solution (matched from community answers)

Python


```python
# """
# This is HtmlParser's API interface.
# You should not implement it, or speculate about its implementation
# """
# class HtmlParser(object):
# def getUrls(self, url):
# """
# :type url: str
# :rtype List[str]
# """


class Solution:
    def crawl(self, startUrl: str, htmlParser: 'HtmlParser') -> List[str]:
        def host(url):
            url = url[7:]
            return url.split('/')[0]

        def dfs(url):
            if url in ans:
                return
            ans.add(url)
            for next in htmlParser.getUrls(url):
                if host(url) == host(next):
                    dfs(next)


        ans = set()
        dfs(startUrl)
        return list(ans)
```

#1242 MEDIUM

Web Crawler Multithreaded

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

DFS · BFS · Concurrency

Lists: Denny Zhang

Problem:

Given a URL startUrl and an interface HtmlParser, implement a Multi-threaded web crawler to crawl all links that are under the same hostname as startUrl.

Return all URLs obtained by your web crawler in any order.

Your crawler should:

- Start from the page: startUrl
- Call HtmlParser.getUrls(url) to get all URLs from a webpage of a given URL.
- Do not crawl the same link twice.
- Explore only the links that are under the same hostname as startUrl.

As shown in the example URL above, the hostname is example.org. For simplicity's sake, you may assume all URLs use HTTP protocol without any port specified. For example, the URLs http://leetcode.com/problems and http://leetcode.com/contest are under the same hostname, while URLs http://example.org/test and http://example.com/abc are not under the same hostname.

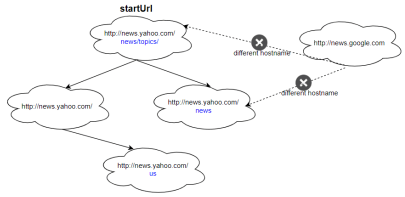
The HtmlParser interface is defined as such:

```
interface HtmlParser {
// Return a list of all urls from a webpage of given url.
// This is a blocking call, that means it will do HTTP request and return when this request
is finished.
public List<String> getUrls(String url);
}
```

Note that getUrls(String url) simulates performing an HTTP request. You can treat it as a blocking function call that waits for an HTTP request to finish. It is guaranteed that getUrls(String url) will return the URLs within 15ms. Single-threaded solutions will exceed the time limit so, can your multi-threaded web crawler do better?

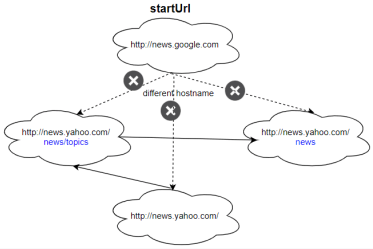
Below are two examples explaining the functionality of the problem. For custom testing purposes, you'll have three variables urls, edges and startUrl. Notice that you will only have access to startUrl in your code, while urls and edges are not directly accessible to you in code.

Example 1:



```
Input:
urls = [
  "http://news.yahoo.com",
  "http://news.yahoo.com/news",
  "http://news.yahoo.com/news/topics/",
  "http://news.google.com",
  "http://news.yahoo.com/us"
]
edges = [[2,0],[2,1],[3,2],[3,1],[0,4]]
startUrl = "http://news.yahoo.com/news/topics/"
Output: [
  "http://news.yahoo.com",
  "http://news.yahoo.com/news",
  "http://news.yahoo.com/news/topics/",
  "http://news.yahoo.com/us"
]
```

Example 2:




```
Input:
urls = [
  "http://news.yahoo.com",
  "http://news.yahoo.com/news",
  "http://news.yahoo.com/news/topics/",
  "http://news.google.com"
]
edges = [[0,2],[2,1],[3,2],[3,1],[3,0]]
startUrl = "http://news.google.com"
Output: ["http://news.google.com"]
Explanation: The startUrl links to all other pages that do not share the same hostname.
```


Constraints:

- 1 <= urls.length <= 1000
- 1 <= urls[i].length <= 300
- startUrl is one of the urls.
- Hostname label must be from 1 to 63 characters long, including the dots, may contain only the ASCII letters from 'a' to 'z', digits from '0' to '9' and the hyphen-minus character ('-').
- The hostname may not start or end with the hyphen-minus character ('-').
- See: https://en.wikipedia.org/wiki/Hostname#Restrictions_on_valid_hostnames
- You may assume there're no duplicates in the URL library.

Follow up:

1. Assume we have 10,000 nodes and 1 billion URLs to crawl. We will deploy the same software onto each node. The software can know about all the nodes. We have to minimize communication between machines and make sure each node does equal amount of work. How would your web crawler design change?
2. What if one node fails or does not work?
3. How do you know when the crawler is done?

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def crawl(self, startUrl, htmlParser):
        # Brute Force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python


```python
import threading
from queue import Queue
from urllib.parse import urlparse

def crawl(startUrl, htmlParser):
    # Extract hostname of the startUrl
    start_host = urlparse(startUrl).netloc
    # Thread-safe set for visited URLs and a lock for safety
    visited = set()
    visited_lock = threading.Lock()

    # Queue to process URLs; initialized with startUrl
    url_queue = Queue()
    url_queue.put(startUrl)

    def worker():
        while True:
            try:
                current_url = url_queue.get(timeout=1) # timeout to avoid blocking indefinitely
            except:
                break
            # Retrieve URLs from the current page using htmlParser
            for next_url in htmlParser.getUrls(current_url):
                # Check if next_url shares the same hostname as startUrl
                if urlparse(next_url).netloc == start_host:
                    with visited_lock:
                        if next_url not in visited:
                            visited.add(next_url)
                            url_queue.put(next_url)
            url_queue.task_done()

    # Mark startUrl as visited
    with visited_lock:
        visited.add(startUrl)

    # Launch worker threads (8 threads used as example)
    threads = []
    for _ in range(8):
        t = threading.Thread(target=worker)
        t.start()
        threads.append(t)

    # Wait until all URLs have been processed
    url_queue.join()

    # Ensure all threads terminate
    for t in threads:
        t.join()

    return list(visited)
```

Python


```python
from threading import Thread, Lock
from urllib.parse import urlparse # To parse URLs and extract hostname for comparison
from typing import List

class Solution:
    def crawl(self, startUrl: str, htmlParser: 'HtmlParser') -> List[str]:
        hostname = urlparse(startUrl).hostname
        seen = set([startUrl]) # Keep track of seen URLs to avoid re-crawling the same URL
        lock = Lock() # Lock for thread-safe operations on the seen set

        def dfs(url):
            for next_url in htmlParser.getUrls(url):
                if urlparse(next_url).hostname == hostname and next_url not in seen:
                    with lock: # Ensure thread-safe write to the seen set
                        if next_url not in seen:
                            seen.add(next_url)
                            dfs(next_url)

        def worker():
            while True:
                with lock:
                    if not unseen:
                        return
                    url = unseen.pop()
                dfs(url)

        unseen = [startUrl] # Stack of URLs to be crawled
        threads = []
        num_threads = 10 # Or any number you deem appropriate based on the problem constraints

        # Start threads
        for _ in range(num_threads):
            thread = Thread(target=worker)
            thread.start()
            threads.append(thread)

        # Wait for all threads to finish
        for thread in threads:
            thread.join()

        return list(seen)

################################
```

Python


```python
# reference https://levelup.gitconnected.com/multi-threaded-python-web-crawler-for-https-pages-e183f0839b71


class Crawler(threading.Thread):
    def __init__(self,base_url, links_to_crawl,have_visited, error_links,url_lock):

        threading.Thread.__init__(self)
        print(f"Web Crawler worker {threading.current_thread()} has Started")
        self.base_url = base_url
        self.links_to_crawl = links_to_crawl
        self.have_visited = have_visited
        self.error_links = error_links
        self.url_lock = url_lock

    def run(self):
            # we create a ssl context so that our script can crawl
            # the https sites with ssl_handshake.

            #Create a SSLContext object with default settings.
            my_ssl = ssl.create_default_context()

            # by default when creating a default ssl context and making an handshake
            # we verify the hostname with the certificate but our objective is to crawl
            # the webpage so we will not be checking the validity of the certificate.
            my_ssl.check_hostname = False


            # in this case we are not verifying the certificate and any
            # certificate is accepted in this mode.
            my_ssl.verify_mode = ssl.CERT_NONE

            # we are defining an infinite while loop so that all the links in our
            # queue are processed.

            while True:

                # In this part of the code we create a global lock on our queue of
                # links so that no two threads can access the queue at same time
                self.url_lock.acquire()
                print(f"Queue Size: {self.links_to_crawl.qsize()}")
                link = self.links_to_crawl.get()
                self.url_lock.release()

                # if the link is None the queue is exhausted or the threads are yet
                # process the links.

                if link is None:
                    break

                # if The link is already visited we break the execution.
                if link in self.have_visited:
```



```python
                    print(f"The link {link} is already visited")
                    break

                try:

                    # This method constructs a full "absolute" URL by combining the
                    # base url with other url. this uses components of the base URL,
                    # in particular the addressing scheme, the network
                    # location and the path, to provide missing components
                    # in the relative URL.
                    # in short we repair our relative url if it is broken.
                    link = urljoin(self.base_url,link)

                    # we use the header parameter to "spoof" the "User-Agent" header
                    # value which is used by the browser to identify itself. This is
                    # because some servers will only allow the connection if it comes
                    # from a verified browser. In this case we are using FirefOx header.
                    req = Request(link, headers= {'User-Agent': 'Mozilla/5.0'})

                    # we are opening the url using a ssl handshake.
                    response = urlopen(req, context=my_ssl)

                    print(f"The URL {response.geturl()} crawled with \
                        status {response.getcode()}")

                    # this returns the html representation of the webpage

                    soup = BeautifulSoup(response.read),"html.parser")

                    # in this case we are finding all the links in the page.
                    for a_tag in soup.find_all('a'):
                        # we are checking of the link is already visited and (network location part) is
our
```

Python

```python
import multiprocessing
from bs4 import BeautifulSoup
from queue import Queue, Empty
from concurrent.futures import ThreadPoolExecutor
from urllib.parse import urljoin, urlparse
import requests

# ThreadPoolExecutor
class MultiThreadedCrawler:

    def __init__(self, seed_url):
        self.seed_url = seed_url
        self.root_url = '{}://{}'.format(urlparse(self.seed_url).scheme,
                                         urlparse(self.seed_url).netloc)

        self.pool = ThreadPoolExecutor(max_workers=5)
        self.scraped_pages = set([])
        self.crawl_queue = Queue()
        self.crawl_queue.put(self.seed_url)

    def parse_links(self, html):
        soup = BeautifulSoup(html, 'html.parser')
        Anchor_Tags = soup.find_all('a', href=True)
        for link in Anchor_Tags:
            url = link['href']
            if url.startswith('/') or url.startswith(self.root_url):

                url = urljoin(self.root_url, url)
                if url not in self.scraped_pages:
                    self.crawl_queue.put(url)

    def scrape_info(self, html):
        soup = BeautifulSoup(html, "html5lib")
        web_page_paragraph_contents = soup('p')
        text = ''
        for para in web_page_paragraph_contents:
            if not ('https:' in str(para.text)):
                text = text + str(para.text).strip()
        print(f'\n <---Text Present in The WebPage is --->\n', text, '\n')
        return

    def post_scrape_callback(self, res):
        result = res.result()
        if result and result.status_code == 200:
            self.parse_links(result.text)
            self.scrape_info(result.text)

    def scrape_page(self, url):
        try:
            res = requests.get(url, timeout=(3, 30))
            return res
        except requests.RequestException:
```



```python
            return

    def run_web_crawler(self):
        while True:
            try:
                print("\n Name of the current executing process: ",
                      multiprocessing.current_process().name, '\n')
                target_url = self.crawl_queue.get(timeout=60)
                if target_url not in self.scraped_pages:
                    print("Scraping URL: {}".format(target_url))
                    self.current_scraping_url = "{}".format(target_url)
                    self.scraped_pages.add(target_url)
                    job = self.pool.submit(self.scrape_page, target_url)
                    job.add_done_callback(self.post_scrape_callback)

            except Empty:
                return
            except Exception as e:
                print(e)
                continue

    def info(self):
        print('\n Seed URL is: ', self.seed_url, '\n')
        print('Scraped pages are: ', self.scraped_pages, '\n')

if __name__ == '__main__':
    cc = MultiThreadedCrawler("https://www.leetcode.ca")
    cc.run_web_crawler()
    cc.info()
```

[*] DFS — Pattern Solution (matched from community answers)

Python


```python
import threading
from queue import Queue
from urllib.parse import urlparse

def crawl(startUrl, htmlParser):
    # Extract hostname of the startUrl
    start_host = urlparse(startUrl).netloc
    # Thread-safe set for visited URLs and a lock for safety
    visited = set()
    visited_lock = threading.Lock()

    # Queue to process URLs; initialized with startUrl
    url_queue = Queue()
    url_queue.put(startUrl)

    def worker():
        while True:
            try:
                current_url = url_queue.get(timeout=1) # timeout to avoid blocking indefinitely
            except:
                break
            # Retrieve URLs from the current page using htmlParser
            for next_url in htmlParser.getUrls(current_url):
                # Check if next_url shares the same hostname as startUrl
                if urlparse(next_url).netloc == start_host:

                    with visited_lock:
                        if next_url not in visited:
                            visited.add(next_url)
                            url_queue.put(next_url)
            url_queue.task_done()

    # Mark startUrl as visited
    with visited_lock:
        visited.add(startUrl)

    # Launch worker threads (8 threads used as example)
    threads = []
    for _ in range(8):
        t = threading.Thread(target=worker)
        t.start()
        threads.append(t)

    # Wait until all URLs have been processed
    url_queue.join()

    # Ensure all threads terminate
    for t in threads:
        t.join()

    return list(visited)
```

#269 HARD

Alien Dictionary

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · String · DFS · BFS · Graph · Topological Sort

Lists: Grind 169 | Premium 98

Problem:

There is a new alien language that uses the English alphabet. However, the order of the letters is unknown to you.

You are given a list of strings words from the alien language's dictionary. Now it is claimed that the strings in words are sorted lexicographically by the rules of this new language.

If this claim is incorrect, and the given arrangement of string in words cannot correspond to any order of letters, return "".

Otherwise, return a string of the unique letters in the new alien language sorted in lexicographically increasing order by the new language's rules. If there are multiple solutions, return any of them.

Example 1:

```
Input: words = ["wrt","wrf","er","ett","rftt"]
Output: "wertf"
```

Example 2:

```
Input: words = ["z","x"]
Output: "zx"
```

Example 3:

```
Input: words = ["z","x","z"]
Output: ""
Explanation: The order is invalid, so return "".
```

Constraints:

- 1 <= words.length <= 100
- 1 <= words[i].length <= 100
- words[i] consists of only lowercase English letters.

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def alienOrder(self, words):
        # Brute Force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python


```python
# Python solution for Alien Dictionary
from collections import defaultdict, deque

def alienOrder(words):
    # Create data structures: graph and in_degree for each unique character.
    graph = defaultdict(set)
    in_degree = {char: 0 for word in words for char in word}

    # Build the graph.
    for i in range(len(words) - 1):
        word1, word2 = words[i], words[i+1]
        # Check for invalid case where prefix of next word is the same but word1 is longer.
        if len(word1) > len(word2) and word1.startswith(word2):
            return ""
        # Compare the words character by character.
        for c1, c2 in zip(word1, word2):
            if c1 != c2:
                # If we haven't already recorded this edge, add it.
                if c2 not in graph[c1]:
                    graph[c1].add(c2)
                    in_degree[c2] += 1
                break

    # Prepare for topological sort by adding all nodes with in_degree == 0.
    queue = deque([char for char in in_degree if in_degree[char] == 0])

    order = []

    # Kahn's algorithm for topological sort.
    while queue:
        current = queue.popleft()
        order.append(current)
        for neighbor in graph[current]:
            in_degree[neighbor] -= 1
            if in_degree[neighbor] == 0:
                queue.append(neighbor)

    # Check if topological sort is possible.
    if len(order) == len(in_degree):
        return "".join(order)
    else:
        return ""

# Example usage:
print(alienOrder(["wrt","wrf","er","ett","rftt"]))
```

Python


```python
class Solution:
  def alienOrder(self, words: list[str]) -> str:
    graph = {}
    inDegrees = [0] * 26

    self._buildGraph(graph, words, inDegrees)
    return self._topology(graph, inDegrees)

  def _buildGraph(
      self,
      graph: dict[str, set[str]],
      words: list[str],
      inDegrees: list[int],
  ) -> None:
    # Create a node for each character in each word.
    for word in words:
      for c in word:
        if c not in graph:
          graph[c] = set()

    for first, second in zip(words, words[1:]):
      length = min(len(first), len(second))
      for j in range(length):
        u = first[j]
        v = second[j]

        if u != v:
          if v not in graph[u]:
            graph[u].add(v)
            inDegrees[ord(v) - ord('a')] += 1
          break # The order of characters after this are meaningless.
      # First = 'ab', second = 'a' - invalid
      if j == length - 1 and len(first) > len(second):
        graph.clear()
        return

  def _topology(self, graph: dict[str, set[str]], inDegrees: list[int]) -> str:
    s = ''
    q = collections.deque()

    for c in graph:
      if inDegrees[ord(c) - ord('a')] == 0:
        q.append(c)

    while q:
      u = q.pop()
      s += u
      for v in graph[u]:
        inDegrees[ord(v) - ord('a')] -= 1
        if inDegrees[ord(v) - ord('a')] == 0:
          q.append(v)

    # Words = ['a', 'x', 'y', 'x']
    return s if len(s) == len(graph) else ''
```

Python

```python
'''
>>> a = ['a','b','c']
>>> zip(a, a[1:])
[('a', 'b'), ('b', 'c')]


# same as pairwise()
>>> from itertools import pairwise
>>> pairwise(a)
<itertools.pairwise object at 0x10845B730>
>>> list(pairwise(a))
[('a', 'b'), ('b', 'c')]
'''

'''
>>> inDegree = {'a':11, 'b':22, 'c':33}

>>> [print(c) for c in inDegree.keys()]
a
b
c
[None, None, None]

# same as for c in inDegree.keys()
>>> [print(c) for c in inDegree]
```

```python
a
b
c
[None, None, None]

>>> [print(c) for c in inDegree.values()]
11
22
33
[None, None, None]

>>> [print(c) for c in inDegree.items()]
('a', 11)
('b', 22)
('c', 33)
[None, None, None]
'''
from collections import defaultdict, deque
from typing import Dict, List, Set

class Solution:
    def alienOrder(self, words: List[str]) -> str:
        graph = defaultdict(set)
        inDegree = defaultdict(int)
```



```python
        self._buildGraph(graph, words, inDegree)
        return self._topology(graph, inDegree)

    def _buildGraph(self, graph: Dict[str, Set[str]], words: List[str], inDegree: Dict[str, int]) -> None:
        # Create a node for each character in each word
        for word in words:
            for c in word:
                inDegree[c] = 0 # necessary for final char counting

        for first, second in zip(words, words[1:]): # or pairwise(words)
            length = min(len(first), len(second))
            for j in range(length):
                u = first[j]
                v = second[j]
                if u != v:
                    if v not in graph[u]:
                        graph[u].add(v)
                        inDegree[v] += 1
                    break # Later characters' order is meaningless
                if j == length - 1 and len(first) > len(second):
                    # If 'ab' comes before 'a', it's an invalid order
                    graph.clear()
                    return

    def _topology(self, graph: Dict[str, Set[str]], inDegree: Dict[str, int]) -> str:

        result = ''
        q = deque([c for c in inDegree if inDegree[c] == 0])

        while q:
            u = q.popleft()
```

[*] DFS — Pattern Solution (matched from community answers)

Python


```python
'''
>>> a = ['a','b','c']
>>> zip(a, a[1:])
[('a', 'b'), ('b', 'c')]


# same as pairwise()
>>> from itertools import pairwise
>>> pairwise(a)
<itertools.pairwise object at 0x10845B730>
>>> list(pairwise(a))
[('a', 'b'), ('b', 'c')]
'''

'''
>>> inDegree = {'a':11, 'b':22, 'c':33}

>>> [print(c) for c in inDegree.keys()]
a
b
c
[None, None, None]

# same as for c in inDegree.keys()
>>> [print(c) for c in inDegree]
```

```python
a
b
c
[None, None, None]

>>> [print(c) for c in inDegree.values()]
11
22
33
[None, None, None]

>>> [print(c) for c in inDegree.items()]
('a', 11)
('b', 22)
('c', 33)
[None, None, None]
'''
from collections import defaultdict, deque
from typing import Dict, List, Set

class Solution:
    def alienOrder(self, words: List[str]) -> str:
        graph = defaultdict(set)
        inDegree = defaultdict(int)
```



```python
        self._buildGraph(graph, words, inDegree)
        return self._topology(graph, inDegree)

    def _buildGraph(self, graph: Dict[str, Set[str]], words: List[str], inDegree: Dict[str, int]) -> None:
        # Create a node for each character in each word
        for word in words:
            for c in word:
                inDegree[c] = 0 # necessary for final char counting

        for first, second in zip(words, words[1:]): # or pairwise(words)
            length = min(len(first), len(second))
            for j in range(length):
                u = first[j]
                v = second[j]
                if u != v:
                    if v not in graph[u]:
                        graph[u].add(v)
                        inDegree[v] += 1
                    break # Later characters' order is meaningless
                if j == length - 1 and len(first) > len(second):
                    # If 'ab' comes before 'a', it's an invalid order
                    graph.clear()
                    return

    def _topology(self, graph: Dict[str, Set[str]], inDegree: Dict[str, int]) -> str:

        result = ''
        q = deque([c for c in inDegree if inDegree[c] == 0])

        while q:
            u = q.popleft()
```


#329 HARD

Longest Increasing Path in a Matrix

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · DFS · BFS · Graph · Topological Sort · Memoization · Matrix

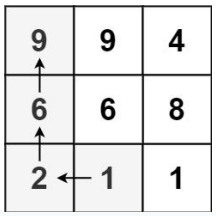
Lists: Grind 169

Problem:

Given an m x n integers matrix, return the length of the longest increasing path in matrix.

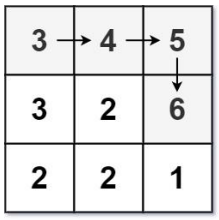
From each cell, you can either move in four directions: left, right, up, or down. You may not move diagonally or move outside the boundary (i.e., wrap-around is not allowed).

Example 1:




```
Input: matrix = [[9,9,4],[6,6,8],[2,1,1]]
Output: 4
Explanation: The longest increasing path is [1, 2, 6, 9].
```

Example 2:




```
Input: matrix = [[3,4,5],[3,2,6],[2,2,1]]
Output: 4
Explanation: The longest increasing path is [3, 4, 5, 6]. Moving diagonally is not allowed.
```

Example 3:

```
Input: matrix = [[1]]
Output: 1
```


Constraints:

- m == matrix.length
- n == matrix[i].length
- 1 <= m, n <= 200
- 0 <= matrix[i][j] <= 231 - 1

Python

Python

```python
# Python implementation for Longest Increasing Path in a Matrix

class Solution:
    def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
        # Return 0 if matrix is empty
        if not matrix or not matrix[0]:
            return 0

        # Dimensions of the matrix
        m, n = len(matrix), len(matrix[0])
        # Create a memoization table (same dimensions as the matrix) initialized to 0
        memo = [[0] * n for _ in range(m)]

        # Define the DFS function to explore the matrix
        def dfs(i, j):
            # If result is already computed, return it
            if memo[i][j]:
                return memo[i][j]

            # Start with a path length of 1 (the cell itself)
            max_len = 1
            # Define directions: up, down, left, right
            directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

            # Explore each direction

            for dx, dy in directions:
                ni, nj = i + dx, j + dy
                # If neighbor is within bounds and its value is greater
                if 0 <= ni < m and 0 <= nj < n and matrix[ni][nj] > matrix[i][j]:
                    length = 1 + dfs(ni, nj)
                    max_len = max(max_len, length)

            # Store computed longest length in memoization table
            memo[i][j] = max_len
            return max_len

        # Compute the longest increasing path starting from each cell
        longest_path = 0
        for i in range(m):
            for j in range(n):
                longest_path = max(longest_path, dfs(i, j))

        return longest_path
```

Python

```python
class Solution:
    def longestIncreasingPath(self, matrix: list[list[int]]) -> int:
        m = len(matrix)
        n = len(matrix[0])

        @functools.lru_cache(None)
        def dfs(i: int, j: int, prev: int) -> int:
            if i < 0 or i == m or j < 0 or j == n:
                return 0
            if matrix[i][j] <= prev:
                return 0

            curr = matrix[i][j]
            return 1 + max(dfs(i + 1, j, curr),
                           dfs(i - 1, j, curr),
                           dfs(i, j + 1, curr),
                           dfs(i, j - 1, curr))

        return max(dfs(i, j, -math.inf) for i in range(m) for j in range(n))
```

Python

```python
class Solution:
    def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
        @cache
        def dfs(i: int, j: int) -> int:
            ans = 0
            for a, b in pairwise((-1, 0, 1, 0, -1)):
                x, y = i + a, j + b
                if 0 <= x < m and 0 <= y < n and matrix[x][y] > matrix[i][j]:
                    ans = max(ans, dfs(x, y))
            return ans + 1

        m, n = len(matrix), len(matrix[0])
        return max(dfs(i, j) for i in range(m) for j in range(n))
```

Python

```python
class Solution:
    def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
        @cache
        def dfs(i: int, j: int) -> int:
            ans = 0
            for a, b in pairwise((-1, 0, 1, 0, -1)):
                x, y = i + a, j + b
                if 0 <= x < m and 0 <= y < n and matrix[x][y] > matrix[i][j]:
                    ans = max(ans, dfs(x, y))
            return ans + 1

        m, n = len(matrix), len(matrix[0])
        return max(dfs(i, j) for i in range(m) for j in range(n))
```

[*] DFS — Pattern Solution (matched from community answers)

Python


```python
# Python implementation for Longest Increasing Path in a Matrix

class Solution:
    def longestIncreasingPath(self, matrix: List[List[int]]) -> int:
        # Return 0 if matrix is empty
        if not matrix or not matrix[0]:
            return 0

        # Dimensions of the matrix
        m, n = len(matrix), len(matrix[0])
        # Create a memoization table (same dimensions as the matrix) initialized to 0
        memo = [[0] * n for _ in range(m)]

        # Define the DFS function to explore the matrix
        def dfs(i, j):
            # If result is already computed, return it
            if memo[i][j]:
                return memo[i][j]

            # Start with a path length of 1 (the cell itself)
            max_len = 1
            # Define directions: up, down, left, right
            directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

            # Explore each direction

            for dx, dy in directions:
                ni, nj = i + dx, j + dy
                # If neighbor is within bounds and its value is greater
                if 0 <= ni < m and 0 <= nj < n and matrix[ni][nj] > matrix[i][j]:
                    length = 1 + dfs(ni, nj)
                    max_len = max(max_len, length)

            # Store computed longest length in memoization table
            memo[i][j] = max_len
            return max_len

        # Compute the longest increasing path starting from each cell
        longest_path = 0
        for i in range(m):
            for j in range(n):
                longest_path = max(longest_path, dfs(i, j))

        return longest_path
```

#685 HARD

Redundant Connection II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

DFS · BFS · Union Find · Graph

Lists: Denny Zhang

Problem:

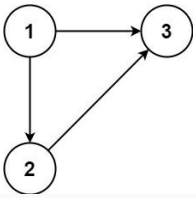
In this problem, a rooted tree is a directed graph such that, there is exactly one node (the root) for which all other nodes are descendants of this node, plus every node has exactly one parent, except for the root node which has no parents.

The given input is a directed graph that started as a rooted tree with n nodes (with distinct values from 1 to n), with one additional directed edge added. The added edge has two different vertices chosen from 1 to n, and was not an edge that already existed.

The resulting graph is given as a 2D-array of edges. Each element of edges is a pair [ui, vi] that represents a directed edge connecting nodes ui and vi, where ui is a parent of child vi.

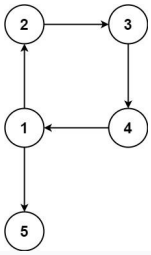
Return an edge that can be removed so that the resulting graph is a rooted tree of n nodes. If there are multiple answers, return the answer that occurs last in the given 2D-array.

Example 1:




```
Input: edges = [[1,2],[1,3],[2,3]]
Output: [2,3]
```

Example 2:




```
Input: edges = [[1,2],[2,3],[3,4],[4,1],[1,5]]
Output: [4,1]
```

Constraints:

- n == edges.length
- 3 <= n <= 1000
- edges[i].length == 2
- 1 <= ui, vi <= n
- ui != vi

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findRedundantDirectedConnection(self, edges):
        # Brute force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python


```python
# Python solution with line-by-line comments
class Solution:
    def findRedundantDirectedConnection(self, edges):
        n = len(edges)
        candidate1 = None
        candidate2 = None
        # Dictionary to record the parent for each child node.
        childParent = {}
        # First pass: detect if a node has two parents.
        for u, v in edges:
            if v in childParent:
                candidate1 = [childParent[v], v] # earlier edge providing parent
                candidate2 = [u, v] # later edge providing parent
            else:
                childParent[v] = u

        # Initialize Union-Find structure: each node is its own parent.
        root = [i for i in range(n+1)]

        # Find function with path compression.
        def find(x):
            while root[x] != x:
                root[x] = root[root[x]]
                x = root[x]
            return x


        cycle_edge = None
        # Second pass: union-find to detect cycle, skip candidate2 if conflict exists.
        for u, v in edges:
            # If candidate2 exists and current edge is candidate2, skip it.
            if candidate2 and [u, v] == candidate2:
                continue
            parent_u = find(u)
            parent_v = find(v)
            # If they have the same root, a cycle is detected.
            if parent_u == parent_v:
                cycle_edge = [u, v]
            else:
                root[parent_v] = parent_u

        # Decide which edge to remove based on conflict and cycle detection.
        if candidate1:
            return candidate1 if cycle_edge else candidate2
        return cycle_edge
```

Python


```python
class UnionFind:
    def __init__(self, n: int):
        self.id = list(range(n))
        self.rank = [0] * n

    def unionByRank(self, u: int, v: int) -> bool:
        i = self._find(u)
        j = self._find(v)
        if i == j:
            return False
        if self.rank[i] < self.rank[j]:
            self.id[i] = j
        elif self.rank[i] > self.rank[j]:
            self.id[j] = i
        else:
            self.id[i] = j
            self.rank[j] += 1
        return True

    def _find(self, u: int) -> int:
        if self.id[u] != u:
            self.id[u] = self._find(self.id[u])
        return self.id[u]
```

```python
class Solution:
    def findRedundantDirectedConnection(
        self, edges: list[list[int]],
    ) -> list[int]:
        ids = [0] + (len(edges) + 1)
        nodeWithTwoParents = 0

        for _, v in edges:
            ids[v] += 1
            if ids[v] == 2:
                nodeWithTwoParents = v

        def findRedundantDirectedConnection(skippedEdgeIndex: int) -> list[int]:
            uf = UnionFind(len(edges) + 1)

            for i, edge in enumerate(edges):
                if i == skippedEdgeIndex:
                    continue
                if not uf.unionByRank(edge[0], edge[1]):
                    return edge

            return []

        # If there is no edge with two ids, don't skip any edge.
        if nodeWithTwoParents == 0:
```

```python
            return findRedundantDirectedConnection(-1)

        for i in reversed(range(len(edges))):
            _, v = edges[i]
            if v == nodeWithTwoParents:
                # Try to delete the edge[i]
                if not findRedundantDirectedConnection(i):
                    return edges[i]
```

Python

```python
class Solution:
    def findRedundantDirectedConnection(self, edges: List[List[int]]) -> List[int]:
        def find(x: int) -> int:
            if p[x] != x:
                p[x] = find(p[x])
            return p[x]

        n = len(edges)
        ind = [0] * n
        for _, v in edges:
            ind[v - 1] += 1
        dup = [i for i, (_, v) in enumerate(edges) if ind[v - 1] == 2]
        p = list(range(n))
        if dup:
            for i, (u, v) in enumerate(edges):
                if i == dup[1]:
                    continue
                pu, pv = find(u - 1), find(v - 1)
                if pu == pv:
                    return edges[dup[0]]
                p[pu] = pv
        for i, (u, v) in enumerate(edges):
            pu, pv = find(u - 1), find(v - 1)
            if pu == pv:

                return edges[i]
            p[pu] = pv
```

Python

```python
class UnionFind:
    def __init__(self, n):
        self.p = list(range(n))
        self.n = n

    def union(self, a, b):
        if self.find(a) == self.find(b):
            return False
        self.p[self.find(a)] = self.find(b)
        self.n -= 1
        return True

    def find(self, x):
        if self.p[x] != x:
            self.p[x] = self.find(self.p[x])
        return self.p[x]


class Solution:
    def findRedundantDirectedConnection(self, edges: List[List[int]]) -> List[int]:
        n = len(edges)
        p = list(range(n + 1))
        uf = UnionFind(n + 1)
        conflict = cycle = None
        for i, (u, v) in enumerate(edges):
```



```python
            if p[v] != v:
                conflict = i
            else:
                p[v] = u
                if not uf.union(u, v):
                    cycle = i
        if conflict is None:
            return edges[cycle]
        v = edges[conflict][1]
        if cycle is not None:
            return [p[v], v]
        return edges[conflict]
```

[*] DFS — Pattern Solution (stored optimal)

Python

```python
# Python solution with line-by-line comments
class Solution:
    def findRedundantDirectedConnection(self, edges):
        n = len(edges)
        candidate1 = None
        candidate2 = None
        # Dictionary to record the parent for each child node.
        childParent = {}
        # First pass: detect if a node has two parents.
        for u, v in edges:
            if v in childParent:
                candidate1 = [childParent[v], v] # earlier edge providing parent
                candidate2 = [u, v] # later edge providing parent
            else:
                childParent[v] = u

        # Initialize Union-Find structure: each node is its own parent.
        root = [i for i in range(n+1)]

        # Find function with path compression.
        def find(x):
            while root[x] != x:
                root[x] = root[root[x]]
                x = root[x]
            return x
```



```python
        cycle_edge = None
        # Second pass: union-find to detect cycle, skip candidate2 if conflict exists.
        for u, v in edges:
            # If candidate2 exists and current edge is candidate2, skip it.
            if candidate2 and [u, v] == candidate2:
                continue
            parent_u = find(u)
            parent_v = find(v)
            # If they have the same root, a cycle is detected.
            if parent_u == parent_v:
                cycle_edge = [u, v]
            else:
                root[parent_v] = parent_u

        # Decide which edge to remove based on conflict and cycle detection.
        if candidate1:
            return candidate1 if cycle_edge else candidate2
        return cycle_edge
```

#1036 HARD

Escape a Large Maze

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · DFS · BFS

Lists: Denny Zhang

Problem:

There is a 1 million by 1 million grid on an XY-plane, and the coordinates of each grid square are (x, y).

We start at the source = [sx, sy] square and want to reach the target = [tx, ty] square. There is also an array of blocked squares, where each blocked[i] = [xi, yi] represents a blocked square with coordinates (xi, yi).

Each move, we can walk one square north, east, south, or west if the square is not in the array of blocked squares. We are also not allowed to walk outside of the grid.

Return true if and only if it is possible to reach the target square from the source square through a sequence of valid moves.

Example 1:

```
Input: blocked = [[0,1],[1,0]], source = [0,0], target = [0,2]
Output: false
Explanation: The target square is inaccessible starting from the source square because we
cannot move.
We cannot move north or east because those squares are blocked.
We cannot move south or west because we cannot go outside of the grid.
```

Example 2:

```
Input: blocked = [], source = [0,0], target = [999999,999999]
Output: true
Explanation: Because there are no blocked cells, it is possible to reach the target square.
```

Constraints:

- 0 <= blocked.length <= 200
- blocked[i].length == 2
- 0 <= xi, yi < 106
- source.length == target.length == 2
- 0 <= sx, sy, tx, ty < 106
- source != target
- It is guaranteed that source and target are not blocked.

>> Brute Force [DFS] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def isEscapePossible(self, blocked, source, target):
        # Brute force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python

```python
from collections import deque

def isEscapePossible(blocked, source, target):
    blocked_set = {(x, y) for x, y in blocked}
    n = len(blocked)
    if n < 2:
        return True
    max_steps = n + (n - 1) // 2

    def bfs(start, finish):
        queue = deque([start])
        seen = {tuple(start)}
        directions = [(0, 1), (1, 0), (0, -1), (-1, 0)]
        steps = 0

        while queue:
            x, y = queue.popleft()
            if [x, y] == finish:
                return True
            for dx, dy in directions:
                nx, ny = x + dx, y + dy
                if 0 <= nx < 10**6 and 0 <= ny < 10**6 and (nx, ny) not in blocked_set and (nx, ny) not in
seen:
                    queue.append((nx, ny))
                    seen.add((nx, ny))
                    steps += 1
```



```python
                if steps > max_steps:
                    return True
        return False

    return bfs(source, target) and bfs(target, source)

# Example usage:
print(isEscapePossible([[0,1],[1,0]], [0,0], [0,2])) # Expected output: false
print(isEscapePossible([], [0,0], [999999,999999])) # Expected output: true
```

Python

```python
class Solution:
    def isEscapePossible(
        self,
        blocked: list[list[int]],
        source: list[int],
        target: list[int]
    ) -> bool:
        def dfs(i: int, j: int, target: list[int], seen: set) -> bool:
            if i < 0 or i >= 10**6 or j < 0 or j >= 10**6:
                return False
            if (i, j) in blocked or (i, j) in seen:
                return False
            seen.add((i, j))
            return (len(seen) > (1 + 199) + 199 // 2 or [i, j] == target or
                    dfs(i + 1, j, target, seen) or
                    dfs(i - 1, j, target, seen) or
                    dfs(i, j + 1, target, seen) or
                    dfs(i, j - 1, target, seen))

        blocked = set(tuple(b) for b in blocked)
        return (dfs(source[0], source[1], target, set()) and
                dfs(target[0], target[1], source, set()))
```

Python

```python
class Solution:
    def isEscapePossible(
        self, blocked: List[List[int]], source: List[int], target: List[int]
    ) -> bool:
        def dfs(source: List[int], target: List[int], vis: set) -> bool:
            vis.add(tuple(source))
            if len(vis) > m:
                return True
            for a, b in pairwise(dirs):
                x, y = source[0] + a, source[1] + b
                if 0 <= x < n and 0 <= y < n and (x, y) not in s and (x, y) not in vis:
                    if [x, y] == target or dfs([x, y], target, vis):
                        return True
            return False

        s = {(x, y) for x, y in blocked}
        dirs = (-1, 0, 1, 0, -1)
        n = 10**6
        m = len(blocked) ** 2 // 2
        return dfs(source, target, set()) and dfs(target, source, set())
```

Python


```python
class Solution:
    def isEscapePossible(
        self, blocked: List[List[int]], source: List[int], target: List[int]
    ) -> bool:
        def dfs(source, target, seen):
            x, y = source
            if (
                not (0 <= x < 10**6 and 0 <= y < 10**6)
                or (x, y) in blocked
                or (x, y) in seen
            ):
                return False
            seen.add((x, y))
            if len(seen) > 20000 or source == target:
                return True
            for a, b in [(0, -1), (0, 1), (1, 0), (-1, 0)]:
                next = [x + a, y + b]
                if dfs(next, target, seen):
                    return True
            return False

        blocked = set((x, y) for x, y in blocked)
        return dfs(source, target, set()) and dfs(target, source, set())
```

[*] DFS — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def isEscapePossible(
        self,
        blocked: list[list[int]],
        source: list[int],
        target: list[int]
    ) -> bool:
        def dfs(i: int, j: int, target: list[int], seen: set) -> bool:
            if i < 0 or i >= 10**6 or j < 0 or j >= 10**6:
                return False
            if (i, j) in blocked or (i, j) in seen:
                return False
            seen.add((i, j))
            return (len(seen) > (1 + 199) + 199 // 2 or [i, j] == target or
                    dfs(i + 1, j, target, seen) or
                    dfs(i - 1, j, target, seen) or
                    dfs(i, j + 1, target, seen) or
                    dfs(i, j - 1, target, seen))

        blocked = set(tuple(b) for b in blocked)
        return (dfs(source[0], source[1], target, set()) and
                dfs(target[0], target[1], source, set()))
```

#1192 HARD

Critical Connections in a Network

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

DFS · Graph · Biconnected Component

Lists: Denny Zhang

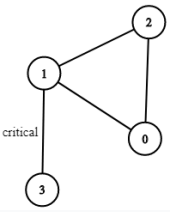
Problem:

There are n servers numbered from 0 to n - 1 connected by undirected server-to-server connections forming a network where connections[i] = [ai, bi] represents a connection between servers ai and bi. Any server can reach other servers directly or indirectly through the network.

A critical connection is a connection that, if removed, will make some servers unable to reach some other server.

Return all critical connections in the network in any order.

Example 1:




```
Input: n = 4, connections = [[0,1],[1,2],[2,0],[1,3]]
Output: [[1,3]]
Explanation: [[3,1]] is also accepted.
```

Example 2:

```
Input: n = 2, connections = [[0,1]]
Output: [[0,1]]
```

Constraints:

- 2 <= n <= 105
- n - 1 <= connections.length <= 105
- 0 <= ai, bi <= n - 1
- ai != bi
- There are no repeated connections.

>> Brute Force [DFS] Interview Prep

Python


```
# Brute Force Approach
class Solution:
    def criticalConnections(self, n, connections):
        # Brute Force - DFS from every unvisited node
        visited = set()
        count = 0
        def dfs(node):
            visited.add(node)
            for nb in graph.get(node, []):
                if nb not in visited: dfs(nb)
        for node in range(n):
            if node not in visited:
                dfs(node); count += 1
        return count
```

Python

Python

```
def criticalConnections(n, connections):
    # Build the adjacency list for the graph
    graph = [[] for _ in range(n)]
    for u, v in connections:
        graph[u].append(v)
        graph[v].append(u)

    # Initialize discovery time and low arrays
    disc = [-1] * n # Discovery times, -1 indicates unvisited
    low = [0] * n # Low values for each node
    time = 0 # Global timer
    bridges = [] # List to store critical connections (bridges)

    def dfs(u, parent):
        nonlocal time
        disc[u] = low[u] = time
        time += 1

        # Traverse all adjacent nodes
        for v in graph[u]:
            if disc[v] == -1: # If v is not visited
                dfs(v, u)
                low[u] = min(low[u], low[v])
                # If the smallest reachable vertex from v is after u's discovery, edge (u,v) is a bridge
                if low[v] > disc[u]:
                    bridges.append([u, v])
            elif v != parent: # v is visited and is not the parent (back edge)
                low[u] = min(low[u], disc[v])

    # Start DFS from node 0
    dfs(0, -1)
    return bridges

# Example usage:
n = 4
connections = [[0,1],[1,2],[2,0],[1,3]]
print(criticalConnections(n, connections)) # Output: [[1, 3]]
```

Python


```
class Solution:
    def criticalConnections(
        self, n: int, connections: List[List[int]]
    ) -> List[List[int]]:
        def tarjan(a: int, fa: int):
            nonlocal now
            now += 1
            dfn[a] = low[a] = now
            for b in g[a]:
                if b == fa:
                    continue
                if not dfn[b]:
                    tarjan(b, a)
                    low[a] = min(low[a], low[b])
                    if low[b] > dfn[a]:
                        ans.append([a, b])
                else:
                    low[a] = min(low[a], dfn[b])

        g = [[] for _ in range(n)]
        for a, b in connections:
            g[a].append(b)
            g[b].append(a)

        dfn = [0] * n

        low = [0] * n
        now = 0
        ans = []
        tarjan(0, -1)
        return ans
```

Python

```
class Solution:
    def criticalConnections(
        self, n: int, connections: List[List[int]]
    ) -> List[List[int]]:
        def tarjan(a: int, fa: int):
            nonlocal now
            now += 1
            dfn[a] = low[a] = now
            for b in g[a]:
                if b == fa:
                    continue
                if not dfn[b]:
                    tarjan(b, a)
                    low[a] = min(low[a], low[b])
                    if low[b] > dfn[a]:
                        ans.append([a, b])
                else:
                    low[a] = min(low[a], dfn[b])

        g = [[] for _ in range(n)]
        for a, b in connections:
            g[a].append(b)
            g[b].append(a)

        dfn = [0] * n
```



```
        low = [0] * n
        now = 0
        ans = []
        tarjan(0, -1)
        return ans
```

[*] DFS — Pattern Solution (matched from community answers)

Python

```
def criticalConnections(n, connections):
    # Build the adjacency list for the graph
    graph = [[] for _ in range(n)]
    for u, v in connections:
        graph[u].append(v)
        graph[v].append(u)

    # Initialize discovery time and low arrays
    disc = [-1] * n # Discovery times, -1 indicates unvisited
    low = [0] * n # Low values for each node
    time = 0 # Global timer
    bridges = [] # List to store critical connections (bridges)

    def dfs(u, parent):
        nonlocal time
        disc[u] = low[u] = time
        time += 1

        # Traverse all adjacent nodes
        for v in graph[u]:
            if disc[v] == -1: # If v is not visited
                dfs(v, u)
                low[u] = min(low[u], low[v])
                # If the smallest reachable vertex from v is after u's discovery, edge (u,v) is a bridge
                if low[v] > disc[u]:
                    bridges.append([u, v])
            elif v != parent: # v is visited and is not the parent (back edge)
                low[u] = min(low[u], disc[v])

    # Start DFS from node 0
    dfs(0, -1)
    return bridges

# Example usage:
n = 4
connections = [[0,1],[1,2],[2,0],[1,3]]
print(criticalConnections(n, connections)) # Output: [[1, 3]]
```


Quick Review — DFS 23 questions · insights · complexity · solution

#463 Island Perimeter [Easy]

Key Insights

- Each land cell contributes 4 edges if isolated.
- For every adjacent pair of land cells (neighbors horizontally or vertically), the shared edge should be subtracted from the total perimeter.
- Instead of DFS/BFS, we can iterate the grid and check the four neighbors for each land cell.
- Alternatively, one may use DFS/BFS to visit all connected cells if needed, but an iterative solution is straightforward with O(rows*cols) time complexity.

Space & Time

Time Complexity: O(m * m), where n is the number of rows and m is the number of columns.

Space Complexity: O(1) for the iterative solution (excluding the input storage).

Solution

We iterate through every cell in the grid. For each land cell, we initially add 4 to the perimeter. Then we check its four neighbors (up, down, left, right). For every neighbor that is also land, we subtract 1 from the current cell's contribution since that side is not exposed. This simple approach ensures that shared borders between adjacent land cells are not over-counted.

#733 Flood Fill [Easy]

Key Insights

- The flood fill operation can be executed using either DFS (Depth First Search) or BFS (Breadth First Search).
- A check is needed at the beginning to determine if the new color is the same as the original; if so, no changes are needed.
- The algorithm recursively or iteratively visits each pixel that is directly adjacent and matches the original color, then updates its color.
- The problem has a worst-case scenario where all pixels are the same color, leading to a full traversal of the grid.

Space & Time

Time Complexity: O(m * n), where m and n are the dimensions of the image grid, because in the worst-case all cells are visited.

Space Complexity: O(m * n) in the worst-case due to the recursion stack (or BFS queue) if the entire image needs to be filled.

Solution

We use a DFS approach to perform the flood fill. The algorithm starts at the given starting pixel and uses recursion to fill connected pixels with the same original color. The following steps are key:

- Check if the starting pixel's color is already the target color; if so, return the image immediately. - Define a recursive helper function that: Checks boundary conditions. Updates the current pixel's color. Recursively calls itself on the four adjacent directions (up, down, left, right). - Return the modified image after the recursion completes.

Data structures used:

- Recursion call stack for the DFS implementation. - Variables to store the image dimensions and original color.

#130 Surrounded Regions [Medium]

Key Insights

- Regions connected to the border are not captured.
- Use DFS/BFS from border 'O's to mark safe cells.
- After marking, flip all remaining 'O's to 'X' and then revert the marked cells back to 'O'.

Space & Time

Time Complexity: O(m * n)

Space Complexity: O(m * n) in the worst case due to recursion stack or queue storage for DFS/BFS

Solution

The solution involves two main steps. First, traverse the board's borders and run a DFS or BFS from any encountered 'O', marking it (e.g., using '*') to indicate that it should not be flipped. Next, go through each cell in the board: flip any remaining 'O' (i.e., not marked safe) to 'X' and change the temporary marker '*' back to 'O'. This approach ensures only the regions completely surrounded by 'X' are captured.

#133 Clone Graph [Medium]

Key Insights

- The graph is undirected and connected.
- Each node must be cloned exactly once to prevent cycles and repetitions.
- Use a mapping/dictionary to keep track of cloned nodes.
- A depth-first search (DFS) or breadth-first search (BFS) can be used to traverse and clone the graph.
- Handle edge cases where the graph might be empty.

Space & Time

Time Complexity: O(N + E), where N is the number of nodes and E is the number of edges, because each node and each edge are traversed once.

Space Complexity: O(N), to store the mapping from original nodes to their clones, and for the recursion/queue used in traversal.

Solution

We employ a depth-first search (DFS) approach to clone the graph. The main idea is to create a mapping (hash table) that links each original node to its corresponding cloned node. During the DFS traversal, if a node is encountered for the first time, a new node is created and stored in the mapping; then, the algorithm recursively clones all of its neighbors. For an already visited node, the cloned node is directly returned from the mapping. This ensures that every node is cloned only once and correctly handles cycles in the graph.

#200 Number of Islands [Medium]

Key Insights

- Use graph traversal (DFS/BFS) to explore connected components of land.
- Iterate over the matrix and start a traversal (e.g., DFS) when encountering an unvisited "1".
- During traversal, mark visited cells to avoid duplicate counting.
- Each DFS/BFS call completely explores one island.
- Alternative approach can use Union Find to group connected lands.

Space & Time

Time Complexity: O(m * n), as each cell is visited at most once.

Space Complexity: O(m * n) in the worst-case scenario due to recursion stack (DFS) or auxiliary data structures.

Solution

We solve the problem using a DFS-based approach. The algorithm iterates over each cell in the grid; when a "1" (land) is encountered, the DFS is initiated to mark the entire island (all connected "1"s) as visited (by setting them to "0"). This prevents recounting the same island. The island count is incremented each time a new island is discovered and fully explored.

#207 Course Schedule [Medium]

Key Insights

- This problem can be modeled as a directed graph where courses are nodes and prerequisites are edges.
- If the graph has a cycle, then it is not possible to complete all courses.

- We can detect cycles using either Topological Sort (BFS/Kahn's algorithm) or DFS-based cycle detection.
- The time complexity is O(V + E), where V is the number of courses and E is the number of prerequisite pairs.

Space & Time

Time Complexity: O(V + E)

Space Complexity: O(V + E)

Solution

We solve the problem using the Topological Sort algorithm (BFS/Kahn's algorithm). First, we build a graph (using an adjacency list) and track the indegree (number of prerequisites) for each course. We then find all courses with an indegree of zero (courses with no prerequisites) and add them to a processing queue. While the queue is not empty, we remove a course from the queue, reduce the indegree for its neighboring courses, and if any neighbor's indegree becomes zero, we add it to the queue as well. If we successfully process all courses, then it means that the graph was acyclic and a valid schedule exists, returning true. If not, a cycle exists, and we return false.

#210 Course Schedule II [Medium]

Key Insights

- This is a topological sort problem on a directed graph.
- The courses are nodes, and prerequisites form directed edges.
- We can solve the problem by either:
- Using DFS with cycle detection and a stack for topological ordering.
- Using BFS (Kahn's Algorithm) to process nodes with zero in-degrees.
- If at any point no node with a zero in-degree is available, a cycle exists and no valid order can be produced.

Space & Time

Time Complexity: O(V + E) where V is the number of courses and E is the number of prerequisite pairs.

Space Complexity: O(V + E) for storing the graph and in-degree array.

Solution

We treat the problem as a topological sort on a directed acyclic graph (DAG). For a BFS approach using Kahn's algorithm:

- Build an adjacency list for the graph and an in-degree array to record the number of prerequisites (incoming edges) for each course. - Initialize a queue with all courses that have an in-degree of zero (i.e., courses that can be taken without prerequisites). - Dequeue nodes from the queue, add them to the ordering, and reduce the in-degree of their neighbors. - If a neighbor's in-degree becomes zero, add it to the queue. - If the final ordering includes all courses, return the ordering; otherwise, return an empty array indicating a cycle. The chosen data structures are:

- A list (or array) for the in-degree counts. - A dictionary (or array of lists) for the adjacency list. - A queue to process nodes with zero in-degree.

#261 Graph Valid Tree [Medium]

Key Insights

- A tree must be acyclic and connected.
- For n nodes, a valid tree must have exactly n - 1 edges.
- A DFS/BFS traversal can check connectivity and simultaneously detect cycles.
- The Union-Find (disjoint set) approach provides an alternative for cycle detection and connectivity verification.

Space & Time

Time Complexity: O(n + e), where e is the number of edges

Space Complexity: O(n + e) for storing the graph and traversal/union-find structures

Solution

We can solve the problem using either a DFS/BFS approach or a Union-Find algorithm. For the DFS/BFS method, we first check if the number of edges is exactly n - 1. If not, it cannot be a tree. Then, we build an adjacency list for the

graph and perform a DFS starting from node 0 while keeping track of visited nodes and parent information to avoid false-positive cycle detections. If during DFS we encounter a node that has been visited (and is not the parent), then a cycle exists. Finally, if all nodes are reached from node 0 and no cycle is found, the graph is a valid tree.

Using Union-Find, we iterate over each edge and try to union the two nodes. If they already share the same parent (i.e., are in the same set), a cycle exists. After processing all edges, we confirm that exactly one connected component exists.

#310 Minimum Height Trees [Medium]

Key Insights

- A tree can have at most two centers that will serve as the roots for its minimum height trees.
- Instead of trying all possible roots, we can remove leaves layer by layer until we are left with the central node(s).
- The process is similar to a topological sort on the tree.
- When there are 2 or fewer nodes left, these nodes are the centers (roots) of MHTs.

Space & Time

Time Complexity: O(n) where n is the number of nodes, since we process each node and edge once.

Space Complexity: O(n) for storing the graph and additional data structures like the leaves list.

Solution

The solution builds an undirected graph from the input, then identifies and removes node-leaves iteratively. A leaf is defined as a node connected to only one other node. By repeatedly removing the leaves and updating the graph (decreasing neighbor degrees), the algorithm peels away the outer layers of the tree. When only one or two nodes remain, these nodes are the centers of the tree and form the roots of the minimum height trees.

#323 Number of Connected Components in an Undirected Graph [Medium]

Key Insights

- Use an adjacency list to represent the graph.
- A DFS (or BFS) traversal from an unvisited node will traverse an entire connected component.
- Each new DFS/BFS call from an unvisited node signals a separate connected component.
- Alternatively, the Union-Find data structure can be used to union connected nodes and count distinct sets.

Space & Time

Time Complexity: O(n + e), where n is the number of nodes and e is the number of edges.

Space Complexity: O(n + e) due to the storage of the graph and tracking of visited nodes.

Solution

We solve the problem by iterating over each node and performing a DFS for any node that hasn't been visited. The DFS uses an iterative approach with a stack to traverse all nodes in a connected component. Each DFS initiation corresponds to one connected component. This method efficiently explores the graph using an adjacency list and marks nodes as visited to prevent redundant traversals.

#417 Pacific Atlantic Water Flow [Medium]

Key Insights

- Reverse the typical DFS/BFS perspective: start from the oceans and work inward.
- Maintain two matrices (or sets) to keep track of cells reachable from the Pacific and Atlantic edges.
- A cell can be added to the result if it is reached in both traversals.
- The height constraint ensures water flows only in non-increasing order.

Space & Time

Time Complexity: O(m * n) where m is the number of rows and n is the number of columns. Each cell is processed up to twice (once from each ocean).

Space Complexity: O(m * n) for the visited matrices and the recursion/queue used in DFS/BFS.

Solution

The solution uses depth-first search (DFS) from the ocean boundaries. We initialize two boolean matrices, one for each ocean. For the Pacific, we start DFS from the leftmost column and top row; for the Atlantic, we start from the rightmost column and bottom row. In each DFS, if the next cell has a height equal to or greater than the current cell and hasn't been visited, we continue the search. Finally, we scan through the grid to collect cells that are reachable from both oceans. This reverse search simplifies the path validations and leverages the allowed water flow conditions.

#490 The Maze [Medium]

Key Insights

- The ball continues moving in one direction until it hits a wall (or the boundary, which is considered a wall).
- You must simulate the ball's rolling rather than taking a single step.
- Even if the ball passes through the destination, it is only valid if it can stop exactly there.
- A breadth-first search (BFS) or depth-first search (DFS) strategy is ideal because we need to explore all stopping positions.
- Use a visited structure to avoid revisiting positions and thus prevent infinite loops.

Space & Time

Time Complexity: O(m * n) where m and n are the dimensions of the maze.

Space Complexity: O(m * n) due to the storage used by the visited matrix and the BFS/DFS queue or recursion stack.

Solution

We simulate the ball's movement using a breadth-first search (BFS) approach. Each time we pick a stop position, we try to roll in all four directions until the ball hits a wall. At that point, if the new stopping position has not been visited, we mark it as visited and add it to our queue. If we ever reach the destination exactly, we return true. The key is simulating the "roll" by iteratively moving in a direction until no further move is possible.

We store the coordinates in a queue (or stack for DFS) along with a 2D visited array to monitor which positions have been processed. The directions are maintained in a list/array, and for each direction, we keep moving until the next cell would be out of bounds or a wall.

#694 Number of Distinct Islands [Medium]

Key Insights

- Use DFS/BFS to traverse and mark each island.
- For each island, record its shape relative to a starting coordinate to allow translation invariance.
- Represent the island's shape as a unique signature (e.g., tuple or string of relative positions).
- Use a set (or hash table) to store unique shapes.

Space & Time

Time Complexity: O(m * n) as each cell is visited once.

Space Complexity: O(m * n) in the worst-case due to recursion stack and storing island shapes.

Solution

Use depth-first search (DFS) to explore the grid. When an unvisited land cell (1) is encountered, initiate a DFS to traverse the entire island. Record each land cell's coordinates relative to the starting position of the island. This relative representation is invariant under translation, making it possible to compare island shapes. Store the canonical shape signature (for example, as a tuple or string) in a set to ensure uniqueness. Finally, return the size of the set as the number of distinct islands.

#695 Max Area of Island [Medium]

Key Insights

- Traverse each cell in the grid.
- Use Depth-First Search (DFS) or Breadth-First Search (BFS) to explore connected 1's.
- Mark visited cells to prevent double counting (e.g., by setting them to 0).

- Maintain and update a maximum area count during traversal.
- The grid's boundaries are naturally water, so no extra edge processing is needed.

Space & Time

Time Complexity: O(m * n) where m and n are the dimensions of the grid since every cell is visited at most once.

Space Complexity: O(m * n) in the worst-case scenario for the recursion stack (or auxiliary data structures) when the grid is entirely land.

Solution

The solution leverages a DFS approach to iterate over each cell in the given grid. When a cell with a value of 1 is encountered, a DFS is initiated from that cell to calculate the area of the island connected to it. Cells are marked as visited (by changing them from 1 to 0) during the DFS to avoid revisiting. The DFS explores all four directions (up, down, left, right) and accumulates the count of connected cells. The maximum area found among all islands is returned at the end. This approach efficiently computes the largest island area with a simple traversal mechanism.

#721 Accounts Merge [Medium]

Key Insights

- Two accounts are merged if they share at least one common email.
- Treat each email as a node in a graph; accounts represent connected components.
- Both DFS/BFS and Union-Find can be used to find connected components.
- Keep a mapping from email to name because all emails in a connected component belong to the same person.
- After grouping, sort the emails and prepend the name to form the final account list.

Space & Time

Time Complexity: O(N * M * log(M)) where N is the number of accounts and M is the number of emails per account (owing to sorting and union operations).

Space Complexity: O(N * M), used for storing email mappings and union-find structures.

Solution

The solution involves using a Union-Find (Disjoint Set Union) data structure to merge emails that are connected by being on the same account. For each account, we union the first email with every other email in that account. We also maintain a mapping from email to owner name. After processing all accounts, we group the emails by their root parent derived from the union-find structure. For each group, we sort the emails and prepend the owner's name. Finally, we return the merged list of accounts.

#785 Is Graph Bipartite? [Medium]

Key Insights

- A graph is bipartite if you can assign two colors to each node such that no two adjacent nodes share the same color.
- The challenge may require exploring disconnected components.
- BFS or DFS can be used to attempt the two-coloring, and a conflict during coloring indicates the graph is not bipartite.
- Union Find is an alternative approach, but BFS/DFS is more intuitive for graph coloring problems.

Space & Time

Time Complexity: O(V + E) where V is the number of nodes and E is the number of edges, since every node and edge is traversed at most once.

Space Complexity: O(V), for the color array and the queue (or recursion stack) used in BFS (or DFS).

Solution

We use a BFS/DFS approach to assign colors (0 and 1) to nodes. A color array, initially set to -1 for all nodes, is maintained. For each unvisited node (color -1), we start a BFS (or DFS) and assign it an initial color (e.g., 0). For every visited neighbor, if it has not been colored, assign it the opposite color; if it is already colored with the same color as the current node, a conflict has occurred, and the graph is not bipartite. This method gracefully handles

disconnected components by iterating over all nodes.

#1236 Web Crawler [Medium]

Key Insights

- Treat the set of webpages as a graph where each URL is a node and an edge exists from one URL to another if it is returned by HtmlParser.getUrls(url).
- Use either breadth-first search (BFS) or depth-first search (DFS) to traverse the webpages.
- Extract the hostname from a URL to ensure that only URLs on the same domain are crawled.
- Maintain a visited set (or equivalent) to avoid processing the same URL multiple times.

Space & Time

Time Complexity: O(N + E) where N is the number of nodes (URLs) and E is the number of edges (links between URLs) encountered during the crawl.

Space Complexity: O(N) for storing the visited URLs and the queue/stack used for traversal.

Solution

The solution uses a BFS (or DFS) approach to traverse the web graph:

- Extract the hostname from startUrl (for instance, by splitting on "://" and then the first "/" that follows). - Initialize a visited set to keep track of URLs that have been crawled. - Use a queue to implement the BFS. Enqueue the startUrl. - While the queue is not empty, dequeue a URL and retrieve its outgoing links using HtmlParser.getUrls(url). - For each URL obtained: Extract its hostname and compare it with the hostname of startUrl. If it matches and it has not been visited before, add it to the visited set and enqueue it. - Finally, return the set of visited URLs.

This approach guarantees that each URL is processed only once and that only URLs from the same hostname are crawled.

#1242 Web Crawler Multithreaded [Medium]

Key Insights

- Use multiple threads to perform concurrent crawling, which speeds up the process over a single-threaded approach.
- Maintain a thread-safe set (visited) to ensure no URL is processed twice.
- Utilize a work queue to manage URLs pending processing.
- Filter URLs based on the hostname extracted from startUrl.
- Apply synchronization (locks or concurrent data structures) to avoid race conditions when multiple threads access shared data.

Space & Time

Time Complexity: O(N) where N is the number of URLs processed, with additional constant factors due to threading overhead.

Space Complexity: O(N) for storing the visited URLs and the work queue.

Solution

The solution initializes a shared work queue with the startUrl and a thread-safe visited set. Each worker thread dequeues a URL, retrieves its linked URLs via the blocking HtmlParser.getUrls() call, and filters the resulting URLs by comparing their hostnames with the start URL's hostname. New URLs that have not yet been visited are added to both the visited set and the work queue. The use of a thread pool or multiple worker threads along with synchronization ensures that the crawler processes URLs concurrently while avoiding race conditions. The approach follows a BFS-like pattern spread across multiple threads.

#269 Alien Dictionary [Hard]

Key Insights

- Build a graph where each unique character is a node.
- Compare adjacent words to determine the first different character, which gives the ordering (directed edge).

- Ensure all characters, even those without edges, are represented.
- Use a topological sort (using DFS or BFS) to determine a valid character ordering.
- Handle edge cases, such as words where a longer word appears before its prefix which is invalid.

Space & Time

Time Complexity: O(N * L + V + E) where N is the number of words, L is the average length of each word, V is the number of unique characters, and E is the number of edges derived from adjacent word comparisons.

Space Complexity: O(V + E) for storing the graph and additional data structures for topological sorting.

Solution

We use a graph-based approach to solve the problem. First, we initialize a graph (adjacency list) and in-degree dictionary for all unique characters. We then iterate through each pair of consecutive words, comparing them character by character to find the first pair that differs. This difference implies an order: the first character should come before the second character. We add a directed edge accordingly and update the in-degree for the target character.

To get the final ordering, we perform a topological sort using Kahn's algorithm:

- Enqueue characters with in-degree zero. - Remove characters from the queue and append them to our result. - Decrement the in-degree of neighboring nodes. - If a neighbor's in-degree becomes zero, enqueue it. If the result's length equals the number of unique nodes, we have a valid ordering; otherwise, a cycle exists, and we return an empty string.

#329 Longest Increasing Path in a Matrix [Hard]

Key Insights

- The problem can be modeled as a graph where each cell is a node connected to its neighbors if the neighbor's value is greater.
- A Depth First Search (DFS) with memoization is effective to avoid recalculations.
- Dynamic Programming (DP) is used to cache the longest path found from each cell.
- Since each cell can be a start point, compute DFS for every cell and update the global maximum.

Space & Time

Time Complexity: O(m * n) - each cell is computed once due to memoization.

Space Complexity: O(m * n) - space is used for the memoization cache and recursion stack.

Solution

We solve the problem using a DFS approach combined with memoization to keep track of already computed longest paths from each cell. For each cell in the matrix, a DFS is initiated to explore all four valid directions (up, down, left, right). If a neighboring cell has a larger value, the DFS recurses from that neighbor. The result for each cell is cached in a 2D memoization table, ensuring that each cell is processed only once. After processing all cells, the maximum value in the memo table represents the longest increasing path in the entire matrix.

#685 Redundant Connection II [Hard]

Key Insights

- The extra edge introduces one of two issues:
- A node has two parents.
- A cycle exists in the graph.
- First, scan through edges to detect if any node gets two parents. If found, there are two candidate edges.
- Use the Union-Find data structure (Disjoint Set Union) to detect cycles.
- Depending on whether a conflict (two parents) exists and/or a cycle is detected, decide which edge to remove:
- If a node has two parents, test by ignoring the second candidate edge during union-find. If no cycle forms, the second candidate is removable; otherwise, remove the first candidate.
- If no conflict exists, simply remove the edge that causes a cycle.

Space & Time

Time Complexity: O(n)

Space Complexity: O(n)

Solution

The solution consists of two main steps:

- Traverse the list of edges to detect whether any node has two parents. If such a node is found, mark the two candidate edges (candidate1: the earlier edge, candidate2: the later edge). - Use the Union-Find algorithm to iterate over the edges: If there is a conflict (i.e., a node with two parents), skip candidate2 initially. While processing edges with Union-Find, check if adding an edge would create a cycle. Finally, based on the presence of a cycle and the two-parent conflict, decide which candidate edge to return.

Key Data Structures:

- An array (or hash map) to track the parent of each node. - The Union-Find (Disjoint Set Union) structure to union sets and detect cycles.

#1036 Escape a Large Maze [Hard]

Key Insights

- The maximum number of blocked cells is 200, which limits the size of a potential completely enclosed region.
- Even if the target is unreachable by direct search, if the BFS (or DFS) explores more than a calculated threshold (based on the blocked cells count), then the region is not fully enclosed.
- Perform two BFS explorations: one starting from the source and one from the target. If neither search is trapped within a small region, then an escape/path exists.
- Early exit in BFS if the target is reached or if the number of visited nodes exceeds the maximum possible enclosed area.

Space & Time

Time Complexity: O(B^2) in the worst case, where B is the number of blocked cells (with B ≤ 200).

Space Complexity: O(B^2) due to the storage of visited cells in the worst-case scenario.

Solution

The solution uses the Breadth-First Search (BFS) algorithm. A key observation here is that the blocked cells can only enclose a limited area. We compute a threshold (maxSteps) based on the number of blocked cells (n*(n-1)/2) to determine if the search is confined. The BFS from the source (and similarly from the target) stops in two cases: if the target is reached or if the visited area exceeds the threshold, indicating that the region is not fully enclosed by blocked cells. Running BFS from both the source and target ensures that neither is trapped inside a small banned region.

#1192 Critical Connections in a Network [Hard]

Key Insights

- The problem is about identifying bridges in an undirected graph.
- Use Depth-First Search (DFS) to explore the graph.
- During DFS, track discovery times and low-link values for each node.
- For an edge (u, v), if low[v] > disc[u] then (u, v) is a critical connection.
- Tarjan's algorithm is well-suited for solving this problem in O(n + m) time.

Space & Time

Time Complexity: O(n + m) where n is the number of nodes and m is the number of connections.

Space Complexity: O(n + m) for the graph representation and DFS recursion stack.

Solution

We solve this problem using a DFS-based approach (Tarjan's algorithm). First, we transform the list of connections into an adjacency list for efficient graph traversal. We then start a DFS from an arbitrary node (node 0, since the graph is connected) while maintaining two arrays: one for discovery times (disc) and one for low-link values (low).

The discovery time is the time when the node is first visited, and the low-link value is the smallest discovery time that can be reached from that node (including via back edges). For each edge (u, v), after a DFS call on v, if low[v] > disc[u] then the edge (u, v) is a bridge because v (and its descendants) cannot reach u or any of its ancestors without this edge. This edge is then recorded as a critical connection.

#18 Binary Search — 24 questions · #18 of 21

#35 EASY

Search Insert Position

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search

Lists: Denny Zhang

Problem:

Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

Example 1:

```
Input: nums = [1,3,5,6], target = 5
Output: 2
```

Example 2:

```
Input: nums = [1,3,5,6], target = 2
Output: 1
```

Example 3:

```
Input: nums = [1,3,5,6], target = 7
Output: 4
```

Constraints:

- 1 <= nums.length <= 104
- -104 <= nums[i] <= 104
- nums contains distinct values sorted in ascending order.
- -104 <= target <= 104

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def searchInsert(self, nums, target):
        # Brute Force O(n) - linear scan instead of binary search
        for i, val in enumerate(nums):
            if val == target:
                return i
        return -1
```

Python

Python


```python
# Python solution using binary search
def searchInsert(nums, target):
    # Initialize low and high pointers for binary search
    low, high = 0, len(nums) - 1
    # Perform binary search until low pointer exceeds high pointer
    while low <= high:
        mid = (low + high) // 2 # Calculate the middle index
        if nums[mid] == target:
            return mid # Return mid if target is found
        elif target < nums[mid]:
            high = mid - 1 # Narrow search on the left side
        else:
            low = mid + 1 # Narrow search on the right side
    # Return low pointer as the insertion index when target is not found
    return low

# Example usage
print(searchInsert([1, 3, 5, 6], 5)) # Output: 2
```

Python

```python
class Solution:
  def searchInsert(self, nums: list[int], target: int) -> int:
    l = 0
    r = len(nums)

    while l < r:
      m = (l + r) // 2
      if nums[m] == target:
        return m
      if nums[m] < target:
        l = m + 1
      else:
        r = m

    return l
```

Python

```python
class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        l, r = 0, len(nums)
        while l < r:
            mid = (l + r) >> 1
            if nums[mid] >= target:
                r = mid
            else:
                l = mid + 1
        return l
```

Python


```python
class Solution:
    def searchInsert(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums)
        while left < right:
            mid = (left + right) >> 1
            if nums[mid] >= target:
                right = mid
            else:
                left = mid + 1
        return left
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
# Python solution using binary search
def searchInsert(nums, target):
    # Initialize low and high pointers for binary search
    low, high = 0, len(nums) - 1
    # Perform binary search until low pointer exceeds high pointer
    while low <= high:
        mid = (low + high) // 2 # Calculate the middle index
        if nums[mid] == target:
            return mid # Return mid if target is found
        elif target < nums[mid]:
            high = mid - 1 # Narrow search on the left side
        else:
            low = mid + 1 # Narrow search on the right side
    # Return low pointer as the insertion index when target is not found
    return low

# Example usage
print(searchInsert([1, 3, 5, 6], 5)) # Output: 2
```

#69 EASY

Sqrt(x)

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math · Binary Search

Lists: Denny Zhang

Problem:

Given a non-negative integer x, return the square root of x rounded down to the nearest integer. The returned integer should be non-negative as well.

You must not use any built-in exponent function or operator.

- For example, do not use pow(x, 0.5) in c++ or x ** 0.5 in python.

Example 1:

```
Input: x = 4
Output: 2
Explanation: The square root of 4 is 2, so we return 2.
```

Example 2:


```
Input: x = 8
Output: 2
Explanation: The square root of 8 is 2.82842..., and since we round it down to the nearest
integer, 2 is returned.
```

Constraints:

- 0 <= x <= 231 - 1

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def mySqrt(self, x):
        # Brute Force O(n) - linear scan instead of binary search
        for i, val in enumerate(x):
            if val == target:
                return i
        return -1
```

Python

Python

```python
# Binary search solution to find the integer square root
def mySqrt(x):
    # Edge case for small numbers
    if x < 2:
        return x

    low, high = 0, x
    ans = 0
    # Binary search loop
    while low <= high:
        mid = (low + high) // 2
        # If mid squared equals x, we found the exact square root
        if mid * mid == x:
            return mid
        # If mid squared is less than x, store mid and search right half
        elif mid * mid < x:
            ans = mid # Candidate answer
            low = mid + 1
        else:
            # If mid squared is more than x, search left half
            high = mid - 1
    return ans

# Example usage:
print(mySqrt(4)) # Output: 2

print(mySqrt(8)) # Output: 2
```

Python

```python
class Solution:
  def mySqrt(self, x: int) -> int:
    return bisect.bisect_right(range(x + 1), x,
                               key=lambda m: m * m) - 1
```

Python

```python
class Solution:
    def mySqrt(self, x: int) -> int:
        l, r = 0, x
        while l < r:
            mid = (l + r + 1) >> 1
            if mid > x // mid:
                r = mid - 1
            else:
                l = mid
        return l
```

Python

```python
class Solution:
    def mySqrt(self, x: int) -> int:
        l, r = 0, x
        while l < r:
            mid = (l + r + 1) >> 1
            if mid > x // mid:
                r = mid - 1
            else:
                l = mid
        return l
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
# Binary search solution to find the integer square root
def mySqrt(x):
    # Edge case for small numbers
    if x < 2:
        return x

    low, high = 0, x
    ans = 0
    # Binary search loop
    while low <= high:
        mid = (low + high) // 2
        # If mid squared equals x, we found the exact square root
        if mid * mid == x:
            return mid
        # If mid squared is less than x, store mid and search right half
        elif mid * mid < x:
            ans = mid # candidate answer
            low = mid + 1
        else:
            # If mid squared is more than x, search left half
            high = mid - 1
    return ans

# Example usage:
print(mySqrt(4)) # Output: 2

print(mySqrt(8)) # Output: 2
```


#278 EASY

First Bad Version

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Binary Search · Interactive

Lists: Grind 169

Problem:

You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which returns whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

Example 1:

```
Input: n = 5, bad = 4
Output: 4
Explanation:
call isBadVersion(3) -> false
call isBadVersion(5) -> true
call isBadVersion(4) -> true
Then 4 is the first bad version.
```

Example 2:

```
Input: n = 1, bad = 1
Output: 1
```

Constraints:

- 1 <= bad <= n <= 231 - 1

Python

Python

```python
# Python implementation of First Bad Version
class Solution:
    def firstBadVersion(self, n: int) -> int:
        # Initialize the left pointer to the first version and right pointer to the last version
        left, right = 1, n
        while left < right:
            # Calculate the mid point to avoid potential overflow
            mid = left + (right - left) // 2
            # Call the API isBadVersion to check if mid version is bad
            if isBadVersion(mid):
                # If mid is bad, search the left half including mid
                right = mid
            else:
                # If mid is not bad, search the right half excluding mid
                left = mid + 1
        # When left equals right, we found the first bad version
        return left
```

Python


```python
class Solution:
   def firstBadVersion(self, n: int) -> int:
     l = 1
     r = n

     while l < r:
       m = (l + r) >> 1
       if isBadVersion(m):
         r = m
       else:
         l = m + 1

     return l
```

Python

```python
# The isBadVersion API is already defined for you.
# def isBadVersion(version: int) -> bool:


class Solution:
    def firstBadVersion(self, n: int) -> int:
        l, r = 1, n
        while l < r:
            mid = (l + r) >> 1
            if isBadVersion(mid):
                r = mid
            else:
                l = mid + 1
        return l
```

Python

```python
# The isBadVersion API is already defined for you.
# @param version, an integer
# @return an integer
# def isBadVersion(version):


class Solution:
    def firstBadVersion(self, n):
        """
        :type n: int
        :rtype: int
        """
        left, right = 1, n
        while left < right:
            mid = (left + right) >> 1
            if isBadVersion(mid):
                right = mid
            else:
                left = mid + 1
        return left
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python


```python
# Python implementation of First Bad Version
class Solution:
    def firstBadVersion(self, n: int) -> int:
        # Initialize the left pointer to the first version and right pointer to the last version
        left, right = 1, n
        while left < right:
            # Calculate the mid point to avoid potential overflow
            mid = left + (right - left) // 2
            # Call the API isBadVersion to check if mid version is bad
            if isBadVersion(mid):
                # If mid is bad, search the left half including mid
                right = mid
            else:
                # If mid is not bad, search the right half excluding mid
                left = mid + 1
        # When left equals right, we found the first bad version
        return left
```

#374 EASY

Guess Number Higher or Lower

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math · Binary Search · Interactive

Lists: Denny Zhang

Problem:

We are playing the Guess Game. The game is as follows:

I pick a number from 1 to n. You have to guess which number I picked (the number I picked stays the same throughout the game).

Every time you guess wrong, I will tell you whether the number I picked is higher or lower than your guess.

You call a pre-defined API int guess(int num), which returns three possible results:

- -1: Your guess is higher than the number I picked (i.e. num > pick).
- 1: Your guess is lower than the number I picked (i.e. num < pick).
- 0: your guess is equal to the number I picked (i.e. num == pick).

Return the number that I picked.

Example 1:

```
Input: n = 10, pick = 6
Output: 6
```

Example 2:

```
Input: n = 1, pick = 1
Output: 1
```

Example 3:

```
Input: n = 2, pick = 1
Output: 1
```

Constraints:

- 1 <= n <= 231 - 1

- 1 <= pick <= n

Python

Python

```python
# Python solution using binary search
class Solution:
    def guessNumber(self, n: int) -> int:
        # Initialize low and high pointers
        low, high = 1, n
        # Continue binary search until low exceeds high
        while low <= high:
            # Calculate midpoint safely to prevent overflow
            mid = low + (high - low) // 2
            # Call the guess API with mid
            result = guess(mid)
            # If the guess is correct, return mid
            if result == 0:
                return mid
            # If guess was too high, adjust the high pointer
            elif result == -1:
                high = mid - 1
            # If guess was too low, adjust the low pointer
            else:
                low = mid + 1
        # Return low (ideally should never reach here if pick exists in [1, n])
        return low
```

Python

```python
# The guess API is already defined for you.
# @param num, your guess
# @return -1 if num is higher than the picked number
# 1 if num is lower than the picked number
# otherwise return 0
# def guess(num: int) -> int:


class Solution:
   def guessNumber(self, n: int) -> int:
     l = 1
     r = n

     # Find the first guess number that >= the target number
     while l < r:
       m = (l + r) // 2
       if guess(m) <= 0: # -1, 0
         r = m
       else:
         l = m + 1

     return l
```

Python


```python
# The guess API is already defined for you.
# @param num, your guess
# @return -1 if num is higher than the picked number
# 1 if num is lower than the picked number
# otherwise return 0
# def guess(num: int) -> int:


class Solution:
    def guessNumber(self, n: int) -> int:
        return bisect.bisect(range(1, n + 1), 0, key=lambda x: -guess(x))
```

Python

```python
# The guess API is already defined for you.
# @param num, your guess
# @return -1 if num is higher than the picked number
# 1 if num is lower than the picked number
# otherwise return 0
# def guess(num: int) -> int:


class Solution:
    def guessNumber(self, n: int) -> int:
        return bisect.bisect(range(1, n + 1), 0, key=lambda x: -guess(x))
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
# Python solution using binary search
class Solution:
    def guessNumber(self, n: int) -> int:
        # Initialize low and high pointers
        low, high = 1, n
        # Continue binary search until low exceeds high
        while low <= high:
            # Calculate midpoint safely to prevent overflow
            mid = low + (high - low) // 2
            # Call the guess API with mid
            result = guess(mid)
            # If the guess is correct, return mid
            if result == 0:
                return mid
            # If guess was too high, adjust the high pointer
            elif result == -1:
                high = mid - 1
            # If guess was too low, adjust the low pointer
            else:
                low = mid + 1
        # Return low (ideally should never reach here if pick exists in [1, n])
        return low
```

#704 EASY

Binary Search

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search

Lists: Grind 169

Problem:

Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1.

You must write an algorithm with O(log n) runtime complexity.

Example 1:

```
Input: nums = [-1,0,3,5,9,12], target = 9
Output: 4
Explanation: 9 exists in nums and its index is 4
```

Example 2:

```
Input: nums = [-1,0,3,5,9,12], target = 2
Output: -1
Explanation: 2 does not exist in nums so return -1
```

Constraints:

- 1 <= nums.length <= 104
- -104 < nums[i], target < 104
- All the integers in nums are unique.
- nums is sorted in ascending order.

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def search(self, nums, target):
        # Brute Force O(n) - linear scan ignoring sorted property
        for i, val in enumerate(nums):
            if val == target:
                return i
        return -1
```

Python

Python


```python
# Python code for binary search

def search(nums, target):
    # Set initial boundaries for the search
    left, right = 0, len(nums) - 1

    # Loop until the search space is empty
    while left <= right:
        # Calculate middle index to avoid overflow
        mid = left + (right - left) // 2

        # Check if the mid element matches the target
        if nums[mid] == target:
            return mid
        # If target is smaller than mid element, narrow search to left half
        elif target < nums[mid]:
            right = mid - 1
        # Else, narrow search to right half
        else:
            left = mid + 1

    # Target not found, return -1
    return -1

# Example usage:
print(search([-1,0,3,5,9,12], 9)) # Expected output: 4
```

Python

```python
class Solution:
    def search(self, nums: list[int], target: int) -> int:
        i = bisect.bisect_left(nums, target)
        return -1 if i == len(nums) or nums[i] != target else i
```

Python

```python
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        l, r = 0, len(nums) - 1
        while l < r:
            mid = (l + r) >> 1
            if nums[mid] >= target:
                r = mid
            else:
                l = mid + 1
        return l if nums[l] == target else -1
```

Python

```python
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1
        while left < right:
            mid = (left + right) >> 1
            if nums[mid] >= target:
                right = mid
            else:
                left = mid + 1
        return left if nums[left] == target else -1
```


[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
# Python code for binary search

def search(nums, target):
    # Set initial boundaries for the search
    left, right = 0, len(nums) - 1

    # Loop until the search space is empty
    while left <= right:
        # Calculate middle index to avoid overflow
        mid = left + (right - left) // 2

        # Check if the mid element matches the target
        if nums[mid] == target:
            return mid
        # If target is smaller than mid element, narrow search to left half
        elif target < nums[mid]:
            right = mid - 1
        # Else, narrow search to right half
        else:
            left = mid + 1

    # Target not found, return -1
    return -1

# Example usage:
print(search([-1,0,3,5,9,12], 9)) # Expected output: 4
```

#33 MEDIUM

Search in Rotated Sorted Array

LeetCode · SimpleLast · WalkCC · LeetCode · Editorial

Array · Binary Search

Lists: Grind 169

Problem:

There is an integer array nums sorted in ascending order (with distinct values).

Prior to being passed to your function, nums is possibly left rotated at an unknown index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (0-indexed). For example, [0,1,2,4,5,6,7] might be left rotated by 3 indices and become [4,5,6,7,0,1,2].

Given the array nums after the possible rotation and an integer target, return the index of target if it is in nums, or -1 if it is not in nums.

You must write an algorithm with O(log n) runtime complexity.

Example 1:

```
Input: nums = [4,5,6,7,0,1,2], target = 0
Output: 4
```


Example 2:

```
Input: nums = [4,5,6,7,0,1,2], target = 3
Output: -1
```

Example 3:

```
Input: nums = [1], target = 0
Output: -1
```

Constraints:

- 1 <= nums.length <= 5000
- -104 <= nums[i] <= 104
- All values of nums are unique.
- nums is an ascending array that is possibly rotated.
- -104 <= target <= 104

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def search(self, nums, target):
        # Brute Force O(n) - linear scan
        for i, val in enumerate(nums):
            if val == target:
                return i
        return -1
```

Python

Python

```python
# Define a function to search for the target in a rotated sorted array.
def search(nums, target):
    # Initialize pointers for binary search.
    left, right = 0, len(nums) - 1

    # Loop until the search space is exhausted.
    while left <= right:
        # Compute the middle index.
        mid = (left + right) // 2

        # If the mid element is the target, return its index.
        if nums[mid] == target:
            return mid

        # Check if the left part of the array is sorted.
        if nums[left] <= nums[mid]:
            # If the target is within the sorted left half, adjust the right pointer.
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                # Otherwise, search the right half.
                left = mid + 1
        else:
            # The right half must be sorted.
            if nums[mid] < target <= nums[right]:
```



```
                left = mid + 1
            else:
                right = mid - 1

    # If the target is not found, return -1.
    return -1

# Example usage:
# result = search([4,5,6,7,0,1,2], 0)
# print(result) # Expected output: 4
```

Python

```python
class Solution:
    def search(self, nums: list[int], target: int) -> int:
        l = 0
        r = len(nums) - 1

        while l <= r:
            m = (l + r) // 2
            if nums[m] == target:
                return m
            if nums[l] <= nums[m]: # nums[l..m] are sorted.
                if nums[l] <= target < nums[m]:
                    r = m - 1
                else:
                    l = m + 1
            else: # nums[m..n - 1] are sorted.
                if nums[m] < target <= nums[r]:
                    l = m + 1
                else:
                    r = m - 1

        return -1
```

Python

```python
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        n = len(nums)
        left, right = 0, n - 1
        while left < right:
            mid = (left + right) >> 1
            if nums[0] <= nums[mid]:
                if nums[0] <= target <= nums[mid]:
                    right = mid
                else:
                    left = mid + 1
            else:
                if nums[mid] < target <= nums[n - 1]:
                    left = mid + 1
                else:
                    right = mid
        return left if nums[left] == target else -1
```

Python


```python
# below 2 solutions, diff is while condition: left '<' or '<=' right

# I like this better
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        n = len(nums)
        left, right = 0, n - 1
        while left <= right:
            mid = (left + right) >> 1
            if nums[mid] == target:
                return mid
            elif nums[0] <= nums[mid]: # left half sorted
                if nums[0] <= target <= nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            else: # right half sorted
                if nums[mid] < target <= nums[n - 1]:
                    left = mid + 1
                else:
                    right = mid -1
        return -1

###########

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        n = len(nums)
        left, right = 0, n - 1
        while left < right:
            mid = (left + right) >> 1
            if nums[0] <= nums[mid]: # left half sorted
                if nums[0] <= target <= nums[mid]:
                    right = mid
                else:
                    left = mid + 1
            else: # right half sorted
                if nums[mid] < target <= nums[n - 1]:
                    left = mid + 1
                else:
                    right = mid
        return left if nums[left] == target else -1
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
# Define a function to search for the target in a rotated sorted array.
def search(nums, target):
    # Initialize pointers for binary search.
    left, right = 0, len(nums) - 1

    # Loop until the search space is exhausted.
    while left <= right:
        # Compute the middle index.
        mid = (left + right) // 2

        # If the mid element is the target, return its index.
        if nums[mid] == target:
            return mid

        # Check if the left part of the array is sorted.
        if nums[left] <= nums[mid]:
            # If the target is within the sorted left half, adjust the right pointer.
            if nums[left] <= target < nums[mid]:
                right = mid - 1
            else:
                # Otherwise, search the right half.
                left = mid + 1
        else:
            # The right half must be sorted.
            if nums[mid] < target <= nums[right]:

                left = mid + 1
            else:
                right = mid - 1

    # If the target is not found, return -1.
    return -1

# Example usage:
# result = search([4,5,6,7,0,1,2], 0)
# print(result) # Expected output: 4
```

#34 MEDIUM

Find First and Last Position of Element in Sorted Array

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search

Lists: Denny Zhang

Problem:

Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

You must write an algorithm with O(log n) runtime complexity.

Example 1:

```
Input: nums = [5,7,7,8,8,10], target = 8
Output: [3,4]
```


Example 2:

```
Input: nums = [5,7,7,8,8,10], target = 6
Output: [-1,-1]
```

Example 3:

```
Input: nums = [], target = 0
Output: [-1,-1]
```

Constraints:

- 0 <= nums.length <= 105
- -109 <= nums[i] <= 109
- nums is a non-decreasing array.
- -109 <= target <= 109

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def searchRange(self, nums, target):
        # Brute Force O(n) - linear scan instead of binary search
        for i, val in enumerate(nums):
            if val == target:
                return i
        return -1
```

Python

Python

```python
def searchRange(nums, target):
    # Helper function to find the boundary index (leftmost or rightmost) for target.
    def findBoundary(isLeft):
        left, right = 0, len(nums) - 1
        boundary = -1
        while left <= right:
            mid = (left + right) // 2
            # If the target is found, update boundary and continue to search on one side.
            if nums[mid] == target:
                boundary = mid
                if isLeft:
                    right = mid - 1 # Continue search on left half
                else:
                    left = mid + 1 # Continue search on right half
            elif nums[mid] < target:
                left = mid + 1
            else:
                right = mid - 1
        return boundary

    # Find the leftmost index
    leftIndex = findBoundary(True)
    if leftIndex == -1:
        return [-1, -1]
    # Find the rightmost index
```



```python
    rightIndex = findBoundary(False)
    return [leftIndex, rightIndex]

# Example usage:
# nums = [5,7,7,8,8,10]
# target = 8
# print(searchRange(nums, target)) # Output: [3,4]
```

Python

```python
class Solution:
    def searchRange(self, nums: list[int], target: int) -> list[int]:
        l = bisect_left(nums, target)
        if l == len(nums) or nums[l] != target:
            return -1, -1
        r = bisect_right(nums, target) - 1
        return l, r
```

Python

```python
class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        l = bisect_left(nums, target)
        r = bisect_left(nums, target + 1)
        return [-1, -1] if l == r else [l, r - 1]
```

Python

```python
>>> bisect.bisect_left([1,1,1,2,2,2,7,7,7], 2)
3
>>> bisect.bisect_left([1,1,1,2,2,2,7,7,7], 2+1)
6


>>> bisect.bisect_right([1,1,1,2,2,2,7,7,7], 2)
6
>>> bisect.bisect_right([1,1,1,2,2,2,7,7,7], 2+1)
6


>>> bisect.bisect_left([1,1,1,2,2,2,7,7,7], 0)
0
>>> bisect.bisect_left([1,1,1,2,2,2,7,7,7], 1)
0

# below, even single value for 2, after +1 the index will be different
>>> bisect.bisect_left([1,2,3,4,5], 2)
1
>>> bisect.bisect_left([1,2,3,4,5], 2+1)
2
'''
```



```python
import bisect

class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        l = bisect_left(nums, target)
        r = bisect_left(nums, target + 1)
        return [-1, -1] if l == r else [l, r - 1]

###########

class Solution:
    def searchRange(self, nums: List[int], target: int) -> List[int]:
        def findRange(to_left):
            l, r = 0, len(nums) - 1
            m = 0
            while l <= r:
                m = l + (r - l) // 2
                if nums[m] < target:
                    l = m + 1
                elif nums[m] > target:
                    r = m - 1
                elif to_left and m > 0 and nums[m - 1] == nums[m]: # so now nums[m] == target, but maybe
not the leftmost or rightmost
                    r = m - 1
                elif not to_left and m + 1 < len(nums) and nums[m + 1] == nums[m]: # so now nums[m] ==
target, but maybe not the leftmost or rightmost
                    l = m + 1

            else:
                return m
            if m < len(nums) and nums[m] == target:
            # 1. cover not-found,
            # or 2. cover input=[3] and target=3, first and last both 0, return [0,0]
                return m
            else:
                return -1
        return [findRange(True), findRange(False)]
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python


```python
def searchRange(nums, target):
    # Helper function to find the boundary index (leftmost or rightmost) for target.
    def findBoundary(isLeft):
        left, right = 0, len(nums) - 1
        boundary = -1
        while left <= right:
            mid = (left + right) // 2
            # If the target is found, update boundary and continue to search on one side.
            if nums[mid] == target:
                boundary = mid
                if isLeft:
                    right = mid - 1 # Continue search on left half
                else:
                    left = mid + 1 # Continue search on right half
            elif nums[mid] < target:
                left = mid + 1
            else:
                right = mid - 1
        return boundary

    # Find the leftmost index
    leftIndex = findBoundary(True)
    if leftIndex == -1:
        return [-1, -1]
    # Find the rightmost index

    rightIndex = findBoundary(False)
    return [leftIndex, rightIndex]

# Example usage:
# nums = [5,7,7,8,8,10]
# target = 8
# print(searchRange(nums, target)) # Output: [3,4]
```

#153 MEDIUM

Find Minimum in Rotated Sorted Array

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search

Lists: Grind 169

Problem:

Suppose an array of length n sorted in ascending order is rotated between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

- [4,5,6,7,0,1,2] if it was rotated 4 times.
- [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that rotating an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of unique elements, return the minimum element of this array.

You must write an algorithm that runs in O(log n) time.

Example 1:


```
Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.
```

Example 2:

```
Input: nums = [4,5,6,7,0,1,2]
Output: 0
Explanation: The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.
```

Example 3:

```
Input: nums = [11,13,15,17]
Output: 11
Explanation: The original array was [11,13,15,17] and it was rotated 4 times.
```

Constraints:

- n == nums.length
- 1 <= n <= 5000
- -5000 <= nums[i] <= 5000
- All the integers of nums are unique.
- nums is sorted and rotated between 1 and n times.

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findMin(self, nums):
        # Brute Force O(n) - just return the minimum (ignores sorted structure)
        return min(nums)
```

Python

Python

```python
def findMin(nums):
    # Initialize left and right pointers
    left, right = 0, len(nums) - 1
    # Continue while the search space is valid
    while left < right:
        mid = (left + right) // 2 # Compute middle index
        # If middle element is greater than the rightmost element,
        # the minimum must be in the right half
        if nums[mid] > nums[right]:
            left = mid + 1
        else:
            # Otherwise, the minimum is in the left half (including mid)
            right = mid
    # When loop ends, left == right, pointing to the minimum element
    return nums[left]
```

Python

```python
class Solution:
    def findMin(self, nums: list[int]) -> int:
        l = 0
        r = len(nums) - 1

        while l < r:
            m = (l + r) // 2
            if nums[m] < nums[r]:
                r = m
            else:
                l = m + 1

        return nums[l]
```

Python

```python
class Solution:
    def findMin(self, nums: List[int]) -> int:
        if nums[0] <= nums[-1]:
            return nums[0]
        left, right = 0, len(nums) - 1
        while left < right:
            mid = (left + right) >> 1
            if nums[0] <= nums[mid]:
                left = mid + 1
            else:
                right = mid
        return nums[left]
```

Python

```python
class Solution:
    def findMin(self, nums: List[int]) -> int:
        if nums[0] <= nums[-1]:
            return nums[0]
        left, right = 0, len(nums) - 1
        while left < right:
            mid = (left + right) >> 1
            if nums[0] <= nums[mid]:
                left = mid + 1
            else:
                right = mid
        return nums[left]
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python


```python
def findMin(nums):
    # Initialize left and right pointers
    left, right = 0, len(nums) - 1
    # Continue while the search space is valid
    while left < right:
        mid = (left + right) // 2 # Compute middle index
        # If middle element is greater than the rightmost element,
        # the minimum must be on the right half
        if nums[mid] > nums[right]:
            left = mid + 1
        else:
            # Otherwise, the minimum is in the left half (including mid)
            right = mid
    # When loop ends, left == right, pointing to the minimum element
    return nums[left]
```

#300 MEDIUM

Longest Increasing Subsequence

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search · Dynamic Programming

Lists: Grind 169

Problem:

Given an integer array nums, return the length of the longest strictly increasing subsequence.

Example 1:

```
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.
```

Example 2:

```
Input: nums = [0,1,0,3,2,3]
Output: 4
```

Example 3:

```
Input: nums = [7,7,7,7,7,7,7]
Output: 1
```

Constraints:

- 1 <= nums.length <= 2500
- -104 <= nums[i] <= 104

Follow up: Can you come up with an algorithm that runs in O(n log(n)) time complexity?

>> Brute Force [Binary Search] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def lengthOfLIS(self, nums):
        # Brute Force O(2^n) - generate every subsequence, find longest increasing
        def lis(i, prev_val):
            if i == len(nums): return 0
            skip = lis(i + 1, prev_val)
            take = 0
            if nums[i] > prev_val:
                take = 1 + lis(i + 1, nums[i])
            return max(skip, take)
        return lis(0, float('-inf'))
```

Python

Python

```python
import bisect

def lengthOfLIS(nums):
    dp = [] # O(n) array stores smallest tail for each subsequence length
    for num in nums:
        # Find the leftmost insertion point using binary search
        idx = bisect.bisect_left(dp, num)
        if idx == len(dp):
            dp.append(num) # Extend dp if no larger tail exists
        else:
            dp[idx] = num # Replace to maintain the smallest tail
    return len(dp)

# Example usage:
# print(lengthOfLIS([10,9,2,5,3,7,101,18])) # Output: 4
```

Python

```python
class Solution:
    def lengthOfLIS(self, nums: list[int]) -> int:
        if not nums:
            return 0

        # dp[i] := the length of LIS ending in nums[i]
        dp = [1] * len(nums)

        for i in range(1, len(nums)):
            for j in range(i):
                if nums[j] < nums[i]:
                    dp[i] = max(dp[i], dp[j] + 1)

        return max(dp)
```

Python


```python
class Solution:
    def lengthOfLIS(self, nums: list[int]) -> int:
        # tails[i] := the minimum tails of all the increasing subsequences having
        # length i + 1
        tails = []

        for num in nums:
          if not tails or num > tails[-1]:
            tails.append(num)
          else:
            tails[bisect.bisect_left(tails, num)] = num

        return len(tails)
```

Python

```python
class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        n = len(nums)
        f = [1] * n
        for i in range(1, n):
            for j in range(i):
                if nums[j] < nums[i]:
                    f[i] = max(f[i], f[j] + 1)
        return max(f)
```

Python

```python
class BinaryIndexedTree:
    def __init__(self, n: int):
        self.n = n
        self.c = [0] * (n + 1)

    def update(self, x: int, v: int):
        while x <= self.n:
            self.c[x] = max(self.c[x], v)
            x += x & -x

    def query(self, x: int) -> int:
        mx = 0
        while x:
            mx = max(mx, self.c[x])
            x -= x & -x
        return mx


class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        s = sorted(set(nums))
        m = len(s)
        tree = BinaryIndexedTree(m)
        for x in nums:
            x = bisect_left(s, x) + 1

            t = tree.query(x - 1) + 1
            tree.update(x, t)
        return tree.query(m)
```

Python


```python
# find the largest end element in tails that is smaller than nums[i]
# and then replace it with nums[i] and discard the list in the same length
# which is implemented by `tail[idx] = num`

class Solution(object):
    def lengthOfLIS(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        tail = []
        for num in nums:
            # if using bisect_right(tail, num),
            # then input=[7,7,7,7,7,7,7] will output 7 but expected result is 1
            idx = bisect.bisect_left(tail, num)
            if idx == len(tail): # same as in java: if (i == len) len++;
                tail.append(num)
            else:
                tail[idx] = num
        return len(tail)


# implementation of bisect.bisect_left()
# similar to Leetcode-302, find left/right/top/bottom callable
def bisect_left(a, x, lo=0, hi=None):

    if hi is None:
        hi = len(a)

    while lo < hi:
        mid = (lo + hi) // 2
        if a[mid] < x:
            lo = mid + 1
        else:
            hi = mid

    return lo

###########

class BinaryIndexedTree:
    def __init__(self, n: int):
        self.n = n
        self.c = [0] * (n + 1)

    def update(self, x: int, v: int):
        while x <= self.n:
            self.c[x] = max(self.c[x], v)
            x += x & -x

    def query(self, x: int) -> int:
```



```python
        mx = 0
        while x:
            mx = max(mx, self.c[x])
            x -= x & -x
        return mx


class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        s = sorted(set(nums))
        m = len(s)
        tree = BinaryIndexedTree(m)
        for x in nums:
            x = bisect_left(s, x) + 1
            t = tree.query(x - 1) + 1
            tree.update(x, t)
        return tree.query(m)
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
# find the largest end element in tails that is smaller than nums[i]
# and then replace it with nums[i] and discard the list in the same length
# which is implemented by `tail[idx] = num`

class Solution(object):
    def lengthOfLIS(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        tail = []
        for num in nums:
            # if using bisect_right(tail, num),
            # then input=[7,7,7,7,7,7,7] will output 7 but expected result is 1
            idx = bisect.bisect_left(tail, num)
            if idx == len(tail): # same as in java: if (i == len) len++;
                tail.append(num)
            else:
                tail[idx] = num
        return len(tail)


# implementation of bisect.bisect_left()
# similar to Leetcode-302, find left/right/top/bottom callable
def bisect_left(a, x, lo=0, hi=None):
```

```python
    if hi is None:
        hi = len(a)

    while lo < hi:
        mid = (lo + hi) // 2
        if a[mid] < x:
            lo = mid + 1
        else:
            hi = mid

    return lo

###########

class BinaryIndexedTree:
    def __init__(self, n: int):
        self.n = n
        self.c = [0] * (n + 1)

    def update(self, x: int, v: int):
        while x <= self.n:
            self.c[x] = max(self.c[x], v)
            x += x & -x

    def query(self, x: int) -> int:

        mx = 0
        while x:
            mx = max(mx, self.c[x])
            x -= x & -x
        return mx


class Solution:
    def lengthOfLIS(self, nums: List[int]) -> int:
        s = sorted(set(nums))
        m = len(s)
        tree = BinaryIndexedTree(m)
        for x in nums:
            x = bisect_left(s, x) + 1
            t = tree.query(x - 1) + 1
            tree.update(x, t)
        return tree.query(m)
```

#362 MEDIUM

Design Hit Counter

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Binary Search · Design · Queue

Lists: Grind 169

Problem:

Design a hit counter which counts the number of hits received in the past 5 minutes (i.e., the past 300 seconds).

Your system should accept a timestamp parameter (in seconds granularity), and you may assume that calls are being made to the system in chronological order (i.e., timestamp is monotonically increasing). Several hits may arrive roughly at the same time.

Implement the HitCounter class:

- HitCounter() Initializes the object of the hit counter system.
- void hit(int timestamp) Records a hit that happened at timestamp (in seconds). Several hits may happen at the same timestamp.
- int getHits(int timestamp) Returns the number of hits in the past 5 minutes from timestamp (i.e., the past 300 seconds).

Example 1:

```
Input
["HitCounter", "hit", "hit", "hit", "getHits", "hit", "getHits", "getHits"]
[[], [1], [2], [3], [4], [300], [300], [301]]
Output
[null, null, null, null, 3, null, 4, 3]

Explanation
HitCounter hitCounter = new HitCounter();
hitCounter.hit(1); // hit at timestamp 1.
hitCounter.hit(2); // hit at timestamp 2.
hitCounter.hit(3); // hit at timestamp 3.
hitCounter.getHits(4); // get hits at timestamp 4, return 3.
hitCounter.hit(300); // hit at timestamp 300.
hitCounter.getHits(300); // get hits at timestamp 300, return 4.
hitCounter.getHits(301); // get hits at timestamp 301, return 3.
```

Constraints:

- 1 <= timestamp <= 2 * 109
- All the calls are being made to the system in chronological order (i.e., timestamp is monotonically increasing).
- At most 300 calls will be made to hit and getHits.

Follow up: What if the number of hits per second could be huge? Does your design scale?

Python

Python


```python
# Python implementation of HitCounter

from collections import deque

class HitCounter:
    def __init__(self):
        # Initialize the deque to store (timestamp, count) pairs.
        self.hits_queue = deque()

    def hit(self, timestamp: int) -> None:
        # If the last recorded hit is at the same timestamp, increment its count.
        if self.hits_queue and self.hits_queue[-1][0] == timestamp:
            self.hits_queue[-1][1] += 1
        else:
            # Append new (timestamp, count) pair.
            self.hits_queue.append((timestamp, 1))

    def getHits(self, timestamp: int) -> int:
        # Remove outdated hits from the front of the queue.
        while self.hits_queue and self.hits_queue[0][0] <= timestamp - 300:
            self.hits_queue.popleft()
        # Sum counts of the remaining hit entries.
        return sum(count for _, count in self.hits_queue)
```

Python

```python
class HitCounter:
  def __init__(self):
    self.timestamps = [0] * 300
    self.hits = [0] * 300

  def hit(self, timestamp: int) -> None:
    i = timestamp % 300
    if self.timestamps[i] == timestamp:
      self.hits[i] += 1
    else:
      self.timestamps[i] = timestamp
      self.hits[i] = 1 # Reset the hit count to 1.

  def getHits(self, timestamp: int) -> int:
    return sum(h for t, h in zip(self.timestamps, self.hits)
               if timestamp - t < 300)
```

Python


```python
class HitCounter:

    def __init__(self):
        self.ts = []

    def hit(self, timestamp: int) -> None:
        self.ts.append(timestamp)

    def getHits(self, timestamp: int) -> int:
        return len(self.ts) - bisect_left(self.ts, timestamp - 300 + 1)

# Your HitCounter object will be instantiated and called as such:
# obj = HitCounter()
# obj.hit(timestamp)
# param_2 = obj.getHits(timestamp)
```

Python

```python
from collections import deque

class HitCounter: # queue

    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.hits = deque()

    def hit(self, timestamp: int) -> None:
        """
        Record a hit.
        @param timestamp - The current timestamp (in seconds granularity).
        """
        self.hits.append(timestamp)

    def getHits(self, timestamp: int) -> int:
        """
        Return the number of hits in the past 5 minutes.
        @param timestamp - The current timestamp (in seconds granularity).
        """
        while self.hits and self.hits[0] <= timestamp - 300:
            self.hits.popleft()
        return len(self.hits)
```



```python
class HitCounter: # Follow-up
    def __init__(self):
        self.times = [0] * 300
        self.hits = [0] * 300

    def hit(self, timestamp: int) -> None:
        idx = timestamp % 300
        if self.times[idx] != timestamp:
            self.times[idx] = timestamp
            self.hits[idx] = 1
        else:
            self.hits[idx] += 1

    def getHits(self, timestamp: int) -> int:
        res = 0
        for i in range(300):
            if timestamp - self.times[i] < 300:
                res += self.hits[i]
        return res

# Your HitCounter object will be instantiated and called as such:
# obj = HitCounter()
# obj.hit(timestamp)

# param_2 = obj.getHits(timestamp)

#############

class HitCounter: # extra O(N) space
    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.counter = Counter()

    def hit(self, timestamp: int) -> None:
        """
        Record a hit.
        @param timestamp - The current timestamp (in seconds granularity).
        """
        self.counter[timestamp] += 1

    def getHits(self, timestamp: int) -> int:
        """
        Return the number of hits in the past 5 minutes.
        @param timestamp - The current timestamp (in seconds granularity).
        """
        return sum([v for t, v in self.counter.items() if t + 300 > timestamp])

# Your HitCounter object will be instantiated and called as such:
# obj = HitCounter()
# obj.hit(timestamp)
# param_2 = obj.getHits(timestamp)
```

[*] Binary Search — Pattern Solution (stored optimal)

Python

```python
# Python implementation of HitCounter

from collections import deque

class HitCounter:
    def __init__(self):
        # Initialize the deque to store (timestamp, count) pairs.
        self.hits_queue = deque()

    def hit(self, timestamp: int) -> None:
        # If the last recorded hit is at the same timestamp, increment its count.
        if self.hits_queue and self.hits_queue[-1][0] == timestamp:
            self.hits_queue[-1][1] += 1
        else:
            # Append new (timestamp, count) pair.
            self.hits_queue.append((timestamp, 1))

    def getHits(self, timestamp: int) -> int:
        # Remove outdated hits from the front of the queue.
        while self.hits_queue and self.hits_queue[0][0] <= timestamp - 300:
            self.hits_queue.popleft()
        # Sum counts of the remaining hit entries.
        return sum(count for _, count in self.hits_queue)
```

#528 MEDIUM

Random Pick with Weight

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math · Binary Search · Prefix Sum · Randomized

Lists: Grind 169

Problem:

You are given a 0-indexed array of positive integers w where w[i] describes the weight of the ith index.

You need to implement the function pickIndex(), which randomly picks an index in the range [0, w.length − 1] (inclusive) and returns it. The probability of picking an index i is w[i] / sum(w).

- For example, if w = [1, 3], the probability of picking index 0 is 1 / (1 + 3) = 0.25 (i.e., 25%), and the probability of picking index 1 is 3 / (1 + 3) = 0.75 (i.e., 75%).

Example 1:

```
Input
["Solution","pickIndex"]
[[[1]],[]]
Output
[null,0]

Explanation
Solution solution = new Solution([1]);
solution.pickIndex(); // return 0. The only option is to return 0 since there is only one
element in w.
```

Example 2:

```
Input
["Solution","pickIndex","pickIndex","pickIndex","pickIndex","pickIndex"]
[[[1,3]],[],[],[],[],[]]
Output
[null,1,1,1,1,0]

Explanation
Solution solution = new Solution([1, 3]);
solution.pickIndex(); // return 1. It is returning the second element (index = 1) that has a
probability of 3/4.
solution.pickIndex(); // return 1
solution.pickIndex(); // return 1
solution.pickIndex(); // return 1
solution.pickIndex(); // return 0. It is returning the first element (index = 0) that has a
probability of 1/4.

Since this is a randomization problem, multiple answers are allowed.
All of the following outputs can be considered correct:
[null,1,1,1,1,0]
[null,1,1,1,1,1]
[null,1,1,1,0,0]
[null,1,1,1,0,1]
[null,1,0,1,0,0]
......
and so on.
```

Constraints:

- 1 <= w.length <= 104
- 1 <= w[i] <= 105
- pickIndex will be called at most 104 times.

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def __init__(self, w):
        # Brute Force O(n) - linear scan instead of binary search
        for i, val in enumerate(w):
            if val == target:
                return i
        return -1
```

Python

Python


```python
import random
import bisect

class Solution:
    def __init__(self, w):
        # Build the prefix sum array.
        self.prefix_sums = []
        current_sum = 0
        for weight in w:
            current_sum += weight
            self.prefix_sums.append(current_sum)
        # Store the total sum of weights.
        self.total_sum = current_sum

    def pickIndex(self):
        # Get a random target in the range [1, total_sum]
        target = random.randint(1, self.total_sum)
        # Use bisect_left to find the proper index.
        index = bisect.bisect_left(self.prefix_sums, target)
        return index

# Example usage:
# solution = Solution([1, 3])
# print(solution.pickIndex())
```

Python

```python
class Solution:
  def __init__(self, w: list[int]):
    self.prefix = list(itertools.accumulate(w))

  def pickIndex(self) -> int:
    target = random.randint(0, self.prefix[-1] - 1)
    return bisect.bisect_right(range(len(self.prefix)), target,
                               key=lambda m: self.prefix[m])
```

Python

```python
class Solution:
    def __init__(self, w: List[int]):
        self.s = [0]
        for c in w:
            self.s.append(self.s[-1] + c)

    def pickIndex(self) -> int:
        x = random.randint(1, self.s[-1])
        left, right = 1, len(self.s) - 1
        while left < right:
            mid = (left + right) >> 1
            if self.s[mid] >= x:
                right = mid
            else:
                left = mid + 1
        return left - 1


# Your Solution object will be instantiated and called as such:
# obj = Solution(w)
# param_1 = obj.pickIndex()
```


Python

```python
class Solution:
    def __init__(self, w: List[int]):
        self.s = [0]
        for c in w:
            self.s.append(self.s[-1] + c)

    def pickIndex(self) -> int:
        x = random.randint(1, self.s[-1])
        left, right = 1, len(self.s) - 1
        while left < right:
            mid = (left + right) >> 1
            if self.s[mid] >= x:
                right = mid
            else:
                left = mid + 1
        return left - 1


# Your Solution object will be instantiated and called as such:
# obj = Solution(w)
# param_1 = obj.pickIndex()
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
import random
import bisect

class Solution:
    def __init__(self, w):
        # Build the prefix sum array.
        self.prefix_sums = []
        current_sum = 0
        for weight in w:
            current_sum += weight
            self.prefix_sums.append(current_sum)
        # Store the total sum of weights.
        self.total_sum = current_sum

    def pickIndex(self):
        # Get a random target in the range [1, total_sum]
        target = random.randint(1, self.total_sum)
        # Use bisect_left to find the proper index.
        index = bisect.bisect_left(self.prefix_sums, target)
        return index

# Example usage:
# solution = Solution([1, 3])
# print(solution.pickIndex())
```

#852 MEDIUM

Peak Index in a Mountain Array

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search

Lists: Denny Zhang

Problem:

You are given an integer mountain array arr of length n where the values increase to a peak element and then decrease.

Return the index of the peak element.

Your task is to solve it in O(log(n)) time complexity.

Example 1:

Input: arr = [0,1,0]

Output: 1

Example 2:

Input: arr = [0,2,1,0]

Output: 1

Example 3:

Input: arr = [0,10,5,2]

Output: 1

Constraints:

- 3 <= arr.length <= 105
- 0 <= arr[i] <= 106
- arr is guaranteed to be a mountain array.

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def peakIndexInMountainArray(self, arr):
        # Brute Force O(n) - linear scan instead of binary search
        for i, val in enumerate(arr):
            if val == target:
                return i
        return -1
```

Python

Python


```python
def peakIndexInMountainArray(arr):
    # Initialize pointers for the binary search
    left, right = 0, len(arr) - 1
    # Continue while there is more than one element in the search space
    while left < right:
        mid = (left + right) // 2 # Find the mid index
        # If the sequence is still increasing, move the left pointer to mid+1
        if arr[mid] < arr[mid + 1]:
            left = mid + 1
        else:
            # Otherwise, the peak is at mid or to the left of mid
            right = mid
    # When left and right converge, we have the peak index
    return left

# Example usage:
# print(peakIndexInMountainArray([0, 1, 0])) # Output: 1
```

Python

```python
class Solution:
  def peakIndexInMountainArray(self, arr: list[int]) -> int:
    l = 0
    r = len(arr) - 1

    while l < r:
      m = (l + r) // 2
      if arr[m] >= arr[m + 1]:
        r = m
      else:
        l = m + 1

    return l
```

Python

```python
class Solution:
    def peakIndexInMountainArray(self, arr: List[int]) -> int:
        left, right = 1, len(arr) - 2
        while left < right:
            mid = (left + right) >> 1
            if arr[mid] > arr[mid + 1]:
                right = mid
            else:
                left = mid + 1
        return left
```

Python

```python
class Solution:
    def peakIndexInMountainArray(self, arr: List[int]) -> int:
        left, right = 1, len(arr) - 2
        while left < right:
            mid = (left + right) >> 1
            if arr[mid] > arr[mid + 1]:
                right = mid
            else:
                left = mid + 1
        return left
```


[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
def peakIndexInMountainArray(arr):
    # Initialize pointers for the binary search
    left, right = 0, len(arr) - 1
    # Continue while there is more than one element in the search space
    while left < right:
        mid = (left + right) // 2 # Find the mid index
        # If the sequence is still increasing, move the left pointer to mid+1
        if arr[mid] < arr[mid + 1]:
            left = mid + 1
        else:
            # Otherwise, the peak is at mid or to the left of mid
            right = mid
    # When left and right converge, we have the peak index
    return left

# Example usage:
# print(peakIndexInMountainArray([0, 1, 0])) # Output: 1
```

#981 MEDIUM

Time Based Key-Value Store

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · String · Binary Search · Design

Lists: Grind 169

Problem:

Design a time-based key-value data structure that can store multiple values for the same key at different time stamps and retrieve the key's value at a certain timestamp.

Implement the TimeMap class:

- TimeMap() Initializes the object of the data structure.
- void set(String key, String value, int timestamp) Stores the key key with the value value at the given time timestamp.
- String get(String key, int timestamp) Returns a value such that set was called previously, with timestamp_prev <= timestamp. If there are multiple such values, it returns the value associated with the largest timestamp_prev. If there are no values, it returns "".

Example 1:

```
Input
["TimeMap", "set", "get", "get", "set", "get", "get"]
[[], ["foo", "bar", 1], ["foo", 1], ["foo", 3], ["foo", "bar2", 4], ["foo", 4], ["foo", 5]]
Output
[null, null, "bar", "bar", null, "bar2", "bar2"]

Explanation
TimeMap timeMap = new TimeMap();
timeMap.set("foo", "bar", 1); // store the key "foo" and value "bar" along with timestamp =
1.
timeMap.get("foo", 1); // return "bar"
```

```
timeMap.get("foo", 3); // return "bar", since there is no value corresponding to foo at
timestamp 3 and timestamp 2, then the only value is at timestamp 1 is "bar".
timeMap.set("foo", "bar2", 4); // store the key "foo" and value "bar2" along with timestamp =
4.
timeMap.get("foo", 4); // return "bar2"
timeMap.get("foo", 5); // return "bar2"
```

Constraints:

- 1 <= key.length, value.length <= 100
- key and value consist of lowercase English letters and digits.
- 1 <= timestamp <= 107
- All the timestamps timestamp of set are strictly increasing.
- At most 2 ^ 105 calls will be made to set and get.

Python

Python

```python
# TimeMap class in Python with line-by-line comments.
class TimeMap:
    def __init__(self):
        # Initialize the dictionary to hold key and list of (timestamp, value) pairs.
        self.store = {}

    def set(self, key: str, value: str, timestamp: int) -> None:
        # Append the new (timestamp, value) pair for the given key.
        if key not in self.store:
            self.store[key] = []
        self.store[key].append((timestamp, value))

    def get(self, key: str, timestamp: int) -> str:
        # Return the value for the largest timestamp <= the given timestamp.
        if key not in self.store:
            return ""

        values = self.store[key]
        left, right = 0, len(values) - 1

        # Binary search for the rightmost index with timestamp <= given timestamp.
        while left <= right:
            mid = (left + right) // 2
            if values[mid][0] <= timestamp:
                left = mid + 1
```



```python
            else:
                right = mid - 1

        # If no valid timestamp exists, right will be -1.
        if right == -1:
            return ""
        return values[right][1]

# Example usage:
# timeMap = TimeMap()
# timeMap.set("foo", "bar", 1)
# print(timeMap.get("foo", 1)) # Output: "bar"
# print(timeMap.get("foo", 3)) # Output: "bar"
# timeMap.set("foo", "bar2", 4)
# print(timeMap.get("foo", 4)) # Output: "bar2"
# print(timeMap.get("foo", 5)) # Output: "bar2"
```

Python

```python
class TimeMap:
    def __init__(self):
        self.values = collections.defaultdict(list)
        self.timestamps = collections.defaultdict(list)

    def set(self, key: str, value: str, timestamp: int) -> None:
        self.values[key].append(value)
        self.timestamps[key].append(timestamp)

    def get(self, key: str, timestamp: int) -> str:
        if key not in self.timestamps:
            return ''
        i = bisect.bisect(self.timestamps[key], timestamp)
        return self.values[key][i - 1] if i > 0 else ''
```

Python

```python
class TimeMap:
    def __init__(self):
        self.ktv = defaultdict(list)

    def set(self, key: str, value: str, timestamp: int) -> None:
        self.ktv[key].append((timestamp, value))

    def get(self, key: str, timestamp: int) -> str:
        if key not in self.ktv:
            return ''
        tv = self.ktv[key]
        i = bisect_right(tv, (timestamp, chr(127)))
        return tv[i - 1][1] if i else ''


# Your TimeMap object will be instantiated and called as such:
# obj = TimeMap()
# obj.set(key,value,timestamp)
# param_2 = obj.get(key,timestamp)
```

Python


```python
class TimeMap:
    def __init__(self):
        self.ktv = defaultdict(list)

    def set(self, key: str, value: str, timestamp: int) -> None:
        self.ktv[key].append((timestamp, value))

    def get(self, key: str, timestamp: int) -> str:
        if key not in self.ktv:
            return ''
        tv = self.ktv[key]
        i = bisect_right(tv, (timestamp, chr(127)))
        return tv[i - 1][1] if i else ''


# Your TimeMap object will be instantiated and called as such:
# obj = TimeMap()
# obj.set(key,value,timestamp)
# param_2 = obj.get(key,timestamp)
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
# TimeMap class in Python with line-by-line comments.
class TimeMap:
    def __init__(self):
        # Initialize the dictionary to hold key and list of (timestamp, value) pairs.
        self.store = {}

    def set(self, key: str, value: str, timestamp: int) -> None:
        # Append the new (timestamp, value) pair for the given key.
        if key not in self.store:
            self.store[key] = []
        self.store[key].append((timestamp, value))

    def get(self, key: str, timestamp: int) -> str:
        # Return the value for the largest timestamp <= the given timestamp.
        if key not in self.store:
            return ""

        values = self.store[key]
        left, right = 0, len(values) - 1

        # Binary search for the rightmost index with timestamp <= given timestamp.
        while left <= right:
            mid = (left + right) // 2
            if values[mid][0] <= timestamp:
                left = mid + 1
```



```python
            else:
                right = mid - 1

        # If no valid timestamp exists, right will be -1.
        if right == -1:
            return ""
        return values[right][1]

# Example usage:
# timeMap = TimeMap()
# timeMap.set("foo", "bar", 1)
# print(timeMap.get("foo", 1)) # Output: "bar"
# print(timeMap.get("foo", 3)) # Output: "bar"
# timeMap.set("foo", "bar2", 4)
# print(timeMap.get("foo", 4)) # Output: "bar2"
# print(timeMap.get("foo", 5)) # Output: "bar2"
```

#1011 MEDIUM

Capacity to Ship Packages Within D Days

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search

Lists: Denny Zhang

Problem:

A conveyor belt has packages that must be shipped from one port to another within days days.

The ith package on the conveyor belt has a weight of weights[i]. Each day, we load the ship with packages on the conveyor belt (in the order given by weights). We may not load more weight than the maximum weight capacity of the ship.

Return the least weight capacity of the ship that will result in all the packages on the conveyor belt being shipped within days days.

Example 1:

```
Input: weights = [1,2,3,4,5,6,7,8,9,10], days = 5
Output: 15
Explanation: A ship capacity of 15 is the minimum to ship all the packages in 5 days like
this:
1st day: 1, 2, 3, 4, 5
2nd day: 6, 7
3rd day: 8
4th day: 9
5th day: 10

Note that the cargo must be shipped in the order given, so using a ship of capacity 14 and
splitting the packages into parts like (2, 3, 4, 5), (1, 6, 7), (8), (9), (10) is not
allowed.
```

Example 2:

```
Input: weights = [3,2,2,4,1,4], days = 3
Output: 6
Explanation: A ship capacity of 6 is the minimum to ship all the packages in 3 days like
```



```
this:
1st day: 3, 2
2nd day: 2, 4
3rd day: 1, 4
```

Example 3:

```
Input: weights = [1,2,3,1,1], days = 4
Output: 3
Explanation:
1st day: 1
2nd day: 2
3rd day: 3
4th day: 1, 1
```

Constraints:

- 1 <= days <= weights.length <= 5 ^ 104
- 1 <= weights[i] <= 500

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def shipWithinDays(self, weights, days):
        # Brute Force O(n) - linear scan instead of binary search
        for i, val in enumerate(weights):
            if val == target:
                return i
        return -1
```

Python

Python


```python
def shipWithinDays(weights, days):
    # Helper function to determine if a given capacity can ship packages within 'days' days.
    def canShip(capacity):
        current_load = 0 # Current total weight on the ship for the current day.
        required_days = 1 # Start with the first day.
        for weight in weights:
            # If adding the current weight exceeds capacity, increment the day counter.
            if current_load + weight > capacity:
                required_days += 1
                current_load = 0
            current_load += weight
        return required_days <= days

    # Establish lower and upper bounds for binary search.
    low, high = max(weights), sum(weights)
    while low < high:
        mid = (low + high) // 2
        # If mid capacity can ship all packages within the allowed days, try a lower capacity.
        if canShip(mid):
            high = mid
        else:
            low = mid + 1
    return low

# Example usage:

weights = [1,2,3,3,4,5,6,7,8,9,10]
days = 5
print(shipWithinDays(weights, days))
```

Python

```python
class Solution:
    def shipWithinDays(self, weights: list[int], days: int) -> int:
        def shipDays(shipCapacity: int) -> int:
            shipDays = 1
            capacity = 0
            for weight in weights:
                if capacity + weight > shipCapacity:
                    shipDays += 1
                    capacity = weight
                else:
                    capacity += weight
            return shipDays

        l = max(weights)
        r = sum(weights)
        return bisect.bisect_left(range(l, r), True,
                                  key=lambda m: shipDays(m) <= days) + l
```

Python

```python
class Solution:
    def shipWithinDays(self, weights: List[int], days: int) -> int:
        def check(mx):
            ws, cnt = 0, 1
            for w in weights:
                ws += w
                if ws > mx:
                    cnt += 1
                    ws = w
            return cnt <= days

        left, right = max(weights), sum(weights) + 1
        return left + bisect_left(range(left, right), True, key=check)
```

Python

```python
class Solution:
    def shipWithinDays(self, weights: List[int], days: int) -> int:
        def check(mx):
            ws, cnt = 0, 1
            for w in weights:
                ws += w
                if ws > mx:
                    cnt += 1
                    ws = w
            return cnt <= days

        left, right = max(weights), sum(weights) + 1
        return left + bisect_left(range(left, right), True, key=check)
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
def shipWithinDays(weights, days):
    # Helper function to determine if a given capacity can ship packages within 'days' days.
    def canShip(capacity):
        current_load = 0 # Current total weight on the ship for the current day.
        required_days = 1 # Start with the first day.
        for weight in weights:
            # If adding the current weight exceeds capacity, increment the day counter.
            if current_load + weight > capacity:
                required_days += 1
                current_load = 0
            current_load += weight
        return required_days <= days

    # Establish lower and upper bounds for binary search.
    low, high = max(weights), sum(weights)
    while low < high:
        mid = (low + high) // 2
        # If mid capacity can ship all packages within the allowed days, try a lower capacity.
        if canShip(mid):
            high = mid
        else:
            low = mid + 1
    return low

# Example usage:
```



```python
weights = [1,2,3,4,5,6,7,8,9,10]
days = 5
print(shipWithinDays(weights, days))
```

#1060 MEDIUM

Missing Element in Sorted Array

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search

Lists: Premium 98

Problem:

Given an integer array nums which is sorted in ascending order and all of its elements are unique and given also an integer k, return the kth missing number starting from the leftmost number of the array.

Example 1:

```
Input: nums = [4,7,9,10], k = 1
Output: 5
Explanation: The first missing number is 5.
```

Example 2:

```
Input: nums = [4,7,9,10], k = 3
Output: 8
Explanation: The missing numbers are [5,6,8,...], hence the third missing number is 8.
```

Example 3:

```
Input: nums = [1,2,4], k = 3
Output: 6
Explanation: The missing numbers are [3,5,6,7,...], hence the third missing number is 6.
```

Constraints:

- 1 <= nums.length <= 5 * 104
- 1 <= nums[i] <= 107
- nums is sorted in ascending order, and all the elements are unique.
- 1 <= k <= 108

Follow up: Can you find a logarithmic time complexity (i.e., O(log(n))) solution?

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def missingElement(self, nums, k):
        # Brute Force O(n) - linear scan instead of binary search
        for i, val in enumerate(nums):
            if val == target:
                return i
        return -1
```

Python

Python

```python
# Python solution with comprehensive comments

def missingElement(nums, k):
    # Calculate the missing count until index i: missing(i) = nums[i] - nums[0] - i.
    def missing_count(index):
        return nums[index] - nums[0] - index

    n = len(nums)

    # If kth missing number is beyond the last element.
    if k > missing_count(n - 1):
        return nums[-1] + k - missing_count(n - 1)

    # Perform binary search to find the smallest index where missing_count(index) >= k.
    left, right = 0, n - 1
    while left < right:
        mid = left + (right - left) // 2
        if missing_count(mid) < k:
            left = mid + 1
        else:
            right = mid

    # After binary search, left is the smallest index such that missing_count(left) >= k.
    # The kth missing number is after nums[left - 1].
    return nums[left - 1] + k - missing_count(left - 1)


# Example usage:
print(missingElement([4,7,9,10], 3)) # Output: 8
```

Python

```python
class Solution:
    def missingElement(self, nums: List[int], k: int) -> int:
        def missing(i: int) -> int:
            return nums[i] - nums[0] - i

        n = len(nums)
        if k > missing(n - 1):
            return nums[n - 1] + k - missing(n - 1)
        l, r = 0, n - 1
        while l < r:
            mid = (l + r) >> 1
            if missing(mid) >= k:
                r = mid
            else:
                l = mid + 1
        return nums[l - 1] + k - missing(l - 1)
```

Python


```python
class Solution:
    def missingElement(self, nums: List[int], k: int) -> int:
        def missing(i: int) -> int:
            return nums[i] - nums[0] - i

        n = len(nums)
        if k > missing(n - 1):
            return nums[n - 1] + k - missing(n - 1)
        l, r = 0, n - 1
        while l < r:
            mid = (l + r) >> 1
            if missing(mid) >= k:
                r = mid
            else:
                l = mid + 1
        return nums[l - 1] + k - missing(l - 1)
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
# Python solution with comprehensive comments

def missingElement(nums, k):
    # Calculate the missing count until index i: missing(i) = nums[i] - nums[0] - i.
    def missing_count(index):
        return nums[index] - nums[0] - index

    n = len(nums)

    # If kth missing number is beyond the last element.
    if k > missing_count(n - 1):
        return nums[-1] + k - missing_count(n - 1)

    # Perform binary search to find the smallest index where missing_count(index) >= k.
    left, right = 0, n - 1
    while left < right:
        mid = left + (right - left) // 2
        if missing_count(mid) < k:
            left = mid + 1
        else:
            right = mid

    # After binary search, left is the smallest index such that missing_count(left) >= k.
    # The kth missing number is after nums[left - 1].
    return nums[left - 1] + k - missing_count(left - 1)


# Example usage:
print(missingElement([4,7,9,10], 3)) # Output: 8
```

#1062 MEDIUM

Longest Repeating Substring

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Binary Search · Dynamic Programming · Rolling Hash · Suffix Array · Hash Function

Lists: Denny Zhang

Problem:

Given a string s, return the length of the longest repeating substrings. If no repeating substring exists, return 0.

Example 1:

```
Input: s = "abcd"
Output: 0
Explanation: There is no repeating substring.
```

Example 2:

```
Input: s = "abbaba"
Output: 2
Explanation: The longest repeating substrings are "ab" and "ba", each of which occurs twice.
```

Example 3:

```
Input: s = "aabcaabdaab"
Output: 3
Explanation: The longest repeating substring is "aab", which occurs 3 times.
```

Constraints:

- 1 <= s.length <= 2000
- s consists of lowercase English letters.

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def longestRepeatingSubstring(self, s):
        # Brute Force O(n) - linear scan instead of binary search
        for i, val in enumerate(s):
            if val == target:
                return i
        return -1
```

Python

Python


```python
def longestRepeatingSubstring(s):
    n = len(s)
    # Convert characters to integers for rolling hash computation.
    nums = [ord(c) - ord('a') for c in s]

    # Base value for rolling hash.
    a = 26
    # Modulus value to avoid overflow.
    mod = 2**32

    # Helper function to check if any substring of given length L repeats.
    def search(L):
        h = 0
        for i in range(L):
            h = (h * a + nums[i]) % mod
        seen = {h}
        # Precompute a^L % mod for removing the first digit in the rolling hash.
        aL = pow(a, L, mod)
        for start in range(1, n - L + 1):
            h = (h * a - nums[start - 1] * aL + nums[start + L - 1]) % mod
            if h in seen:
                return True
            seen.add(h)
        return False

    left, right = 1, n
    ans = 0
    while left <= right:
        mid = left + (right - left) // 2
        if search(mid):
            ans = mid
            left = mid + 1
        else:
            right = mid - 1
    return ans

# Example usage:
s = "aabcaabdaab"
print(longestRepeatingSubstring(s)) # Expected output: 3
```

Python

```python
class Solution:
    def longestRepeatingSubstring(self, s: str) -> int:
        n = len(s)
        ans = 0
        # dp[i][j] := the number of repeating characters of s[0..i) and s[0..j)
        dp = [[0] * (n + 1) for _ in range(n + 1)]

        for i in range(1, n + 1):
          for j in range(i + 1, n + 1):
            if s[i - 1] == s[j - 1]:
              dp[i][j] = 1 + dp[i - 1][j - 1]
              ans = max(ans, dp[i][j])

        return ans
```

Python

```python
class Solution:
    def longestRepeatingSubstring(self, s: str) -> int:
        n = len(s)
        f = [[0] * n for _ in range(n)]
        ans = 0
        for i in range(1, n):
            for j in range(i):
                if s[i] == s[j]:
                    f[i][j] = 1 + (f[i - 1][j - 1] if j else 0)
                    ans = max(ans, f[i][j])
        return ans
```

Python

```python
class Solution:
    def longestRepeatingSubstring(self, s: str) -> int:
        n = len(s)
        dp = [[0] * n for _ in range(n)]
        ans = 0
        for i in range(n):
            for j in range(i + 1, n):
                if s[i] == s[j]:
                    dp[i][j] = dp[i - 1][j - 1] + 1 if i else 1
                    ans = max(ans, dp[i][j])
        return ans
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
def longestRepeatingSubstring(s):
    n = len(s)
    # Convert characters to integers for rolling hash computation.
    nums = [ord(c) - ord('a') for c in s]

    # Base value for rolling hash.
    a = 26
    # Modulus value to avoid overflow.
    mod = 2**32

    # Helper function to check if any substring of given length L repeats.
    def search(L):
        h = 0
        for i in range(L):
            h = (h * a + nums[i]) % mod
        seen = {h}
        # Precompute a^L % mod for removing the first digit in the rolling hash.
        aL = pow(a, L, mod)
        for start in range(1, n - L + 1):
            h = (h * a - nums[start - 1] * aL + nums[start + L - 1]) % mod
            if h in seen:
                return True
            seen.add(h)
        return False
```



```python
    left, right = 1, n
    ans = 0
    while left <= right:
        mid = left + (right - left) // 2
        if search(mid):
            ans = mid
            left = mid + 1
        else:
            right = mid - 1
    return ans

# Example usage:
s = "aabcaabdaab"
print(longestRepeatingSubstring(s)) # Expected output: 3
```

#1533 MEDIUM

Find the Index of the Large Integer

LeetCodeCx · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search · Interactive

Lists: Premium 98

Problem:

We have an integer array arr, where all the integers in arr are equal except for one integer which is larger than the rest of the integers. You will not be given direct access to the array, instead, you will have an API ArrayReader which have the following functions:

- int compareSub(int l, int r, int x, int y): where 0 <= l, r, x, y < ArrayReader.length(), l <= r and x <= y. The function compares the sum of sub-array arr[l..r] with the sum of the sub-array arr[x..y] and returns: 1 if arr[l]+arr[l+1]+...+arr[r] > arr[x]+arr[x+1]+...+arr[y].
- 0 if arr[l]+arr[l+1]+...+arr[r] == arr[x]+arr[x+1]+...+arr[y].
- -1 if arr[l]+arr[l+1]+...+arr[r] < arr[x]+arr[x+1]+...+arr[y].

You are allowed to call compareSub() 20 times at most. You can assume both functions work in O(1) time.

Return the index of the array arr which has the largest integer.

Example 1:

```
Input: arr = [7,7,7,7,10,7,7,7]
Output: 4
Explanation: The following calls to the API
reader.compareSub(0, 0, 1, 1) // returns 0 this is a query comparing the sub-array (0, 0)
with the sub array (1, 1), (i.e. compares arr[0] with arr[1]).
Thus we know that arr[0] and arr[1] doesn't contain the largest element.
reader.compareSub(2, 2, 3, 3) // returns 0, we can exclude arr[2] and arr[3].
reader.compareSub(4, 4, 5, 5) // returns 1, thus for sure arr[4] is the largest element in
the array.
Notice that we made only 3 calls, so the answer is valid.
```

Example 2:

```
Input: nums = [6,6,12]
Output: 2
```


Constraints:

- 2 <= arr.length <= 5 * 105
- 1 <= arr[i] <= 100
- All elements of arr are equal except for one element which is larger than all other elements.

Follow up:

- What if there are two numbers in arr that are bigger than all other numbers?
- What if there is one number that is bigger than other numbers and one number that is smaller than other numbers?

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def getIndex(self, reader):
        # Brute force: linear scan instead of binary search
        for i, val in enumerate(reader):
            if val == target:
                return i
        return -1
```

Python

Python

```python
# Python solution with line-by-line comments
class Solution:
    def getIndex(self, reader):
        # Initialize search bounds
        left, right = 0, reader.length() - 1

        # Binary search for the candidate index
        while left < right:
            length = right - left + 1
            if length % 2 == 0:
                # Even length: split into two equal halves.
                mid = left + length // 2
                # Compare left half [left, mid-1] with right half [mid, right]
                comp = reader.compareSub(left, mid - 1, mid, right)
                if comp == 1:
                    # Large integer is in the left half
                    right = mid - 1
                else:
                    # Large integer is in the right half
                    left = mid
            else:
                # Odd length: divide into two halves of equal size, ignoring middle element.
                half = length // 2
                # Compare left half [left, left+half-1] with right half [right-half+1, right]
                comp = reader.compareSub(left, left + half - 1, right - half + 1, right)
```



```python
                if comp == 1:
                    # Large integer in left half
                    right = left + half - 1
                elif comp == -1:
                    # Large integer in right half
                    left = right - half + 1
                else:
                    # Halves are equal, so the middle element is the large one.
                    return left + half
        return left
```

Python

```python
# """
# This is ArrayReader's API interface.
# You should not implement it, or speculate about its implementation
# """
# class ArrayReader(object):
# # Compares the sum of arr[l..r] with the sum of arr[x..y]
# # return 1 if sum(arr[l..r]) > sum(arr[x..y])
# # return 0 if sum(arr[l..r]) == sum(arr[x..y])
# # return -1 if sum(arr[l..r]) < sum(arr[x..y])
# def compareSub(self, l: int, r: int, x: int, y: int) -> int:
#
# # Returns the length of the array
# def length(self) -> int:
#


class Solution:
    def getIndex(self, reader: 'ArrayReader') -> int:
        l = 0
        r = reader.length() - 1

        while l < r:
            m = (l + r) // 2
            if (r - l) % 2 == 0:
                res = reader.compareSub(l, m - 1, m + 1, r)

                if res == 0:
                    return m
                if res == 1:
                    r = m - 1
                else: # res == -1
                    l = m + 1
            else:
                res = reader.compareSub(l, m, m + 1, r)
                # res is either 1 or -1.
                if res == 1:
                    r = m
                else: # res == -1
                    l = m + 1
        return l
```

Python


```python
# """
# This is ArrayReader's API interface.
# You should not implement it, or speculate about its implementation
# """
# class ArrayReader(object):
# # Compares the sum of arr[l..r] with the sum of arr[x..y]
# # return 1 if sum(arr[l..r]) > sum(arr[x..y])
# # return 0 if sum(arr[l..r]) == sum(arr[x..y])
# # return -1 if sum(arr[l..r]) < sum(arr[x..y])
# def compareSub(self, l: int, r: int, x: int, y: int) -> int:
#
# # Returns the length of the array
# def length(self) -> int:
#


class Solution:
    def getIndex(self, reader: 'ArrayReader') -> int:
        left, right = 0, reader.length() - 1
        while left < right:
            t1, t2, t3 = (
                left,
                left + (right - left) // 3,
                left + ((right - left) // 3) * 2 + 1,
            )

            cmp = reader.compareSub(t1, t2, t2 + 1, t3)
            if cmp == 0:
                left = t3 + 1
            elif cmp == 1:
                right = t2
            else:
                left, right = t2 + 1, t3
        return left
```

Python


```python
# """
# This is ArrayReader's API interface.
# You should not implement it, or speculate about its implementation
# """
# class ArrayReader(object):
# # Compares the sum of arr[l..r] with the sum of arr[x..y]
# # return 1 if sum(arr[l..r]) > sum(arr[x..y])
# # return 0 if sum(arr[l..r]) == sum(arr[x..y])
# # return -1 if sum(arr[l..r]) < sum(arr[x..y])
# def compareSub(self, l: int, r: int, x: int, y: int) -> int:
#
# # Returns the length of the array
# def length(self) -> int:
#


class Solution:
    def getIndex(self, reader: 'ArrayReader') -> int:
        left, right = 0, reader.length() - 1
        while left < right:
            t1, t2, t3 = (
                left,
                left + (right - left) // 3,
                left + ((right - left) // 3) * 2 + 1,
            )

            cmp = reader.compareSub(t1, t2, t2 + 1, t3)
            if cmp == 0:
                left = t3 + 1
            elif cmp == 1:
                right = t2
            else:
                left, right = t2 + 1, t3
        return left
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
# Python solution with line-by-line comments
class Solution:
    def getIndex(self, reader):
        # Initialize search bounds
        left, right = 0, reader.length() - 1

        # Binary search for the candidate index
        while left < right:
            length = right - left + 1
            if length % 2 == 0:
                # Even length: split into two equal halves.
                mid = left + length // 2
                # Compare left half [left, mid-1] with right half [mid, right]
                comp = reader.compareSub(left, mid - 1, mid, right)
                if comp == 1:
                    # Large integer is in the left half
                    right = mid - 1
                else:
                    # Large integer is in the right half
                    left = mid
            else:
                # Odd length: divide into two halves of equal size, ignoring middle element.
                half = length // 2
                # Compare left half [left, left+half-1] with right half [right-half+1, right]
                comp = reader.compareSub(left, left + half - 1, right - half + 1, right)

                if comp == 1:
                    # Large integer in left half
                    right = left + half - 1
                elif comp == -1:
                    # Large integer in right half
                    left = right - half + 1
                else:
                    # Halves are equal, so the middle element is the large one.
                    return left + half
        return left
```

#4 HARD

Median of Two Sorted Arrays

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search · Divide and Conquer

Lists: Grind 169

Problem:

Given two sorted arrays nums1 and nums2 of size m and n respectively, return the median of the two sorted arrays.

The overall run time complexity should be O(log (m+n)).

Example 1:

```
Input: nums1 = [1,3], nums2 = [2]
Output: 2.00000
Explanation: merged array = [1,2,3] and median is 2.
```

Example 2:


```
Input: nums1 = [1,2], nums2 = [3,4]
Output: 2.50000
Explanation: merged array = [1,2,3,4] and median is (2 + 3) / 2 = 2.5.
```

Constraints:

- nums1.length == m
- nums2.length == n
- 0 <= m <= 1000
- 0 <= n <= 1000
- 1 <= m + n <= 2000
- -106 <= nums1[i], nums2[i] <= 106

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findMedianSortedArrays(self, nums1, nums2):
        # Brute Force O((m+n) log(m+n)) - merge both arrays, sort, find median
        merged = sorted(nums1 + nums2)
        n = len(merged)
        if n % 2 == 1:
            return float(merged[n // 2])
        return (merged[n/2 - 1] + merged[n/2]) / 2.0
```

Python

Python

```python
# Function to find the median of two sorted arrays
def findMedianSortedArrays(nums1, nums2):
    # Ensure nums1 is the smaller array for faster binary search
    if len(nums1) > len(nums2):
        nums1, nums2 = nums2, nums1

    m, n = len(nums1), len(nums2)
    low, high = 0, m

    while low <= high:
        # Partition indexes for nums1 and nums2
        partition_nums1 = (low + high) // 2
        partition_nums2 = (m + n + 1) // 2 - partition_nums1

        # If partition is 0 then there are no elements on left side. Use -infinity for comparison.
        maxLeftNums1 = float('-inf') if partition_nums1 == 0 else nums1[partition_nums1 - 1]
        # If partition is length then there are no elements on right side. Use infinity for comparison.
        minRightNums1 = float('inf') if partition_nums1 == m else nums1[partition_nums1]

        # Similarly for nums2
        maxLeftNums2 = float('-inf') if partition_nums2 == 0 else nums2[partition_nums2 - 1]
        minRightNums2 = float('inf') if partition_nums2 == n else nums2[partition_nums2]

        # Check if we have found the correct partition.
        if maxLeftNums1 <= minRightNums2 and maxLeftNums2 <= minRightNums1:
```



```python
            # If total number of elements is even
            if (m + n) % 2 == 0:
                return (max(maxLeftNums1, maxLeftNums2) + min(minRightNums1, minRightNums2)) / 2.0
            else:
                return float(max(maxLeftNums1, maxLeftNums2))
        # If not, adjust binary search range
        elif maxLeftNums1 > minRightNums2:
            # Move partition in nums1 to left
            high = partition_nums1 - 1
        else:
            # Move partition in nums1 to right
            low = partition_nums1 + 1

# Example usage:
print(findMedianSortedArrays([1,3], [2])) # Output: 2.0
print(findMedianSortedArrays([1,2], [3,4])) # Output: 2.5
```

Python

```python
class Solution:
    def findMedianSortedArrays(self, nums1: list[int], nums2: list[int]) -> float:
      n1 = len(nums1)
      n2 = len(nums2)
      if n1 > n2:
        return self.findMedianSortedArrays(nums2, nums1)

      l = 0
      r = n1

      while l <= r:
        partition1 = (l + r) // 2
        partition2 = (n1 + n2 + 1) // 2 - partition1
        maxLeft1 = -2**31 if partition1 == 0 else nums1[partition1 - 1]
        maxLeft2 = -2**31 if partition2 == 0 else nums2[partition2 - 1]
        minRight1 = 2**31 - 1 if partition1 == n1 else nums1[partition1]
        minRight2 = 2**31 - 1 if partition2 == n2 else nums2[partition2]
        if maxLeft1 <= minRight2 and maxLeft2 <= minRight1:
          return (max(maxLeft1, maxLeft2) + min(minRight1, minRight2)) * 0.5 if (n1 + n2) % 2 == 0 else
max(maxLeft1, maxLeft2)
        elif maxLeft1 > minRight2:
          r = partition1 - 1
        else:
          l = partition1 + 1
```

Python


```python
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        def f(i: int, j: int, k: int) -> int:
            if i >= m:
                return nums2[j + k - 1]
            if j >= n:
                return nums1[i + k - 1]
            if k == 1:
                return min(nums1[i], nums2[j])
            p = k // 2
            x = nums1[i + p - 1] if i + p - 1 < m else inf
            y = nums2[j + p - 1] if j + p - 1 < n else inf
            return f(i + p, j, k - p) if x < y else f(i, j + p, k - p)

        m, n = len(nums1), len(nums2)
        a = f(0, 0, (m + n + 1) // 2)
        b = f(0, 0, (m + n + 2) // 2)
        return (a + b) / 2
```

Python

```python
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:

        def findKth(i: int, j: int, k: int) -> float:
            if i >= m:
                return nums2[j + k - 1]
            if j >= n:
                return nums1[i + k - 1]
            if k == 1:
                return min(nums1[i], nums2[j])

            # Division a // b : floordiv(a, b)
            midVal1 = nums1[i + k // 2 - 1] if i + k // 2 - 1 < m else math.inf
            midVal2 = nums2[j + k // 2 - 1] if j + k // 2 - 1 < n else math.inf

            if midVal1 < midVal2:
                return findKth(i + k // 2, j, k - k // 2) # '+' or '-' k//2
            else:
                return findKth(i, j + k // 2, k - k // 2)

        m = len(nums1)
        n = len(nums2)
        # Division a // b : floordiv(a, b)
        left = (m + n + 1) // 2
        right = (m + n + 2) // 2
```



```python
        return (findKth(0, 0, left) + findKth(0, 0, right)) / 2.0

############

# iteration version
class Solution(object):
    def findMedianSortedArrays(self, nums1, nums2):
        a, b = sorted((nums1, nums2), key=len)
        m, n = len(a), len(b)
        after = (m + n - 1) / 2
        lo, hi = 0, m
        while lo < hi:
          i = (lo + hi) / 2
          if after - i - 1 < 0 or a[i] >= b[after - i - 1]:
            hi = i
          else:
            lo = i + 1
        i = lo
        nextfew = sorted(a[i:i + 2] + b[after - i:after - i + 2])
        return (nextfew[0] + nextfew[1 - (m + n) % 2]) / 2.0
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:

        def findKth(i: int, j: int, k: int) -> float:
            if i >= m:
                return nums2[j + k - 1]
            if j >= n:
                return nums1[i + k - 1]
            if k == 1:
                return min(nums1[i], nums2[j])

            # Division a // b : floordiv(a, b)
            midVal1 = nums1[i + k // 2 - 1] if i + k // 2 - 1 < m else math.inf
            midVal2 = nums2[j + k // 2 - 1] if j + k // 2 - 1 < n else math.inf

            if midVal1 < midVal2:
                return findKth(i + k // 2, j, k - k // 2) # '+' or '-' k//2
            else:
                return findKth(i, j + k // 2, k - k // 2)

        m = len(nums1)
        n = len(nums2)
        # Division a // b : floordiv(a, b)
        left = (m + n + 1) // 2
        right = (m + n + 2) // 2
```



```python
        return (findKth(0, 0, left) + findKth(0, 0, right)) / 2.0

############

# iteration version
class Solution(object):
    def findMedianSortedArrays(self, nums1, nums2):
        a, b = sorted((nums1, nums2), key=len)
        m, n = len(a), len(b)
        after = (m + n - 1) / 2
        lo, hi = 0, m
        while lo < hi:
          i = (lo + hi) / 2
          if after - i - 1 < 0 or a[i] >= b[after - i - 1]:
            hi = i
          else:
            lo = i + 1
        i = lo
        nextfew = sorted(a[i:i + 2] + b[after - i:after - i + 2])
        return (nextfew[0] + nextfew[1 - (m + n) % 2]) / 2.0
```

#315 HARD

Count of Smaller Numbers After Self

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search · Divide and Conquer · Binary Indexed Tree · Segment Tree · Merge Sort · Ordered Set

Lists: Denny Zhang

Problem:

Given an integer array nums, return an integer array counts where counts[i] is the number of smaller elements to the right of nums[i].

Example 1:

```
Input: nums = [5,2,6,1]
Output: [2,1,1,0]
Explanation:
To the right of 5 there are 2 smaller elements (2 and 1).
To the right of 2 there is only 1 smaller element (1).
To the right of 6 there is 1 smaller element (1).
To the right of 1 there is 0 smaller element.
```

Example 2:

```
Input: nums = [-1]
Output: [0]
```

Example 3:

```
Input: nums = [-1,-1]
Output: [0,0]
```

Constraints:

- 1 <= nums.length <= 105
- -104 <= nums[i] <= 104

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def countSmaller(self, nums):
        # Brute Force O(n) - linear scan instead of binary search
        for i, val in enumerate(nums):
            if val == target:
                return i
        return -1
```

Python

Python

```python
# Python solution using merge sort approach

def countSmaller(nums):
    n = len(nums)
    # Initialize counts array for results
    counts = [0] * n
    # Pair each element with its original index
    enum = [(num, i) for i, num in enumerate(nums)]

    # Recursive function for merge sort with counting
    def mergeSort(arr):
        if len(arr) <= 1:
            return arr
        # Split the array into two halves
        mid = len(arr) // 2
        left = mergeSort(arr[:mid])
        right = mergeSort(arr[mid:])
        merged = []
        l = r = 0
        # r_count counts number of elements taken from right array
        r_count = 0
        while l < len(left) and r < len(right):
            # If left element is less or equal, it means current right elements are smaller
            if left[l][0] <= right[r][0]:
                # update count for original index
```



```python
                counts[left[l][1]] += r_count
                merged.append(left[l])
                l += 1
            else:
                # right element is smaller, so increment r_count and add it to merged
                merged.append(right[r])
                r += 1
                r_count += 1
        # Append remaining elements from left reduce count by accumulated right elements
        while l < len(left):
            counts[left[l][1]] += r_count
            merged.append(left[l])
            l += 1
        # Append any remaining elements from right side
        while r < len(right):
            merged.append(right[r])
            r += 1
        return merged

    # Start the merge sort process
    mergeSort(enum)
    return counts

# Example test
print(countSmaller([5, 2, 6, 1])) # Expected output: [2, 1, 1, 0]
```

Python

```python
class FenwickTree:
  def __init__(self, n: int):
    self.sums = [0] * (n + 1)

  def add(self, i: int, delta: int) -> None:
    while i < len(self.sums):
      self.sums[i] += delta
      i += FenwickTree.lowbit(i)

  def get(self, i: int) -> int:
    summ = 0
    while i > 0:
      summ += self.sums[i]
      i -= FenwickTree.lowbit(i)
    return summ

  @staticmethod
  def lowbit(i: int) -> int:
    return i & -i


class Solution:
  def countSmaller(self, nums: list[int]) -> list[int]:
    ans = []
    ranks = self._getRanks(nums)
```



```python
    tree = FenwickTree(len(ranks))

    for num in reversed(nums):
      ans.append(tree.get(ranks[num] - 1))
      tree.add(ranks[num], 1)

    return ans[::-1]

  def _getRanks(self, nums: list[int]) -> dict[int, int]:
    ranks = collections.Counter()
    rank = 0
    for num in sorted(set(nums)):
      rank += 1
      ranks[num] = rank
    return ranks
```

Python

```python
class BinaryIndexedTree:
    def __init__(self, n):
        self.n = n
        self.c = [0] * (n + 1)

    @staticmethod
    def lowbit(x):
        return x & -x

    def update(self, x, delta):
        while x <= self.n:
            self.c[x] += delta
            x += BinaryIndexedTree.lowbit(x)

    def query(self, x):
        s = 0
        while x > 0:
            s += self.c[x]
            x -= BinaryIndexedTree.lowbit(x)
        return s


class Solution:
    def countSmaller(self, nums: List[int]) -> List[int]:
        alls = sorted(set(nums))
```

```python
        m = {v: i for i, v in enumerate(alls, 1)}
        tree = BinaryIndexedTree(len(m))
        ans = []
        for v in nums[::-1]:
            x = m[v]
            tree.update(x, 1)
            ans.append(tree.query(x - 1))
        return ans[::-1]
```

Python


```
'''
bisect: maintaining a list in sorted order without having to sort the list after each insertion.
https://docs.python.org/3/library/bisect.html

bisect.bisect_left()
Locate the insertion point for x in a to maintain sorted order.

bisect.bisect_right() or bisect.bisect()
Similar to bisect_left(), but returns an insertion point which comes after (to the right of) any existing
entries of x in a.


bisect.insort_left(a, x, lo=0, hi=len(a), *, key=None)
Insert x in a in sorted order.
Keep in mind that the O(log n) search is dominated by the slow O(n) insertion step.


bisect.insort_right(a, x, lo=0, hi=len(a), *, key=None)
bisect.insort(a, x, lo=0, hi=len(a), *, key=None)
Similar to insort_left(), but inserting x in a after any existing entries of x.


>>> import bisect
>>> bisect.bisect_left([1,2,3], 2)
1
>>> bisect.bisect_right([1,2,3], 2)
```

```
2

>>> a = [1, 1, 1, 2, 3]
>>> bisect.insort_left(a, 1.0)
>>> a
[1.0, 1, 1, 1, 2, 3]

>>> a = [1, 1, 1, 2, 3]
>>> bisect.insort_right(a, 1.0)
>>> a
[1, 1, 1, 1.0, 2, 3]

>>> a = [1, 1, 1, 2, 3]
>>> bisect.insort(a, 1.0)
>>> a
[1, 1, 1, 1.0, 2, 3]
'''
```

```
import bisect

class Solution(object):
  def countSmaller(self, nums):
    '''
    :type nums: List[int]
    :rtype: List[int]
```



```
    '''
    ans = []
    bst = []
    for num in reversed(nums):
      idx = bisect.bisect_left(bst, num)
      ans.append(idx)
      bisect.insort(bst, num)
    return ans[::-1]

###########

class BinaryIndexedTree:
    def __init__(self, n):
        self.n = n
        self.c = [0] * (n + 1)

    @staticmethod
    def lowbit(x):
        return x & -x

    def update(self, x, delta):
        while x <= self.n:
            self.c[x] += delta
            x += BinaryIndexedTree.lowbit(x)

    def query(self, x):
        s = 0
        while x > 0:
            s += self.c[x]
            x -= BinaryIndexedTree.lowbit(x)
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python


```
'''
bisect: maintaining a list in sorted order without having to sort the list after each insertion.
https://docs.python.org/3/library/bisect.html

bisect.bisect_left()
Locate the insertion point for x in a to maintain sorted order.

bisect.bisect_right() or bisect.bisect()
Similar to bisect_left(), but returns an insertion point which comes after (to the right of) any existing
entries of x in a.


bisect.insort_left(a, x, lo=0, hi=len(a), *, key=None)
Insert x in a in sorted order.
Keep in mind that the O(log n) search is dominated by the slow O(n) insertion step.


bisect.insort_right(a, x, lo=0, hi=len(a), *, key=None)
bisect.insort(a, x, lo=0, hi=len(a), *, key=None)
Similar to insort_left(), but inserting x in a after any existing entries of x.


>>> import bisect
>>> bisect.bisect_left([1,2,3], 2)
1
>>> bisect.bisect_right([1,2,3], 2)
```

```
2

>>> a = [1, 1, 1, 2, 3]
>>> bisect.insort_left(a, 1.0)
>>> a
[1.0, 1, 1, 1, 2, 3]

>>> a = [1, 1, 1, 2, 3]
>>> bisect.insort_right(a, 1.0)
>>> a
[1, 1, 1, 1.0, 2, 3]

>>> a = [1, 1, 1, 2, 3]
>>> bisect.insort(a, 1.0)
>>> a
[1, 1, 1, 1.0, 2, 3]
'''
```

```
import bisect

class Solution(object):
  def countSmaller(self, nums):
    '''
    :type nums: List[int]
    :rtype: List[int]
```

```
    """
    ans = []
    bst = []
    for num in reversed(nums):
      idx = bisect.bisect_left(bst, num)
      ans.append(idx)
      bisect.insort(bst, num)
    return ans[::-1]

###########

class BinaryIndexedTree:
    def __init__(self, n):
        self.n = n
        self.c = [0] * (n + 1)

    @staticmethod
    def lowbit(x):
        return x & -x

    def update(self, x, delta):
        while x <= self.n:
            self.c[x] += delta
            x += BinaryIndexedTree.lowbit(x)

    def query(self, x):
        s = 0
        while x > 0:
            s += self.c[x]
            x -= BinaryIndexedTree.lowbit(x)
```

#410 HARD

Split Array Largest Sum

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search · Dynamic Programming · Greedy · Prefix Sum

Lists: Denny Zhang

Problem:

Given an integer array nums and an integer k, split nums into k non-empty subarrays such that the largest sum of any subarray is minimized.

Return the minimized largest sum of the split.

A subarray is a contiguous part of the array.

Example 1:

```
Input: nums = [7,2,5,10,8], k = 2
Output: 18
Explanation: There are four ways to split nums into two subarrays.
The best way is to split it into [7,2,5] and [10,8], where the largest sum among the two
subarrays is only 18.
```

Example 2:


```
Input: nums = [1,2,3,4,5], k = 2
Output: 9
Explanation: There are four ways to split nums into two subarrays.
The best way is to split it into [1,2,3] and [4,5], where the largest sum among the two
subarrays is only 9.
```

Constraints:

- 1 <= nums.length <= 1000
- 0 <= nums[i] <= 106
- 1 <= k <= min(50, nums.length)

> > Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def splitArray(self, nums, k):
        # Brute-force find - linear scan instead of binary search
        for i, val in enumerate(nums):
            if val == target:
                return i
        return -1
```

Python

Python

```python
# Python solution for Split Array Largest Sum

def splitArray(nums, k):
    # Helper function to determine if we can split array into <= k subarrays with max subarray sum <=
max_sum
    def canSplit(max_sum):
        subarray_count = 1 # start with one subarray
        current_sum = 0
        for num in nums:
            if current_sum + num > max_sum:
                subarray_count += 1 # start a new subarray
                current_sum = num # reset the sum with the current number
            else:
                current_sum += num # add num to current subarray
        return subarray_count <= k # valid if we used k or fewer subarrays

    low, high = max(nums), sum(nums) # boundaries for binary search
    while low < high:
        mid = (low + high) // 2 # candidate maximum subarray sum
        if canSplit(mid):
            high = mid # candidate is valid, try to find a smaller one
        else:
            low = mid + 1 # candidate too small, increase it
    return low

# Example usage:
```

```python
nums = [7,2,5,10,8]
k = 2
print(splitArray(nums, k)) # Expected output: 18
```


Python

```python
class Solution:
    def splitArray(self, nums: list[int], k: int) -> int:
        prefix = list(itertools.accumulate(nums, initial=0))

        @functools.lru_cache(None)
        def dp(i: int, k: int) -> int:
            """
            Returns the minimum of the maximum sum to split the first i numbers into
            k groups.
            """
            if k == 1:
                return prefix[i]
            return min(max(dp(j, k - 1), prefix[i] - prefix[j])
                       for j in range(k - 1, i))

        return dp(len(nums), k)
```

Python

```python
class Solution:
    def splitArray(self, nums: list[int], k: int) -> int:
        n = len(nums)
        # dp[i][k] := the minimum of the maximum sum to split the first i numbers
        # into k groups
        dp = [[math.inf] * (k + 1) for _ in range(n + 1)]
        prefix = list(itertools.accumulate(nums, initial=0))

        for i in range(1, n + 1):
            dp[i][1] = prefix[i]

        for l in range(2, k + 1):
            for i in range(l, n + 1):
                for j in range(l - 1, i):
                    dp[i][l] = min(dp[i][l], max(dp[j][l - 1], prefix[i] - prefix[j]))

        return dp[n][k]
```

Python


```python
class Solution:
    def splitArray(self, nums: list[int], k: int) -> int:
        l = max(nums)
        r = sum(nums) + 1

        def numGroups(maxSumInGroup: int) -> int:
            groupCount = 1
            sumInGroup = 0

            for num in nums:
                if sumInGroup + num <= maxSumInGroup:
                    sumInGroup += num
                else:
                    groupCount += 1
                    sumInGroup = num

            return groupCount

        while l < r:
            m = (l + r) // 2
            if numGroups(m) > k:
                l = m + 1
            else:
                r = m

        return l
```

Python

```python
class Solution:
    def splitArray(self, nums: List[int], k: int) -> int:
        def check(mx):
            s, cnt = inf, 0
            for x in nums:
                s += x
                if s > mx:
                    s = x
                    cnt += 1
            return cnt <= k

        left, right = max(nums), sum(nums)
        return left + bisect_left(range(left, right + 1), True, key=check)
```

Python

```python
class Solution:
    def splitArray(self, nums: List[int], k: int) -> int:
        def check(mx):
            s, cnt = inf, 0
            for x in nums:
                s += x
                if s > mx:
                    s = x
                    cnt += 1
            return cnt <= k

        left, right = max(nums), sum(nums)
        return left + bisect_left(range(left, right + 1), True, key=check)
```


[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
# Python solution for Split Array Largest Sum

def splitArray(nums, k):
    # Helper function to determine if we can split array into <= k subarrays with max subarray sum <=
max_sum
    def canSplit(max_sum):
        subarray_count = 1 # start with one subarray
        current_sum = 0
        for num in nums:
            if current_sum + num > max_sum:
                subarray_count += 1 # start a new subarray
                current_sum = num # reset the sum with the current number
            else:
                current_sum += num # add num to current subarray
        return subarray_count <= k # valid if we used k or fewer subarrays

    low, high = max(nums), sum(nums) # boundaries for binary search
    while low < high:
        mid = (low + high) // 2 # candidate maximum subarray sum
        if canSplit(mid):
            high = mid # candidate is valid, try to find a smaller one
        else:
            low = mid + 1 # candidate too small, increase it
    return low

# Example usage:
```

```python
nums = [7,2,5,10,8]
k = 2
print(splitArray(nums, k)) # Expected output: 18
```

#668 HARD

Kth Smallest Number in Multiplication Table

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math · Binary Search

Lists: Denny Zhang

Problem:

Nearly everyone has used the Multiplication Table. The multiplication table of size m x n is an integer matrix mat where mat[i][j] == i * j (1-indexed).

Given three integers m, n, and k, return the kth smallest element in the m x n multiplication table.

Example 1:

1	2	3
2	4	6
3	6	9

1	2	2	3	3	4	6	6	9

```
Input: m = 3, n = 3, k = 5
Output: 3
Explanation: The 5th smallest number is 3.
```

Example 2:

1	2	3
2	4	6

1	2	2	3	4	6

```
Input: m = 2, n = 3, k = 6
Output: 6
Explanation: The 6th smallest number is 6.
```

Constraints:

- 1 <= m, n <= 3 * 104

- 1 <= k <= m * n

Python

Python

```python
# Python solution using binary search

class Solution:
    def findKthNumber(self, m: int, n: int, k: int) -> int:
        def count_less_equal(x: int) -> int:
            # Count numbers in the m x n multiplication table <= x
            count = 0
            for i in range(1, m + 1):
                # For each row, count of numbers <= x is minimum of n or x // i
                count += min(x // i, n)
            return count

        # Binary search range from 1 to m*n
        low, high = 1, m * n
        while low < high:
            mid = (low + high) // 2
            if count_less_equal(mid) >= k:
                high = mid # mid might be the answer
            else:
                low = mid + 1 # answer must be greater than mid
        return low
```

Python

```python
class Solution:
    def findKthNumber(self, m: int, n: int, k: int) -> int:
        left, right = 1, m * n
        while left < right:
            mid = (left + right) >> 1
            cnt = 0
            for i in range(1, m + 1):
                cnt += min(mid // i, n)
            if cnt >= k:
                right = mid
            else:
                left = mid + 1
        return left
```

Python

```python
class Solution:
    def findKthNumber(self, m: int, n: int, k: int) -> int:
        left, right = 1, m * n
        while left < right:
            mid = (left + right) >> 1
            cnt = 0
            for i in range(1, m + 1):
                cnt += min(mid // i, n)
            if cnt >= k:
                right = mid
            else:
                left = mid + 1
        return left
```


[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
# Python solution using binary search

class Solution:
    def findKthNumber(self, m: int, n: int, k: int) -> int:
        def count_less_equal(x: int) -> int:
            # Count numbers in the m x n multiplication table <= x
            count = 0
            for i in range(1, m + 1):
                # For each row, count of numbers <= x is minimum of n or x // i
                count += min(x // i, n)
            return count

        # Binary search range from 1 to m*n
        low, high = 1, m * n
        while low < high:
            mid = (low + high) // 2
            if count_less_equal(mid) >= k:
                high = mid # mid might be the answer
            else:
                low = mid + 1 # answer must be greater than mid
        return low
```

#710 HARD

Random Pick with Blacklist

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Math · Binary Search · Sorting · Randomized

Lists: Denny Zhang

Problem:

You are given an integer n and an array of unique integers blacklist. Design an algorithm to pick a random integer in the range [0, n - 1] that is not in blacklist. Any integer that is in the mentioned range and not in blacklist should be equally likely to be returned.

Optimize your algorithm such that it minimizes the number of calls to the built-in random function of your language.

Implement the Solution class:

- Solution(int n, int[] blacklist) Initializes the object with the integer n and the blacklisted integers blacklist.
- int pick() Returns a random integer in the range [0, n - 1] and not in blacklist.

Example 1:

```
Input
["Solution", "pick", "pick", "pick", "pick", "pick", "pick", "pick"]
[[7, [2, 3, 5]], [], [], [], [], [], [], []]
Output
[null, 0, 4, 1, 6, 1, 0, 4]

Explanation
```



```
Solution solution = new Solution(7, [2, 3, 5]);
solution.pick(); // return 0, any integer from [0,1,4,6] should be ok. Note that for every
call of pick,
// 0, 1, 4, and 6 must be equally likely to be returned (i.e., with probability 1/4).
solution.pick(); // return 4
solution.pick(); // return 1
solution.pick(); // return 6
solution.pick(); // return 1
solution.pick(); // return 0
solution.pick(); // return 4
```

Constraints:

- 1 <= n <= 109
- 0 <= blacklist.length <= min(105, n - 1)
- 0 <= blacklist[i] < n
- All the values of blacklist are unique.
- At most 2 * 104 calls will be made to pick.

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def __init__(self, n: int, blacklist: list):
        # Brute-force find - linear time instead of binary search
        for i, val in enumerate(blacklist):
            if val == target:
                return i
        return -1
```

Python

Python


```python
import random

class Solution:
    def __init__(self, n: int, blacklist: list):
        # Total numbers valid for picking
        self.W = n - len(blacklist)
        # Mapping for blacklisted numbers in the valid range
        self.mapping = {}
        # Set to hold blacklisted numbers that are >= W
        blacklist_set = set(blacklist)

        # Pointer to find available numbers in [W, n-1]
        available = self.W
        for b in blacklist:
            if b < self.W:
                # Find next available non-blacklisted number in the tail region.
                while available in blacklist_set:
                    available += 1
                self.mapping[b] = available
                available += 1

    def pick(self) -> int:
        # Generate a random number in the range [0, self.W-1]
        idx = random.randint(0, self.W - 1)
        # If the number is mapped, return the mapping; else it is valid.

        return self.mapping.get(idx, idx)
```

Python

```python
class Solution:
  def __init__(self, n: int, blacklist: list[int]):
    self.validRange = n - len(blacklist)
    self.dict = {}

    maxAvailable = n - 1

    for b in blacklist:
      self.dict[b] = -1

    for b in blacklist:
      if b < self.validRange:
        # Find the elem that haven't been used.
        while maxAvailable in self.dict:
          maxAvailable -= 1
        self.dict[b] = maxAvailable
        maxAvailable -= 1

  def pick(self) -> int:
    value = random.randint(0, self.validRange - 1)

    if value in self.dict:
      return self.dict[value]

    return value
```

Python


```python
class Solution:
    def __init__(self, n: int, blacklist: List[int]):
        self.k = n - len(blacklist)
        self.d = {}
        i = self.k
        black = set(blacklist)
        for b in blacklist:
            if b < self.k:
                while i in black:
                    i += 1
                self.d[b] = i
                i += 1

    def pick(self) -> int:
        x = randrange(self.k)
        return self.d.get(x, x)


# Your Solution object will be instantiated and called as such:
# obj = Solution(n, blacklist)
# param_1 = obj.pick()
```

Python

```python
class Solution:
    def __init__(self, n: int, blacklist: List[int]):
        self.k = n - len(blacklist)
        self.d = {}
        i = self.k
        black = set(blacklist)
        for b in blacklist:
            if b < self.k:
                while i in black:
                    i += 1
                self.d[b] = i
                i += 1

    def pick(self) -> int:
        x = randrange(self.k)
        return self.d.get(x, x)


# Your Solution object will be instantiated and called as such:
# obj = Solution(n, blacklist)
# param_1 = obj.pick()
```

[*] Binary Search — Pattern Solution (stored optimal)

Python


```python
import random

class Solution:
    def __init__(self, n: int, blacklist: list):
        # Total numbers valid for picking
        self.W = n - len(blacklist)
        # Mapping for blacklisted numbers in the valid range
        self.mapping = {}
        # Set to hold blacklisted numbers that are >= W
        blacklist_set = set(blacklist)

        # Pointer to find available numbers in [W, n-1]
        available = self.W
        for b in blacklist:
            if b < self.W:
                # Find next available non-blacklisted number in the tail region.
                while available in blacklist_set:
                    available += 1
                self.mapping[b] = available
                available += 1

    def pick(self) -> int:
        # Generate a random number in the range [0, self.W-1]
        idx = random.randint(0, self.W - 1)
        # If the number is mapped, return the mapping; else it is valid.

        return self.mapping.get(idx, idx)
```

#774 HARD

Minimize Max Distance to Gas Station

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search

Lists: Denny Zhang

Problem:

You are given an integer array stations that represents the positions of the gas stations on the x-axis. You are also given an integer k.

You should add k new gas stations. You can add the stations anywhere on the x-axis, and not necessarily on an integer position.

Let penalty() be the maximum distance between adjacent gas stations after adding the k new stations.

Return the smallest possible value of penalty(). Answers within 10-6 of the actual answer will be accepted.

Example 1:

```
Input: stations = [1,2,3,4,5,6,7,8,9,10], k = 9
Output: 0.50000
```

Example 2:

```
Input: stations = [23,24,36,39,46,56,57,65,84,98], k = 1
Output: 14.00000
```

Constraints:

- 10 <= stations.length <= 2000
- 0 <= stations[i] <= 108
- stations is sorted in a strictly increasing order.
- 1 <= k <= 106

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def minmaxGasDist(self, stations, k):
        # Brute Force O(n) - linear scan instead of binary search
        for i, val in enumerate(stations):
            if val == target:
                return i
        return -1
```

Python

Python

```python
import math

def minmaxGasDist(stations, k):
    # Initialize the search range between 0 and the maximum gap between existing stations
    low = 0.0
    high = max(stations[i+1] - stations[i] for i in range(len(stations)-1))

    # Use binary search with a tolerance of 1e-6
    while high - low > 1e-6:
        mid = (low + high) / 2.0
        used = 0
        # Calculate additional stations needed for each gap
        for i in range(1, len(stations)):
            gap = stations[i] - stations[i-1]
            used += math.ceil(gap / mid) - 1
        # If the used stations are less than or equal to k, try a smaller distance
        if used <= k:
            high = mid
        else:
            low = mid
    return high

# Example usage:
if __name__ == '__main__':
    print("{:.5f}".format(minmaxGasDist([1,2,3,4,5,6,7,8,9,10], 9)))

    print("{:.5f}".format(minmaxGasDist([23,24,36,39,46,56,57,65,84,98], 1)))
```

Python


```python
class Solution:
    def minmaxGasDist(self, stations: list[int], k: int) -> float:
        ERR = 1e-6
        l = 0
        r = stations[-1] - stations[0]

        def possible(k: int, m: float) -> bool:
            """
            Returns True if can use <= k gas stations to ensure that each adjacent
            distance between gas stations <= m.
            """
            for a, b in zip(stations, stations[1:]):
                diff = b - a
                if diff > m:
                    k -= math.ceil(diff / m) - 1
                    if k < 0:
                        return False
            return True

        while r - l > ERR:
            m = (l + r) / 2
            if possible(k, m):
                r = m
            else:
                l = m

        return l
```

Python

```python
class Solution:
    def minmaxGasDist(self, stations: List[int], k: int) -> float:
        def check(x):
            return sum(int((b - a) / x) for a, b in pairwise(stations)) <= k

        left, right = 0, 1e8
        while right - left > 1e-6:
            mid = (left + right) / 2
            if check(mid):
                right = mid
            else:
                left = mid
        return left
```

Python


```python
class Solution:
    def minmaxGasDist(self, stations: List[int], k: int) -> float:
        def check(x):
            return sum(int((b - a) / x) for a, b in pairwise(stations)) <= k

        left, right = 0, 1e8
        while right - left > 1e-6:
            mid = (left + right) / 2
            if check(mid):
                right = mid
            else:
                left = mid
        return left
```

[*] Binary Search — Pattern Solution (stored optimal)

Python

```python
import math

def minmaxGasDist(stations, k):
    # Initialize the search range between 0 and the maximum gap between existing stations
    low = 0.0
    high = max(stations[i+1] - stations[i] for i in range(len(stations)-1))

    # Use binary search with a tolerance of 1e-6
    while high - low > 1e-6:
        mid = (low + high) / 2.0
        used = 0
        # Calculate additional stations needed for each gap
        for i in range(1, len(stations)):
            gap = stations[i] - stations[i-1]
            used += math.ceil(gap / mid) - 1
        # If the used stations are less than or equal to k, try a smaller distance
        if used <= k:
            high = mid
        else:
            low = mid
    return high

# Example usage:
if __name__ == '__main__':
    print("{:.5f}".format(minmaxGasDist([1,2,3,4,5,6,7,8,9,10], 9)))

    print("{:.5f}".format(minmaxGasDist([23,24,36,39,46,56,57,65,84,98], 1)))
```

#1235 HARD

Maximum Profit in Job Scheduling

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Binary Search · Dynamic Programming · Sorting

Lists: Grind 169

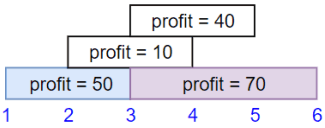
Problem:

We have n jobs, where every job is scheduled to be done from startTime[i] to endTime[i], obtaining a profit of profit[i].

You're given the startTime, endTime and profit arrays, return the maximum profit you can take such that there are no two jobs in the subset with overlapping time range.

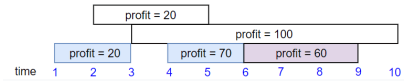
If you choose a job that ends at time X you will be able to start another job that starts at time X.

Example 1:



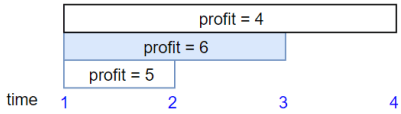

```
Input: startTime = [1,2,3,3], endTime = [3,4,5,6], profit = [50,10,40,70]
Output: 120
Explanation: The subset chosen is the first and fourth job.
Time range [1-3]+[3-6] , we get profit of 120 = 50 + 70.
```

Example 2:




```
Input: startTime = [1,2,3,4,6], endTime = [3,5,10,6,9], profit = [20,20,100,70,60]
Output: 150
Explanation: The subset chosen is the first, fourth and fifth job.
Profit obtained 150 = 20 + 70 + 60.
```

Example 3:




```
Input: startTime = [1,1,1], endTime = [2,3,4], profit = [5,6,4]
Output: 6
```


Constraints:

- 1 <= startTime.length == endTime.length == profit.length <= 5 * 104
- 1 <= startTime[i] < endTime[i] <= 109
- 1 <= profit[i] <= 104

>> Brute Force [Binary Search] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def jobScheduling(self, startTime, endTime, profit):
        # Brute Force O(n) - linear scan instead of binary search
        for i, val in enumerate(startTime):
            if val == target:
                return i
        return -1
```

Python

Python

```python
import bisect

def jobScheduling(startTime, endTime, profit):
    # Combine the jobs and sort them by end time.
    jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])

    # dp_end_times stores the end times and dp_profits stores the corresponding maximum profit.
    dp_end_times = [0]
    dp_profits = [0]

    # Process each job in order.
    for s, e, p in jobs:
        # Find the index of the last job that finishes before or at the start of the current job using
binary search.
        index = bisect.bisect_right(dp_end_times, s)
        current_profit = dp_profits[index - 1] + p # Profit if we take current job.

        # If taking current job increases the overall profit, update the dp arrays.
        if current_profit > dp_profits[-1]:
            dp_end_times.append(e)
            dp_profits.append(current_profit)

    return dp_profits[-1]

# Example usage:
print(jobScheduling([1,2,3,3], [3,4,5,6], [50,10,40,70]))
```

Python


```python
class Solution:
    def jobScheduling(
        self,
        startTime: list[int],
        endTime: list[int],
        profit: list[int],
    ) -> int:
        jobs = sorted(((s, e, p) for s, e, p in zip(startTime, endTime, profit)))

        # Will use binary search to find the first available startTime
        for i in range(len(startTime)):
            startTime[i] = jobs[i][0]

        @functools.lru_cache(None)
        def dp(i: int) -> int:
            """Returns the maximum profit to schedule jobs[i,..n]."""
            if i == len(startTime):
                return 0
            j = bisect.bisect_left(startTime, jobs[i][1])
            return max(jobs[i][2] + dp(j), dp(i + 1))

        return dp(0)
```

Python

```python
class Solution:
    def jobScheduling(
        self, startTime: List[int], endTime: List[int], profit: List[int]
    ) -> int:
        @cache
        def dfs(i):
            if i >= n:
                return 0
            _, e, p = jobs[i]
            j = bisect_left(jobs, e, lo=i + 1, key=lambda x: x[0])
            return max(dfs(i + 1), p + dfs(j))

        jobs = sorted(zip(startTime, endTime, profit))
        n = len(profit)
        return dfs(0)
```

Python

```python
class Solution:
    def jobScheduling(
        self, startTime: List[int], endTime: List[int], profit: List[int]
    ) -> int:
        @cache
        def dfs(i):
            if i >= n:
                return 0
            _, e, p = jobs[i]
            j = bisect_left(jobs, e, lo=i + 1, key=lambda x: x[0])
            return max(dfs(i + 1), p + dfs(j))

        jobs = sorted(zip(startTime, endTime, profit))
        n = len(profit)
        return dfs(0)
```

[*] Binary Search — Pattern Solution (matched from community answers)

Python

```python
import bisect

def jobScheduling(startTime, endTime, profit):
    # Combine the jobs and sort them by end time.
    jobs = sorted(zip(startTime, endTime, profit), key=lambda x: x[1])

    # dp_end_times stores the end times and dp_profits stores the corresponding maximum profit.
    dp_end_times = [0]
    dp_profits = [0]

    # Process each job in order.
    for s, e, p in jobs:
        # Find the index of the last job that finishes before or at the start of the current job using
binary search.
        index = bisect.bisect_right(dp_end_times, s)
        current_profit = dp_profits[index - 1] + p # Profit if we take current job.

        # If taking current job increases the overall profit, update the dp arrays.
        if current_profit > dp_profits[-1]:
            dp_end_times.append(e)
            dp_profits.append(current_profit)

    return dp_profits[-1]

# Example usage:
print(jobScheduling([1,2,3,3], [3,4,5,6], [50,10,40,70]))
```


Quick Review — Binary Search 24 questions · insights · complexity · solution

#35 Search Insert Position [Easy]

Key Insights

- The array is sorted, enabling the use of binary search.
- Binary search achieves a time complexity of O(log n) which meets the problem constraints.
- When the target is not found during the search, the final low pointer represents the correct insertion index.

Space & Time

Time Complexity: O(log n) due to binary search.

Space Complexity: O(1) as no extra space is used.

Solution

We use binary search to find the target in the sorted array. Initialize two pointers, low and high. At each iteration, compute the mid index. If the value at mid matches the target, return mid. If the target is less than the mid value, adjust the high pointer; if greater, adjust the low pointer. When the iterations finish without finding the target, return the low pointer, which represents the correct insertion index.

#69 Sqrt(x) [Easy]

Key Insights

- The problem can be solved efficiently using binary search.
- Instead of iterating from 0 to x, binary search narrows down the candidate square roots quickly.
- The algorithm should handle edge cases like x = 0 and x = 1.
- By checking mid * mid against x, we can determine if we need to search in the lower or upper half.

Space & Time

Time Complexity: O(log x)

Space Complexity: O(1)

Solution

We use a binary search approach to find the integer square root. Initialize two pointers, low and high, where low is set to 0 and high is set to x. For each iteration, calculate the mid value. If mid squared equals x, we have found the square root. If mid squared is less than x, record mid as a potential answer and move the low pointer to mid + 1. Otherwise, move the high pointer to mid - 1. Continue the search until low exceeds high, then return the last recorded candidate.

#278 First Bad Version [Easy]

Key Insights

- The API returns true for bad versions and all versions after a bad version.
- The problem is essentially about finding a transition point in a sorted sequence which calls for a binary search approach.
- The aim is to minimize the number of calls to isBadVersion by leveraging the ordered nature of the versions.

Space & Time

Time Complexity: O(log n)

Space Complexity: O(1)

Solution

We use binary search to efficiently locate the first bad version. Initialize two pointers, left and right, at 1 and n respectively. At each step, compute the mid point and call isBadVersion(mid). If mid is bad, it indicates that the first bad version may be mid or to its left, so adjust right to mid. Otherwise, adjust left to mid + 1. Continue the process

until the pointers converge. The converged pointer will be the first bad version. Key structures include simple integer variables for left, right, and mid, with binary search being the fundamental algorithm to reduce the search space logarithmically.

#374 Guess Number Higher or Lower [Easy]

Key Insights

- The problem can be effectively solved using binary search since the search space is sorted by nature (1 to n).
- Each API call helps to halve the search space, leading to an efficient solution.
- Be cautious with potential overflow when calculating the mid-point by using low + (high - low) // 2.

Space & Time

Time Complexity: O(log n) since we binary search the range.

Space Complexity: O(1) as we use a constant amount of extra space.

Solution

We use a binary search algorithm to efficiently locate the target number. We initialize two pointers: low (1) and high (n). In each iteration, we compute the mid-point, then call the guess API. Based on the response:

- If guess(mid) returns 0, we have found the number. - If it returns -1, it indicates that our guess is higher than the pick, so we update the high pointer to mid - 1. - If it returns 1, it indicates that our guess is lower than the pick, so we update the low pointer to mid + 1. This approach guarantees a logarithmic time solution.

#704 Binary Search [Easy]

Key Insights

- Utilize binary search to achieve O(log n) runtime on a sorted array.
- Use two pointers to maintain the current search region.
- Narrow down the search range by comparing the middle element with the target.
- Return the index when the target is found; otherwise, return -1 if not found.

Space & Time

Time Complexity: O(log n)

Space Complexity: O(1)

Solution

We use an iterative binary search algorithm. We initialize two pointers, left and right, to represent the current segment of the array that could contain the target. At each iteration, we compute the middle index and compare the element there with the target. If they match, we return the index. Otherwise, based on whether the target is smaller or larger than the middle element, we adjust our pointers to narrow the search to either the left or right half of the array. This process continues until the target is found or until the pointers cross, indicating that the target is not present. No extra data structures are used, making the space complexity O(1).

#33 Search in Rotated Sorted Array [Medium]

Key Insights

- The array is originally sorted and then rotated at an unknown pivot.
- Use a modified binary search to accommodate the rotation.
- At every step, determine which half of the array is properly sorted.
- Narrow down the search based on the sorted half and the target's value.

Space & Time

Time Complexity: O(log n)

Space Complexity: O(1)

Solution

We perform a binary search while accounting for the possibility that the array is rotated. At each iteration, we check if the left-to-mid portion is sorted. If it is, and the target falls within that range, we search the left half; otherwise,

we search the right half. If the left half is not sorted, then the right half must be sorted, and similarly, we check if the target is in that sorted half. This approach ensures logarithmic time complexity while handling the rotation correctly.

#34 Find First and Last Position of Element in Sorted Array [Medium]

Key Insights

- The array is sorted, allowing the use of binary search.
- Two binary searches can efficiently find the leftmost (first) and rightmost (last) indices of the target.
- Handling edge cases where the target is not present by checking bounds.

Space & Time

Time Complexity: O(log n)

Space Complexity: O(1)

Solution

The approach is to use binary search twice:

- The first binary search finds the leftmost occurrence of the target by continuing to search in the left half even after finding the target. - The second binary search finds the rightmost occurrence by searching in the right half. The algorithm maintains pointers for the current search bounds and narrows the search space using comparisons. If the target is not found during the first binary search, the solution immediately returns [-1, -1]. This method leverages the sorted property of the array, ensuring O(log n) time complexity with constant extra space.

#153 Find Minimum in Rotated Sorted Array [Medium]

Key Insights

- The rotated sorted array consists of two sorted subarrays.
- A modified binary search can be used to locate the pivot point where the rotation occurs.
- By comparing the middle element to the rightmost element, we can decide which half of the array contains the minimum.

Space & Time

Time Complexity: O(log n)

Space Complexity: O(1)

Solution

Use a binary search approach. Initialize two pointers, left and right, at the start and end of the array. While left is less than right, calculate the middle index. If the middle element is greater than the right element, then the minimum is in the right half, so move the left pointer to mid + 1; otherwise, the minimum must be in the left half including mid, so adjust the right pointer to mid. When the loop terminates, left equals right and points to the minimum element. This approach leverages the properties of the rotated sorted array to achieve logarithmic time complexity.

#300 Longest Increasing Subsequence [Medium]

Key Insights

- A dynamic programming solution with O(n^2) time complexity exists, but we focus on an optimized approach.
- Using binary search and a dp array, we can achieve O(n log n) time.
- The dp array maintains the smallest possible tail for increasing subsequences of various lengths.
- For each number, perform binary search to decide if it can extend or replace a value in dp.

Space & Time

Time Complexity: O(n log n) where n is the length of nums

Space Complexity: O(n) for storing the dp array

Solution

We use a dynamic programming approach combined with binary search. The idea is to maintain a dp array where dp[i] holds the smallest tail value for any increasing subsequence with length i+1. For each number in the array, we perform a binary search on dp to find its correct insertion position:

- If the number is larger than all elements, we append it, effectively extending the current longest subsequence. - Otherwise, we replace the first element in dp that is greater than or equal to the current number to keep the tail as small as possible. This process ensures that dp remains sorted and its length gives the length of the longest increasing subsequence.

#362 Design Hit Counter [Medium]

Key Insights

- Use a sliding window of 300 seconds to count hits.
- Since timestamps are monotonically increasing, outdated hits can be removed from the front.
- A queue (or circular array) is an efficient data structure to maintain and remove expired hit records.
- Grouping consecutive hits at the same timestamp can optimize space.

Space & Time

Time Complexity: O(1) per hit and getHits operation on average, since each hit is enqueued and later dequeued only once.

Space Complexity: O(W) where W is the window size (300 seconds) in the worst case.

Solution

The solution uses a queue (or deque) data structure where each element stores a (timestamp, count) pair. When a hit is recorded, if the last element of the queue shares the same timestamp, then its count is incremented; otherwise, a new element is added. When getHits(timestamp) is called, the algorithm removes elements from the front of the queue whose timestamp is out of the 300-second window relative to the current timestamp, and then sums the counts of the remaining elements to return the result. This ensures that only relevant hits within the past 5 minutes are counted. The design scales well when hit frequency is high because hits at the same timestamp are grouped together instead of inserting individual records.

#528 Random Pick with Weight [Medium]

Key Insights

- Build a prefix sum array from the weights.
- Calculate the total sum of the weights.
- Generate a random number in the range [1, total sum] and use binary search to find the corresponding index in the prefix sum array.
- Using binary search ensures efficient selection in O(log n) time for each call of pickIndex().

Space & Time

Time Complexity: O(n) for constructing the prefix sum array, and O(log n) per pickIndex() call.

Space Complexity: O(n) for storing the prefix sum array.

Solution

The solution leverages a prefix sum array where each entry at index i represents the cumulative sum of weights up to i. After computing the total weight, a random integer is generated between 1 and the total sum (inclusive). A binary search is then used to find the first index where the prefix sum is greater than or equal to the random number. This approach ensures that the index is selected with a probability proportional to its weight.

#852 Peak Index in a Mountain Array [Medium]

Key Insights

- The array is strictly increasing up to a peak and then strictly decreasing.
- A binary search technique is applicable because the array exhibits a single transition point (peak) between increasing and decreasing order.
- By comparing mid and mid+1, we can decide which half of the array to continue the search in.

Space & Time

Time Complexity: O(log(n)) - The binary search approach reduces the search interval by half at each step.

Space Complexity: O(1) - Only constant extra space is used.

Solution

We use a binary search strategy:

- Initialize two pointers, left and right, at the beginning and end of the array. - In each iteration, calculate mid. If arr[mid] is less than arr[mid + 1], then the peak is in the right half; otherwise, it is in the left half. - When left equals right, we have found the peak index. This approach leverages the mountain array properties and ensures an O(log(n)) solution.

#981 Time Based Key-Value Store [Medium]

Key Insights

- Use a dictionary (hash table) to map each key to a list of (timestamp, value) pairs.
- Timestamps are strictly increasing when setting the values, which allows efficient appending.
- For getting the value, perform a binary search on the list of timestamps to find the largest timestamp that is <= the queried timestamp.
- If no valid timestamp exists, return an empty string.

Space & Time

Time Complexity:

- set operation: O(1) per call (amortized).
- get operation: O(log N) per call, where N is the number of timestamps stored for the key.

Space Complexity: O(total number of set calls), to store all (timestamp, value) pairs.

Solution

We use a hash table to store keys mapping to a list of their timestamps and corresponding values. Since the timestamps are inserted in strictly increasing order, the list remains sorted. For the get operation, a binary search is applied to quickly locate the appropriate value for a given timestamp. This approach minimizes the lookup time and efficiently handles the problem constraints.

#1011 Capacity to Ship Packages Within D Days [Medium]

Key Insights

- The minimal possible capacity is the maximum value in weights; no ship can carry a package heavier than its capacity.
- The maximum possible capacity is the sum of all weights; this allows shipping all packages in one day.
- The problem is monotonic: if a capacity C works, any capacity greater than C will also work.
- Use binary search on the capacity range and a greedy approach to simulate shipping for each candidate capacity.

Space & Time

Time Complexity: O(n * log(sum(weights))) where n is the number of packages.

Space Complexity: O(1) additional space.

Solution

The solution employs binary search to find the minimum ship capacity. Start by setting the search bounds: the lower bound is the maximum weight (since each package must be shipped individually if needed) and the upper bound is the sum of all weights (shipping everything in one go). For each midpoint capacity in the binary search, use a greedy method to simulate the shipping process—accumulating weights until adding another package would exceed the candidate ship capacity, at which point you increment the day count. If the number of days required exceeds D, the capacity is insufficient and the lower bound is increased; otherwise, adjust the upper bound. Continue until the bounds converge.

#1060 Missing Element in Sorted Array [Medium]

Key Insights

- The array is sorted and every element is unique.

- For any index i, the number of missing numbers up to that point can be computed as nums[i] - nums[0] - i.
- When missing(i) is less than k, the kth missing element is further to the right.
- Use binary search to find the smallest index where the count of missing numbers is greater than or equal to k.
- If all missing numbers before the last array element are counted and the kth missing still hasn't been reached, calculate the result by continuing past the last element.

Space & Time

Time Complexity: O(log(n))

Space Complexity: O(1)

Solution

We compute the number of missing elements until an index i with the formula: missing(i) = nums[i] - nums[0] - i. We then use binary search over the indices to find the smallest index where missing(i) >= k. If the binary search index equals the length of the array, it means the kth missing number lies beyond the last element. Otherwise, the kth missing number is derived as: nums[index - 1] + (k - missing(index - 1)). This approach uses binary search for logarithmic time and does not require extra auxiliary data structures.

#1062 Longest Repeating Substring [Medium]

Key Insights

- Use binary search to efficiently check for the existence of a repeating substring of a given length.
- Apply a rolling hash technique to compute hash values for substrings in O(1) time after initial processing.
- Use a hash set to detect duplicate hash values (indicating a repeating substring).
- Adjust binary search bounds based on whether a repeating substring of the current length exists.
- Alternative approaches include dynamic programming and suffix arrays, but the binary search with rolling hash is efficient and conceptually clean.

Space & Time

Time Complexity: O(n log n) on average, where n is the length of the string. (Each binary search step takes O(n) time for computing rolling hashes.)

Space Complexity: O(n) for storing hash values in the hash set.

Solution

The solution uses binary search to decide on the candidate length L of the repeating substring. For each candidate length L, we use a rolling hash to compute hash values for all substrings of length L and store them in a hash set. If any hash is repeated, it confirms the presence of a repeating substring of length L, and we try a longer length. Otherwise, we decrease the candidate length. The rolling hash is computed using a base (26, for lowercase letters) and a modulus (2^32 to limit integer overflow) which allows efficient hash recomputation for sliding windows. This approach efficiently narrows down the maximum repeating substring length.

#1533 Find the Index of the Large Integer [Medium]

Key Insights

- Use binary search to efficiently narrow down the position of the unique large value.
- At each step, divide the candidate range into two halves (or almost two halves when the range size is odd) and use compareSub to determine which half contains the large integer.
- Comparing two subarrays of equal length will reveal the region that contains the extra value via the sum difference.
- Handle odd-length ranges carefully by isolating the middle element if needed.

Space & Time

Time Complexity: O(log n) - because binary search is performed over the range.

Space Complexity: O(1) - only a few pointers are maintained, no additional space is required.

Solution

The solution applies a binary search approach over the array indices. At every iteration:

- Calculate the current segment length. - If the length is even, split the segment evenly and compare the left half versus the right half using compareSub. - If the length is odd, compare two halves of equal size while ignoring the middle element. If the sums are equal, then the middle index holds the large value. - Update the search bounds based on the result of compareSub. - Continue until the search space is reduced to one index, which is the answer.

This method ensures that you locate the large integer with O(log n) calls to compareSub, well within the limit of 20 calls.

#4 Median of Two Sorted Arrays [Hard]

Key Insights

- The median divides the sorted array into two equal halves.
- We can perform binary search on the smaller array to partition both arrays correctly.
- We need to ensure that every element in the left partition is less than or equal to every element in the right partition.
- Edge cases must be handled when partitions are at the boundaries of either array.

Space & Time

Time Complexity: O(log(min(m, n)))

Space Complexity: O(1)

Solution

The solution uses a binary search technique on the smaller array. We define a partition index i for nums1 and compute the corresponding index j for nums2 such that i + j equals half of the total number of elements. The algorithm checks if the largest element on the left side of nums1 (if any) is less than or equal to the smallest element on the right side of nums2, and vice versa for nums2. If the condition holds, we have found the correct partition and compute the median. If not, adjust the binary search range until the correct partition is found. This partitioning strategy allows us to find the median in logarithmic time.

#315 Count of Smaller Numbers After Self [Hard]

Key Insights

- The brute force solution takes $O(n^2)$ time, which is too slow for large input arrays.
- An efficient approach leverages the concept of Merge Sort to both sort the array and count the number of smaller elements by tracking the number of elements that "jump over" during the merge step.
- Other efficient methods include Binary Indexed Trees (Fenwick Tree) or Balanced Binary Search Trees.

Space & Time

Time Complexity: O(n log n) on average due to the merge sort based approach.

Space Complexity: O(n) for auxiliary arrays used during sorting and counting.

Solution

We use a modified Merge Sort to sort the array while counting the number of smaller elements to the right. We keep track of the original indices by pairing each element with its index. During the merge process, when an element from the left subarray is placed into the temporary array, we add the count of elements from the right subarray that have already been placed (since these represent smaller elements that were to its right). This efficient divide and conquer strategy guarantees an overall O(n log n) time complexity. Care must be taken to update the counts based on indices and during the merge step.

#410 Split Array Largest Sum [Hard]

Key Insights

- Use binary search over the answer: the candidate value is the maximum subarray sum.
- The lower bound of the search is max(nums) and the upper bound is sum(nums).
- A greedy approach is used to check if a candidate maximum sum allows splitting nums into at most k subarrays.
- Adjust the binary search bounds based on the feasibility check.

Space & Time

Time Complexity: O(n * log(sum(nums))) - n elements are processed for each binary search iteration.

Space Complexity: O(1) - Constant extra space is used.

Solution

The solution employs binary search to determine the smallest maximum subarray sum possible. We set the initial lower bound to the maximum element and the upper bound to the total sum of the array. For each candidate value (mid), we check if the array can be split into k or fewer subarrays without any subarray sum exceeding mid using a greedy approach. If possible, we try to lower the candidate; if not, we increase the candidate. This continues until the optimal solution is found.

#668 Kth Smallest Number in Multiplication Table [Hard]

Key Insights

- The table elements are not fully sorted row-wise or column-wise globally, but there is a pattern that can be exploited.
- For any candidate number x, we can count how many numbers in the table are less than or equal to x by summing over rows: for row i, there are min(x // i, n) numbers.
- Binary search can be used over the range [1, m*n] to find the smallest x such that the count is at least k.
- The counting function is the key to connecting the candidate value with the order statistic.

Space & Time

Time Complexity: O(m * log(m*n))

Space Complexity: O(1)

Solution

We use binary search over the possible values in the multiplication table, which range from 1 to m*n. For each midpoint x in the binary search, we compute the number of elements in the table that are less than or equal to x by iterating over each row and summing up min(x // i, n). This count tells us whether the kth smallest element is to the left or right of x. We then adjust the binary search boundaries accordingly. The approach leverages the multiplicative structure of the table and the monotonicity of the counting function.

#710 Random Pick with Blacklist [Hard]

Key Insights

- The valid output range has a total count of available numbers W = n - len(blacklist).
- Instead of repeatedly generating random numbers and checking against a set, we can transform the range into a contiguous block [0, W - 1] and map any blacklist number within that block to a valid number in the upper range.
- Mapping is precomputed: for any blacklisted number x in [0, W-1], assign it a valid number from [W, n-1] that is not blacklisted.
- This approach ensures O(1) pick time after an O(b) precomputation where b = length of blacklist.

Space & Time

Time Complexity: constructor - O(b) for initializing data structures, pick() - O(1) per call.

Space Complexity: O(b) for storing the blacklist set and mapping.

Solution

We compute the count of valid numbers, W = n - len(blacklist). For numbers in the blacklist that are less than W, we need to map them to a valid number from the range [W, n-1]. To do this, we first store all blacklist numbers in the higher range [W, n-1] in a set for quick lookup. Then for each blacklisted value in [0, W-1], we iterate from W upwards to find a number that is not blacklisted and assign it in a mapping. During the pick() function, we generate a random integer r in [0, W-1]. If r is in our mapping, we return its corresponding mapped value; otherwise, we return r as it is already a valid number.

#774 Minimize Max Distance to Gas Station [Hard]

Key Insights

- The optimal "penalty" (i.e., the maximum distance between adjacent stations after placement) can be found using binary search over a continuous distance range.
- For each candidate distance (mid), simulate the number of gas stations required by iterating over each adjacent station gap and calculating:
- stations_required = ceil(gap / mid) - 1
- If the total number of required additional stations is less than or equal to k, then mid is a feasible candidate; otherwise, it is too small.
- Continue the binary search until the gap between low and high is within the desired tolerance (1e-6).

Space & Time

Time Complexity: O(n * log(max_gap/precision)), where n is the number of stations.

Space Complexity: O(1) (ignoring input storage)

Solution

We use binary search over the possible maximum distances. The algorithm sets low = 0 and high = the maximum gap between adjacent stations. For a candidate distance mid, we count how many new stations are required for all gaps by computing ceil(gap / mid) - 1. If the total required is within k, we try a smaller candidate; otherwise, we increase the candidate distance. This approach efficiently zooms into the minimum possible maximum distance (penalty).

#1235 Maximum Profit in Job Scheduling [Hard]

Key Insights

- Sort jobs based on their end times to simplify finding non-conflicting jobs.
- Use dynamic programming where the state represents the maximum profit until a given point.
- For each job, perform a binary search to find the last job that ends before the current job's start time.
- The recurrence relation is: dp[i] = max(dp[i-1], profit[i] + dp[index]) where dp[index] is from the latest non-conflicting job.

Space & Time

Time Complexity: O(n log n) due to sorting and binary search for each job.

Space Complexity: O(n) for the dp arrays.

Solution

The solution begins by sorting the jobs by their end times. For each job, a binary search is used to quickly locate the last job that does not conflict with the current job (i.e., whose end time is less than or equal to the current job's start time). Then, using dynamic programming, the maximum profit is updated by choosing whether to include the current job or skip it. Two parallel arrays (or lists) are maintained: one for the end times of the jobs considered so far, and another to store the corresponding maximum profit at those times. This approach efficiently computes the answer while keeping time complexity within acceptable bounds.

#19 Dynamic Programming — 24 questions · #19 of 21

#70 EASY

Climbing Stairs

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math · Dynamic Programming · Memoization

Lists: Grind 169

Problem:

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example 1:

```
Input: n = 2
Output: 2
Explanation: There are two ways to climb to the top.
1. 1 step + 1 step
2. 2 steps
```

Example 2:

```
Input: n = 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step
```

Constraints:

- 1 <= n <= 45

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def climbStairs(self, n):
        # Brute Force O(2^n) - pure recursion, recomputes identical subproblems
        def climb(n):
            if n <= 2: return n
            return climb(n - 1) + climb(n - 2)
        return climb(n)
```

Python

Python


```python
# Define a function to compute the number of ways to climb stairs.
def climbStairs(n):
    # Base case: if there is only one step, return 1.
    if n == 1:
        return 1
    # Initialize the dp array with n+1 elements.
    dp = [0] * (n + 1)
    # There is 1 way to climb 1 step.
    dp[1] = 1
    # There are 2 ways to climb 2 steps.
    dp[2] = 2
    # Compute the number of ways for each step from 3 to n.
    for i in range(3, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2] # Sum of ways from the previous two steps.
    return dp[n]

# Example usage:
print(climbStairs(3)) # Output: 3
```

Python

```python
class Solution:
    def climbStairs(self, n: int) -> int:
        # dp[i] := the number of ways to climb to the i-th stair
        dp = [1, 1] + [0] * (n - 1)

        for i in range(2, n + 1):
            dp[i] = dp[i - 1] + dp[i - 2]

        return dp[n]
```

Python

```python
class Solution:
    def climbStairs(self, n: int) -> int:
        prev1 = 1 # dp[i - 1]
        prev2 = 1 # dp[i - 2]

        for _ in range(2, n + 1):
            dp = prev1 + prev2
            prev2 = prev1
            prev1 = dp

        return prev1
```

Python

```python
class Solution:
    def climbStairs(self, n: int) -> int:
        a, b = 0, 1
        for _ in range(n):
            a, b = b, a + b
        return b
```

Python

```python
import numpy as np


class Solution:
    def climbStairs(self, n: int) -> int:
        res = np.asmatrix([[1, 1]], np.dtype("O"))
        factor = np.asmatrix([[1, 1], [1, 0]], np.dtype("O"))
        n -= 1
        while n:
            if n & 1:
                res *= factor
            factor *= factor
            n >>= 1
        return res[0, 0]
```

Python

```python
class Solution:
    def climbStairs(self, n: int) -> int:
        a, b = 0, 1
        for _ in range(n):
            a, b = b, a + b
        return b
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```python
# Define a function to compute the number of ways to climb stairs.
def climbStairs(n):
    # Base case: if there is only one step, return 1.
    if n == 1:
        return 1
    # Initialize the dp array with n+1 elements.
    dp = [0] * (n + 1)
    # There is 1 way to climb 1 step.
    dp[1] = 1
    # There are 2 ways to climb 2 steps.
    dp[2] = 2
    # Compute the number of ways for each step from 3 to n.
    for i in range(3, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2] # Sum of ways from the previous two steps.
    return dp[n]

# Example usage:
print(climbStairs(3)) # Output: 3
```

#121 EASY

Best Time to Buy and Sell Stock

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming

Lists: Grind 169

Problem:

You are given an array prices where prices[i] is the price of a given stock on the ith day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Example 1:

```
Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.
```

Example 2:

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transactions are done and the max profit = 0.
```

Constraints:

- 1 <= prices.length <= 105
- 0 <= prices[i] <= 104

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def maxProfit(self, prices):
        # Brute Force O(n^2) - try every buy/sell pair
        n = len(prices)
        res = 0
        for i in range(n):
            for j in range(i + 1, n):
                res = max(res, prices[j] - prices[i])
        return res
```

Python

Python

```python
def maxProfit(prices):
    # Initialize min_price as infinity and max_profit as 0
    min_price = float('inf')
    max_profit = 0
    # Iterate through each price in the list
    for price in prices:
        # Update the minimum price if current price is lower
        if price < min_price:
            min_price = price
        else:
            # Calculate current profit and update max_profit if it is higher
            max_profit = max(max_profit, price - min_price)
    return max_profit

# Example usage:
# result = maxProfit([7,1,5,3,6,4])
# print(result) # Output should be 5
```


Python

```python
class Solution:
   def maxProfit(self, prices: list[int]) -> int:
     sellOne = 0
     holdOne = -math.inf

     for price in prices:
       sellOne = max(sellOne, holdOne + price)
       holdOne = max(holdOne, -price)

     return sellOne
```

Python

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        ans, mi = 0, inf
        for v in prices:
            ans = max(ans, v - mi)
            mi = min(mi, v)
        return ans
```

Python

```python
'''
>>> import math
>>> a = math.inf
>>> a
inf

>>> x = float('-inf')
>>> y = float('inf')
>>> print(x < y) # Output: True
True
>>>
>>> z = -math.inf
>>> x==z
True

'''
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        ans, mi = 0, inf
        for v in prices:
            mi = min(mi, v)
            ans = max(ans, v - mi)
        return ans

###########
```



```python
class Solution(object):
  def maxProfit(self, prices):
    """
    :type prices: List[int]
    :rtype: int
    """
    if not prices:
      return 0
    ans = 0
    pre = prices[0]
    for i in range(1, len(prices)):
      pre = min(pre, prices[i])
      ans = max(prices[i] - pre, ans)
    return ans
```

[*] Dynamic Programming — Pattern Solution (stored optimal)

Python

```python
def maxProfit(prices):
    # Initialize min_price as infinity and max_profit as 0
    min_price = float('inf')
    max_profit = 0
    # Iterate through each price in the list
    for price in prices:
        # Update the minimum price if current price is lower
        if price < min_price:
            min_price = price
        else:
            # Calculate current profit and update max_profit if it is higher
            max_profit = max(max_profit, price - min_price)
    return max_profit

# Example usage:
# result = maxProfit([7,1,5,3,6,4])
# print(result) # Output should be 5
```

#5 MEDIUM

Longest Palindromic Substring

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Dynamic Programming

Lists: Grind 169

Problem:

Given a string s, return the longest palindromic substring in s.

Example 1:

```
Input: s = "babad"
Output: "bab"
Explanation: "aba" is also a valid answer.
```

Example 2:


```
Input: s = "cbbd"
Output: "bb"
```

Constraints:

- 1 <= s.length <= 1000
- s consist of only digits and English letters.

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def expand_from_center(self, s, left, right):
        # Brute Force O(n^3) - check every substring for palindrome property
        def is_palindrome(sub):
            return sub == sub[::-1]
        res = ''
        for i in range(len(s)):
            for j in range(i + 1, len(s) + 1):
                sub = s[i:j]
                if is_palindrome(sub) and len(sub) > len(res):
                    res = sub
        return res
```

Python

Python

```python
# Function to expand from the center and return the palindrome substring
def expand_from_center(s, left, right):
    # Expand while the characters on both sides are the same
    while left >= 0 and right < len(s) and s[left] == s[right]:
        left -= 1
        right += 1
    # Return the palindromic substring found after expansion
    return s[left + 1:right]

def longest_palindromic_substring(s):
    if not s:
        return ""
    longest = ""
    # Iterate over each index in string as potential center
    for i in range(len(s)):
        # Check for odd-length palindrome
        odd_palindrome = expand_from_center(s, i, i)
        if len(odd_palindrome) > len(longest):
            longest = odd_palindrome
        # Check for even-length palindrome
        even_palindrome = expand_from_center(s, i, i + 1)
        if len(even_palindrome) > len(longest):
            longest = even_palindrome
    return longest

# Example usage
print(longest_palindromic_substring("babad"))
```

Python


```python
class Solution:
    def longestPalindrome(self, s: str) -> str:
        if not s:
            return ''

        # (start, end) indices of the longest palindrome in s
        indices = [0, 0]

        def extend(s: str, i: int, j: int) -> tuple[int, int]:
            """
            Returns the (start, end) indices of the longest palindrome extended from
            the substring s[i..j].
            """
            while i >= 0 and j < len(s):
                if s[i] != s[j]:
                    break
                i -= 1
                j += 1
            return i + 1, j - 1

        for i in range(len(s)):
            l1, r1 = extend(s, i, i)
            if r1 - l1 > indices[1] - indices[0]:
                indices = l1, r1
            if i + 1 < len(s) and s[i] == s[i + 1]:
                l2, r2 = extend(s, i, i + 1)
                if r2 - l2 > indices[1] - indices[0]:
                    indices = l2, r2

        return s[indices[0]:indices[1] + 1]
```

Python

```python
class Solution:
    def longestPalindrome(self, s: str) -> str:
        t = '#'.join('@' + s + '$')
        p = self._manacher(t)
        maxPalindromeLength, bestCenter = max((extend, i)
                                              for i, extend in enumerate(p))
        l = (bestCenter - maxPalindromeLength) // 2
        r = (bestCenter + maxPalindromeLength) // 2
        return s[l:r]

    def _manacher(self, t: str) -> list[int]:
        """
        Returns an array `p` s.t. `p[i]` is the length of the longest palindrome
        centered at `t[i]`, where `t` is a string with delimiters and sentinels.
        """
        p = [0] * len(t)
        center = 0
        for i in range(1, len(t) - 1):
            rightBoundary = center + p[center]
            mirrorIndex = center - (i - center)
            if rightBoundary > i:
                p[i] = min(rightBoundary - i, p[mirrorIndex])
            # Try to expand the palindrome centered at i.
            while t[i + 1 + p[i]] == t[i - 1 - p[i]]:
                p[i] += 1
```

```
        # If a palindrome centered at i expands past `rightBoundary`, adjust
        # the center based on the expanded palindrome.
        if i + p[i] > rightBoundary:
            center = i
    return p
```

Python

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        n = len(s)
        f = [[True] * n for _ in range(n)]
        k, mx = 0, 1
        for i in range(n - 2, -1, -1):
            for j in range(i + 1, n):
                f[i][j] = False
                if s[i] == s[j]:
                    f[i][j] = f[i + 1][j - 1]
                    if f[i][j] and mx < j - i + 1:
                        k, mx = i, j - i + 1
        return s[k : k + mx]
```

Python

```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        def f(l, r):
            while l >= 0 and r < n and s[l] == s[r]:
                l, r = l - 1, r + 1
            return r - l - 1

        n = len(s)
        start, mx = 0, 1
        for i in range(n):
            a = f(i, i)
            b = f(i, i + 1)
            t = max(a, b)
            if mx < t:
                mx = t
                start = i - ((t - 1) >> 1)
        return s[start : start + mx]
```

Python


```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        mlen = 0
        start = end = 0
        n = len(s)
        dp = [ [False] * n for i in range(n) ]

        for j in range(n):
            for i in range(j + 1):
                dp[i][j] = (i == j) or (s[i] == s[j] and j - i == 1) or (s[i] == s[j] and dp[i + 1][j - 1])
                if dp[i][j] is True and j - i + 1 > mlen:
                    mlen = j - i + 1
                    start = i
                    end = j

        return s[start: end + 1]

######

class Solution:
    def longestPalindrome(self, s: str) -> str:
        n = len(s)
        f = [[True] * n for _ in range(n)]
        k, mx = 0, 1
        for i in range(n - 2, -1, -1):

            for j in range(i + 1, n):
                f[i][j] = False
                if s[i] == s[j]:
                    f[i][j] = f[i + 1][j - 1]
                    if f[i][j] and mx < j - i + 1:
                        k, mx = i, j - i + 1
        return s[k : k + mx]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python


```
class Solution:
    def longestPalindrome(self, s: str) -> str:
        mlen = 0
        start = end = 0
        n = len(s)
        dp = [ [False] * n for i in range(n) ]

        for j in range(n):
            for i in range(j + 1):
                dp[i][j] = (i == j) or (s[i] == s[j] and j - i == 1) or (s[i] == s[j] and dp[i + 1][j - 1])
                if dp[i][j] is True and j - i + 1 > mlen:
                    mlen = j - i + 1
                    start = i
                    end = j

        return s[start: end + 1]

######

class Solution:
    def longestPalindrome(self, s: str) -> str:
        n = len(s)
        f = [[True] * n for _ in range(n)]
        k, mx = 0, 1
        for i in range(n - 2, -1, -1):

            for j in range(i + 1, n):
                f[i][j] = False
                if s[i] == s[j]:
                    f[i][j] = f[i + 1][j - 1]
                    if f[i][j] and mx < j - i + 1:
                        k, mx = i, j - i + 1
        return s[k : k + mx]
```

#53 MEDIUM

Maximum Subarray

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming · Divide and Conquer

Lists: Grind 169

Problem:

Given an integer array nums, find the subarray with the largest sum, and return its sum.

Example 1:

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: The subarray [4,-1,2,1] has the largest sum 6.
```

Example 2:

```
Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.
```

Example 3:


```
Input: nums = [5,4,-1,7,8]
Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.
```

Constraints:

- 1 <= nums.length <= 105
- -104 <= nums[i] <= 104

Follow up: If you have figured out the O(n) solution, try coding another solution using the divide and conquer approach, which is more subtle.

>> Brute Force [Dynamic Programming] Interview Prep

Python

```
# Brute Force Approach
class Solution:
    def maxSubArray(self, nums):
        # Brute Force O(n^2) - compute sum of every contiguous subarray
        res = nums[0]
        for i in range(len(nums)):
            curr = 0
            for j in range(i, len(nums)):
                curr += nums[j]
                res = max(res, curr)
        return res
```

Python

Python

```
# Python solution using Kadane's Algorithm
def maxSubArray(nums):
    # Initializing currentSum and maxSum to the first element
    current_sum = max_sum = nums[0]
    # Traverse through the array starting from the second element
    for num in nums[1:]:
        # If current_sum + num is less than num, restart at num
        current_sum = max(num, current_sum + num)
        # Update max_sum if current_sum is greater
        max_sum = max(max_sum, current_sum)
    return max_sum

# Example usage
if __name__ == "__main__":
    sample = [-2,1,-3,4,-1,2,1,-5,4]
    print(maxSubArray(sample)) # Output: 6
```

Python

```
class Solution:
    def maxSubArray(self, nums: list[int]) -> int:
        # dp[i] := the maximum sum subarray ending in i
        dp = [0] * len(nums)

        dp[0] = nums[0]
        for i in range(1, len(nums)):
            dp[i] = max(nums[i], dp[i - 1] + nums[i])

        return max(dp)
```


Python

```
class Solution:
    def maxSubArray(self, nums: list[int]) -> int:
        ans = -math.inf
        sum = 0

        for num in nums:
            sum = max(num, sum + num)
            ans = max(ans, sum)

        return ans
```

Python

```
from dataclasses import dataclass


@dataclass(frozen=True)
class T:
    sum: int
    # the sum of the subarray starting from the first number
    maxSubarraySumLeft: int
    # the sum of the subarray ending in the last number
    maxSubarraySumRight: int
    maxSubarraySum: int


class Solution:
    def maxSubArray(self, nums: list[int]) -> int:
        def divideAndConquer(l: int, r: int) -> T:
            if l == r:
                return T(nums[l], nums[l], nums[l], nums[l])
            m = (l + r) // 2
            left = divideAndConquer(l, m)
            right = divideAndConquer(m + 1, r)
            maxSubarraySumLeft = max(left.maxSubarraySumLeft,
                                     left.sum + right.maxSubarraySumLeft)
            maxSubarraySumRight = max(
                left.maxSubarraySumRight + right.sum, right.maxSubarraySumRight)

            maxSubarraySum = max(left.maxSubarraySumRight + right.maxSubarraySumLeft,
                                 left.maxSubarraySum, right.maxSubarraySum)
            sum = left.sum + right.sum
            return T(sum, maxSubarraySumLeft, maxSubarraySumRight, maxSubarraySum)

        return divideAndConquer(0, len(nums) - 1).maxSubarraySum
```

Python

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        ans = f = nums[0]
        for x in nums[1:]:
            f = max(f, 0) + x
            ans = max(ans, f)
        return ans
```

Python


```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        def crossMaxSub(nums, left, mid, right):
            lsum = rsum = 0
            lmx = rmx = -inf
            for i in range(mid, left - 1, -1):
                lsum += nums[i]
                lmx = max(lmx, lsum)
            for i in range(mid + 1, right + 1):
                rsum += nums[i]
                rmx = max(rmx, rsum)
            return lmx + rmx

        def maxSub(nums, left, right):
            if left == right:
                return nums[left]
            mid = (left + right) >> 1
            lsum = maxSub(nums, left, mid)
            rsum = maxSub(nums, mid + 1, right)
            csum = crossMaxSub(nums, left, mid, right)
            return max(lsum, rsum, csum)

        left, right = 0, len(nums) - 1
        return maxSub(nums, left, right)
```

Python

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        res = cur_sum = nums[0]
        for num in nums[1:]:
            cur_sum = num + max(cur_sum, 0)
            res = max(res, cur_sum)
        return res
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```
class Solution:
    def maxSubArray(self, nums: list[int]) -> int:
        # dp[i] := the maximum sum subarray ending in i
        dp = [0] * len(nums)

        dp[0] = nums[0]
        for i in range(1, len(nums)):
            dp[i] = max(nums[i], dp[i - 1] + nums[i])

        return max(dp)
```

#55 MEDIUM

Jump Game

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming · Greedy

Lists: Grind 169

Problem:
You are given an integer array nums. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position.

Return true if you can reach the last index, or false otherwise.

Example 1:

```
Input: nums = [2,3,1,1,4]
Output: true
Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.
```

Example 2:

```
Input: nums = [3,2,1,0,4]
Output: false
Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0,
which makes it impossible to reach the last index.
```

Constraints:

- 1 <= nums.length <= 104
- 0 <= nums[i] <= 105

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def canJump(self, nums):
        # Brute Force O(2^n) - recursion: try every reachable position
        def can_reach(i):
            if i >= len(nums) - 1: return True
            for step in range(1, nums[i] + 1):
                if can_reach(i + step): return True
            return False
        return can_reach(0)
```

Python

Python


```python
# Function to determine if you can reach the last index of the array
def canJump(nums):
    # Initialize maxReach which is the max index reachable at start (index 0)
    maxReach = 0
    # Iterate through each element and its index
    for i, jump in enumerate(nums):
        # If current index is greater than maximum reachable index, return False
        if i > maxReach:
            return False
        # Update maxReach with the maximum of current maxReach and i + jump length at index i
        maxReach = max(maxReach, i + jump)
        # If current maxReach is already at or beyond the last index, return True
        if maxReach >= len(nums) - 1:
            return True
    # Return True if the end is reachable after processing all indices
    return True

# Example usage:
# print(canJump([2,3,1,1,4])) # Expected output: True
# print(canJump([3,2,1,0,4])) # Expected output: False
```

Python

```python
class Solution:
  def canJump(self, nums: list[int]) -> bool:
    i = 0
    reach = 0

    while i < len(nums) and i <= reach:
        reach = max(reach, i + nums[i])
      i += 1

    return i == len(nums)
```

Python

```python
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        mx = 0
        for i, x in enumerate(nums):
            if mx < i:
                return False
            mx = max(mx, i + x)
        return True
```

Python

```python
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        mx = 0
        for i, x in enumerate(nums):
            if mx < i:
                return False
            mx = max(mx, i + x)
        return True
```

[*] Dynamic Programming — Pattern Solution (stored optimal)

Python


```python
# Function to determine if you can reach the last index of the array
def canJump(nums):
    # Initialize maxReach which is the max index reachable at start (index 0)
    maxReach = 0
    # Iterate through each element and its index
    for i, jump in enumerate(nums):
        # If current index is greater than maximum reachable index, return False
        if i > maxReach:
            return False
        # Update maxReach with the maximum of current maxReach and i + jump length at index i
        maxReach = max(maxReach, i + jump)
        # If current maxReach is already at or beyond the last index, return True
        if maxReach >= len(nums) - 1:
            return True
    # Return True if the end is reachable after processing all indices
    return True

# Example usage:
# print(canJump([2,3,1,1,4])) # Expected output: True
# print(canJump([3,2,1,0,4])) # Expected output: False
```

#62 MEDIUM

Unique Paths

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math · Dynamic Programming · Combinatorics

Lists: Grind 169

Problem:
There is a robot on an m x n grid. The robot is initially located at the top-left corner (i.e., grid[0][0]). The robot tries to move to the bottom-right corner (i.e., grid[m - 1][n - 1]). The robot can only move either down or right at any point in time.

Given the two integers m and n, return the number of possible unique paths that the robot can take to reach the bottom-right corner.

The test cases are generated so that the answer will be less than or equal to 2 * 109.

Example 1:

```
Input: m = 3, n = 7
Output: 28
```

Example 2:

```
Input: m = 3, n = 2
Output: 3
Explanation: From the top-left corner, there are a total of 3 ways to reach the bottom-right
corner:
1. Right -> Down -> Down
2. Down -> Down -> Right
3. Down -> Right -> Down
```

Constraints:

- 1 <= m, n <= 100

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def uniquePaths(self, m, n):
        # Brute Force O(2^(m+n)) - recursion without memo; each step right or down
        def paths(r, c):
            if r == m - 1 and c == n - 1: return 1
            if r >= m or c >= n: return 0
            return paths(r + 1, c) + paths(r, c + 1)
        return paths(0, 0)
```

Python

Python

```python
# Python solution using Dynamic Programming

def uniquePaths(m, n):
    # Initialize a 2D list with 1's for the first row and first column
    dp = [[1] + n for _ in range(m)]

    # Fill in the DP table
    for i in range(1, m):
        for j in range(1, n):
            # The number of ways to reach this cell is the sum of the ways from the cell above and the cell
on the left
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    # Return the value in the bottom-right cell
    return dp[m-1][n-1]

# Example usage:
print(uniquePaths(3, 7))
```

Python


```python
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        # dp[i][j] := the number of unique paths from (0, 0) to (i, j)
        dp = [[1] + n for _ in range(m)]

        for i in range(1, m):
            for j in range(1, n):
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1]

        return dp[-1][-1]
```

Python

```python
class Solution:
  def uniquePaths(self, m: int, n: int) -> int:
    dp = [1] * n

    for _ in range(1, m):
      for j in range(1, n):
        dp[j] += dp[j - 1]

    return dp[n - 1]
```

Python

```python
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        f = [[0] * n for _ in range(m)]
        f[0][0] = 1
        for i in range(m):
            for j in range(n):
                if i:
                    f[i][j] += f[i - 1][j]
                if j:
                    f[i][j] += f[i][j - 1]
        return f[-1][-1]
```

Python

```python
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        f = [1] * n
        for _ in range(1, m):
            for j in range(1, n):
                f[j] += f[j - 1]
        return f[-1]
```

Python

```python
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        # avoid setting dp[i][j] to 1 for i==0 or j==0 as initialization
        dp = [[1] * n for _ in range(m)]
        for i in range(1, m):
            for j in range(1, n):
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1]
        return dp[-1][-1]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```python
# Python solution using Dynamic Programming

def uniquePaths(m, n):
    # Initialize a 2D list with 1's for the first row and first column
    dp = [[1] * n for _ in range(m)]

    # Fill in the DP table
    for i in range(1, m):
        for j in range(1, n):
            # The number of ways to reach this cell is the sum of the ways from the cell above and the cell
on the left
            dp[i][j] = dp[i-1][j] + dp[i][j-1]
    # Return the value in the bottom-right cell
    return dp[m-1][n-1]

# Example usage:
print(uniquePaths(3, 7))
```

#72 MEDIUM

Edit Distance

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Dynamic Programming

Lists: Denny Zhang

Problem:
Given two strings word1 and word2, return the minimum number of operations required to convert word1 to word2.

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Example 1:

```
Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')
```

Example 2:

```
Input: word1 = "intention", word2 = "execution"
Output: 5
Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
exection -> execution (insert 'u')
```

Constraints:

- 0 <= word1.length, word2.length <= 500
- word1 and word2 consist of lowercase English letters.

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def minDistance(self, word1, word2):
        # Brute Force O(3^(m+n)) - recursion without memo; try insert/delete/replace
        def dp(i, j):
            if i == len(word1): return len(word2) - j
            if j == len(word2): return len(word1) - i
            if word1[i] == word2[j]:
                return dp(i + 1, j + 1)
            return 1 + min(dp(i + 1, j), # delete
                           dp(i, j + 1), # insert
                           dp(i + 1, j + 1)) # replace
        return dp(0, 0)
```

Python

Python

```python
# Python solution to compute the Edit Distance
def minDistance(word1, word2):
    m = len(word1)
    n = len(word2)

    # Initialize the DP table with size (m+1) x (n+1)
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Base cases: transform word1[0:i] to empty word2 requires i deletions
    for i in range(m + 1):
        dp[i][0] = i
    # Base cases: transform empty word1 to word2[0:j] requires j insertions
    for j in range(n + 1):
        dp[0][j] = j

    # Fill up the dp table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            # If the characters are the same, no new operation is needed.
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                # consider insertion, deletion, and replacement
                dp[i][j] = 1 + min(dp[i][j - 1], # Insert
                                   dp[i - 1][j], # Delete
                                   dp[i - 1][j - 1]) # Replace
    return dp[m][n]

# Example usage:
print(minDistance("horse", "ros"))
```

Python


```python
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m = len(word1)
        n = len(word2)
        # dp[i][j] := the minimum number of operations to convert word1[0..i) to
        # word2[0..j)
        dp = [[0] + (n + 1) for _ in range(m + 1)]

        for i in range(1, m + 1):
          dp[i][0] = i

        for j in range(1, n + 1):
          dp[0][j] = j

        for i in range(1, m + 1):
          for j in range(1, n + 1):
            if word1[i - 1] == word2[j - 1]:
              dp[i][j] = dp[i - 1][j - 1]
            else:
              dp[i][j] = min(dp[i - 1][j - 1], dp[i - 1][j], dp[i][j - 1]) + 1

        return dp[m][n]
```

Python

```python
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m, n = len(word1), len(word2)
        f = [[0] + (n + 1) for _ in range(m + 1)]
        for j in range(n + 1):
            f[0][j] = j
        for i, a in enumerate(word1, 1):
            f[i][0] = i
            for j, b in enumerate(word2, 1):
                if a == b:
                    f[i][j] = f[i - 1][j - 1]
                else:
                    f[i][j] = min(f[i - 1][j], f[i][j - 1], f[i - 1][j - 1]) + 1
        return f[m][n]
```

Python


```python
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m, n = len(word1), len(word2)
        f = [[0] + (n + 1) for _ in range(m + 1)]
        for j in range(n + 1):
            f[0][j] = j
        for i, a in enumerate(word1, 1):
            f[i][0] = i
            for j, b in enumerate(word2, 1):
                if a == b:
                    f[i][j] = f[i - 1][j - 1]
                else:
                    f[i][j] = min(f[i - 1][j], f[i][j - 1], f[i - 1][j - 1]) + 1
        return f[m][n]

##########

class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        m, n = len(word1), len(word2)
        f = [[0] + (n + 1) for _ in range(m + 1)]

        for j in range(1, n + 1):
            f[0][j] = j
        for i, a in enumerate(word1, 1):
            f[i][0] = i # merged for loops from above solution, but not good for readability
            for j, b in enumerate(word2, 1):
                if a == b:
                    f[i][j] = f[i - 1][j - 1]
                else:
                    f[i][j] = min(f[i - 1][j], f[i][j - 1], f[i - 1][j - 1]) + 1
        return f[m][n]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python


```python
# Python solution to compute the Edit Distance
def minDistance(word1, word2):
    m = len(word1)
    n = len(word2)

    # Initialize the DP table with size (m+1) x (n+1)
    dp = [[0] + (n + 1) for _ in range(m + 1)]

    # Base cases: transform word1[0:i] to empty word2 requires i deletions
    for i in range(m + 1):
        dp[i][0] = i
    # Base cases: transform empty word1 to word2[0:j] requires j insertions
    for j in range(n + 1):
        dp[0][j] = j

    # Fill up the dp table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            # If the characters are the same, no new operation is needed.
            if word1[i - 1] == word2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1]
            else:
                # consider insertion, deletion, and replacement
                dp[i][j] = 1 + min(dp[i][j - 1], # Insert
                                   dp[i - 1][j], # Delete
                                   dp[i - 1][j - 1]) # Replace
    return dp[m][n]

# Example usage:
print(minDistance("horse", "ros"))
```

#91 MEDIUM

Decode Ways

LeetCodeCx · SimplyLeet · WalkCC · LeetCode · Editorial

String · Dynamic Programming

Lists: Grind 169

Problem:

You have intercepted a secret message encoded as a string of numbers. The message is decoded via the following mapping:

"1" -> 'A' "2" -> 'B' ... "25" -> 'Y' "26" -> 'Z'

However, while decoding the message, you realize that there are many different ways you can decode the message because some codes are contained in other codes ("2" and "5" vs "25").

For example, "11106" can be decoded into:

- "AAJF" with the grouping (1, 1, 10, 6)
- "KJF" with the grouping (11, 10, 6)
- The grouping (1, 11, 06) is invalid because "06" is not a valid code (only "6" is valid).

Note: there may be strings that are impossible to decode. Given a string s containing only digits, return the number of ways to decode it. If the entire string cannot be decoded in any valid way, return 0.

The test cases are generated so that the answer fits in a 32-bit integer.

Example 1:

Input: s = "12"

Output: 2

Explanation:

"12" could be decoded as "AB" (1 2) or "L" (12).

Example 2:

Input: s = "226"

Output: 3

Explanation:

"226" could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

Example 3:

Input: s = "06"

Output: 0

Explanation:

"06" cannot be mapped to "F" because of the leading zero ("6" is different from "06"). In this case, the string is not a valid encoding, so return 0.

Constraints:

- 1 <= s.length <= 100
- s contains only digits and may contain leading zero(s).

Python

Python


```python
# Python solution using dynamic programming
def num_decodings(s: str) -> int:
    # If the string starts with '0', it's not decodable
    if not s or s[0] == '0':
        return 0

    # dp array where dp[i] represents the number of ways to decode s[:i]
    n = len(s)
    dp = [0] * (n + 1)

    # Base case: an empty string has one decoding
    dp[0] = 1

    for i in range(1, n + 1):
        # Check for single digit decoding (s[i-1] is valid if not '0')
        if s[i - 1] != '0':
            dp[i] += dp[i - 1]
        # Check for two-digit decoding (if i>=2, s[i-2:i] forms a valid number between 10 and 26)
        if i >= 2 and s[i - 2] != '0':
            two_digit = int(s[i - 2:i])
            if 10 <= two_digit <= 26:
                dp[i] += dp[i - 2]
    return dp[n]

# Example usage:

print(num_decodings("12")) # Expected output: 2
```

Python

```python
class Solution:
  def numDecodings(self, s: str) -> int:
    n = len(s)
    # dp[i] := the number of ways to decode s[i..n)
    dp = [0] * n + [1]

    def isValid(a: str, b=None) -> bool:
      if b:
        return a == '1' or a == '2' and b < '7'
      return a != '0'

    if isValid(s[-1]):
      dp[n - 1] = 1

    for i in reversed(range(n - 1)):
      if isValid(s[i]):
        dp[i] += dp[i + 1]
      if isValid(s[i], s[i + 1]):
        dp[i] += dp[i + 2]

    return dp[0]
```

Python

```python
class Solution:
    def numDecodings(self, s: str) -> int:
        n = len(s)
        f = [1] + [0] * n
        for i, c in enumerate(s, 1):
            if c != "0":
                f[i] = f[i - 1]
            if i > 1 and s[i - 2] != "0" and int(s[i - 2 : i]) <= 26:
                f[i] += f[i - 2]
        return f[n]
```

Python

```python
class Solution:
    def numDecodings(self, s: str) -> int:
        f, g = 0, 1
        for i, c in enumerate(s, 1):
            h = g if c != "0" else 0
            if i > 1 and s[i - 2] != "0" and int(s[i - 2 : i]) <= 26:
                h += f
            f, g = g, h
        return g
```

Python

```python
'''
>>> s = "abcdef"
>>> [(i,c) for i, c in enumerate(s, 1)]
[(1, 'a'), (2, 'b'), (3, 'c'), (4, 'd'), (5, 'e'), (6, 'f')]
'''

class Solution:
    def numDecodings(self, s: str) -> int:
        # f: i-2
        # g: i-1
        f, g = 0, 1
        for i, c in enumerate(s, 1):
            h = g if c != "0" else 0
            if i > 1 and s[i - 2] != "0" and int(s[i - 2 : i]) <= 26:
                h += f
            f, g = g, h
        return g

#############

class Solution:
    def numDecodings(self, s: str) -> int:
        if len(s) == 0:
            return 0

        length = len(s)
```



```python
        dp = [0] * (length + 1) # dp[i] => at index i, its decode ways

        dp[length] = 1 # initiator, just make forloop flow working
        dp[length - 1] = 0 if s[length - 1] == '0' else 1 # assumption is starting at i, so starting as 0
is not decodable

        # start at 'len-2'
        # or else below 'dp[i+2]' will out of index boundary
        for i in range(length - 2, -1, -1):
            if s[i] == '0':
                continue
            tem = int(s[i:i + 2])

            if tem > 26:
                dp[i] = dp[i + 1]
            else:
                dp[i] = dp[i + 1] + dp[i + 2]
        return dp[0]

# optimized from above solution, no dp array
class Solution:
    def numDecodings(self, s: str) -> int:
        n = len(s)
        # a: dp[i-2], b: dp[i-1], c: count for current index i
        a, b, c = 0, 1, 0
        for i in range(1, n + 1):
```

```python
            c = 0
            if s[i - 1] != '0':
                c += b
            if i > 1 and s[i - 2] != '0' and (int(s[i - 2]) * 10 + int(s[i - 1]) <= 26):
                c += a
            a, b = b, c
        return c

#############

class Solution:
    def numDecodings(self, s: str) -> int:
        n = len(s)
        dp = [0] * (n + 1)
        dp[0] = 1
        for i in range(1, n + 1):
            if s[i - 1] != '0':
                dp[i] += dp[i - 1]
            if i > 1 and s[i - 2] != '0' and (int(s[i - 2]) * 10 + int(s[i - 1]) <= 26):
                dp[i] += dp[i - 2]
        return dp[n]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python


```python
# Python solution using dynamic programming
def num_decodings(s: str) -> int:
    # If the string starts with '0', it's not decodable
    if not s or s[0] == '0':
        return 0

    # dp array where dp[i] represents the number of ways to decode s[:i]
    n = len(s)
    dp = [0] * (n + 1)

    # Base case: an empty string has one decoding
    dp[0] = 1

    for i in range(1, n + 1):
        # Check for single digit decoding (s[i-1] is valid if not '0')
        if s[i - 1] != '0':
            dp[i] += dp[i - 1]
        # Check for two-digit decoding (if i>=2, s[i-2:i] forms a valid number between 10 and 26)
        if i >= 2 and s[i - 2] != '0':
            two_digit = int(s[i - 2:i])
            if 10 <= two_digit <= 26:
                dp[i] += dp[i - 2]
    return dp[n]

# Example usage:

print(num_decodings("12")) # Expected output: 2
```

#152 MEDIUM

Maximum Product Subarray

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming

Lists: Grind 169

Problem:

Given an integer array nums, find a subarray that has the largest product, and return the product.

The test cases are generated so that the answer will fit in a 32-bit integer.

Note that the product of an array with a single element is the value of that element.

Example 1:

```
Input: nums = [2,3,-2,4]
Output: 6
Explanation: [2,3] has the largest product 6.
```

Example 2:

```
Input: nums = [-2,0,-1]
Output: 0
Explanation: The result cannot be 2, because [-2,-1] is not a subarray.
```

Constraints:

- 1 <= nums.length <= 2 * 104
- -10 <= nums[i] <= 10

- The product of any subarray of nums is guaranteed to fit in a 32-bit integer.

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def maxProduct(self, nums):
        # Brute-force O(n^2) - compute product of every contiguous subarray
        res = nums[0]
        for i in range(len(nums)):
            prod = 1
            for j in range(i, len(nums)):
                prod *= nums[j]
                res = max(res, prod)
        return res
```

Python

Python

```python
def maxProduct(nums):
    # If the input array is empty, return 0 (though problem guarantees at least one element)
    if not nums:
        return 0

    # Initialize maximum, minimum product ending here and the result.
    max_ending_here = nums[0]
    min_ending_here = nums[0]
    global_max = nums[0]

    # Iterate over the array starting from the second element.
    for num in nums[1:]:
        # If the current number is negative, swap max and min.
        if num < 0:
            max_ending_here, min_ending_here = min_ending_here, max_ending_here
        # Update the max product at the current index.
        max_ending_here = max(num, max_ending_here * num)
        # Update the min product at the current index.
        min_ending_here = min(num, min_ending_here * num)
        # Update the global maximum product.
        global_max = max(global_max, max_ending_here)

    return global_max

# Example usage:
# print(maxProduct([2,3,-2,4])) # Output: 6
```

Python


```python
class Solution:
    def maxProduct(self, nums: list[int]) -> int:
        ans = nums[0]
        dpMin = nums[0] # the minimum so far
        dpMax = nums[0] # the maximum so far

        for i in range(1, len(nums)):
          num = nums[i]
          prevMin = dpMin # dpMin[i - 1]
          prevMax = dpMax # dpMax[i - 1]
          if num < 0:
            dpMin = min(prevMax * num, num)
            dpMax = max(prevMin * num, num)
          else:
            dpMin = min(prevMin * num, num)
            dpMax = max(prevMax * num, num)

          ans = max(ans, dpMax)

        return ans
```

Python

```python
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        ans = f = g = nums[0]
        for x in nums[1:]:
            ff, gg = f, g
            f = max(x, ff * x, gg * x)
            g = min(x, ff * x, gg * x)
            ans = max(ans, f)
        return ans
```

Python

```python
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        ans = f = g = nums[0]
        for x in nums[1:]:
            ff, gg = f, g
            f = max(x, ff * x, gg * x)
            g = min(x, ff * x, gg * x)
            ans = max(ans, f)
        return ans
```

[*] Dynamic Programming — Pattern Solution (stored optimal)

Python


```python
def maxProduct(nums):
    # If the input array is empty, return 0 (though problem guarantees at least one element)
    if not nums:
        return 0
    # Initialize maximum, minimum product ending here and the result.
    max_ending_here = nums[0]
    min_ending_here = nums[0]
    global_max = nums[0]

    # Iterate over the array starting from the second element.
    for num in nums[1:]:
        # If the current number is negative, swap max and min.
        if num < 0:
            max_ending_here, min_ending_here = min_ending_here, max_ending_here
        # Update the max product at the current index.
        max_ending_here = max(num, max_ending_here * num)
        # Update the min product at the current index.
        min_ending_here = min(num, min_ending_here * num)
        # Update the global maximum product.
        global_max = max(global_max, max_ending_here)

    return global_max

# Example usage:
# print(maxProduct([2,3,-2,4])) # Output: 6
```

#198 MEDIUM

House Robber

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming

Lists: Grind 169

Problem:

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and it will automatically contact the police if two adjacent houses were broken into on the same night.

Given an integer array nums representing the amount of money of each house, return the maximum amount of money you can rob tonight without alerting the police.

Example 1:

```
Input: nums = [1,2,3,1]
Output: 4
Explanation: Rob house 1 (money = 1) and then rob house 3 (money = 3).
Total amount you can rob = 1 + 3 = 4.
```

Example 2:

```
Input: nums = [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9) and rob house 5 (money = 1).
Total amount you can rob = 2 + 9 + 1 = 12.
```

Constraints:

- 1 <= nums.length <= 100
- 0 <= nums[i] <= 400

Python

Python

```python
# Definition of the Solution class
class Solution:
    def rob(self, nums: List[int]) -> int:
        # Edge case: if there are no houses, return 0
        if not nums:
            return 0

        # Initialize two variables to keep track of the max amounts
        prev1, prev2 = 0, 0

        # Iterate over each house's money
        for money in nums:
            # Calculate the maximum money that can be robbed if we consider the current house
            current = max(prev1, prev2 + money) # Either skip current house or rob it
            # Update variables: prev2 becomes previous best, and prev1 becomes current best
            prev2 = prev1
            prev1 = current

        # Return the maximum money that can be robbed
        return prev1
```

Python

```python
class Solution:
    def rob(self, nums: list[int]) -> int:
        if not nums:
            return 0
        if len(nums) == 1:
            return nums[0]

        # dp[i]:= max money of robbing nums[0..i]
        dp = [0] * len(nums)
        dp[0] = nums[0]
        dp[1] = max(nums[0], nums[1])

        for i in range(2, len(nums)):
            dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])

        return dp[-1]
```

Python


```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        prev1 = 0 # dp[i - 1]
        prev2 = 0 # dp[i - 2]

        for num in nums:
            dp = max(prev1, prev2 + num)
            prev2 = prev1
            prev1 = dp

        return prev1
```

Python

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        @cache
        def dfs(i: int) -> int:
            if i >= len(nums):
                return 0
            return max(nums[i] + dfs(i + 2), dfs(i + 1))

        return dfs(0)
```

Python

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        n = len(nums)
        f = [0] * (n + 1)
        f[1] = nums[0]
        for i in range(2, n + 1):
            f[i] = max(f[i - 1], f[i - 2] + nums[i - 1])
        return f[n]
```

Python

```python
class Solution:
    def rob(self, nums: List[int]) -> int:
        f = g = 0
        for x in nums:
            f, g = max(f, g), f + x
        return max(f, g)
```

Python


```python
# greedy
class Solution:
    def rob(self, nums: List[int]) -> int:
        not_rob, rob = 0, nums[0]
        for num in nums[1:]:
            # must max check
            # eg. first robbed 99999, then following ones are just 3,3,3,3,3
            not_rob, rob = rob, max(num + not_rob, rob)
        return rob

###########

class Solution(object):
    def rob(self, nums):
        """
        :type nums: List[int]
        :rtype: int
        """
        if len(nums) == 0:
            return 0
        if len(nums) <= 2:
            return max(nums)
        dp = [0 for i in range(0, 2)]
        dp[0] = nums[0]
        dp[1] = max(nums[1], nums[0])

        for i in range(2, len(nums)):
            dp[i % 2] = max(dp[(i - 1) % 2], dp[(i - 2) % 2] + nums[i])
        return dp[(len(nums) - 1) % 2]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def rob(self, nums: list[int]) -> int:
        if not nums:
            return 0
        if len(nums) == 1:
            return nums[0]

        # dp[i]:= max money of robbing nums[0..i]
        dp = [0] * len(nums)
        dp[0] = nums[0]
        dp[1] = max(nums[0], nums[1])

        for i in range(2, len(nums)):
            dp[i] = max(dp[i - 1], dp[i - 2] + nums[i])

        return dp[-1]
```

#256 MEDIUM

Paint House

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming

Lists: Premium 98

Problem:

There is a row of n houses, where each house can be painted one of three colors: red, blue, or green. The cost of painting each house with a certain color is different. You have to paint all the houses such that no two adjacent houses have the same color.

The cost of painting each house with a certain color is represented by an n x 3 cost matrix costs.

- For example, costs[0][0] is the cost of painting house 0 with the color red; costs[1][2] is the cost of painting house 1 with color green, and so on...

Return the minimum cost to paint all houses.

Example 1:

```
Input: costs = [[17,2,17],[16,16,5],[14,3,19]]
Output: 10
Explanation: Paint house 0 into blue, paint house 1 into green, paint house 2 into blue.
Minimum cost: 2 + 5 + 3 = 10.
```

Example 2:

```
Input: costs = [[7,6,2]]
Output: 2
```

Constraints:

- costs.length == n
- costs[i].length == 3
- 1 <= n <= 100
- 1 <= costs[i][j] <= 20

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def minCost(self, costs):
        # Brute Force 指数级 - plain recursion, no memoisation
        def recurse(i):
            if i == len(costs): return 0
            take = costs[i] + recurse(i + 1)
            skip = recurse(i + 1)
            return max(take, skip)
        return recurse(0)
```

Python

Python


```python
# Python solution for the Paint House problem

def minCost(costs):
    # Get the number of houses
    n = len(costs)
    # Base case: If there are no houses, return 0 cost
    if n == 0:
        return 0

    # Iterate from the second house to the last house
    for i in range(1, n):
        # Update the cost for painting the current house red
        costs[i][0] += min(costs[i-1][1], costs[i-1][2])
        # Update the cost for painting the current house blue
        costs[i][1] += min(costs[i-1][0], costs[i-1][2])
        # Update the cost for painting the current house green
        costs[i][2] += min(costs[i-1][0], costs[i-1][1])

    # The answer is the minimum cost of painting the last house with any of the colors
    return min(costs[-1])

# Example usage:
example_costs = [[17,2,17],[16,16,5],[14,3,19]]
print(minCost(example_costs)) # Output: 10
```

Python

```python
class Solution:
    def minCost(self, costs: list[list[int]]) -> list[list[int]]:
    for i in range(1, len(costs)):
      costs[i][0] += min(costs[i - 1][1], costs[i - 1][2])
      costs[i][1] += min(costs[i - 1][0], costs[i - 1][2])
      costs[i][2] += min(costs[i - 1][0], costs[i - 1][1])

    return min(costs[-1])
```

Python

```python
class Solution:
    def minCost(self, costs: List[List[int]]) -> int:
        a = b = c = 0
        for ca, cb, cc in costs:
            a, b, c = min(b, c) + ca, min(a, c) + cb, min(a, b) + cc
        return min(a, b, c)
```

Python


```python
class Solution:
    def minCost(self, costs: List[List[int]]) -> int:
        a = b = c = 0
        for ca, cb, cc in costs:
            a, b, c = min(b, c) + ca, min(a, c) + cb, min(a, b) + cc
        return min(a, b, c)

############
'''
Two-dimensional dynamic array dp,
where dp[i][j] represents the minimum cost of brushing to the i+1th house with color j,

and the state transition equation is:
dp[i][j] = cost[i][j] + min(dp[i-1][(j + 1)% 3], dp[i-1][(j + 2)% 3])
'''
class Solution(object):
    def minCost(self, costs):
        """
        :type costs: List[List[int]]
        :rtype: int
        """
        if not costs:
            return 0

        dp = [0] * (len(costs[0]))
        dp[:] = costs[0]
        for i in range(1, len(costs)):
            d0 = d1 = d2 = 0
            for j in range(0, 3):
                if j == 0:
                    d0 = min(dp[1], dp[2]) + costs[i][0]
                elif j == 1:
                    d1 = min(dp[0], dp[2]) + costs[i][1]
                else:
                    d2 = min(dp[0], dp[1]) + costs[i][2]
            dp[0], dp[1], dp[2] = d0, d1, d2
        return min(dp)
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def minCost(self, costs: List[List[int]]) -> int:
        a = b = c = 0
        for ca, cb, cc in costs:
            a, b, c = min(b, c) + ca, min(a, c) + cb, min(a, b) + cc
        return min(a, b, c)

#############

'''
Two-dimensional dynamic array dp,
where dp[i][j] represents the minimum cost of brushing to the i-th house with color j,

and the state transition equation is:
dp[i][j] = cost[i][j] + min(dp[i-1][(j + 1)% 3], dp[i-1][(j + 2)% 3])

'''
class Solution(object):
  def minCost(self, costs):
    """
    :type costs: List[List[int]]
    :rtype: int
    """
    if not costs:
      return 0

    dp = [0] * len(costs[0])
    dp[:] = costs[0]
    for i in range(1, len(costs)):
      d0 = d1 = d2 = 0
      for j in range(0, 3):
        if j == 0:
          d0 = min(dp[1], dp[2]) + costs[i][0]
        elif j == 1:
          d1 = min(dp[0], dp[2]) + costs[i][1]
        else:
          d2 = min(dp[0], dp[1]) + costs[i][2]
      dp[0], dp[1], dp[2] = d0, d1, d2
    return min(dp)
```

#309 MEDIUM

Best Time to Buy and Sell Stock with Cooldown

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming

Lists: Denny Zhang

Problem:

You are given an array prices where prices[i] is the price of a given stock on the ith day.

Find the maximum profit you can achieve. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times) with the following restrictions:

- After you sell your stock, you cannot buy stock on the next day (i.e., cooldown one day).

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: prices = [1,2,3,0,2]
Output: 3
Explanation: transactions = [buy, sell, cooldown, buy, sell]
```

Example 2:

```
Input: prices = [1]
Output: 0
```

Constraints:

- 1 <= prices.length <= 5000
- 0 <= prices[i] <= 1000

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def maxProfit(self, prices):
        # Brute Force O(3^n) - recursion: at each day choose buy/sell/rest
        def dfs(i, holding, cooldown):
            if i >= len(prices): return 0
            if cooldown: return dfs(i+1, holding, False) # must rest
            if holding:
                sell = prices[i] + dfs(i+1, False, True) # sell today
                rest = dfs(i+1, True, False) # hold
                return max(sell, rest)
            else:
                buy = -prices[i] + dfs(i+1, True, False) # buy today
                rest = dfs(i+1, False, False) # wait
                return max(buy, rest)
        return dfs(0, False, False)
```

Python

Python


```python
# Python Solution with line-by-line comments

def maxProfit(prices):
    # Handle edge case: if there are no prices, profit is 0
    if not prices:
        return 0

    # Initialize states:
    # s0: Maximum profit when ready to buy
    # s1: Maximum profit when holding a stock (bought)
    # s2: Maximum profit during cooldown after a sale
    s0, s1, s2 = 0, -prices[0], 0

    # Iterate over prices starting from day 1
    for price in prices[1:]:
        # Store previous state values
        prev_s0, prev_s1, prev_s2 = s0, s1, s2
        # s0 can come either from staying in s0 or coming from a cooldown (s2)
        s0 = max(prev_s0, prev_s2)
        # s1 is updated by either keeping the stock or buying today with profit from s0
        s1 = max(prev_s1, prev_s0 - price)
        # s2 is achieved by selling the stock that we were holding (prev_s1 + price)
        s2 = prev_s1 + price
    # The final maximum profit is the maximum of not holding a stock (s0 or s2)
    return max(s0, s2)


# Example usage:
print(maxProfit([1,2,3,0,2]))
```

Python

```python
class Solution:
    def maxProfit(self, prices: list[int]) -> int:
        sell = 0
        hold = -math.inf
        prev = 0

        for price in prices:
          cache = sell
          sell = max(sell, hold + price)
          hold = max(hold, prev - price)
          prev = cache

        return sell
```

Python


```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        @cache
        def dfs(i: int, j: int) -> int:
            if i >= len(prices):
                return 0
            ans = dfs(i + 1, j)
            if j:
                ans = max(ans, prices[i] + dfs(i + 2, 0))
            else:
                ans = max(ans, -prices[i] + dfs(i + 1, 1))
            return ans

        return dfs(0, 0)
```

Python

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        n = len(prices)
        f = [[0] * 2 for _ in range(n)]
        f[0][1] = -prices[0]
        for i in range(1, n):
            f[i][0] = max(f[i - 1][0], f[i - 1][1] + prices[i])
            f[i][1] = max(f[i - 1][1], f[i - 2][0] - prices[i])
        return f[n - 1][0]
```

Python

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        f, f0, f1 = 0, 0, -prices[0]
        for x in prices[1:]:
            f, f0, f1 = f0, max(f0, f1 + x), max(f1, f - x)
        return f0
```

Python


```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        f1, f2, f3 = -prices[0], 0, 0
        for price in prices[1:]:
            pf1, pf2, pf3 = f1, f2, f3
            f1 = max(pf1, pf3 - price)
            f2 = max(pf2, pf1 + price)
            f3 = max(pf3, pf2) # cooldown
        return f2

###########

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        f, f0, f1 = 0, 0, -prices[0]
        for x in prices[1:]:
            f, f0, f1 = f0, max(f0, f1 + x), max(f1, f - x)
        return f0

###########

class Solution(object):
    def maxProfit(self, prices):
        """
        :type prices: List[int]

        :rtype: int
        """
        if len(prices) < 2:
            return 0
        buy = [0] * len(prices)
        sell = [0] * len(prices)
        buy[0] = -prices[0]
        buy[1] = max(-prices[1], buy[0])
        sell[0] = 0
        sell[1] = max(prices[1] - prices[0], 0)
        for i in range(2, len(prices)):
          buy[i] = max(sell[i - 2] - prices[i], buy[i - 1])
          sell[i] = max(prices[i] + buy[i - 1], sell[i - 1])
        return max(sell)
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        @cache
        def dfs(i: int, j: int) -> int:
            if i >= len(prices):
                return 0
            ans = dfs(i + 1, j)
            if j:
                ans = max(ans, prices[i] + dfs(i + 2, 0))
            else:
                ans = max(ans, -prices[i] + dfs(i + 1, 1))
            return ans

        return dfs(0, 0)
```


#377 MEDIUM

Combination Sum IV

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming

Lists: Grind 169

Problem:

Given an array of distinct integers nums and a target integer target, return the number of possible combinations that add up to target.

The test cases are generated so that the answer can fit in a 32-bit integer.

Example 1:

```
Input: nums = [1,2,3], target = 4
Output: 7
Explanation:
The possible combination ways are:
(1, 1, 1, 1)
(1, 1, 2)
(1, 2, 1)
(1, 3)
(2, 1, 1)
(2, 2)
(3, 1)
Note that different sequences are counted as different combinations.
```

Example 2:

```
Input: nums = [9], target = 3
Output: 0
```

Constraints:

- 1 <= nums.length <= 200
- 1 <= nums[i] <= 1000
- All the elements of nums are unique.
- 1 <= target <= 1000

Follow up: What if negative numbers are allowed in the given array? How does it change the problem? What limitation we need to add to the question to allow negative numbers?

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def combinationSum4(self, nums, target):
        # Brute Force O(2^n) - plain recursion, no memoisation
        def recurse(i):
            if i >= len(nums): return 0
            take = nums[i] + recurse(i + 1)
            skip = recurse(i + 1)
            return max(take, skip)
        return recurse(0)
```

Python

Python

```python
# Python solution using dynamic programming

class Solution:
    def combinationSum4(self, nums, target):
        # Initialize dp array where dp[i] stores the number of ways to reach sum i
        dp = [0] * (target + 1)
        # Base case: there is one way to form a sum of 0 (use no elements)
        dp[0] = 1

        # Fill the dp table for each sum from 1 to target
        for i in range(1, target + 1):
            for num in nums:
                if i - num >= 0:
                    dp[i] += dp[i - num]
        return dp[target]
```

Python

```python
class Solution:
   def combinationSum4(self, nums: list[int], target: int) -> int:
    dp = [1] + [-1] * target

    def dfs(target: int) -> int:
      if target < 0:
        return 0
      if dp[target] != -1:
        return dp[target]

      dp[target] = sum(dfs(target - num) for num in nums)
      return dp[target]

    return dfs(target)
```

Python

```python
class Solution:
    def combinationSum4(self, nums: List[int], target: int) -> int:
        f = [1] + [0] * target
        for i in range(1, target + 1):
            for x in nums:
                if i >= x:
                    f[i] += f[i - x]
        return f[target]
```

Python


```python
class Solution:
    def combinationSum4(self, nums: List[int], target: int) -> int:
        f = [1] + [0] * target
        for i in range(1, target + 1):
            for x in nums:
                if i >= x:
                    f[i] += f[i - x]
        return f[target]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```python
# Python solution using dynamic programming

class Solution:
    def combinationSum4(self, nums, target):
        # Initialize dp array where dp[i] stores the number of ways to reach sum i
        dp = [0] * (target + 1)
        # Base case: there is one way to form a sum of 0 (use no elements)
        dp[0] = 1

        # Fill the dp table for each sum from 1 to target
        for i in range(1, target + 1):
            for num in nums:
                if i - num >= 0:
                    dp[i] += dp[i - num]
        return dp[target]
```

#416 MEDIUM

Partition Equal Subset Sum

LeetDoocs · SimplyLatet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming

Lists: Grind 169

Problem:

Given an integer array nums, return true if you can partition the array into two subsets such that the sum of the elements in both subsets is equal or false otherwise.

Example 1:

```
Input: nums = [1,5,11,5]
Output: true
Explanation: The array can be partitioned as [1, 5, 5] and [11].
```

Example 2:

```
Input: nums = [1,2,3,5]
Output: false
Explanation: The array cannot be partitioned into equal sum subsets.
```

Constraints:

- 1 <= nums.length <= 200
- 1 <= nums[i] <= 100

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def canPartition(self, nums):
        # Brute Force O(2^n) - try including/excluding each element for target sum
        total = sum(nums)
        if total % 2: return False
        target = total // 2
        def can_reach(i, rem):
            if rem == 0: return True
            if i == len(nums) or rem < 0: return False
            return can_reach(i+1, rem - nums[i]) or can_reach(i+1, rem)
        return can_reach(0, target)
```

Python

Python

```python
# Python solution for Partition Equal Subset Sum

def canPartition(nums):
    # Calculate the total sum of the array
    total_sum = sum(nums)

    # If the total sum is odd, equal partition is not possible
    if total_sum % 2 != 0:
        return False

    # Set the target sum to half of the total sum
    target = total_sum // 2

    # Initialize dp array where dp[i] is True if a subset sum i is achievable
    dp = [False] * (target + 1)
    dp[0] = True # Base case: subset sum of 0 is always possible

    # Iterate through each number in the array
    for num in nums:
        # Iterate backwards to avoid overwriting dp values needed for current iteration
        for i in range(target, num - 1, -1):
            # Update dp[i] if dp[i-num] is True
            if dp[i - num]:
                dp[i] = True


    # dp[target] is True if a subset adds up to target sum
    return dp[target]

# Example usage:
# print(canPartition([1, 5, 11, 5]))
```

Python


```python
class Solution:
   def canPartition(self, nums: list[int]) -> bool:
    summ = sum(nums)
    if summ % 2 == 1:
      return False
    return self.knapsack_(nums, summ // 2)

   def knapsack_(self, nums: list[int], subsetSum: int) -> bool:
    n = len(nums)
    # dp[i][j] := True if j can be formed by nums[0..i]
    dp = [[False] * (subsetSum + 1) for _ in range(n + 1)]
    dp[0][0] = True

    for i in range(1, n + 1):
      num = nums[i - 1]
      for j in range(subsetSum + 1):
        if j < num:
          dp[i][j] = dp[i - 1][j]
        else:
          dp[i][j] = dp[i - 1][j] or dp[i - 1][j - num]

    return dp[n][subsetSum]
```

Python

```python
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        m, mod = divmod(sum(nums), 2)
        if mod:
            return False
        n = len(nums)
        f = [[False] * (m + 1) for _ in range(n + 1)]
        f[0][0] = True
        for i, x in enumerate(nums, 1):
            for j in range(m + 1):
                f[i][j] = f[i - 1][j] or (j >= x and f[i - 1][j - x])
        return f[n][m]
```

Python


```python
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        s = sum(nums)
        if s % 2 != 0:
            return False
        n = s >> 1
        dp = [False] * (n + 1)
        dp[0] = True
        for v in nums:
            for target in range(n, v - 1, -1): # including v itself
                dp[target] = dp[target] or dp[target - v]
        return dp[-1]


class Solution: # recursive, but over time limit in O3
    def canPartition(self, nums: List[int]) -> bool:
        total_sum = sum(nums)
        if total_sum % 2 != 0:
            return False
        target_sum = total_sum // 2
        memo = {} # ==> or just use @cache for dfs()

        def dfs(idx, curr_sum):
            if curr_sum == target_sum:
                return True

            if curr_sum > target_sum or idx >= len(nums):
                return False
            if (idx, curr_sum) in memo:
                return memo[(idx, curr_sum)]

            # Explore two options: include the current number or exclude it
            include = dfs(idx + 1, curr_sum + nums[idx])
            exclude = dfs(idx + 1, curr_sum)

            memo[(idx, curr_sum)] = include or exclude
            return memo[(idx, curr_sum)]

        return dfs(0, 0)

###########

class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        m, mod = divmod(sum(nums), 2)
        if mod:
            return False
        f = [True] + [False] * m
        for x in nums:
            for j in range(m, x - 1, -1):
                f[j] = f[j] or f[j - x]

        return f[m]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python


```python
class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        s = sum(nums)
        if s % 2 != 0:
            return False
        n = s >> 1
        dp = [False] * (n + 1)
        dp[0] = True
        for v in nums:
            for target in range(n, v - 1, -1): # including v itself
                dp[target] = dp[target] or dp[target - v]
        return dp[-1]


class Solution: # recursive, but over time limit in O3
    def canPartition(self, nums: List[int]) -> bool:
        total_sum = sum(nums)
        if total_sum % 2 != 0:
            return False
        target_sum = total_sum // 2
        memo = {} # ==> or just use @cache for dfs()

        def dfs(idx, curr_sum):
            if curr_sum == target_sum:
                return True

            if curr_sum > target_sum or idx >= len(nums):
                return False
            if (idx, curr_sum) in memo:
                return memo[(idx, curr_sum)]

            # Explore two options: include the current number or exclude it
            include = dfs(idx + 1, curr_sum + nums[idx])
            exclude = dfs(idx + 1, curr_sum)

            memo[(idx, curr_sum)] = include or exclude
            return memo[(idx, curr_sum)]

        return dfs(0, 0)

###########

class Solution:
    def canPartition(self, nums: List[int]) -> bool:
        m, mod = divmod(sum(nums), 2)
        if mod:
            return False
        f = [True] + [False] * m
        for x in nums:
            for j in range(m, x - 1, -1):
                f[j] = f[j] or f[j - x]

        return f[m]
```

#435 MEDIUM

Non-overlapping Intervals

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming · Greedy · Sorting

Lists: Grind 169

Problem:

Given an array of intervals intervals where intervals[i] = [starti, endi], return the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping.

Note that intervals which only touch at a point are non-overlapping. For example, [1, 2] and [2, 3] are non-overlapping.

Example 1:

```
Input: intervals = [[1,2],[2,3],[3,4],[1,3]]
Output: 1
Explanation: [1,3] can be removed and the rest of the intervals are non-overlapping.
```

Example 2:

```
Input: intervals = [[1,2],[1,2],[1,2]]
Output: 2
Explanation: You need to remove two [1,2] to make the rest of the intervals non-overlapping.
```

Example 3:

```
Input: intervals = [[1,2],[2,3]]
Output: 0
Explanation: You don't need to remove any of the intervals since they're already
non-overlapping.
```

Constraints:

- 1 <= intervals.length <= 105
- intervals[i].length == 2
- -5 * 104 <= starti < endi <= 5 * 104

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def eraseOverlapIntervals(self, intervals):
        # Brute Force O(2^n) - try removing every subset, keep smallest valid
        # (In practice: sort by end, greedy is O(n log n); brute shown for intuition)
        intervals.sort(key=lambda x: x[1])
        removed = 0
        end = float('-inf')
        for iv in intervals:
            if iv[0] >= end:
                end = iv[1] # keep this interval
            else:
                removed += 1 # drop overlapping interval
        return removed
```

Python

Python


```python
# Python implementation for Non-overlapping Intervals

def eraseOverlapIntervals(intervals):
    # Sort intervals based on the end time
    intervals.sort(key=lambda x: x[1])

    # Initialize the count of intervals in the non-overlapping set
    count = 0
    # Initialize the end of the last added interval to the minimum possible value
    last_included_end = float('-inf')

    # Iterate over every interval in the sorted list
    for interval in intervals:
        # If the current interval does not overlap with the last added interval
        if interval[0] >= last_included_end:
            count += 1 # We include this interval
            last_included_end = interval[1] # Update end for subsequent comparisons
    # The minimum number of removals is total intervals minus non-overlapping intervals count
    return len(intervals) - count

# Example usage:
print(eraseOverlapIntervals([[1,2],[2,3],[3,4],[1,3]])) # Expected output: 1
```

Python

```python
class Solution:
  def eraseOverlapIntervals(self, intervals: list[list[int]]) -> int:
    ans = 0
    currentEnd = -math.inf

    for interval in sorted(intervals, key=lambda x: x[1]):
      if interval[0] >= currentEnd:
        currentEnd = interval[1]
      else:
        ans += 1

    return ans
```

Python

```python
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        intervals.sort(key=lambda x: x[1])
        ans = len(intervals)
        pre = -inf
        for l, r in intervals:
            if pre <= l:
                ans -= 1
                pre = r
        return ans
```

Python


```python
class Solution:
    def eraseOverlapIntervals(self, intervals: List[List[int]]) -> int:
        intervals.sort(key=lambda x: x[1])
        ans, t = 0, intervals[0][1]
        for s, e in intervals[1:]:
            if s >= t:
                t = e
            else:
                ans += 1
        return ans
```

[*] Dynamic Programming — Pattern Solution (stored optimal)

Python

```python
# Python implementation for Non-overlapping Intervals

def eraseOverlapIntervals(intervals):
    # Sort intervals based on the end time
    intervals.sort(key=lambda x: x[1])

    # Initialize the count of intervals in the non-overlapping set
    count = 0
    # Initialize the end of the last added interval to the minimum possible value
    last_included_end = float('-inf')

    # Iterate over every interval in the sorted list
    for interval in intervals:
        # If the current interval does not overlap with the last added interval
        if interval[0] >= last_included_end:
            count += 1 # We include this interval
            last_included_end = interval[1] # Update end for subsequent comparisons
    # The minimum number of removals is total intervals minus non-overlapping intervals count
    return len(intervals) - count

# Example usage:
print(eraseOverlapIntervals([[1,2],[2,3],[3,4],[1,3]])) # Expected output: 1
```

#516 MEDIUM

Longest Palindromic Subsequence

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Dynamic Programming

Lists: Denny Zhang

Problem:

Given a string s, find the longest palindromic subsequence's length in s.

A subsequence is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

Example 1:

```
Input: s = "bbbab"
Output: 4
Explanation: One possible longest palindromic subsequence is "bbbb".
```


Example 2:

```
Input: s = "cbbd"
Output: 2
Explanation: One possible longest palindromic subsequence is "bb".
```

Constraints:

- 1 <= s.length <= 1000
- s consists only of lowercase English letters.

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def longest_palindromic_subseq(self, s):
        # Brute Force O(2^n) - plain recursion, no memoisation
        def recurse(i):
            if i >= len(s): return 0
            take = s[i] + recurse(i + 1)
            skip = recurse(i + 1)
            return max(take, skip)
        return recurse(0)
```

Python

Python

```python
# Function to compute the longest palindromic subsequence length
def longest_palindromic_subseq(s):
    n = len(s)
    # Create a 2D DP table initialized to 0
    dp = [[0] * n for _ in range(n)]

    # Base case: every single character is a palindrome of length 1
    for i in range(n):
        dp[i][i] = 1

    # Build the table. 'sub_len' is the current length of substring we are considering.
    for sub_len in range(2, n+1):
        for i in range(n - sub_len + 1):
            j = i + sub_len - 1 # Compute the end index of the substring
            if s[i] == s[j]:
                # Characters match, add 2 to the result from the inner substring if available
                dp[i][j] = 2 + (dp[i+1][j-1] if sub_len > 2 else 0)
            else:
                # Characters don't match, choose the maximum from either excluding left or right char
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])
    # The answer for the full string is stored at dp[0][n-1]
    return dp[0][n-1]

# Example usage:
s = "bbbab"

print(longest_palindromic_subseq(s)) # Expected output: 4
```

Python


```python
class Solution:
    def longestPalindromeSubseq(self, s: str) -> int:
        @functools.lru_cache(None)
        def dp(i: int, j: int) -> int:
            """Returns the length of LPS(s[i..j])."""
            if i > j:
                return 0
            if i == j:
                return 1
            if s[i] == s[j]:
                return 2 + dp(i + 1, j - 1)
            return max(dp(i + 1, j), dp(i, j - 1))

        return dp(0, len(s) - 1)
```

Python

```python
class Solution:
    def longestPalindromeSubseq(self, s: str) -> int:
        n = len(s)
        # dp[i][j] := the length of LPS(s[i..j])
        dp = [[0] * n for _ in range(n)]

        for i in range(n):
            dp[i][i] = 1

        for d in range(1, n):
            for i in range(n - d):
                j = i + d
                if s[i] == s[j]:
                    dp[i][j] = 2 + dp[i + 1][j - 1]
                else:
                    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])

        return dp[0][n - 1]
```

Python

```python
class Solution:
    def longestPalindromeSubseq(self, s: str) -> int:
        n = len(s)
        f = [[0] * n for _ in range(n)]
        for i in range(n):
            f[i][i] = 1
        for i in range(n - 1, -1, -1):
            for j in range(i + 1, n):
                if s[i] == s[j]:
                    f[i][j] = f[i + 1][j - 1] + 2
                else:
                    f[i][j] = max(f[i + 1][j], f[i][j - 1])
        return f[0][-1]
```

Python


```python
class Solution:
    def longestPalindromeSubseq(self, s: str) -> int:
        n = len(s)
        dp = [[0] * n for _ in range(n)]
        for i in range(n - 1, -1, -1):
            dp[i][i] = 1 # pre-set before inner loop
            for j in range(i + 1, n):
                if s[i] == s[j]:
                    dp[i][j] = dp[i + 1][j - 1] + 2
                else:
                    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])
        return dp[0][-1]

##########

class Solution:
    def longestPalindromeSubseq(self, s: str) -> int:
        n = len(s)
        dp = [[0] * n for _ in range(n)]
        for i in range(n):
            dp[i][i] = 1 # cannot move it inside 'for j in range(i, n)'
            # because below 'for j in range' starting at i
            # so cannot put dp[j][j]=1 inside below for loop
        for j in range(1, n):
            for i in range(j - 1, -1, -1):

                if s[i] == s[j]:
                    dp[i][j] = dp[i + 1][j - 1] + 2
                    # "aa" => i=0,j=1 => dp[i+1][0] is 0, still works
                else:
                    dp[i][j] = max(dp[i + 1][j], dp[i][j - 1])
        return dp[0][-1]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```python
# Function to compute the longest palindromic subsequence length
def longest_palindromic_subseq(s):
    n = len(s)
    # Create a 2D DP table initialized to 0
    dp = [[0] * n for _ in range(n)]

    # Base case: every single character is a palindrome of length 1
    for i in range(n):
        dp[i][i] = 1

    # Build the table. 'sub_len' is the current length of substring we are considering.
    for sub_len in range(2, n+1):
        for i in range(n - sub_len + 1):
            j = i + sub_len - 1 # Compute the end index of the substring
            if s[i] == s[j]:
                # Characters match, add 2 to the result from the inner substring if available
                dp[i][j] = 2 + (dp[i+1][j-1] if sub_len > 2 else 0)
            else:
                # Characters don't match, choose the maximum from either excluding left or right char
                dp[i][j] = max(dp[i+1][j], dp[i][j-1])
    # The answer for the full string is stored at dp[0][n-1]
    return dp[0][n-1]

# Example usage:
s = "bbbab"

print(longest_palindromic_subseq(s)) # Expected output: 4
```

#576 MEDIUM

Out of Boundary Paths

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Dynamic Programming

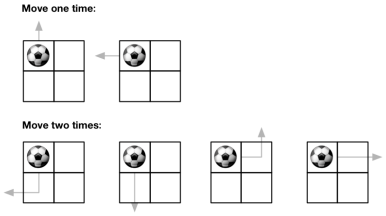
Lists: Denny Zhang

Problem:

There is an m x n grid with a ball. The ball is initially at the position [startRow, startColumn]. You are allowed to move the ball to one of the four adjacent cells in the grid (possibly out of the grid crossing the grid boundary). You can apply at most maxMove moves to the ball.

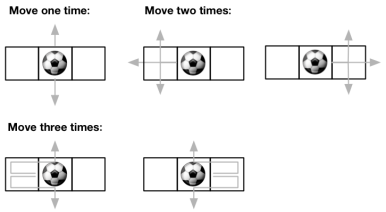
Given the five integers m, n, maxMove, startRow, startColumn, return the number of paths to move the ball out of the grid boundary. Since the answer can be very large, return it modulo 109 + 7.

Example 1:




```
Input: m = 2, n = 2, maxMove = 2, startRow = 0, startColumn = 0
Output: 6
```

Example 2:




```
Input: m = 1, n = 3, maxMove = 3, startRow = 0, startColumn = 1
Output: 12
```

Constraints:

- 1 <= m, n <= 50
- 0 <= maxMove <= 50
- 0 <= startRow < m
- 0 <= startColumn < n

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findPaths(self, m, n, maxMove, startRow, startColumn):
        # Brute Force O(3^n) - plain recursion, no memoization
        def recurse(i):
            if i >= len(maxMove): return 0
            take = maxMove[i] + recurse(i + 1)
            skip = recurse(i + 1)
            return max(take, skip)
        return recurse(0)
```

Python

Python

```python
MOD = 10**9 + 7

def findPaths(m, n, maxMove, startRow, startColumn):
    # Initialize memo dictionary to cache the states
    memo = {}

    def dp(i, j, movesLeft):
        # If out of grid bounds, it's a valid path
        if i < 0 or i >= m or j < 0 or j >= n:
            return 1
        # No moves left and still inside the grid means no valid path out
        if movesLeft == 0:
            return 0
        # Check if we already computed this state
        if (i, j, movesLeft) in memo:
            return memo[(i, j, movesLeft)]

        count = 0
        # Four possible directions: up, down, left, right
        count = (dp(i - 1, j, movesLeft - 1) +
                 dp(i + 1, j, movesLeft - 1) +
                 dp(i, j - 1, movesLeft - 1) +
                 dp(i, j + 1, movesLeft - 1)) % MOD

        memo[(i, j, movesLeft)] = count

        return count

    return dp(startRow, startColumn, maxMove)

# Example usage
if __name__ == "__main__":
    print(findPaths(2, 2, 2, 0, 0)) # Expected output: 6
```

Python


```python
class Solution:
    def findPaths(self, m, n, maxMove, startRow, startColumn):
        MOD = 1000000007

        @functools.lru_cache(None)
        def dp(k: int, i: int, j: int) -> int:
            """
            Returns the number of paths to move the ball at (i, j) out-of-bounds with
            k moves.
            """
            if i < 0 or i >= m or j < 0 or j >= n:
                return 1
            if k == 0:
                return 0
            return (dp(k - 1, i + 1, j) + dp(k - 1, i - 1, j) +
                    dp(k - 1, i, j + 1) + dp(k - 1, i, j - 1)) % MOD

        return dp(maxMove, startRow, startColumn)
```

Python

```python
class Solution:
    def findPaths(
        self,
        m: int,
        n: int,
        maxMove: int,
        startRow: int,
        startColumn: int,
    ) -> int:
        DIRS = ((0, 1), (1, 0), (0, -1), (-1, 0))
        MOD = 1_000_000_007
        ans = 0
        # dp[i][j] := the number of paths to move the ball (i, j) out-of-bounds
        dp = [[0] * n for _ in range(m)]
        dp[startRow][startColumn] = 1

        for _ in range(maxMove):
          newDp = [[0] * n for _ in range(m)]
          for i in range(m):
            for j in range(n):
              if dp[i][j] > 0:
                for dx, dy in DIRS:
                  x = i + dx
                  y = j + dy
                  if x < 0 or x >= m or y < 0 or y >= n:
                    ans = (ans + dp[i][j]) % MOD
                  else:
                    newDp[x][y] = (newDp[x][y] + dp[i][j]) % MOD
          dp = newDp

        return ans
```

Python


```python
class Solution:
    def findPaths(
        self, m: int, n: int, maxMove: int, startRow: int, startColumn: int
    ) -> int:
        @cache
        def dfs(i: int, j: int, k: int) -> int:
            if not 0 <= i < m or not 0 <= j < n:
                return int(k >= 0)
            if k <= 0:
                return 0
            ans = 0
            for a, b in pairwise(dirs):
                x, y = i + a, j + b
                ans = (ans + dfs(x, y, k - 1)) % mod
            return ans

        mod = 10**9 + 7
        dirs = (-1, 0, 1, 0, -1)
        return dfs(startRow, startColumn, maxMove)
```

Python

```python
class Solution:
    def findPaths(
        self, m: int, n: int, maxMove: int, startRow: int, startColumn: int
    ) -> int:
        @cache
        def dfs(i, j, k):
            if i < 0 or j < 0 or i >= m or j >= n:
                return 1
            if k == 0:
                return 0
            res = 0
            for a, b in [(-1, 0), (1, 0), (0, 1), (0, -1)]:
                x, y = i + a, j + b
                res += dfs(x, y, k - 1)
                res %= mod
            return res

        mod = 10**9 + 7
        return dfs(startRow, startColumn, maxMove)
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python


```python
class Solution:
    def findPaths(
        self,
        m: int,
        n: int,
        maxMove: int,
        startRow: int,
        startColumn: int,
    ) -> int:
        DIRS = ((0, 1), (1, 0), (0, -1), (-1, 0))
        MOD = 1_000_000_007
        ans = 0
        # dp[i][j] := the number of paths to move the ball (i, j) out-of-bounds
        dp = [[0] * n for _ in range(m)]
        dp[startRow][startColumn] = 1

        for _ in range(maxMove):
          newDp = [[0] * n for _ in range(m)]
          for i in range(m):
            for j in range(n):
              if dp[i][j] > 0:
                for dx, dy in DIRS:
                  x = i + dx
                  y = j + dy
                  if x < 0 or x >= m or y < 0 or y >= n:
                    ans = (ans + dp[i][j]) % MOD
                  else:
                    newDp[x][y] = (newDp[x][y] + dp[i][j]) % MOD
          dp = newDp

        return ans
```

#1143 MEDIUM

Longest Common Subsequence

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Dynamic Programming

Lists: Denny Zhang

Problem:

Given two strings text1 and text2, return the length of their longest common subsequence. If there is no common subsequence, return 0.

A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, "ace" is a subsequence of "abcde".

A common subsequence of two strings is a subsequence that is common to both strings.

Example 1:

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.
```

Example 2:

```
Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.
```

Example 3:

```
Input: text1 = "abc", text2 = "def"
Output: 0
Explanation: There is no such common subsequence, so the result is 0.
```

Constraints:

- 1 <= text1.length, text2.length <= 1000
- text1 and text2 consist of only lowercase English characters.

Python

Python

```python
# Function to find the longest common subsequence length
def longestCommonSubsequence(text1: str, text2: str) -> int:
    m, n = len(text1), len(text2)
    # Initialize a (m+1) x (n+1) DP table filled with zeros
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Build the DP table bottom-up
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            # If characters match, extend the subsequence length by 1
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                # Otherwise, choose the maximum from the left or top cell
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]

# Example usage
if __name__ == "__main__":
    text1 = "abcde"
    text2 = "ace"
    print(longestCommonSubsequence(text1, text2)) # Output: 3
```

Python

```python
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        m = len(text1)
        n = len(text2)
        # dp[i][j] := the length of LCS(text1[0..i), text2[0..j))
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        for i in range(m):
          for j in range(n):
            dp[i + 1][j + 1] = (1 + dp[i][j] if text1[i] == text2[j]
                                else max(dp[i][j + 1], dp[i + 1][j]))

        return dp[m][n]
```

Python


```python
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        m, n = len(text1), len(text2)
        f = [[0] * (n + 1) for _ in range(m + 1)]
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if text1[i - 1] == text2[j - 1]:
                    f[i][j] = f[i - 1][j - 1] + 1
                else:
                    f[i][j] = max(f[i - 1][j], f[i][j - 1])
        return f[m][n]
```

Python

```python
class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        m, n = len(text1), len(text2)
        f = [[0] * (n + 1) for _ in range(m + 1)]
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if text1[i - 1] == text2[j - 1]:
                    f[i][j] = f[i - 1][j - 1] + 1
                else:
                    f[i][j] = max(f[i - 1][j], f[i][j - 1])
        return f[m][n]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```python
# Function to find the longest common subsequence length
def longestCommonSubsequence(text1: str, text2: str) -> int:
    m, n = len(text1), len(text2)
    # Initialize a (m+1) x (n+1) DP table filled with zeros
    dp = [[0] * (n + 1) for _ in range(m + 1)]

    # Build the DP table bottom-up
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            # If characters match, extend the subsequence length by 1
            if text1[i - 1] == text2[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + 1
            else:
                # Otherwise, choose the maximum from the left or top cell
                dp[i][j] = max(dp[i - 1][j], dp[i][j - 1])

    return dp[m][n]

# Example usage
if __name__ == "__main__":
    text1 = "abcde"
    text2 = "ace"
    print(longestCommonSubsequence(text1, text2)) # Output: 3
```

#10 HARD

Regular Expression Matching

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Dynamic Programming · Recursion

Lists: Denny Zhang

Problem:

Given an input string s and a pattern p, implement regular expression matching with support for '.' and '*' where:

- '.' Matches any single character.
- '*' Matches zero or more of the preceding element.

Return a boolean indicating whether the matching covers the entire input string (not partial).

Example 1:

```
Input: s = "aa", p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".
```

Example 2:

```
Input: s = "aa", p = "a*"
Output: true
Explanation: '*' means zero or more of the preceding element, 'a'. Therefore, by repeating
 'a' once, it becomes "aa".
```

Example 3:

```
Input: s = "ab", p = ".*"
Output: true
Explanation: ".*" means "zero or more (*) of any character (.)".
```

Constraints:

- 1 <= s.length <= 20
- 1 <= p.length <= 20
- s contains only lowercase English letters.
- p contains only lowercase English letters, '.', and '*'.
- It is guaranteed for each appearance of the character '*', there will be a previous valid character to match.

Python

Python


```python
# Python solution using recursion with memoization
def isMatch(s: str, p: str) -> bool:
    memo = {}

    def dp(i: int, j: int) -> bool:
        # If result is already computed, return it.
        if (i, j) in memo:
            return memo[(i, j)]

        # If we have reached the end of the pattern, s must also be at the end.
        if j == len(p):
            ans = i == len(s)
        else:
            # Check if current characters match
            first_match = i < len(s) and (p[j] == s[i] or p[j] == '.')

            # If there is a '*' coming after the current pattern character
            if j + 1 < len(p) and p[j+1] == '*':
                # Two cases:
                # 1. '*' stands for zero occurrence -> skip current pattern char and '*'
                # 2. first_match is true, so we try to consume one character in s and remain at same
pattern index
                ans = dp(i, j+2) or (first_match and dp(i+1, j))
            else:
                # No '*' means we directly check next characters in both s and p.
                ans = first_match and dp(i+1, j+1)

        memo[(i, j)] = ans
        return ans

    return dp(0, 0)


# Example test run
#print(isMatch("aa", "a*")) # Expected output: True
```

Python


```python
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        m = len(s)
        n = len(p)
        # dp[i][j] := True if s[0..i) matches p[0..j)
        dp = [[False] * (n + 1) for _ in range(m + 1)]
        dp[0][0] = True

        def isMatch(i: int, j: int) -> bool:
          return j >= 0 and p[j] == '.' or s[i] == p[j]

        for j, c in enumerate(p):
          if c == '*' and dp[0][j - 1]:
            dp[0][j + 1] = True

        for i in range(m):
          for j in range(n):
            if p[j] == '*':
              # The minimum index of '*' is 1.
              noRepeat = dp[i + 1][j - 1]
              doRepeat = isMatch(i, j - 1) and dp[i][j + 1]
              dp[i + 1][j + 1] = noRepeat or doRepeat
            elif isMatch(i, j):
              dp[i + 1][j + 1] = dp[i][j]

        return dp[m][n]
```

Python

```python
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        @cache
        def dfs(i, j):
            if j >= n:
                return i == m
            if j + 1 < n and p[j + 1] == '*':
                return dfs(i, j + 2) or (
                    i < m and (s[i] == p[j] or p[j] == '.') and dfs(i + 1, j)
                )
            return i < m and (s[i] == p[j] or p[j] == '.') and dfs(i + 1, j + 1)

        m, n = len(s), len(p)
        return dfs(0, 0)
```

Python


```python
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        m, n = len(s), len(p)
        f = [[False] * (n + 1) for _ in range(m + 1)]
        f[0][0] = True
        for i in range(m + 1):
            for j in range(1, n + 1):
                if p[j - 1] == "*":
                    f[i][j] = f[i][j - 2]
                    if i > 0 and (p[j - 2] == "." or s[i - 1] == p[j - 2]):
                        f[i][j] |= f[i - 1][j]
                elif i > 0 and (p[j - 1] == "." or s[i - 1] == p[j - 1]):
                    f[i][j] = f[i - 1][j - 1]
        return f[m][n]
```

Python

```python
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        m, n = len(s), len(p)
        f = [[False] * (n + 1) for _ in range(m + 1)]
        f[0][0] = True
        for i in range(m + 1):
            for j in range(1, n + 1):
                if p[j - 1] == "*":
                    f[i][j] = f[i][j - 2]
                    if i > 0 and (p[j - 2] == "." or s[i - 1] == p[j - 2]):
                        f[i][j] |= f[i - 1][j]
                elif i > 0 and (p[j - 1] == "." or s[i - 1] == p[j - 1]):
                    f[i][j] = f[i - 1][j - 1]
        return f[m][n]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        m = len(s)
        n = len(p)
        # dp[i][j] = True if s[0..i] matches p[0..j]
        dp = [[False] * (n + 1) for _ in range(m + 1)]
        dp[0][0] = True

        def isMatch(i: int, j: int) -> bool:
            return j >= 0 and p[j] == '.' or s[i] == p[j]

        for j, c in enumerate(p):
            if c == '*' and dp[0][j - 1]:
                dp[0][j + 1] = True

        for i in range(m):
            for j in range(n):
                if p[j] == '*':
                    # The minimum index of '*' is 1.
                    noRepeat = dp[i + 1][j - 1]
                    doRepeat = isMatch(i, j - 1) and dp[i][j + 1]
                    dp[i + 1][j + 1] = noRepeat or doRepeat
                elif isMatch(i, j):
                    dp[i + 1][j + 1] = dp[i][j]

        return dp[m][n]
```

#115 HARD

Distinct Subsequences

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Dynamic Programming

Lists: Denny Zhang

Problem:

Given two strings s and t, return the number of distinct subsequences of s which equals t.

The test cases are generated so that the answer fits on a 32-bit signed integer.

Example 1:

```
Input: s = "rabbbit", t = "rabbit"
Output: 3
Explanation:
As shown below, there are 3 ways you can generate "rabbit" from s.
rabbbit
rabbbit
rabbbit
```

Example 2:

```
Input: s = "babgbag", t = "bag"
Output: 5
Explanation:
As shown below, there are 5 ways you can generate "bag" from s.
babgbag
```



```
babgbag
babgbag
babgbag
babgbag
```

Constraints:

- 1 <= s.length, t.length <= 1000
- s and t consist of English letters.

Python

Python

```python
# Python solution for Distinct Subsequences
def numDistinct(s: str, t: str) -> int:
    # Get lengths of the strings s and t
    m, n = len(s), len(t)
    # Initialize a dp table with (m+1) rows and (n+1) columns filled with 0
    dp = [[0] + (n + 1) for _ in range(m + 1)]

    # Base case: An empty string t (j = 0) is a subsequence of any prefix of s, so set dp[i][0] = 1
    for i in range(m + 1):
        dp[i][0] = 1

    # Fill the dp table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            # If the current characters match, add the count of sequences by either using or skipping
s[i-1]
            if s[i - 1] == t[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]
            else:
                # If they do not match, skip s[i-1]
                dp[i][j] = dp[i - 1][j]
    # The answer is the way to form all of t from all of s
    return dp[m][n]

# Example usage:
print(numDistinct("rabbbit", "rabbit")) # Output: 3
```

Python


```python
class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        m = len(s)
        n = len(t)
        dp = [[0] + (n + 1) for _ in range(m + 1)]

        for i in range(n + 1):
            dp[i][0] = 1

        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if s[i - 1] == t[j - 1]:
                    dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]
                else:
                    dp[i][j] = dp[i - 1][j]

        return dp[m][n]
```

Python

```python
class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        m, n = len(s), len(t)
        f = [[0] + (n + 1) for _ in range(m + 1)]
        for i in range(m + 1):
            f[i][0] = 1
        for i, a in enumerate(s, 1):
            for j, b in enumerate(t, 1):
                f[i][j] = f[i - 1][j]
                if a == b:
                    f[i][j] += f[i - 1][j - 1]
        return f[m][n]
```

Python

```python
class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        n = len(t)
        f = [1] + [0] * n
        for a in s:
            for j in range(n, 0, -1):
                if a == t[j - 1]:
                    f[j] += f[j - 1]
        return f[n]
```

Python

```python
class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        n = len(t)
        f = [1] + [0] * n
        for a in s:
            for j in range(n, 0, -1):
                if a == t[j - 1]:
                    f[j] += f[j - 1]
        return f[n]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python


```python
# Python solution for Distinct Subsequences
def numDistinct(s: str, t: str) -> int:
    # Get lengths of the strings s and t
    m, n = len(s), len(t)
    # Initialize a dp table with (m+1) rows and (n+1) columns filled with 0
    dp = [[0] + (n + 1) for _ in range(m + 1)]

    # Base case: An empty string t (j = 0) is a subsequence of any prefix of s, so set dp[i][0] = 1
    for i in range(m + 1):
        dp[i][0] = 1

    # Fill the dp table
    for i in range(1, m + 1):
        for j in range(1, n + 1):
            # If the current characters match, add the count of sequences by either using or skipping
s[i-1]
            if s[i - 1] == t[j - 1]:
                dp[i][j] = dp[i - 1][j - 1] + dp[i - 1][j]
            else:
                # If they do not match, skip s[i-1]
                dp[i][j] = dp[i - 1][j]
    # The answer is the way to form all of t from all of s
    return dp[m][n]

# Example usage:
print(numDistinct("rabbbit", "rabbit")) # Output: 3
```

#123 HARD

Best Time to Buy and Sell Stock III

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming

Lists: Denny Zhang

Problem:

You are given an array prices where prices[i] is the price of a given stock on the ith day.

Find the maximum profit you can achieve. You may complete at most two transactions.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

```
Input: prices = [3,3,5,0,0,3,1,4]
Output: 6
Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.
Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.
```

Example 2:

```
Input: prices = [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engaging
multiple transactions at the same time. You must sell before buying again.
```


Example 3:

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transaction is done, i.e. max profit = 0.
```

Constraints:

- 1 <= prices.length <= 105
- 0 <= prices[i] <= 105

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute-Force Approach
class Solution:
    def max_profit(self, prices):
        # Brute Force O(2^n) - plain recursion, no memoisation
        def recurse(i):
            if i >= len(prices): return 0
            take = prices[i] + recurse(i + 1)
            skip = recurse(i + 1)
            return max(take, skip)
        return recurse(0)
```

Python

Python

```python
# Python solution for Best Time to Buy and Sell Stock III

def max_profit(prices):
    # If there are less than 2 days, no transaction can be made.
    if not prices or len(prices) < 2:
        return 0

    n = len(prices)
    # Initialize left and right profit arrays with zeros.
    left_profits = [0] * n
    right_profits = [0] * n

    # Forward pass: compute max profit until day i with one transaction.
    min_price = prices[0]
    for i in range(1, n):
        # Update the min price encountered so far.
        min_price = min(min_price, prices[i])
        # Maximum profit if sold on day i.
        left_profits[i] = max(left_profits[i - 1], prices[i] - min_price)

    # Backward pass: compute max profit from day i to end with one transaction.
    max_price = prices[-1]
    for i in range(n - 2, -1, -1):
        # Update max price seen from the right.
        max_price = max(max_price, prices[i])
```



```
        # Maximum profit possible starting from day i.
        right_profits[i] = max(right_profits[i + 1], max_price - prices[i])

    # Combine the two profits to find the maximum total profit.
    max_total_profit = 0
    for i in range(n):
        max_total_profit = max(max_total_profit, left_profits[i] + right_profits[i])

    return max_total_profit

# Example usage:
prices = [3,3,5,0,0,3,1,4]
print(max_profit(prices)) # Expected output: 6
```

Python

```python
class Solution:
    def maxProfit(self, prices: list[int]) -> int:
        sellTwo = 0
        holdTwo = -math.inf
        sellOne = 0
        holdOne = -math.inf

        for price in prices:
            sellTwo = max(sellTwo, holdTwo + price)
            holdTwo = max(holdTwo, sellOne - price)
            sellOne = max(sellOne, holdOne + price)
            holdOne = max(holdOne, -price)

        return sellTwo
```

Python

```python
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # Initialize profits and prices.
        f1, f2, f3, f4 = -prices[0], 0, -prices[0], 0
        for price in prices[1:]:
            f1 = max(f1, -price)
            f2 = max(f2, f1 + price)
            f3 = max(f3, f2 - price)
            f4 = max(f4, f3 + price)
        return f4
```

Python

```
'''
- f1: 第一天结束时的最大利润
- f2: 第一天开始时的最大利润
- f3: 第二天结束时的最大利润
- f4: 第二天开始时的最大利润
'''

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        # 初始化四个状态
        f1, f2, f3, f4 = -prices[0], 0, -prices[0], 0
        for price in prices[1:]:
            f1 = max(f1, -price)
            f2 = max(f2, f1 + price)
            f3 = max(f3, f2 - price)
            f4 = max(f4, f3 + price)
        return f4


class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        if not prices:
            return 0

        # 0 to i, max profit
        left_max = [0] * len(prices)


        tmp_min = prices[0]
        for i in range(len(prices)):
            tmp_min = min(tmp_min, prices[i])
            left_max[i] = max(left_max[i-1], prices[i]-tmp_min) if i > 0 else 0

        # i to end, max profit
        right_max = [0] * len(prices)

        tmp_max = prices[-1]
        for i in range(len(prices)-1, -1, -1):
            tmp_max = max(tmp_max, prices[i])
            right_max[i] = max(right_max[i+1], tmp_max-prices[i]) if i < len(prices)-1 else 0

        return max(left_max[i] + right_max[i] for i in range(len(prices)))

###########

class Solution(object):
  def maxProfit(self, prices):
    """
    :type prices: List[int]
    :rtype: int
    """
    buy1 = buy2 = float("-inf")
```



```
    sell1 = sell2 = 0

    for i in range(len(prices)):
      sell1 = max(prices[i] + buy1, sell1)
      buy1 = max(buy1, -prices[i])
      sell2 = max(sell2, prices[i] + buy2)
      buy2 = max(sell1 - prices[i], buy2)
    return max(sell1, sell2)
```

[*] Dynamic Programming — Pattern Solution (stored optimal)

Python

```
# Python solution for Best Time to Buy and Sell Stock III

def max_profit(prices):
    # If there are less than 2 days, no transaction can be made.
    if not prices or len(prices) < 2:
        return 0

    n = len(prices)
    # Initialize left and right profit arrays with zeros.
    left_profits = [0] * n
    right_profits = [0] * n

    # Forward pass: compute max profit until day i with one transaction.
    min_price = prices[0]
    for i in range(1, n):
        # Update the min price encountered so far.
        min_price = min(min_price, prices[i])
        # Maximum profit if sold on day i.
        left_profits[i] = max(left_profits[i - 1], prices[i] - min_price)

    # Backward pass: compute max profit from day i to end with one transaction.
    max_price = prices[-1]
    for i in range(n - 2, -1, -1):
        # Update max price seen from the right.
        max_price = max(max_price, prices[i])

        # Maximum profit possible starting from day i.
        right_profits[i] = max(right_profits[i + 1], max_price - prices[i])

    # Combine the two profits to find the maximum total profit.
    max_total_profit = 0
    for i in range(n):
        max_total_profit = max(max_total_profit, left_profits[i] + right_profits[i])

    return max_total_profit

# Example usage:
prices = [3,3,5,0,0,3,1,4]
print(max_profit(prices)) # Expected output: 6
```

#132 HARD

Palindrome Partitioning II

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Dynamic Programming

Lists: Denny Zhang

Problem:

Given a string s, partition s such that every substring of the partition is a palindrome.

Return the minimum cuts needed for a palindrome partitioning of s.

Example 1:

```
Input: s = "aab"
Output: 1
Explanation: The palindrome partitioning ["aa","b"] could be produced using 1 cut.
```

Example 2:

```
Input: s = "a"
Output: 0
```

Example 3:

```
Input: s = "ab"
Output: 1
```

Constraints:

- 1 <= s.length <= 2000
- s consists of lowercase English letters only.

Python

Python

```
def minCut(s: str) -> int:
    n = len(s)
    # Precompute palindrome table where isPal[i][j] is True if s[i:j+1] is a palindrome.
    isPal = [[False] * n for _ in range(n)]
    # dp[i] will store the minimum cuts required for substring s[0...i]
    dp = [0] * n
    for i in range(n):
        min_cut = i # maximum cuts (cut every character)
        for j in range(i + 1):
            # Check if s[j...i] is a palindrome.
            if s[j] == s[i] and (i - j < 2 or isPal[j+1][i-1]):
                isPal[j][i] = True # Mark substring as palindrome.
                # If the palindrome starts at index 0, no cut is needed.
                min_cut = 0 if j == 0 else min(min_cut, dp[j-1] + 1)
        dp[i] = min_cut # Record minimum cuts needed for s[0...i]
    return dp[-1]

# Example usage:
# print(minCut("aab")) # Output: 1
```

Python


```
class Solution:
  def minCut(self, s: str) -> int:
    n = len(s)
    # isPalindrome[i][j] := True if s[i..j] is a palindrome
    isPalindrome = [[True] * n for _ in range(n)]
    # dp[i] := the minimum cuts needed for a palindrome partitioning of s[0..i]
    dp = [n] * n

    for l in range(2, n + 1):
      i = 0
      for j in range(l - 1, n):
        isPalindrome[i][j] = s[i] == s[j] and isPalindrome[i + 1][j - 1]
        i += 1

    for i in range(n):
      if isPalindrome[0][i]:
        dp[i] = 0
        continue

      # Try all the possible partitions.
      for j in range(i):
        if isPalindrome[j + 1][i]:
          dp[i] = min(dp[i], dp[j] + 1)

    return dp[-1]
```

Python

```
class Solution:
    def minCut(self, s: str) -> int:
        n = len(s)
        g = [[True] * n for _ in range(n)]
        for i in range(n - 1, -1, -1):
            for j in range(i + 1, n):
                g[i][j] = s[i] == s[j] and g[i + 1][j - 1]
        f = list(range(n))
        for i in range(1, n):
            for j in range(i + 1):
                if g[j][i]:
                    f[i] = min(f[i], 1 + f[j - 1] if j else 0)
        return f[-1]
```

Python


```
class Solution:
    def minCut(self, s: str) -> int:
        pal = [[False for j in range(0, len(s))] for i in range(0, len(s))]
        dp = [len(s) for _ in range(0, len(s) + 1)]
        for i in range(0, len(s)):
            for j in range(0, i + 1):
                # also ok if j == 0 or j == i - 1
                if (s[i] == s[j]) and ((j + 1 > i - 1) or (pal[i - 1][j + 1])):
                    pal[i][j] = True
                    dp[i + 1] = min(dp[i + 1], dp[j] + 1) if j != 0 else 0
                    # if j != 0 to ensure from start to i is a palindrome
        return dp[-1]

###########

class Solution: # clear cache
    def minCut(self, s: str) -> int:
        @cache
        def dfs(i):
            if i >= n - 1:
                return 0
            ans = inf
            for j in range(i, n):
                if g[i][j]:
                    ans = min(ans, dfs(j + 1) + (j < n - 1))

            return ans

        n = len(s)
        g = [[True] * n for _ in range(n)]
        for i in range(n - 1, -1, -1):
            for j in range(i + 1, n):
                g[i][j] = s[i] == s[j] and g[i + 1][j - 1]
        ans = dfs(0)
        dfs.cache_clear() # nice, clear cache !!!
        return ans

###########

class Solution:
    def minCut(self, s: str) -> int:
        n = len(s)
        is_pal = [[False] * n for _ in range(n)]
        for i in range(n - 1, -1, -1):
            for j in range(i, n):
                is_pal[i][j] = s[i] == s[j] and (j - i < 3 or is_pal[i + 1][j - 1])
        min_cut = [i for i in range(n)]
        for i in range(n):
            for j in range(0, i + 1):
                if is_pal[j][i]:
                    min_cut[i] = min(min_cut[i], min_cut[j - 1] + 1) if j > 0 else 0

        return min_cut[-1]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python


```
class Solution:
    def minCut(self, s: str) -> int:
        pal = [[False for j in range(0, len(s))] for i in range(0, len(s))]
        dp = [len(s) for _ in range(0, len(s) + 1)]
        for i in range(0, len(s)):
            for j in range(0, i + 1):
                # also ok if j == 0 or j == i - 1
                if (s[i] == s[j]) and ((j + 1 > i - 1) or (pal[i - 1][j + 1])):
                    pal[i][j] = True
                    dp[i + 1] = min(dp[i + 1], dp[j] + 1) if j != 0 else 0
                    # if j != 0 to ensure from start to i is a palindrome
        return dp[-1]

###########

class Solution: # clear cache
    def minCut(self, s: str) -> int:
        @cache
        def dfs(i):
            if i >= n - 1:
                return 0
            ans = inf
            for j in range(i, n):
                if g[i][j]:
                    ans = min(ans, dfs(j + 1) + (j < n - 1))

            return ans

        n = len(s)
        g = [[True] * n for _ in range(n)]
        for i in range(n - 1, -1, -1):
            for j in range(i + 1, n):
                g[i][j] = s[i] == s[j] and g[i + 1][j - 1]
        ans = dfs(0)
        dfs.cache_clear() # nice, clear cache !!!
        return ans

###########

class Solution:
    def minCut(self, s: str) -> int:
        n = len(s)
        is_pal = [[False] * n for _ in range(n)]
        for i in range(n - 1, -1, -1):
            for j in range(i, n):
                is_pal[i][j] = s[i] == s[j] and (j - i < 3 or is_pal[i + 1][j - 1])
        min_cut = [i for i in range(n)]
        for i in range(n):
            for j in range(0, i + 1):
                if is_pal[j][i]:
                    min_cut[i] = min(min_cut[i], min_cut[j - 1] + 1) if j > 0 else 0

        return min_cut[-1]
```

#312 HARD

Burst Balloons

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming

Lists: Denny Zhang

Problem:

You are given n balloons, indexed from 0 to n − 1. Each balloon is painted with a number on it represented by an array nums. You are asked to burst all the balloons.

If you burst the ith balloon, you will get nums[i − 1] * nums[i] * nums[i + 1] coins. If i − 1 or i + 1 goes out of bounds of the array, then treat it as if there is a balloon with a 1 painted on it.

Return the maximum coins you can collect by bursting the balloons wisely.

Example 1:

```
Input: nums = [3,1,5,8]
Output: 167
Explanation:
nums = [3,1,5,8] --> [3,5,8] --> [3,8] --> [8] --> []
coins = 3*1*5 + 3*5*8 + 1*3*8 + 1*8*1 = 167
```

Example 2:

```
Input: nums = [1,5]
Output: 10
```

Constraints:

- n == nums.length
- 1 <= n <= 300
- 0 <= nums[i] <= 100

>> Brute Force [Dynamic Programming] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def maxCoins(self, nums):
        # Brute Force O(n! · n) — try every order to burst all balloons
        from itertools import permutations
        nums = [1] + nums + [1]
        best = 0
        # Only feasible for tiny n; actual brute is exponential
        def burst(balloons, coins):
            nonlocal best
            if not balloons:
                best = max(best, coins)
                return
            for i in range(len(balloons)):
                left = balloons[i-1] if i > 0 else 1
                right = balloons[i+1] if i+1 < len(balloons) else 1
                gained = left * balloons[i] * right
                burst(balloons[:i] + balloons[i+1:], coins + gained)
        interior = list(nums[1:-1])
        burst(interior, 0)
        return best
```

Python

Python

```python
# Define the function to compute maximum coins
def maxCoins(nums):
    # Add virtual balloons with value 1 at both boundaries
    n = len(nums)
    extended = [1] + nums + [1]
    # Initialize dp table with 0's; dimensions: (n+2) x (n+2)
    dp = [[0] + (n + 2) for _ in range(n + 2)]

    # Calculate dp values for intervals with length 2 to n+1
    for length in range(2, n + 2): # interval length from 2 to n+1
        for left in range(0, n + 2 - length):
            right = left + length
            # Iterate through all possible last burst balloons in the interval (left, right)
            for k in range(left + 1, right):
                # Coins when bursting balloon k last in the interval:
                # value from bursting: extended[left] * extended[k] * extended[right]
                # plus coins from subintervals [left, k] and [k, right]
                coins = extended[left] * extended[k] * extended[right] + dp[left][k] + dp[k][right]
                dp[left][right] = max(dp[left][right], coins)

    # The final answer is stored in dp[0][n+1] which represents bursting all real balloons.
    return dp[0][n+1]

# Example test cases
print(maxCoins([3,1,5,8])) # Expected output: 167
```

```python
print(maxCoins([1,5])) # Expected output: 10
```

Python

```python
class Solution:
    def maxCoins(self, nums: list[int]) -> int:
        n = len(nums)
        nums = [1] + nums + [1]

        @functools.lru_cache(None)
        def dp(i: int, j: int) -> int:
            """Returns maxCoins(nums[i:j])."""
            if i + 1 >= j:
                return 0
            return max(dp(i, k - 1) +
                       dp(k + 1, j) +
                       nums[i - 1] * nums[k] * nums[j + 1]
                       for k in range(i, j + 1))

        return dp(1, n)
```

Python


```python
class Solution:
    def maxCoins(self, nums: List[int]) -> int:
        n = len(nums)
        arr = [1] + nums + [1]
        f = [[0] + (n + 2) for _ in range(n + 2)]
        for i in range(n - 1, -1, -1):
            for j in range(i + 2, n + 2):
                for k in range(i + 1, j):
                    f[i][j] = max(f[i][j], f[i][k] + f[k][j] + arr[i] * arr[k] * arr[j])
        return f[0][-1]
```

Python

```python
class Solution:
    def maxCoins(self, nums: List[int]) -> int:
        nums = [1] + nums + [1]
        n = len(nums)
        dp = [[0] * n for _ in range(n)]
        for l in range(2, n):
            for i in range(n - l):
                j = i + l
                for k in range(i + 1, j):
                    dp[i][j] = max(
                        dp[i][j], dp[i][k] + dp[k][j] + nums[i] * nums[k] * nums[j]
                    )
        return dp[0][-1]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```python
# Define the function to compute maximum coins
def maxCoins(nums):
    # Add virtual balloons with value 1 at both boundaries
    n = len(nums)
    extended = [1] + nums + [1]
    # Initialize dp table with 0's; dimensions: (n+2) x (n+2)
    dp = [[0] + (n + 2) for _ in range(n + 2)]

    # Calculate dp values for intervals with length 2 to n+1
    for length in range(2, n + 2): # interval length from 2 to n+1
        for left in range(0, n + 2 - length):
            right = left + length
            # Iterate through all possible last burst balloons in the interval (left, right)
            for k in range(left + 1, right):
                # Coins when bursting balloon k last in the interval:
                # value from bursting: extended[left] * extended[k] * extended[right]
                # plus coins from subintervals [left, k] and [k, right]
                coins = extended[left] * extended[k] * extended[right] + dp[left][k] + dp[k][right]
                dp[left][right] = max(dp[left][right], coins)

    # The final answer is stored in dp[0][n+1] which represents bursting all real balloons.
    return dp[0][n+1]

# Example test cases
print(maxCoins([3,1,5,8])) # Expected output: 167
```

```python
print(maxCoins([1,5])) # Expected output: 10
```


#1000 HARD

Minimum Cost to Merge Stones

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Dynamic Programming

Lists: Denny Zhang

Problem:

There are n piles of stones arranged in a row. The ith pile has stones[i] stones.

A move consists of merging exactly k consecutive piles into one pile, and the cost of this move is equal to the total number of stones in these k piles.

Return the minimum cost to merge all piles of stones into one pile. If it is impossible, return -1.

Example 1:

```
Input: stones = [3,2,4,1], k = 2
Output: 20
Explanation: We start with [3, 2, 4, 1].
We merge [3, 2] for a cost of 5, and we are left with [5, 4, 1].
We merge [4, 1] for a cost of 5, and we are left with [5, 5].
We merge [5, 5] for a cost of 10, and we are left with [10].
The total cost was 20, and this is the minimum possible.
```

Example 2:

```
Input: stones = [3,2,4,1], k = 3
Output: -1
Explanation: After any merge operation, there are 2 piles left, and we can't merge anymore.
So the task is impossible.
```

Example 3:

```
Input: stones = [3,5,1,2,6], k = 3
Output: 25
Explanation: We start with [3, 5, 1, 2, 6].
We merge [5, 1, 2] for a cost of 8, and we are left with [3, 8, 6].
We merge [3, 8, 6] for a cost of 17, and we are left with [17].
The total cost was 25, and this is the minimum possible.
```

Constraints:

- n == stones.length
- 1 <= n <= 30
- 1 <= stones[i] <= 100
- 2 <= k <= 30

Python

Python


```python
class Solution:
    def mergeStones(self, stones: List[int], K: int) -> int:
        n = len(stones)
        if (n - 1) % (K - 1):
            return -1
        s = list(accumulate(stones, initial=0))
        f = [[[inf] * (K + 1) for _ in range(n + 1)] for _ in range(n + 1)]
        for i in range(1, n + 1):
            f[i][i][1] = 0
        for l in range(2, n + 1):
            for i in range(1, n - l + 2):
                j = i + l - 1
                for k in range(1, K + 1):
                    for h in range(i, j):
                        f[i][j][k] = min(f[i][j][k], f[i][h][1] + f[h + 1][j][k - 1])
                f[i][j][1] = f[i][j][K] + s[j] - s[i - 1]
        return f[1][n][1]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```python
# Python solution for merging stones with detailed comments.
class Solution:
    def mergeStones(self, stones: List[int], k: int) -> int:
        n = len(stones)
        # Check for merge feasibility.
        if (n - 1) % (k - 1) != 0:
            return -1

        # Precompute prefix sums for fast range sum queries.
        prefix = [0] * (n + 1)
        for i in range(n):
            prefix[i + 1] = prefix[i] + stones[i]

        # Initialize dp table, dp[i][j] represents minimal cost to merge stones[i...j].
        dp = [[0] * n for _ in range(n)]

        # Process all possible subarrays with lengths >= k.
        for length in range(k, n + 1):
            for i in range(n - length + 1):
                j = i + length - 1
                dp[i][j] = float('inf')
                # Try all valid partitions with step size (k-1).
                for mid in range(i, j, k - 1):
                    dp[i][j] = min(dp[i][j], dp[i][mid] + dp[mid+1][j])
                # If this segment can be merged into one pile, add cost.

                if (length - 1) % (k - 1) == 0:
                    dp[i][j] += prefix[j+1] - prefix[i]
        return dp[0][n-1]

# Example usage:
# solution = Solution()
# print(solution.mergeStones([3, 2, 4, 1], 2)) # Expected output: 20
```

Python

```python
class Solution:
    def mergeStones(self, stones: List[int], K: int) -> int:
        n = len(stones)
        if (n - 1) % (K - 1):
            return -1
        s = list(accumulate(stones, initial=0))
        f = [[[inf] * (K + 1) for _ in range(n + 1)] for _ in range(n + 1)]
        for i in range(1, n + 1):
            f[i][i][1] = 0
        for l in range(2, n + 1):
            for i in range(1, n - l + 2):
                j = i + l - 1
                for k in range(1, K + 1):
                    for h in range(i, j):
                        f[i][j][k] = min(f[i][j][k], f[i][h][1] + f[h + 1][j][k - 1])
                f[i][j][1] = f[i][j][K] + s[j] - s[i - 1]
        return f[1][n][1]
```



```python
class Solution:
    def mergeStones(self, stones: List[int], K: int) -> int:
        n = len(stones)
        if (n - 1) % (K - 1):
            return -1
        s = list(accumulate(stones, initial=0))
        f = [[[inf] * (K + 1) for _ in range(n + 1)] for _ in range(n + 1)]
        for i in range(1, n + 1):
            f[i][i][1] = 0
        for l in range(2, n + 1):
            for i in range(1, n - l + 2):
                j = i + l - 1
                for k in range(1, K + 1):
                    for h in range(i, j):
                        f[i][j][k] = min(f[i][j][k], f[i][h][1] + f[h + 1][j][k - 1])
                f[i][j][1] = f[i][j][K] + s[j] - s[i - 1]
        return f[1][n][1]
```

[*] Dynamic Programming — Pattern Solution (matched from community answers)

Python

```python
# Python solution for merging stones with detailed comments.
class Solution:
    def mergeStones(self, stones: List[int], k: int) -> int:
        n = len(stones)
        # Check for merge feasibility.
        if (n - 1) % (k - 1) != 0:
            return -1

        # Precompute prefix sums for fast range sum queries.
        prefix = [0] * (n + 1)
        for i in range(n):
            prefix[i + 1] = prefix[i] + stones[i]

        # Initialize dp table, dp[i][j] represents minimal cost to merge stones[i...j].
        dp = [[0] * n for _ in range(n)]

        # Process all possible subarrays with lengths >= k.
        for length in range(k, n + 1):
            for i in range(n - length + 1):
                j = i + length - 1
                dp[i][j] = float('inf')
                # Try all valid partitions with step size (k-1).
                for mid in range(i, j, k - 1):
                    dp[i][j] = min(dp[i][j], dp[i][mid] + dp[mid+1][j])
                # If this segment can be merged into one pile, add cost.

                if (length - 1) % (k - 1) == 0:
                    dp[i][j] += prefix[j+1] - prefix[i]
        return dp[0][n-1]

# Example usage:
# solution = Solution()
# print(solution.mergeStones([3, 2, 4, 1], 2)) # Expected output: 20
```

Quick Review — Dynamic Programming 24 questions · insights · complexity · solution

#70 Climbing Stairs [Easy]

Key Insights

- The problem can be solved using dynamic programming because the solution for a given step depends on the solutions of the previous two steps.
- The recurrence relation is: ways[i] = ways[i−1] + ways[i−2].
- Base cases: when there is 1 step, there is 1 way; when there are 2 steps, there are 2 ways.
- This is analogous to the Fibonacci sequence, shifting indices accordingly.

Space & Time

Time Complexity: O(n)

Space Complexity: O(n) (can be optimized to O(1) using constant space)

Solution

We use a dynamic programming approach to build a solution gradually. The idea is that to reach step i, you could have come from step i−1 (taking one step) or from step i−2 (taking two steps). Therefore, the total number of ways to reach step i is the sum of ways to reach step i−1 and i−2. We initialize dp[1] = 1 and dp[2] = 2, and iterate from 3 to n to fill dp[]. Each code solution below uses this approach with comments to explain the logic.

#121 Best Time to Buy and Sell Stock [Easy]

Key Insights

- Traverse the list of prices once while keeping track of the minimum price encountered so far.
- At each day, calculate the profit if you sold the stock on that day.
- Update the maximum profit when the current profit exceeds the previous maximum.
- The optimal solution makes a single pass through the array with constant space usage.

Space & Time

Time Complexity: O(n) - requires one pass through the prices array.

Space Complexity: O(1) - only a few extra variables are used.

Solution

We solve the problem using a simple linear scan. We maintain two key variables: one for the minimum price seen so far and another for the maximum profit calculated so far. As we iterate, we continuously update the minimum price if a lower price is found, and then compute the potential profit at the current price. If this profit exceeds the current maximum profit, we update the maximum profit. The final maximum profit is returned after iterating through all prices.

#5 Longest Palindromic Substring [Medium]

Key Insights

- Every palindrome is centered around a character (for odd-length) or a pair of characters (for even-length).
- Expand from each center until the substring is no longer a palindrome.
- Update the longest palindrome when a longer one is found during the expansion.
- The solution employs a two-pointer technique without extra storage for dynamic programming tables.

Space & Time

Time Complexity: O(n^2) in the worst case as we expand around each center.

Space Complexity: O(1), excluding the space required for the output substring.

Solution

The approach is based on expanding around potential centers of palindromes. For each character in the string, consider it as the center for an odd-length palindrome and the gap between it and the next character for an even-length palindrome. Using two pointers, expand outward until the characters differ. Track the longest palindrome found during these expansions. This method is efficient given the input constraints, and no extra data structures are needed besides variables to store indices or substrings.

#53 Maximum Subarray [Medium]

Key Insights

- Use Kadane's Algorithm to solve the problem in O(n) time.
- Keep track of a running sum (currentSum) and update it with the maximum between the current element and the sum including the current element.
- The maximum sum encountered during the iteration (maxSum) is the answer.
- A divide and conquer approach also exists, splitting the array and merging solutions, but Kadane's method is optimal for this problem.

Space & Time

Time Complexity: O(n)

Space Complexity: O(1)

Solution

The solution uses Kadane's Algorithm. Iterate over the array while maintaining two variables: currentSum, which tracks the maximum subarray sum ending at the current position, and maxSum, which stores the best sum seen so far. For each element, update currentSum to be either the current element or the sum of currentSum and the current element, whichever is higher. Update maxSum accordingly. This approach efficiently computes the maximum contiguous subarray sum in a single pass.

#55 Jump Game [Medium]

Key Insights

- The greedy approach is optimal for this problem.
- Track the furthest index reachable as you iterate through the array.
- If the current index exceeds the maximum reachable index, it is impossible to proceed further.
- The solution only requires constant extra space.

Space & Time

Time Complexity: O(n) where n is the length of the array.

Space Complexity: O(1)

Solution

The solution uses a greedy algorithm. As you iterate through each array element, you maintain a variable (maxReach) that represents the furthest index you can reach so far. For each index i, if i is greater than maxReach then it is not possible to reach index i and hence return false immediately. Otherwise, update maxReach to the maximum of its current value and i + nums[i]. If you can iterate through the array without i exceeding maxReach, then you can reach the end and return true.

#62 Unique Paths [Medium]

Key Insights

- The robot is constrained to only move right or down.
- Any valid path consists of exactly (m−1) down moves and (n−1) right moves.
- The problem can be solved using Dynamic Programming where each cell is the sum of the ways to reach the cell from the top and left.
- Alternatively, a combinatorial solution is possible by calculating the number of unique sequences of moves, equivalent to choosing (m−1) moves from (m+n−2) moves.

Space & Time

Time Complexity: O(m

n)

Space Complexity: O(m

n) - can be optimized to O(n) using a 1D DP array.

Solution

We can solve the problem using a dynamic programming approach. The idea is to create a 2D DP table where each cell (i, j) represents the number of ways to reach that cell. Since the robot can only come from above or from the left, we update the DP table as follows:

- Initialize dp[0][j] = 1 for all j (only one way to move right in the first row). - Initialize dp[i][0] = 1 for all i (only one way to move down in the first column). - For each other cell, compute dp[i][j] = dp[i−1][j] + dp[i][j−1]. The final answer will be at dp[m−1][n−1].

The combinatorial alternative computes C(m+n−2, m−1) or C(m+n−2, n−1) using basic factorial arithmetic, which is efficient for the given constraints.

#72 Edit Distance [Medium]

Key Insights

- This is a classic dynamic programming problem.
- Use a 2D DP table where dp[i][j] represents the minimum edit distance between word1[0...i−1] and word2[0...j−1].
- Base cases: converting an empty string to a string of length j requires j insertions, and vice versa.
- Transition: If characters match then dp[i][j] = dp[i−1][j−1], otherwise take minimum among insertion, deletion, and replacement operations and add 1.

Space & Time

Time Complexity: O(m

n), where m and n are the lengths of word1 and word2 respectively.

Space Complexity: O(m

n) due to the DP table used for storing computed distances.

Solution

We use a dynamic programming approach by constructing a 2D table dp of size (m+1) x (n+1), where m and n are the lengths of word1 and word2. Each dp[i][j] holds the minimum number of operations needed to convert the first i characters of word1 to the first j characters of word2. The algorithm includes the following steps:

- Initialize the base cases: dp[0][j] = j (converting an empty word1 to j characters of word2 requires j insertions) dp[i][0] = i (converting i characters of word1 to an empty word2 requires i deletions) - Iterate over each character in word1 and word2: If the current characters are equal, no additional operation is needed and dp[i][j] = dp[i−1][j−1]. Otherwise, choose the best operation among insertion (dp[i][j−1]), deletion (dp[i−1][j]), or substitution (dp[i−1][j−1]) and add 1. - The answer is found in dp[m][n].

This method reliably computes the edit distance using a systematic DP approach, ensuring that all subproblems are solved and reused efficiently.

#91 Decode Ways [Medium]

Key Insights

- A digit '0' cannot be mapped on its own; it must be part of "10" or "20".
- Use dynamic programming where dp[i] represents the number of ways to decode the substring s[0...i−1].
- For each index i, if the digit at position i−1 is non-zero, add dp[i−1] to dp[i].
- If the two-digit number formed by s[i−2] and s[i−1] is between 10 and 26, add dp[i−2] to dp[i].
- Initialize dp[0] as 1 (an empty string has one way to be decoded).

Space & Time

Time Complexity: O(n), where n is the length of the string.

Space Complexity: O(n), due to the dp array (which can be optimized to O(1)).

Solution

We solve the problem using a dynamic programming approach. We initialize a dp array where dp[i] indicates the number of ways to decode the substring s[0...i−1]. We set dp[0] = 1 since an empty string is trivially decodable. We loop through the string, and at each step:

- If s[i−1] is not '0', we add dp[i−1] to dp[i] because s[i−1] can be mapped to a valid letter. - If the substring s[i−2:i] (for i >= 2) forms a valid two-digit number between 10 and 26, we add dp[i−2] to dp[i]. This cumulative approach yields the total ways to decode the string, and if the string starts with '0' or contains invalid sequences, dp[n] will eventually be 0.

#152 Maximum Product Subarray [Medium]

Key Insights

- Maintain both the maximum and minimum product at the current index because a negative number can swap the max and min.
- A subarray must be contiguous.
- Use dynamic programming to update both the maximum product and the minimum product ending at each index.
- Resetting the product in case of a zero is crucial since the product becomes 0.

Space & Time

Time Complexity: O(n) - The solution iterates through the array once.

Space Complexity: O(1) - Constant space is used for tracking products.

Solution

The solution uses an iterative dynamic programming approach. At each index, compute the maximum product (maxEndingHere) and minimum product (minEndingHere) ending at that index. If the current element is negative, swap these two values, because multiplying a negative with a minimum product (possibly negative) might yield a larger product. After processing each element, update the global maximum. This method handles edge cases like zeros and negatives efficiently with constant additional space.

#198 House Robber [Medium]

Key Insights

- The problem can be solved using Dynamic Programming.
- For each house, decide whether to rob it or not.
- Recurrence relation: dp[i] = max(dp[i−1], dp[i−2] + currentHouseMoney)
- Use iterative computation to keep track of the optimal solutions without needing extra space for the entire dp array.

Space & Time

Time Complexity: O(n), where n is the number of houses.

Space Complexity: O(1), using only two variables to track the previous decisions.

Solution

We use a dynamic programming approach by iterating through each house in the array. At each step, we consider two scenarios: either we rob the current house (and add its money to the total from two houses ago) or we skip it (keeping the optimal solution up to the previous house). By using two variables to store these intermediate results, we achieve an optimized space complexity of O(1).

#256 Paint House [Medium]

Key Insights

- The choice for painting each house depends on the previous house's color choices.
- Use dynamic programming to accumulate the cost while ensuring adjacent houses do not have the same color.

- The state of each house can be represented by the minimum cost when the house is painted a specific color, computed using the minimal costs of the previous house from the other two colors.
- The problem only requires updating the cost matrix in-place for optimal space usage.

Space & Time

Time Complexity: O(n), where n is the number of houses.

Space Complexity: O(1) extra space if the input matrix is modified in place.

Solution

We use a dynamic programming approach with iterative in-place updating of the cost matrix. For each house starting from the second one, we update the cost for each color by adding the minimal cost of painting the previous house with either of the other two colors. This guarantees we never use the same color for adjacent houses while accumulating the minimal total cost. The final answer is the minimum value for the last house.

#309 Best Time to Buy and Sell Stock with Cooldown [Medium]

Key Insights

- Use Dynamic Programming to solve the problem.
- Maintain three states to represent:
- s0: the state where you are ready to buy (not holding any stock).
- s1: the state where you are holding a stock.
- s2: the cooldown state encountered right after selling a stock.
- Transition between states:
- Transition into s0 can come from staying in s0 or coming out from the cooldown state s2.
- To enter s1, you either continue holding your stock, or buy a stock using the profit available from s0.
- The only way to reach s2 is by selling the stock from s1.
- The final answer is determined by the maximum profit when you are not holding a stock (i.e., max(s0, s2)).

Space & Time

Time Complexity: O(n), where n is the number of days.

Space Complexity: O(1), since only a few variables are maintained for the states.

Solution

The solution involves updating three state variables for each day:

- s0 (ready to buy) is updated as the maximum of the previous s0 and the previous cooldown s2. - s1 (holding a stock) is updated as the maximum of the previous s1 or buying today with the profit from s0 minus the current price. - s2 (cooldown) is updated by selling the stock held in s1 (i.e., previous s1 plus the current price). At the end of the iteration, the maximum profit will be in state s0 (ready to buy) or s2 (cooldown), since remaining in s1 means you are holding a stock, which does not constitute a realized profit.

#377 Combination Sum IV [Medium]

Key Insights

- Use dynamic programming to count the number of valid combinations.
- The order in which numbers are picked matters, making this akin to permutation counting.
- Initialize a DP table where dp[i] represents the number of ways to form the sum i.
- dp[0] is initialized to 1 since there is one way to form a sum of 0 (by choosing nothing).
- For each sum i from 1 to target, iterate over nums and update dp[i] based on dp[i - num] when i >= num.
- For negative numbers, the problem becomes unbounded due to potential infinite combinations; hence a constraint on the combination length is required to make it well-defined.

Space & Time

Time Complexity: O(target * n), where n is the length of nums.

Space Complexity: O(target), for the dp array of size target + 1.

Solution

The problem is solved using a bottom-up dynamic programming approach. We create a dp array where each dp[i] represents the number of possible combinations to reach the sum i. We iterate from 1 up to target, and for each sum, we check every number in nums. If the number can contribute to the current sum (i - num >= 0), we add the number of ways to form the remainder (i - num) to dp[i]. This ensures that every possible permutation is counted. When negative numbers are allowed in the input, the potential for an infinite number of combinations arises unless we impose additional constraints, such as limiting the length of the combination.

#416 Partition Equal Subset Sum [Medium]

Key Insights

- The total sum of the array must be even, otherwise, partitioning into two equal subsets is impossible.
- The problem can be transformed into a "subset sum" problem where we need to check if there exists a subset of numbers that sums to half of the total sum.
- A dynamic programming approach can efficiently decide if such a subset exists.

Space & Time

Time Complexity: O(n * target), where target is half of the total sum.

Space Complexity: O(target), using a one-dimensional DP array.

Solution

First, compute the total sum of the array. If the total is odd, return false since an odd number cannot be equally partitioned. Next, set the target sum to half of the total. Use a dynamic programming array (dp) where dp[i] represents whether a subset with sum i is achievable. Initialize dp[0] as true because a subset sum of 0 is always possible. Then, for each number in the array, iterate through the dp array backwards from target to the number's value and update dp[i] to true if dp[i - num] is true. Ultimately, if dp[target] is true, the array can be partitioned into two subsets with equal sum.

#435 Non-overlapping Intervals [Medium]

Key Insights

- Sorting is the key: By sorting intervals by their end times, we can greedily choose the optimal set of non-overlapping intervals.
- Greedy approach: Always select the interval that ends the earliest, as it leaves maximum room for subsequent intervals.
- Counting removals: The answer is the total number of intervals minus the count of intervals selected by this greedy approach.
- Edge observation: Intervals that touch at the boundary are not overlapping, so the check uses ">=" instead of ">".

Space & Time

Time Complexity: O(n log n) due to sorting the intervals.

Space Complexity: O(n) if considering the storage for the sorted list (or O(1) if sorting in-place).

Solution

The solution involves first sorting the intervals by their end times. Initialize a variable to track the end of the last chosen interval. Iterate over the sorted intervals: if the current interval's start is greater than or equal to the last chosen interval's end, update the last end and include this interval in the non-overlapping set. Otherwise, count the interval as one that must be removed. The final answer is the removal count, which is computed as the total number of intervals minus the count of non-overlapping intervals selected.

#516 Longest Palindromic Subsequence [Medium]

Key Insights

- The problem can be solved efficiently using dynamic programming.
- A 2D DP table is used where dp[i][j] represents the length of the longest palindromic subsequence in the substring s[i...j].

- If the characters at the two ends of the substring match (s[i] == s[j]), they contribute to the palindrome length.
- Otherwise, the solution is the maximum sequence length by either excluding the left or the right character.
- The approach builds the solution from smaller substrings (length 1) to the entire string.

Space & Time

Time Complexity: O(n^2) where n is the length of the string.

Space Complexity: O(n^2) due to the usage of a 2D DP table.

Solution

We solve this problem using a bottom-up dynamic programming approach. We create a 2D table dp[][] such that dp[i][j] stores the longest palindromic subsequence length for the substring s[i...j]. For any single character, dp[i][i] is 1. For substrings of length greater than 1, if the characters at positions i and j (s[i] and s[j]) match, then dp[i][j] = dp[i+1][j-1] + 2. If they do not match, we take the maximum of dp[i+1][j] and dp[i][j-1]. Finally, the dp[0][n-1] holds the result for the entire string.

#576 Out of Boundary Paths [Medium]

Key Insights

- Use Dynamic Programming (DP) as the state is defined by (current row, current column, moves remaining).
- Transitions are made from a cell to its four adjacent cells.
- If the ball moves out of bounds, it contributes to one valid path.
- States can be memoized to avoid recomputation and optimize the recursive solution.

Space & Time

Time Complexity: O(maxMove * m * n), since we compute each state at most once.

Space Complexity: O(maxMove * m * n) for the memoization table.

Solution

We use a recursive dynamic programming (or memoization) approach. Define a function dp(i, j, movesLeft) that returns the number of ways to move out of bounds starting from cell (i, j) with movesLeft moves remaining. For each move from the current cell, check:

- If the cell is out of bounds, return 1 (base case). - If no moves are left (movesLeft == 0) and we are still inside, return 0. - Otherwise, use recursion to explore all four directions and sum the paths. Store results in a memoization table to avoid recomputation, then return the computed value modulo 10^9 + 7.

#1143 Longest Common Subsequence [Medium]

Key Insights

- The problem is well-suited for a dynamic programming approach.
- Use a 2D table where dp[i][j] stores the length of the longest common subsequence for text1[0...i-1] and text2[0...j-1].
- If the characters match at the current position, extend the subsequence by 1 from dp[i-1][j-1]; otherwise, take the maximum from dp[i-1][j] or dp[i][j-1].
- Filling the table in a bottom-up manner ensures that all subproblems are solved optimally.

Space & Time

Time Complexity: O(m * n), where m and n are the lengths of text1 and text2.

Space Complexity: O(m * n) for maintaining the 2D DP table. Space can be optimized to O(min(m, n)) if needed.

Solution

This solution uses a dynamic programming approach with a 2D array (dp) to record the lengths of common subsequences at each substring pair. We initialize a table of dimensions (m+1) x (n+1) with zeros, where m and n are the lengths of text1 and text2. We iterate over both strings, comparing characters; if they match, we set dp[i][j] = dp[i-1][j-1] + 1, otherwise we choose the maximum from dp[i-1][j] and dp[i][j-1]. The final answer is found at dp[m][n].

#10 Regular Expression Matching [Hard]

Key Insights

- Use recursion or dynamic programming to simulate matching as the '*' operator can affect the current and following characters.
- The '.' operator is a wildcard that matches any single character.
- When encountering a '*' in the pattern, consider both skipping the preceding element or consuming one character from the string if it matches.
- Using memoization helps to save intermediate results and efficiently process overlapping subproblems.

Space & Time

Time Complexity: O(m

n) where m = length(s) and n = length(p), due to exploring each subproblem combination.

Space Complexity: O(m

n) for storing the memoization table in the dynamic programming or recursive approach.

Solution

The solution uses a top-down recursive strategy with memoization to efficiently match the string against the pattern. For each character position in s and p, we check if they match directly or if the pattern character is '.' to allow any match. When encountering '*' in the pattern, two possibilities are explored: either the '*' stands for zero occurrences of the preceding element, or if there is a valid match, it accounts for one occurrence and we proceed further in the string while maintaining the same pattern index. The memoization aspect prevents repeated computations of the same (i, j) state in the recursion, resulting in an improved overall runtime.

#115 Distinct Subsequences [Hard]

Key Insights

- Use dynamic programming to build the solution by comparing characters of s and t.
- Define a 2D dp table where dp[i][j] represents the number of ways to form the first j characters of t using the first i characters of s.
- The recurrence relation:
- if s[i-1] equals t[j-1], then include both the cases of matching this character as well as skipping it: dp[i][j] = dp[i-1][j-1] + dp[i-1][j].
- If they do not match, then simply skip the current character of s: dp[i][j] = dp[i-1][j].
- Initialize dp[i][0] = 1 for all i since an empty t can always be formed.
- The final answer is dp[s.length][t.length].

Space & Time

Time Complexity: O(n * m), where n is the length of s and m is the length of t.

Space Complexity: O(n * m), which can be optimized to O(m) with careful row reuse.

Solution

We use a dynamic programming approach with a 2D table. The table has dimensions (len(s) + 1) x (len(t) + 1). Each cell dp[i][j] stores the number of distinct subsequences from the first i characters of s that match the first j characters of t.

Steps:

- Initialize dp[i][0] = 1 for every i, because an empty t is a subsequence of any prefix of s. - Iterate over each character of s (outer loop) and for each, iterate over t (inner loop). - For every character in s, if it equals the current character in t, update dp[i][j] as the sum of the ways by using or skipping this character. Otherwise, carry over the previous count. - Return dp[len(s)][len(t)] as the result.

#123 Best Time to Buy and Sell Stock III [Hard]

Key Insights

- The problem requires the optimal selection of at most two buy-sell transactions.

- One effective strategy is to split the problem into two parts:
- For each day i, calculate the maximum profit obtainable from day 0 to i with one transaction.
- For each day i, calculate the maximum profit obtainable from day i to the end with one transaction.
- Finally, for each day, the sum of these two profits (left and right) gives a candidate answer.
- Alternatively, a state-based dynamic programming approach can track 4 states corresponding to the two transactions.

Space & Time

Time Complexity: O(n) where n is the number of days

Space Complexity: O(n) due to extra arrays used for forward and backward computations (this can be optimized to O(1) with a state variable approach)

Solution

We use a two-pass dynamic programming approach. In the first pass (forward), we compute an array "leftProfits" where leftProfits[i] is the maximum profit from day 0 to day i with one transaction. We do this by keeping track of the minimum price observed so far and calculating the profit if sold on day i. In the second pass (backward), we compute an array "rightProfits" where rightProfits[i] is the maximum profit obtainable from day i to the last day with one transaction by tracking the maximum price seen in reverse order. Finally, for every index i, the sum leftProfits[i] + rightProfits[i] represents the maximum profit achievable by completing the first transaction on or before day i and the second transaction starting from day i. The answer is the maximum sum over all i.

#132 Palindrome Partitioning II [Hard]

Key Insights

- Precompute a 2D table (isPal) where isPal[i][j] indicates if the substring s[i...j] is a palindrome.
- Use dynamic programming: dp[i] represents the minimum cuts needed for the substring s[0...i].
- For every index i, iterate over all possible starting indices j. If s[j...i] is a palindrome, update dp[i] using dp[j-1] + 1 (or 0 if j is 0).
- Time complexity is dominated by the O(n^2) palindrome checking and state transitions.

Space & Time

Time Complexity: O(n^2)

Space Complexity: O(n^2) (due to the 2D table for palindrome checks, plus O(n) for the dp array)

Solution

The solution relies on dynamic programming with a two-step approach. First, precompute a 2D table (isPal) where each cell indicates if a substring is a palindrome. This is done in O(n^2) time by leveraging the fact that a substring s[j...i] is a palindrome if the characters at the ends match and the inner substring s[j+1...i-1] is also a palindrome (or if the length is less than 3). Then, use a dp array where dp[i] holds the minimum number of cuts needed for the substring s[0...i]. For each i, check every index j (0 ≤ j ≤ i) and if s[j...i] is a palindrome, update dp[i] to be the minimum of its current value and dp[j-1] + 1 (with a special case when j is 0, meaning no cut is needed). This two-phase approach efficiently breaks down the problem and computes the answer while avoiding redundant palindrome checks.

#312 Burst Balloons [Hard]

Key Insights

- Transform the problem by adding virtual balloons with value 1 at both ends to simplify edge handling.
- Use interval dynamic programming where dp[i][j] represents the maximum coins that can be obtained by bursting all balloons between indices i and j (exclusive).
- Realize that the order of bursting balloons matters; choose the last balloon to burst in each interval to break dependencies into independent subproblems.
- The solution involves iterating over all possible intervals and determining the optimal burst order.

Space & Time

Time Complexity: O(n^3) - Three nested loops: one for the interval length, one for the starting index, and one for the final balloon choice.

Space Complexity: O(n^2) - A 2D dynamic programming table is used to store the maximum coins for each interval.

Solution

The problem is solved using interval dynamic programming. First, extend the input array by adding 1 at both ends to handle edge cases seamlessly. Define a dp table where dp[i][j] represents the maximum coins that can be obtained by bursting all the balloons between i and j (not including i and j, as these are the virtual boundaries). For every interval [i, j], iterate through the possible choices for the last balloon k to burst in that interval. The value for bursting balloon k will be determined by the coins from bursting it last (i.e., nums[i] * nums[k] * nums[j]) plus the maximum coins from the two resulting subintervals: (i, k) and (k, j). By filling the dp table in increasing interval lengths, you build up to the solution for the overall interval that covers the entire (extended) array.

#1000 Minimum Cost to Merge Stones [Hard]

Key Insights

- You can only merge stones into one final pile if (n - 1) is divisible by (k - 1). If not, return -1.
- A dynamic programming approach is used where dp[i][j] represents the minimum cost to merge the subarray stones[i...j] into as few piles as possible.
- A prefix sum array is precomputed to quickly calculate the sum of stones in any subarray.
- When a segment can be merged into one pile (i.e. when (length - 1) % (k - 1) == 0), add the sum of stones in that segment to the dp value.

Space & Time

Time Complexity: O(n^3) due to the triple nested loop over segments and splits.

Space Complexity: O(n^2) from the dp table and O(n) for the prefix sum array.

Solution

The problem is solved using a range DP approach. We first verify that merging all piles into one is feasible by checking if (n - 1) % (k - 1) == 0. Next, a prefix sum array is computed to optimize range sum calculations. A 2D dp table dp[i][j] is then used to store the minimum cost to merge stones[i...j]. For each interval, we consider valid split points, updating dp[i][j] by combining sub-problems. If the current segment can form a single merged pile (determined by the condition on the interval length), the cost of merging that segment (the sum of stones) is added to dp[i][j].

#20 Stack — 25 questions · #20 of 21

#20 EASY

Valid Parentheses

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Stack

Lists: Grind 169

Problem:

Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.
3. Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: s = "()"

Output: true

Example 2:

Input: s = "()[]{"

Output: true

Example 3:

Input: s = "(]"

Output: false

Example 4:

Input: s = "([])"

Output: true

Example 5:

Input: s = "({)}"

Output: false

Constraints:

- 1 <= s.length <= 104
- s consists of parentheses only '()[]{}'.

Python

Python


```python
def isValid(s: str) -> bool:
    # Mapping from closing brackets to their corresponding opening brackets.
    paren_map = {')': '(', '}': '{', ']': '['}
    # Initialize an empty list to use as a stack.
    stack = []
    # Process each character in the string.
    for char in s:
        # If the character is a closing bracket.
        if char in paren_map:
            # Pop the top element from the stack if available, else use a placeholder.
            top_element = stack.pop() if stack else '#'
            # Check if the mapping does not match the top element.
            if paren_map[char] != top_element:
                return False
        else:
            # For opening brackets, push onto the stack.
            stack.append(char)
    # Return True if all brackets are matched (i.e., the stack is empty).
    return not stack

# Example usage
print(isValid("()[]{}")) # Expected output: True
```

Python

```python
class Solution:
    def isValid(self, s: str) -> bool:
        stack = []

        for c in s:
            if c == '(':
                stack.append(')')
            elif c == '{':
                stack.append('}')
            elif c == '[':
                stack.append(']')
            elif not stack or stack.pop() != c:
                return False

        return not stack
```

Python

```python
class Solution:
    def isValid(self, s: str) -> bool:
        stk = []
        d = {')', '}', ']', '{}'}
        for c in s:
            if c in '({[':
                stk.append(c)
            elif not stk or stk.pop() + c not in d:
                return False
        return not stk
```

Python

```python
class Solution:
    def isValid(self, s: str) -> bool:
        stk = []
        d = {'(': ')', '[': ']', '{': '}'}
        for c in s:
            if c in '({[':
                stk.append(c)
            elif not stk or stk.pop() + c not in d:
                return False
        return not stk
```

[*] Stack — Pattern Solution (matched from community answers)

Python

```python
def isValid(s: str) -> bool:
    # Mapping from closing brackets to their corresponding opening brackets.
    paren_map = {')': '(', '}': '{', ']': '['}
    # Initialize an empty list to use as a stack.
    stack = []
    # Process each character in the string.
    for char in s:
        # If the character is a closing bracket.
        if char in paren_map:
            # Pop the top element from the stack if available, else use a placeholder.
            top_element = stack.pop() if stack else '#'
            # Check if the mapping does not match the top element.
            if paren_map[char] != top_element:
                return False
        else:
            # For opening brackets, push onto the stack.
            stack.append(char)
    # Return True if all brackets are matched (i.e., the stack is empty).
    return not stack

# Example usage
print(isValid("()[]{}")) # Expected output: True
```

#232 EASY

Implement Queue using Stacks

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Stack · Design · Queue

Lists: Grind 169

Problem:

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).

Implement the MyQueue class:

- void push(int x) Pushes element x to the back of the queue.
- int pop() Removes the element from the front of the queue and returns it.
- int peek() Returns the element at the front of the queue.
- boolean empty() Returns true if the queue is empty, false otherwise.

Notes:

- You must use only standard operations of a stack, which means only push to top, peek/pop from top, size, and is empty operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

Example 1:

```
Input
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
Output
[null, null, null, 1, 1, false]

Explanation
MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false
```

Constraints:

- 1 <= x <= 9
- At most 100 calls will be made to push, pop, peek, and empty.
- All the calls to pop and peek are valid.

Follow-up: Can you implement the queue such that each operation is amortized O(1) time complexity? In other words, performing n operations will take overall O(n) time even if one of those operations may take longer.

Python

Python


```python
class MyQueue:
    def __init__(self):
        # Stack for enqueue operations.
        self.input_stack = []
        # Stack for dequeue operations.
        self.output_stack = []

    def push(self, x: int) -> None:
        # Always push new elements onto the input stack.
        self.input_stack.append(x)

    def pop(self) -> int:
        # Ensure the output stack has the current queue order.
        if not self.output_stack:
            while self.input_stack:
                # Move all elements from input_stack to output_stack.
                self.output_stack.append(self.input_stack.pop())
        # The top element of output_stack is the front of the queue.
        return self.output_stack.pop()

    def peek(self) -> int:
        # Ensure the output stack has the current queue order.
        if not self.output_stack:
            while self.input_stack:
                # Move all elements from input_stack to output_stack.

                self.output_stack.append(self.input_stack.pop())
        # Peek the top of output_stack without popping.
        return self.output_stack[-1]

    def empty(self) -> bool:
        # The queue is empty if both stacks are empty.
        return not self.input_stack and not self.output_stack
```

Python

```python
class MyQueue:
    def __init__(self):
        self.input = []
        self.output = []

    def push(self, x: int) -> None:
        self.input.append(x)

    def pop(self) -> int:
        self.peek()
        return self.output.pop()

    def peek(self) -> int:
        if not self.output:
            while self.input:
                self.output.append(self.input.pop())
        return self.output[-1]

    def empty(self) -> bool:
        return not self.input and not self.output
```

Python


```python
class MyQueue:
    def __init__(self):
        self.stk1 = []
        self.stk2 = []

    def push(self, x: int) -> None:
        self.stk1.append(x)

    def pop(self) -> int:
        self.move()
        return self.stk2.pop()

    def peek(self) -> int:
        self.move()
        return self.stk2[-1]

    def empty(self) -> bool:
        return not self.stk1 and not self.stk2

    def move(self):
        if not self.stk2:
            while self.stk1:
                self.stk2.append(self.stk1.pop())

# Your MyQueue object will be instantiated and called as such:
# obj = MyQueue()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.peek()
# param_4 = obj.empty()
```

Python

```python
class MyQueue:
    def __init__(self):
        self.stk1 = []
        self.stk2 = [] # reversed order

    def push(self, x: int) -> None:
        self.stk1.append(x)

    def pop(self) -> int:
        self.move()
        return self.stk2.pop()

    def peek(self) -> int:
        self.move()
        return self.stk2[-1]

    def empty(self) -> bool:
        return not self.stk1 and not self.stk2

    def move(self):
        if not self.stk2: # only when skt2 is empty
            while self.stk1:
                self.stk2.append(self.stk1.pop())
```



```python
# Your MyQueue object will be instantiated and called as such:
# obj = MyQueue()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.peek()
# param_4 = obj.empty()

##########

class MyQueue:

    def __init__(self):
        self.sk = []
        self.rsk = [] # reversed

    def push(self, x: int) -> None:
        self.sk.append(x);


    def pop(self) -> int:
        self.peek()
        return self.rsk.pop()

    def peek(self) -> int:
        if self.rsk:

            return self.rsk[-1]
        else:
            while self.sk:
                self.rsk.append(self.sk.pop())
            return self.rsk[-1]


    def empty(self) -> bool:

        return not (self.sk or self.rsk)



# Your MyQueue object will be instantiated and called as such:
# obj = MyQueue()
# obj.push(x)
# param_2 = obj.pop()
# param_3 = obj.peek()
# param_4 = obj.empty()
```

[*] Stack — Pattern Solution (matched from community answers)

Python


```python
class MyQueue:
    def __init__(self):
        # Stack for enqueue operations.
        self.input_stack = []
        # Stack for dequeue operations.
        self.output_stack = []

    def push(self, x: int) -> None:
        # Always push new elements onto the input stack.
        self.input_stack.append(x)

    def pop(self) -> int:
        # Ensure the output stack has the current queue order.
        if not self.output_stack:
            while self.input_stack:
                # Move all elements from input_stack to output_stack.
                self.output_stack.append(self.input_stack.pop())
        # The top element of output_stack is the front of the queue.
        return self.output_stack.pop()

    def peek(self) -> int:
        # Ensure the output stack has the current queue order.
        if not self.output_stack:
            while self.input_stack:
                # Move all elements from input_stack to output_stack.

                self.output_stack.append(self.input_stack.pop())
        # Peek the top of output_stack without popping.
        return self.output_stack[-1]

    def empty(self) -> bool:
        # The queue is empty if both stacks are empty.
        return not self.input_stack and not self.output_stack
```

#844 EASY

Backspace String Compare

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Two Pointers · String · Stack · Simulation

Lists: Grind 169

Problem:

Given two strings s and t, return true if they are equal when both are typed into empty text editors. '#' means a backspace character.

Note that after backspacing an empty text, the text will continue empty.

Example 1:

```
Input: s = "ab#c", t = "ad#c"
Output: true
Explanation: Both s and t become "ac".
```

Example 2:

```
Input: s = "ab##", t = "c#d#"
Output: true
```

```
Explanation: Both s and t become "".
```

Example 3:

```
Input: s = "a#c", t = "b"
Output: false
Explanation: s becomes "c" while t becomes "b".
```

Constraints:

- 1 <= s.length, t.length <= 200
- s and t only contain lowercase letters and '#' characters.

Follow up: Can you solve it in O(n) time and O(1) space?

Python

Python

```python
# Function to compare two strings with backspace processing
def backspaceCompare(s: str, t: str) -> bool:
    # Pointers for both strings, starting at the end
    i, j = len(s) - 1, len(t) - 1

    # Process both strings until all characters are compared
    while i >= 0 or j >= 0:
        # Process string s: skip backspaced characters
        skip = 0
        while i >= 0:
            if s[i] == '#':
                skip += 1
                i -= 1
            elif skip > 0:
                skip -= 1
                i -= 1
            else:
                break

        # Process string t: skip backspaced characters
        skip = 0
        while j >= 0:
            if t[j] == '#':
                skip += 1
                j -= 1
```



```
            elif skip > 0:
                skip -= 1
                j -= 1
            else:
                break

        # Compare the current valid characters from s and t
        if i >= 0 and j >= 0:
            if s[i] != t[j]:
                return False
        elif i >= 0 or j >= 0:
            return False

        i -= 1
        j -= 1

    return True

# Example test cases
print(backspaceCompare("ab#c", "ad#c")) # Expected output: True
print(backspaceCompare("ab##", "c#d#")) # Expected output: True
print(backspaceCompare("a#c", "b")) # Expected output: False
```

Python

```python
class Solution:
    def backspaceCompare(self, s: str, t: str) -> bool:
        def backspace(s: str) -> str:
            stack = []
            for c in s:
                if c != '#':
                    stack.append(c)
                elif stack:
                    stack.pop()
            return stack

        return backspace(s) == backspace(t)
```

Python


```python
class Solution:
    def backspaceCompare(self, s: str, t: str) -> bool:
        i = len(s) - 1 # s' index
        j = len(t) - 1 # t's index

        while True:
            # Delete characters of s if needed.
            backspace = 0
            while i >= 0 and (s[i] == '#' or backspace > 0):
                backspace += 1 if s[i] == '#' else -1
                i -= 1
            # Delete characters of t if needed.
            backspace = 0
            while j >= 0 and (t[j] == '#' or backspace > 0):
                backspace += 1 if t[j] == '#' else -1
                j -= 1
            if i >= 0 and j >= 0 and s[i] == t[j]:
                i -= 1
                j -= 1
            else:
                break

        return i == -1 and j == -1
```

Python

```python
class Solution:
    def backspaceCompare(self, s: str, t: str) -> bool:
        i, j, skip1, skip2 = len(s) - 1, len(t) - 1, 0, 0
        while i >= 0 or j >= 0:
            while i >= 0:
                if s[i] == '#':
                    skip1 += 1
                    i -= 1
                elif skip1:
                    skip1 -= 1
                    i -= 1
                else:
                    break
            while j >= 0:
                if t[j] == '#':
                    skip2 += 1
                    j -= 1
                elif skip2:
                    skip2 -= 1
                    j -= 1
                else:
                    break
            if i >= 0 and j >= 0:
                if s[i] != t[j]:
                    return False

            elif i >= 0 or j >= 0:
                return False
            i, j = i - 1, j - 1
        return True
```

Python


```python
class Solution:
    def backspaceCompare(self, s: str, t: str) -> bool:
        i, j, skip1, skip2 = len(s) - 1, len(t) - 1, 0, 0
        while i >= 0 or j >= 0:
            while i >= 0:
                if s[i] == '#':
                    skip1 += 1
                    i -= 1
                elif skip1:
                    skip1 -= 1
                    i -= 1
                else:
                    break
            while j >= 0:
                if t[j] == '#':
                    skip2 += 1
                    j -= 1
                elif skip2:
                    skip2 -= 1
                    j -= 1
                else:
                    break
            if i >= 0 and j >= 0:
                if s[i] != t[j]:
                    return False

            elif i >= 0 or j >= 0:
                return False
            i, j = i - 1, j - 1
        return True
```

[*] Stack — Pattern Solution (matched from community answers)

Python

```python
class Solution:
    def backspaceCompare(self, s: str, t: str) -> bool:
        def backspace(s: str) -> str:
            stack = []
            for c in s:
                if c != '#':
                    stack.append(c)
                elif stack:
                    stack.pop()
            return stack

        return backspace(s) == backspace(t)
```

#143 MEDIUM

Reorder List

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Linked List · Two Pointers · Stack · Recursion

Lists: Grind 169

Problem:

You are given the head of a singly linked-list. The list can be represented as:

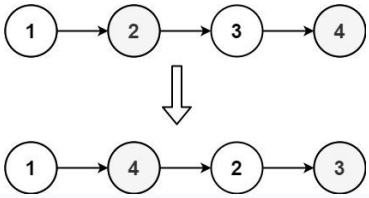
```
L0 → L1 → … → Ln - 1 → Ln
```

Reorder the list to be on the following form:

```
L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …
```

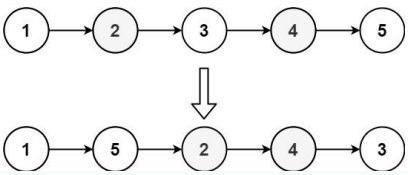
You may not modify the values in the list's nodes. Only nodes themselves may be changed.

Example 1:




```
Input: head = [1,2,3,4]
Output: [1,4,2,3]
```

Example 2:




```
Input: head = [1,2,3,4,5]
Output: [1,5,2,4,3]
```

Constraints:

- The number of nodes in the list is in the range [1, 5 * 104].
- 1 <= Node.val <= 1000

Python

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def reorderList(self, head: ListNode) -> None:
        if not head or not head.next:
            return

        # Step 1: Find the middle of the linked list using slow and fast pointers.
        slow, fast = head, head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # Step 2: Reverse the second half of the list.
        prev, curr = None, slow.next
        slow.next = None # Split the list into two halves.
        while curr:
            next_temp = curr.next # Save next node.
            curr.next = prev # Reverse current pointer.
            prev = curr # Move prev to current.
            curr = next_temp # Move to the next node.

        # Step 3: Merge the two halves, alternating nodes.
        first, second = head, prev
        while second:
            temp1 = first.next # Save next node from first half.
            temp2 = second.next # Save next node from reversed second half.
            first.next = second # Link first node to second list node.
            second.next = temp1 # Link second node to the next node in first half.
            first = temp1 # Move pointer in first half.
            second = temp2 # Move pointer in second half.
```

Python

```python
class Solution:
    def reorderList(self, head: ListNode) -> None:
        def findMid(head: ListNode):
            prev = None
            slow = head
            fast = head

            while fast and fast.next:
                prev = slow
                slow = slow.next
                fast = fast.next.next
            prev.next = None

            return slow

        def reverse(head: ListNode) -> ListNode:
            prev = None
            curr = head

            while curr:
                next = curr.next
                curr.next = prev
                prev = curr
                curr = next

            return prev

        def merge(l1: ListNode, l2: ListNode) -> None:
            while l2:
                next = l1.next
                l1.next = l2
                l1 = l2
                l2 = next

        if not head or not head.next:
            return

        mid = findMid(head)
        reversed = reverse(mid)
        merge(head, reversed)
```

Python


```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def reorderList(self, head: Optional[ListNode]) -> None:
        fast = slow = head
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next

        cur = slow.next
        slow.next = None

        pre = None
        while cur:
            t = cur.next
            cur.next = pre
            pre, cur = cur, t
        cur = head

        while pre:
            t = pre.next
            pre.next = cur.next

            cur.next = pre
            cur, pre = pre.next, t
```

Python


```python
# Definition for singly-linked list.
# class ListNode:
# def __init__(self, val=0, next=None):
# self.val = val
# self.next = next
class Solution:
    def reorderList(self, head: Optional[ListNode]) -> None:
        fast = slow = head
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next

        cur = slow.next
        slow.next = None

        pre = None
        while cur:
            t = cur.next
            cur.next = pre
            pre, cur = cur, t
        cur = head

        while pre:
            t = pre.next
            pre.next = cur.next

            cur.next = pre
            cur, pre = pre.next, t
```


[*] Stack — Pattern Solution (stored optimal)

Python

```python
# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def reorderList(self, head: ListNode) -> None:
        if not head or not head.next:
            return

        # Step 1: Find the middle of the linked list using slow and fast pointers.
        slow, fast = head, head
        while fast and fast.next:
            slow = slow.next
            fast = fast.next.next

        # Step 2: Reverse the second half of the list.
        prev, curr = None, slow.next
        slow.next = None # Split the list into two halves.
        while curr:
            next_temp = curr.next # Save next node.
            curr.next = prev # Reverse current pointer.
            prev = curr # Move prev to current.
            curr = next_temp # Move to the next node.

        # Step 3: Merge the two halves, alternating nodes.
        first, second = head, prev
        while second:
            temp1 = first.next # Save next node from first half.
            temp2 = second.next # Save next node from reversed second half.
            first.next = second # Link first node to second list node.
            second.next = temp1 # Link second node to the next node in first half.
            first = temp1 # Move pointer in first half.
            second = temp2 # Move pointer in second half.
```

#150 MEDIUM

Evaluate Reverse Polish Notation

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Math · Stack

Lists: Grind 169

Problem:

You are given an array of strings tokens that represents an arithmetic expression in a Reverse Polish Notation.

Evaluate the expression. Return an integer that represents the value of the expression.

Note that:

- The valid operators are '+', '-', '*', and '/'.

- Each operand may be an integer or another expression.
- The division between two integers always truncates toward zero.
- There will not be any division by zero.
- The input represents a valid arithmetic expression in a reverse polish notation.
- The answer and all the intermediate calculations can be represented in a 32-bit integer.

Example 1:

```
Input: tokens = ["2","1","+","3","*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9
```

Example 2:

```
Input: tokens = ["4","13","5","/","+"]
Output: 6
Explanation: (4 + (13 / 5)) = 6
```

Example 3:

```
Input: tokens = ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
Output: 22
Explanation: ((10 * (6 / ((9 + 3) * -11))) + 17) + 5
= ((10 * (6 / (12 * -11))) + 17) + 5
= ((10 * (6 / -132)) + 17) + 5
= ((10 * 0) + 17) + 5
= (0 + 17) + 5
= 17 + 5
= 22
```

Constraints:

- 1 <= tokens.length <= 104
- tokens[i] is either an operator: "+", "-", "*", or "/", or an integer in the range [-200, 200].

>> Brute Force [Stack] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def evalRPN(self, tokens):
        # Brute Force (stack is already optimal) - simulate token by token
        stack = []
        ops = {'+': lambda a,b: a+b, '-': lambda a,b: a-b,
               '*': lambda a,b: a*b, '/': lambda a,b: int(a/b)}
        for tok in tokens:
            if tok in ops:
                b, a = stack.pop(), stack.pop()
                stack.append(ops[tok](a, b))
            else:
                stack.append(int(tok))
        return stack[0]
```

Python

Python


```python
def evalRPN(tokens):
    # Initialize a stack to hold numbers
    stack = []
    # Loop through each token in the input list
    for token in tokens:
        if token in {"+", "-", "*", "/"}:
            # Pop the two topmost values; note the order is important
            b = stack.pop()
            a = stack.pop()
            # Apply the operation based on the token
            if token == "+":
                result = a + b
            elif token == "-":
                result = a - b
            elif token == "*":
                result = a * b
            else:
                # Use int division to truncate towards zero
                result = int(a / b)
            # Push the result back onto the stack
            stack.append(result)
        else:
            # Token is a number; push its integer value onto the stack
            stack.append(int(token))
    # The final result is the only element left in the stack

    return stack.pop()

# Example usage:
# tokens = ["2", "1", "+", "3", "*"]
# print(evalRPN(tokens)) # Output: 9
```

Python

```python
class Solution:
    def evalRPN(self, tokens: list[str]) -> int:
        stack = []
        op = {
            '+': lambda a, b: a + b,
            '-': lambda a, b: a - b,
            '*': lambda a, b: a * b,
            '/': lambda a, b: int(a / b),
        }

        for token in tokens:
          if token in op:
            b = stack.pop()
            a = stack.pop()
            stack.append(op[token](a, b))
          else:
            stack.append(int(token))

        return stack.pop()
```

Python


```python
import operator

class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        opt = {
            "+": operator.add,
            "-": operator.sub,
            "*": operator.mul,
            "/": operator.truediv,
        }
        s = []
        for token in tokens:
            if token in opt:
                s.append(int(opt[token](s.pop(-2), s.pop(-1))))
            else:
                s.append(int(token))
        return s[0]
```

Python

```python
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        nums = []
        for t in tokens:
            if len(t) > 1 or t.isdigit():
                nums.append(int(t))
            else:
                if t == "+":
                    nums[-2] += nums[-1]
                elif t == "-":
                    nums[-2] -= nums[-1]
                elif t == "*":
                    nums[-2] *= nums[-1]
                else:
                    nums[-2] = int(nums[-2] / nums[-1])
                nums.pop()
        return nums[0]
```

Python


```python
class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        nums = []
        for t in tokens:
            if len(t) > 1 or t.isdigit():
                nums.append(int(t))
            else:
                if t == "+":
                    nums[-2] += nums[-1]
                elif t == "-":
                    nums[-2] -= nums[-1]
                elif t == "*":
                    nums[-2] *= nums[-1]
                else:
                    nums[-2] = int(nums[-2] / nums[-1])
                nums.pop()
        return nums[0]
```

[*] Stack — Pattern Solution (matched from community answers)

Python

```python
def evalRPN(tokens):
    # Initialize a stack to hold numbers
    stack = []
    # Loop through each token in the input list
    for token in tokens:
        if token in ("+", "-", "*", "/"):
            # Pop the two topmost values; note the order is important
            b = stack.pop()
            a = stack.pop()
            # Apply the operation based on the token
            if token == "+":
                result = a + b
            elif token == "-":
                result = a - b
            elif token == "*":
                result = a * b
            else:
                # Use int division to truncate towards zero
                result = int(a / b)
            # Push the result back onto the stack
            stack.append(result)
        else:
            # Token is a number, push its integer value onto the stack
            stack.append(int(token))
    # The final result is the only element left in the stack
    return stack.pop()

# Example usage:
# tokens = ["2", "1", "+", "3", "*"]
# print(evalRPN(tokens)) # Output: 9
```

#155 MEDIUM

Min Stack

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Stack · Design

Lists: Grind 169

Problem:

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

- MinStack() initializes the stack object.
- void push(int val) pushes the element val onto the stack.
- void pop() removes the element on the top of the stack.
- int top() gets the top element of the stack.
- int getMin() retrieves the minimum element in the stack.

You must implement a solution with O(1) time complexity for each function.

Example 1:


```
Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output
[null,null,null,null,-3,null,0,-2]

Explanation
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top(); // return 0
minStack.getMin(); // return -2
```

Constraints:

- -231 <= val <= 231 - 1
- Methods pop, top and getMin operations will always be called on non-empty stacks.
- At most 3 * 104 calls will be made to push, pop, top, and getMin.

Python

Python

```python
# Python implementation of MinStack with detailed comments

class MinStack:
    def __init__(self):
        # Main stack to store all values
        self.stack = []
        # Auxiliary stack to store minimum values
        self.min_stack = []

    def push(self, val: int) -> None:
        # Push value onto main stack
        self.stack.append(val)
        # If min_stack is empty or the new value is less than or equal to current minimum, push it onto min_stack
        if not self.min_stack or val <= self.min_stack[-1]:
            self.min_stack.append(val)

    def pop(self) -> None:
        # Pop the top value from main stack
        popped_value = self.stack.pop()
        # If the popped value is the current minimum, pop it from min_stack as well
        if popped_value == self.min_stack[-1]:
            self.min_stack.pop()

    def top(self) -> int:
        # Return the top value from main stack without removing it
```



```python
        return self.stack[-1]

    def getMin(self) -> int:
        # Return the minimum value from min_stack
        return self.min_stack[-1]

# Example usage:
# minStack = MinStack()
# minStack.push(-2)
# minStack.push(0)
# minStack.push(-3)
# print(minStack.getMin()) # -> -3
# minStack.pop()
# print(minStack.top()) # -> 0
# print(minStack.getMin()) # -> -2
```

Python

```python
class MinStack:
  def __init__(self):
    self.stack = []

  def push(self, x: int) -> None:
    mm = x if not self.stack else min(self.stack[-1][1], x)
    self.stack.append((x, mm))

  def pop(self) -> None:
    self.stack.pop()

  def top(self) -> int:
    return self.stack[-1][0]

  def getMin(self) -> int:
    return self.stack[-1][1]
```

Python


```python
class MinStack:
    def __init__(self):
        self.stk1 = []
        self.stk2 = [inf]

    def push(self, val: int) -> None:
        self.stk1.append(val)
        self.stk2.append(min(val, self.stk2[-1]))

    def pop(self) -> None:
        self.stk1.pop()
        self.stk2.pop()

    def top(self) -> int:
        return self.stk1[-1]

    def getMin(self) -> int:
        return self.stk2[-1]


# Your MinStack object will be instantiated and called as such:
# obj = MinStack()
# obj.push(val)
# obj.pop()
# param_3 = obj.top()

# param_4 = obj.getMin()
```

Python

```python
class MinStack:
    def __init__(self):
        self.sk = []
        self.minsk = [inf] # trick, avoid later empty-check for min-stack

    def push(self, x: int) -> None:
        self.sk.append(x)
        self.minsk.append(min(x, self.minsk[-1]))

    def pop(self) -> None:
        if not self.sk:
            return
        self.sk.pop()
        self.minsk.pop()

    def top(self) -> int:
        return self.sk[-1]

    def getMin(self) -> int:
        return self.minsk[-1]


# Your MinStack object will be instantiated and called as such:
# obj = MinStack()
# obj.push(x)
```



```python
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.getMin()

##########

class MinStack:

    def __init__(self):
        self.sk = []
        self.minsk = []

    def push(self, val: int) -> None:
        self.sk.append(val)

        # empty check
        self.minsk.append(val if not self.minsk else min(val, self.minsk[-1]))

    def pop(self) -> None:
        if not self.sk:
            return

        self.sk.pop()
        self.minsk.pop()

    def top(self) -> int:
        return self.sk[-1]

    def getMin(self) -> int:
        return self.minsk[-1]

# optimize above, using only 1 stack, with stack element as tuple: (val, its associated min)
class MinStack:

    def __init__(self):
        self._stack = []

    def push(self, x: int) -> None:
        cur_min = self.getMin()
        if x < cur_min:
            cur_min = x
        self._stack.append((x, cur_min))

    def pop(self) -> None:
        self._stack.pop()

    def top(self) -> int:
        if not self._stack:
            return None
        else:

            return self._stack[-1][0]

    def getMin(self) -> int:
        if not self._stack:
            return float('inf')
```

[*] Stack — Pattern Solution (matched from community answers)

Python

```python
# Python implementation of MinStack with detailed comments

class MinStack:
    def __init__(self):
        # Main stack to store all values
        self.stack = []
        # Auxiliary stack to store minimum values
        self.min_stack = []

    def push(self, val: int) -> None:
        # Push value onto main stack
        self.stack.append(val)
        # If min_stack is empty or the new value is less than or equal to current minimum, push it onto min_stack
        if not self.min_stack or val <= self.min_stack[-1]:
            self.min_stack.append(val)

    def pop(self) -> None:
        # Pop the top value from main stack
        popped_value = self.stack.pop()
        # If the popped value is the current minimum, pop it from min_stack as well
        if popped_value == self.min_stack[-1]:
            self.min_stack.pop()

    def top(self) -> int:
        # Return the top value from main stack without removing it

        return self.stack[-1]

    def getMin(self) -> int:
        # Return the minimum value from min_stack
        return self.min_stack[-1]

# Example usage:
# minStack = MinStack()
# minStack.push(-2)
# minStack.push(0)
# minStack.push(-3)
# print(minStack.getMin()) # -> -3
# minStack.pop()
# print(minStack.top()) # -> 0
# print(minStack.getMin()) # -> -2
```

#173 MEDIUM

Binary Search Tree Iterator

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Stack · Tree · Design · Binary Search Tree · Iterator

Lists: Denny Zhang

Problem:

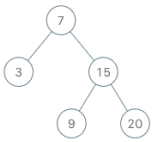
Implement the BSTIterator class that represents an iterator over the in-order traversal of a binary search tree (BST):

- BSTIterator(TreeNode root) Initializes an object of the BSTIterator class. The root of the BST is given as part of the constructor. The pointer should be initialized to a non-existent number smaller than any element in the BST.
- boolean hasNext() Returns true if there exists a number in the traversal to the right of the pointer, otherwise returns false.
- int next() Moves the pointer to the right, then returns the number at the pointer.

Notice that by initializing the pointer to a non-existent smallest number, the first call to next() will return the smallest element in the BST.

You may assume that next() calls will always be valid. That is, there will be at least a next number in the in-order traversal when next() is called.

Example 1:




```
Input
["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "hasNext", "next", "hasNext"]
[[[7, 3, 15, null, null, 9, 20]], [], [], [], [], [], [], [], [], []]
Output
[null, 3, 7, true, 9, true, 15, true, 20, false]

Explanation
BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]);
bSTIterator.next(); // return 3
bSTIterator.next(); // return 7
bSTIterator.hasNext(); // return True
bSTIterator.next(); // return 9
bSTIterator.hasNext(); // return True
bSTIterator.next(); // return 15
bSTIterator.hasNext(); // return True
bSTIterator.next(); // return 20
bSTIterator.hasNext(); // return False
```

Constraints:

- The number of nodes in the tree is in the range [1, 105].
- 0 <= Node.val <= 106
- At most 105 calls will be made to hasNext, and next.

Follow up:

- Could you implement next() and hasNext() to run in average O(1) time and use O(h) memory, where h is the height of the tree?

Python

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class BSTIterator:
    def __init__(self, root):
        # Initialize an empty stack to simulate in-order traversal
        self.stack = []
        # Push all left nodes starting from the root
        self._push_left_branch(root)

    def _push_left_branch(self, node):
        # Push node and all its left children to the stack
        while node:
            self.stack.append(node)
            node = node.left

    def next(self):
        # Pop the top node which represents the next smallest element
        node = self.stack.pop()
        # If the node has a right child, push its left branch onto the stack
        if node.right:
            self._push_left_branch(node.right)
        # Return the node's value
        return node.val

    def hasNext(self):
        # The iterator has a next element if the stack is not empty
        return len(self.stack) > 0
```

Python


```python
class BSTIterator:
    def __init__(self, root: TreeNode | None):
        self.i = 0
        self.vals = []
        self._inorder(root)

    def next(self) -> int:
        self.i += 1
        return self.vals[self.i - 1]

    def hasNext(self) -> bool:
        return self.i < len(self.vals)

    def _inorder(self, root: TreeNode | None) -> None:
        if not root:
            return
        self._inorder(root.left)
        self.vals.append(root.val)
        self._inorder(root.right)
```

Python

```python
class BSTIterator:
    def __init__(self, root: TreeNode | None):
        self.stack = []
        self._pushLeftsUntiNull(root)

    def next(self) -> int:
        root = self.stack.pop()
        self._pushLeftsUntiNull(root.right)
        return root.val

    def hasNext(self) -> bool:
        return self.stack

    def _pushLeftsUntiNull(self, root: TreeNode | None) -> None:
        while root:
            self.stack.append(root)
            root = root.left
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class BSTIterator:
    def __init__(self, root: TreeNode):
        def inorder(root):
            if root:
                inorder(root.left)
                self.vals.append(root.val)
                inorder(root.right)

        self.cur = 0
        self.vals = []
        inorder(root)

    def next(self) -> int:
        res = self.vals[self.cur]
        self.cur += 1
        return res

    def hasNext(self) -> bool:
        return self.cur < len(self.vals)


# Your BSTIterator object will be instantiated and called as such:
# obj = BSTIterator(root)
# param_1 = obj.next()
# param_2 = obj.hasNext()
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class BSTIterator:
    def __init__(self, root: TreeNode):
        self.stack = []
        self._leftmost_inorder(root)

    def _leftmost_inorder(self, node):
        while node:
            self.stack.append(node)
            node = node.left

    def next(self) -> int:
        cur = self.stack.pop()
        node = cur.right
        self._leftmost_inorder(node)
        return cur.val

    def hasNext(self) -> bool:
        return len(self.stack) > 0
```



```python
# Your BSTIterator object will be instantiated and called as such:
# obj = BSTIterator(root)
# param_1 = obj.next()
# param_2 = obj.hasNext()

############

# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class BSTIterator:
    def __init__(self, root: TreeNode):
        self.stack = []
        while root:
            self.stack.append(root)
            root = root.left

    def next(self) -> int:
        cur = self.stack.pop()
        node = cur.right
        while node: # deplicated while block
            self.stack.append(node)
            node = node.left
        return cur.val

    def hasNext(self) -> bool:
        return len(self.stack) > 0


# Your BSTIterator object will be instantiated and called as such:
# obj = BSTIterator(root)
# param_1 = obj.next()
# param_2 = obj.hasNext()
```

[*] Stack — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class BSTIterator:
    def __init__(self, root):
        # Initialize an empty stack to simulate in-order traversal
        self.stack = []
        # Push all left nodes starting from the root
        self._push_left_branch(root)

    def _push_left_branch(self, node):
        # Push node and all its left children to the stack
        while node:
            self.stack.append(node)
            node = node.left

    def next(self):
        # Pop the top node which represents the next smallest element
        node = self.stack.pop()
        # If the node has a right child, push its left branch onto the stack
        if node.right:
            self._push_left_branch(node.right)
        # Return the node's value
        return node.val

    def hasNext(self):
        # The iterator has a next element if the stack is not empty
        return len(self.stack) > 0
```

#227 MEDIUM

Basic Calculator II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math · String · Stack

Lists: Grind 169

Problem:

Given a string s which represents an expression, evaluate this expression and return its value.

The integer division should truncate toward zero.

You may assume that the given expression is always valid. All intermediate results will be in the range of [-231, 231 - 1].

Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().

Example 1:

```
Input: s = "3+2*2"
Output: 7
```

Example 2:

```
Input: s = " 3/2 "
Output: 1
```

Example 3:

```
Input: s = " 3+5 / 2 "
Output: 5
```

Constraints:

- 1 <= s.length <= 3 * 105
- s consists of integers and operators ('+', '-', '*', '/') separated by some number of spaces.
- s represents a valid expression.
- All the integers in the expression are non-negative integers in the range [0, 231 - 1].
- The answer is guaranteed to fit in a 32-bit integer.

Python

Python

```python
# Python solution for Basic Calculator II

def calculate(s: str) -> int:
    # Initialize the stack, current number, and the last seen operator.
    stack = []
    current_number = 0
    previous_operator = '+'

    # Loop over each character in the string along with one extra iteration to handle the end.
    for i, char in enumerate(s):
        if char.isdigit():
            # Build the current number by shifting previous digits.
            current_number = current_number * 10 + int(char)
        # If the character is an operator (or we're at the end), process the previous operator.
        if char in "+-*/" or i == len(s) - 1:
            if previous_operator == '+':
                stack.append(current_number)
            elif previous_operator == '-':
                stack.append(-current_number)
            elif previous_operator == '*':
                last_number = stack.pop()
                stack.append(last_number * current_number)
            elif previous_operator == '/':
                last_number = stack.pop()
                # Python division truncates toward negative infinity by default, so we use int() to
truncate toward zero.

                stack.append(int(last_number / current_number))
            # Reset for the next number and update the operator.
            previous_operator = char
            current_number = 0
    # Sum up all numbers in the stack which gives the final result.
    return sum(stack)

# Example usage:
print(calculate("3+2*2")) # Output: 7
print(calculate(" 3/2 ")) # Output: 1
print(calculate(" 3+5 / 2 ")) # Output: 5
```


Python

```python
class Solution:
    def calculate(self, s: str) -> int:
        ans = 0
        prevNum = 0
        currNum = 0
        op = '+'

        for i, c in enumerate(s):
          if c.isdigit():
            currNum = currNum * 10 + int(c)
          if not c.isdigit() and c != ' ' or i == len(s) - 1:
            if op == '+' or op == '-':
              ans += prevNum
            prevNum = currNum if op == '+' else -currNum
            elif op == '*':
              prevNum = prevNum * currNum
            elif op == '/':
              if prevNum < 0:
                prevNum = math.ceil(prevNum / currNum)
              else:
                prevNum = prevNum // currNum
            op = c
            currNum = 0

        return ans + prevNum
```

Python

```python
class Solution:
    def calculate(self, s: str) -> int:
        v, n = 0, len(s)
        sign = '+'
        stk = []
        for i, c in enumerate(s):
            if c.isdigit():
                v = v * 10 + int(c)
            if i == n - 1 or c in '+-*/':
                match sign:
                    case '+':
                        stk.append(v)
                    case '-':
                        stk.append(-v)
                    case '*':
                        stk.append(stk.pop() * v)
                    case '/':
                        stk.append(int(stk.pop() / v))
                sign = c
                v = 0
        return sum(stk)
```

Python


```python
class Solution:
    def calculate(self, s: str) -> int:
        v, n = 0, len(s)
        sign = '+'
        stk = []
        for i, c in enumerate(s):
            if c.isdigit():
                v = v * 10 + int(c)
            if i == n - 1 or c in '+-*/':
                match sign:
                    case '+':
                        stk.append(v)
                    case '-':
                        stk.append(-v)
                    case '*':
                        stk.append(stk.pop() * v)
                    case '/':
                        stk.append(int(stk.pop() / v))
                sign = c # reset inside 'if'
                v = 0
        return sum(stk)
```

[*] Stack — Pattern Solution (matched from community answers)

Python

```python
# Python solution for Basic Calculator II

def calculate(s: str) -> int:
    # Initialize the stack, current number, and the last seen operator.
    stack = []
    current_number = 0
    previous_operator = '+'

    # Loop over each character in the string along with one extra iteration to handle the end.
    for i, char in enumerate(s):
        if char.isdigit():
            # Build the current number by shifting previous digits.
            current_number = current_number * 10 + int(char)
        # If the character is an operator (or we're at the end), process the previous operator.
        if char in "+-*/" or i == len(s) - 1:
            if previous_operator == '+':
                stack.append(current_number)
            elif previous_operator == '-':
                stack.append(-current_number)
            elif previous_operator == '*':
                last_number = stack.pop()
                stack.append(last_number * current_number)
            elif previous_operator == '/':
                last_number = stack.pop()
                # Python division truncates toward negative infinity by default, so we use int() to
truncate toward zero.
```



```
                stack.append(int(last_number / current_number))
            # Reset for the next number and update the operator.
            previous_operator = char
            current_number = 0
    # Sum up all numbers in the stack which gives the final result.
    return sum(stack)

# Example usage:
print(calculate("3+2*2")) # Output: 7
print(calculate(" 3/2 ")) # Output: 1
print(calculate(" 3+5 / 2 ")) # Output: 5
```

#255 MEDIUM

Verify Preorder Sequence in Binary Search Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

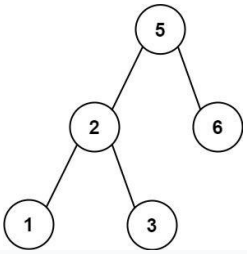
Array · Tree · BST · Stack · Monotonic Stack

Lists: Premium 98

Problem:

Given an array of unique integers preorder, return true if it is the correct preorder traversal sequence of a binary search tree.

Example 1:




```
Input: preorder = [5,2,1,3,6]
Output: true
```

Example 2:


```
Input: preorder = [5,2,6,1,3]
Output: false
```

Constraints:

- 1 <= preorder.length <= 104
- 1 <= preorder[i] <= 104
- All the elements of preorder are unique.

Follow up: Could you do it using only constant space complexity?

>> Brute Force [Stack] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def verifyPreorder(self, preorder):
        # Brute Force O(n^2) - simulate without stack via inner loop
        n = len(preorder)
        result = [-1] * n
        for i in range(n):
            for j in range(i + 1, n):
                if preorder[j] > preorder[i]:
                    result[i] = preorder[j]
                    break
        return result
```

Python

Python

```python
# Python solution using a monotonic stack with detailed comments
def verifyPreorder(preorder):
    lower_bound = float('-inf') # Set initial lower bound to negative infinity
    stack = [] # Use a stack to track the BST construction
    for value in preorder:
        # If the current value is less than lower_bound, the BST property is violated
        if value < lower_bound:
            return False
        # If current value is greater than the top of stack, pop elements and update lower_bound
        while stack and value > stack[-1]:
            lower_bound = stack.pop() # Update lower_bound as we move to the right subtree
        # Push the current value onto the stack
        stack.append(value)
    return True

# Example usage:
# print(verifyPreorder([5,2,1,3,6])) # Expected output: True
```

Python


```python
class Solution:
    def verifyPreorder(self, preorder: list[int]) -> bool:
        i = 0

        def dfs(min: int, max: int) -> None:
          nonlocal i
          if i == len(preorder):
            return
          if preorder[i] < min or preorder[i] > max:
            return

          val = preorder[i]
          i += 1
          dfs(min, val)
          dfs(val, max)

        dfs(-math.inf, math.inf)
        return i == len(preorder)
```

Python

```python
class Solution:
    def verifyPreorder(self, preorder: list[int]) -> list[int]:
        low = -math.inf
        stack = []

        for p in preorder:
          if p < low:
            return False
          while stack and stack[-1] < p:
            low = stack.pop()
          stack.append(p)

        return True
```

Python

```python
class Solution:
    def verifyPreorder(self, preorder: list[int]) -> bool:
        low = -math.inf
        i = -1

        for p in preorder:
          if p < low:
            return False
          while i >= 0 and preorder[i] < p:
            low = preorder[i]
            i -= 1
          i += 1
          preorder[i] = p

        return True
```

Python

```python
class Solution:
    def verifyPreorder(self, preorder: List[int]) -> bool:
        stk = []
        last = -inf
        for x in preorder:
            if x < last:
                return False
            while stk and stk[-1] < x:
                last = stk.pop()
            stk.append(x)
        return True
```

Python

```python
class Solution:
    def verifyPreorder(self, preorder: List[int]) -> bool:
        stk = []
        last = -inf
        for x in preorder:
            if x < last:
                return False
            while stk and stk[-1] < x:
                last = stk.pop()
            stk.append(x)
        return True


class Solution: # O(N) but modifying input list
    def verifyPreorder(self, preorder: List[int]) -> bool:
        low = -math.inf
        i = -1

        for p in preorder:
            if p < low:
                return False
            while i >= 0 and preorder[i] < p:
                low = preorder[i]
                i -= 1
            i += 1
```



```python
            preorder[i] = p

        return True

###########

class Solution(object):

  # O(n) sapce complexity, ++stack++
  def verifyPreorder(self, preorder):
    """
    :type preorder: List[int]
    :rtype: bool
    """
    if len(preorder) <= 1:
      return True
    stack, lastElem = [preorder[0]], None
    for i in range(1, len(preorder)):
      if lastElem > preorder[i]:
        return False
      while len(stack) > 0 and preorder[i] > stack[-1]:
        lastElem = stack.pop()
      stack.append(preorder[i])

    return True
```

[*] Stack — Pattern Solution (matched from community answers)

Python

```python
# Python solution using a monotonic stack with detailed comments
def verifyPreorder(preorder):
    lower_bound = float('-inf') # Set initial lower bound to negative infinity
    stack = [] # Use a stack to track the BST construction
    for value in preorder:
        # If the current value is less than lower_bound, the BST property is violated
        if value < lower_bound:
            return False
        # If current value is greater than the top of stack, pop elements and update lower_bound
        while stack and value > stack[-1]:
            lower_bound = stack.pop() # Update lower_bound as we move to the right subtree
        # Push the current value onto the stack
        stack.append(value)
    return True

# Example usage:
# print(verifyPreorder([5,2,1,3,6])) # Expected output: True
```

#394 MEDIUM

Decode String

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Stack · Recursion

Lists: Grind 169

Problem:

Given an encoded string, return its decoded string.

The encoding rule is: k[encoded_string], where the encoded_string inside the square brackets is being repeated exactly k times. Note that k is guaranteed to be a positive integer.

You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc. Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, k. For example, there will not be input like 3a or 2[4].

The test cases are generated so that the length of the output will never exceed 105.

Example 1:

```
Input: s = "3[a]2[bc]"
Output: "aaabcbc"
```

Example 2:

```
Input: s = "3[a2[c]]"
Output: "accaccacc"
```

Example 3:

```
Input: s = "2[abc]3[cd]ef"
Output: "abcabccdcdcdef"
```

Constraints:

- 1 <= s.length <= 30
- s consists of lowercase English letters, digits, and square brackets '[]'.
- s is guaranteed to be a valid input.
- All the integers in s are in the range [1, 300].

Python

Python


```python
def decodeString(s: str) -> str:
    # index pointer as a list so it can be mutable within recursion
    def helper(i: int) -> (str, int):
        decoded = ""
        while i < len(s):
            if s[i].isdigit():
                # build the number (it might be more than one digit)
                k = 0
                while i < len(s) and s[i].isdigit():
                    k = k * 10 + int(s[i])
                    i += 1
                # skip the '[' character
                i += 1 # skip '['
                # recursively solve the substring inside the brackets
                sub, i = helper(i)
                # append the substring repeated k times
                decoded += sub * k
            elif s[i] == ']':
                # returning decoded string and new index after ']'
                return decoded, i + 1
            else:
                # accumulate alphabetic characters directly
                decoded += s[i]
                i += 1
        return decoded, i


    result, _ = helper(0)
    return result

# Example usage
#print(decodeString("3[a2[c]]")) # Output: "accaccacc"
```

Python

```python
class Solution:
  def decodeString(self, s: str) -> str:
    stack = [] # (prevStr, repeatCount)
    currStr = ''
    currNum = 0

    for c in s:
      if c.isdigit():
        currNum = currNum * 10 + int(c)
      else:
        if c == '[':
          stack.append((currStr, currNum))
          currStr = ''
          currNum = 0
        elif c == ']':
          prevStr, num = stack.pop()
          currStr = prevStr + num * currStr
        else:
          currStr += c

    return currStr
```

Python


```python
class Solution:
  def decodeString(self, s: str) -> str:
    ans = ''

    while self.i < len(s) and s[self.i] != ']':
      if s[self.i].isdigit():
        k = 0
        while self.i < len(s) and s[self.i].isdigit():
          k = k * 10 + int(s[self.i])
          self.i += 1
        self.i += 1 # '['
        decodedString = self.decodeString(s)
        self.i += 1 # ']'
        ans += k * decodedString
      else:
        ans += s[self.i]
        self.i += 1

    return ans

  i = 0
```

Python

```python
class Solution:
    def decodeString(self, s: str) -> str:
        s1, s2 = [], []
        num, res = 0, ''
        for c in s:
            if c.isdigit():
                num = num * 10 + int(c)
            elif c == '[':
                s1.append(num)
                s2.append(res)
                num, res = 0, ''
            elif c == ']':
                res = s2.pop() + res * s1.pop()
            else:
                res += c
        return res
```

Python

```python
class Solution:
    def decodeString(self, s: str) -> str:
        num_stk, str_stk = [], []
        num, res = 0, ''
        for c in s:
            if c.isdigit():
                num = num * 10 + int(c)
            elif c == '[':
                num_stk.append(num)
                str_stk.append(res) # initial, res being pushed is empty str ''
                num, res = 0, ''
            elif c == ']':
                res = str_stk.pop() + res * num_stk.pop()
            else:
                res += c
        return res
```


[*] Stack — Pattern Solution (matched from community answers)

Python

```python
class Solution:
  def decodeString(self, s: str) -> str:
    stack = [] # (prevStr, repeatCount)
    currStr = ''
    currNum = 0

    for c in s:
      if c.isdigit():
        currNum = currNum * 10 + int(c)
      else:
        if c == '[':
          stack.append((currStr, currNum))
          currStr = ''
          currNum = 0
        elif c == ']':
          prevStr, num = stack.pop()
          currStr = prevStr + num * currStr
        else:
          currStr += c

    return currStr
```

#439 MEDIUM

Ternary Expression Parser

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Stack · Recursion

Lists: Premium 98

Problem:

Given a string expression representing arbitrarily nested ternary expressions, evaluate the expression, and return the result of it.

You can always assume that the given expression is valid and only contains digits, '?', ':', 'T', and 'F' where 'T' is true and 'F' is false. All the numbers in the expression are one-digit numbers (i.e., in the range [0, 9]).

The conditional expressions group right-to-left (as usual in most languages), and the result of the expression will always evaluate to either a digit, 'T' or 'F'.

Example 1:

```
Input: expression = "T?2:3"
Output: "2"
Explanation: If true, then result is 2; otherwise result is 3.
```

Example 2:

```
Input: expression = "F?1:T?4:5"
Output: "4"
Explanation: The conditional expressions group right-to-left. Using parenthesis, it is
read/evaluated as:
"(F ? 1 : (T ? 4 : 5))" --> "(F ? 1 : 4)" --> "4"
```

```
or "(F ? 1 : (T ? 4 : 5))" --> "(T ? 4 : 5)" --> "4"
```

Example 3:

```
Input: expression = "T??F:5:3"
Output: "F"
Explanation: The conditional expressions group right-to-left. Using parenthesis, it is
read/evaluated as:
"(T ? (T ? F : 5) : 3)" --> "(T ? F : 3)" --> "F"
"(T ? (T ? F : 5) : 3)" --> "(T ? F : 5)" --> "F"
```

Constraints:

- 5 <= expression.length <= 104
- expression consists of digits, 'T', 'F', '?', and ':'.
- It is guaranteed that expression is a valid ternary expression and that each number is a one-digit number.

Python

Python

```python
def parseTernary(expression: str) -> str:
    # Initialize stack to hold characters/results during evaluation
    stack = []
    # Iterate backwards over the expression
    i = len(expression) - 1
    while i >= 0:
        if expression[i] == '?':
            # Pop the true and false parts from the stack
            true_expr = stack.pop()
            false_expr = stack.pop()
            i -= 1 # Move to the condition character
            # Choose based on the condition
            if expression[i] == 'T':
                stack.append(true_expr)
            else:
                stack.append(false_expr)
        elif expression[i] == ':':
            # Skip the colon as it is only a separator
            pass
        else:
            # Push digits, 'T', and 'F' onto stack as is
            stack.append(expression[i])
        i -= 1 # Move to the next character
    return stack[-1]

# Example usage:
print(parseTernary("T?2:3")) # Expected output "2"
```

Python


```python
class Solution:
    def parseTernary(self, expression: str) -> str:
        c = expression[self.i]

        if self.i + 1 == len(expression) or expression[self.i + 1] == ':':
            self.i += 2
            return str(c)

        self.i += 2
        first = self.parseTernary(expression)
        second = self.parseTernary(expression)

        return first if c == 'T' else second

    i = 0
```

Python

```python
class Solution:
    def parseTernary(self, expression: str) -> str:
        stk = []
        cond = False
        for c in expression[::-1]:
            if c == ':':
                continue
            if c == '?':
                cond = True
            else:
                if cond:
                    if c == 'T':
                        x = stk.pop()
                        stk.pop()
                        stk.append(x)
                    else:
                        stk.pop()
                    cond = False
                else:
                    stk.append(c)
        return stk[0]
```

Python


```python
class Solution:
    def parseTernary(self, expression: str) -> str:
        stk = []
        cond = False
        for c in expression[::-1]:
            if c == ':':
                continue
            if c == '?':
                cond = True
            else:
                if cond:
                    if c == 'T':
                        x = stk.pop()
                        stk.pop()
                        stk.append(x)
                    else:
                        stk.pop()
                    cond = False
                else:
                    stk.append(c)
        return stk[0]
```

[*] Stack — Pattern Solution (matched from community answers)

Python

```python
def parseTernary(expression: str) -> str:
    # Initialize stack to hold characters/results during evaluation
    stack = []
    # Iterate backwards over the expression
    i = len(expression) - 1
    while i >= 0:
        if expression[i] == '?':
            # Pop the true and false parts from the stack
            true_expr = stack.pop()
            false_expr = stack.pop()
            i -= 1 # Move to the condition character
            # Choose based on the condition
            if expression[i] == 'T':
                stack.append(true_expr)
            else:
                stack.append(false_expr)
        elif expression[i] == ':':
            # Skip the colon as it is only a separator
            pass
        else:
            # Push digits, 'T', and 'F' onto stack as is
            stack.append(expression[i])
        i -= 1 # Move to the next character
    return stack[-1]

# Example usage:
print(parseTernary("T?2:3")) # Expected output "2"
```

#484 MEDIUM

Find Permutation

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · String · Stack · Greedy

Lists: Premium 98

Problem:

A permutation perm of n integers of all the integers in the range [1, n] can be represented as a string s of length n - 1 where:

- s[i] == 'I' if perm[i] < perm[i + 1], and
- s[i] == 'D' if perm[i] > perm[i + 1].

Given a string s, reconstruct the lexicographically smallest permutation perm and return it.

Example 1:

```
Input: s = "I"
Output: [1,2]
Explanation: [1,2] is the only legal permutation that can represented by s, where the number
1 and 2 construct an increasing relationship.
```

Example 2:

```
Input: s = "DI"
Output: [2,1,3]
Explanation: Both [2,1,3] and [3,1,2] can be represented as "DI", but since we want to find
the smallest lexicographical permutation, you should return [2,1,3]
```

Constraints:

- 1 <= s.length <= 105
- s[i] is either 'I' or 'D'.

>> Brute Force [Stack] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findPermutation(self, s):
        # Brute Force O(n^2) - simulate without stack via inner loop
        n = len(s)
        result = [-1] * n
        for i in range(n):
            for j in range(i + 1, n):
                if s[j] > s[i]:
                    result[i] = s[j]
                    break
        return result
```

Python

Python


```python
# Function to find the lexicographically smallest permutation
def findPermutation(s):
    stack = [] # Use a stack to hold numbers during a sequence of 'D's
    result = [] # Result list for the final permutation
    # Iterate over 0 to len(s) (inclusive) to process all characters plus one final number.
    for i in range(len(s) + 1):
        # Push i+1 since our numbers range from 1 to n
        stack.append(i + 1)
        # If we are at the end or the current character is 'I', flush the stack.
        if i == len(s) or s[i] == 'I':
            while stack:
                result.append(stack.pop())
    return result

# Example usage:
print(findPermutation("DI"))
```

Python

```python
class Solution:
    def findPermutation(self, s: str) -> list[int]:
        ans = []
        stack = []

        for i, c in enumerate(s):
          stack.append(i + 1)
          if c == 'I':
            while stack: # Consume all decreasings
              ans.append(stack.pop())
        stack.append(len(s) + 1)

        while stack:
          ans.append(stack.pop())

        return ans
```

Python

```python
class Solution:
    def findPermutation(self, s: str) -> list[int]:
        ans = [i for i in range(1, len(s) + 2)]

        # For each D+ group (s[i..j]), reverse ans[i..j + 1].
        i = -1
        j = -1

        def getNextIndex(c: str, start: int) -> int:
          for i in range(start, len(s)):
            if s[i] == c:
              return i
          return len(s)

        while True:
          i = getNextIndex('D', j + 1)
          if i == len(s):
            break
          j = getNextIndex('I', i + 1)
          ans[i:j + 1] = ans[i:j + 1][::-1]

        return ans
```


Python

```python
class Solution:
    def findPermutation(self, s: str) -> List[int]:
        n = len(s)
        ans = list(range(1, n + 2))
        i = 0
        while i < n:
            j = i
            while j < n and s[j] == 'D':
                j += 1
            ans[i : j + 1] = ans[i : j + 1][::-1]
            i = max(i + 1, j)
        return ans
```

Python

```python
class Solution:
    def findPermutation(s: str) -> List[int]:
        n = len(s)
        ans = list(range(1, n + 2))
        i = 0
        while i < n:
            j = i
            while j < n and s[j] == 'D':
                j += 1
            ans[i : j + 1] = ans[i : j + 1][::-1]
            i = max(i + 1, j)
        return ans
```

[*] Stack — Pattern Solution (matched from community answers)

Python

```python
# Function to find the lexicographically smallest permutation
def findPermutation(s):
    stack = [] # Use a stack to hold numbers during a sequence of 'D's
    result = [] # Result list for the final permutation
    # Iterate over 0 to len(s) (inclusive) to process all characters plus one final number.
    for i in range(len(s) + 1):
        # Push i+1 since our numbers range from 1 to n
        stack.append(i + 1)
        # If we are at the end or the current character is 'I', flush the stack.
        if i == len(s) or s[i] == 'I':
            while stack:
                result.append(stack.pop())
    return result

# Example usage:
print(findPermutation("DI"))
```

#735 MEDIUM

Asteroid Collision

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Stack · Simulation

Lists: Grind 169

Problem:
We are given an array asteroids of integers representing asteroids in a row. The indices of the asteroid in the array represent their relative position in space.

For each asteroid, the absolute value represents its size, and the sign represents its direction (positive meaning right, negative meaning left). Each asteroid moves at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

Example 1:

```
Input: asteroids = [5,10,-5]
Output: [5,10]
Explanation: The 10 and -5 collide resulting in 10. The 5 and 10 never collide.
```

Example 2:

```
Input: asteroids = [8,-8]
Output: []
Explanation: The 8 and -8 collide exploding each other.
```

Example 3:

```
Input: asteroids = [10,2,-5]
Output: [10]
Explanation: The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting in 10.
```

Example 4:

```
Input: asteroids = [3,5,-6,2,-1,4]□□□□□□
Output: [-6,2,4]
Explanation: The asteroid -6 makes the asteroid 3 and 5 explode, and then continues going
left. On the other side, the asteroid 2 makes the asteroid -1 explode and then continues
going right, without reaching asteroid 4.
```

Constraints:

- 2 <= asteroids.length <= 104
- -1000 <= asteroids[i] <= 1000
- asteroids[i] != 0

>> Brute Force [Stack] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def asteroidCollision(self, asteroids):
        # Brute Force O(n^2) - simulate without stack via inner loop
        n = len(asteroids)
        result = [-1] * n
        for i in range(n):
            for j in range(i + 1, n):
                if asteroids[j] > asteroids[i]:
                    result[i] = asteroids[j]
                    break
        return result
```

Python

Python

```python
# Function to simulate asteroid collisions
def asteroidCollision(asteroids):
    # Initialize an empty list as a stack to hold surviving asteroids
    stack = []
    # Iterate through each asteroid in the input list
    for ast in asteroids:
        # Flag to check if the current asteroid has exploded
        exploded = False
        # Check for potential collision while there is an asteroid in stack and a collision condition holds
(right-moving on stack and left-moving current)
        while stack and ast < 0 and stack[-1] > 0:
            # If the incoming left-moving asteroid is larger than the right-moving one on stack, pop the
top and continue checking
            if abs(ast) > abs(stack[-1]):
                stack.pop()
                continue # Continue the loop to check next potential collision
            # If they are equal, both explode: pop the stack and mark explosion, break the loop
            elif abs(ast) == abs(stack[-1]):
                stack.pop()
            # Mark the incoming asteroid as exploded and break out of the loop
            exploded = True
            break
        # If the asteroid did not explode during collisions, push it onto the stack
        if not exploded:
            stack.append(ast)
    # Return the surviving asteroids
    return stack


# Example usage:
# print(asteroidCollision([5,10,-5])) # Output: [5,10]
```

Python

```python
class Solution:
    def asteroidCollision(self, asteroids: list[int]) -> list[int]:
        stack = []

        for a in asteroids:
          if a > 0:
            stack.append(a)
          else: # a < 0
            # Destroy the previous positive one(s).
            while stack and stack[-1] > 0 and stack[-1] < -a:
              stack.pop()
            if not stack or stack[-1] < 0:
              stack.append(a)
            elif stack[-1] == -a:
              stack.pop() # Both asteroids explode.
            else: # stack[-1] > the current asteroid.
              pass # Destroy the current asteroid, so do nothing.

        return stack
```

Python


```python
class Solution:
    def asteroidCollision(self, asteroids: List[int]) -> List[int]:
        stk = []
        for x in asteroids:
            if x > 0:
                stk.append(x)
            else:
                while stk and stk[-1] > 0 and stk[-1] < -x:
                    stk.pop()
                if stk and stk[-1] == -x:
                    stk.pop()
                elif not stk or stk[-1] < 0:
                    stk.append(x)
        return stk
```

Python

```python
class Solution:
    def asteroidCollision(self, asteroids: List[int]) -> List[int]:
        stk = []
        for x in asteroids:
            if x > 0:
                stk.append(x)
            else:
                while stk and stk[-1] > 0 and stk[-1] < -x:
                    stk.pop()
                if stk and stk[-1] == -x:
                    stk.pop()
                elif not stk or stk[-1] < 0:
                    stk.append(x)
        return stk
```

[*] Stack — Pattern Solution (matched from community answers)

Python


```python
# Function to simulate asteroid collisions
def asteroidCollision(asteroids):
    # Initialize an empty list as a stack to hold surviving asteroids
    stack = []
    # Iterate through each asteroid in the input list
    for ast in asteroids:
        # Flag to check if the current asteroid has exploded
        exploded = False
        # Check for potential collision while there is an asteroid in stack and a collision condition holds
(right-moving on stack and left-moving current)
        while stack and ast < 0 and stack[-1] > 0:
            # If the incoming left-moving asteroid is larger than the right-moving one on stack, pop the
top and continue checking
            if abs(ast) > abs(stack[-1]):
                stack.pop()
                continue # Continue the loop to check next potential collision
            # If they are equal, both explode: pop the stack and mark explosion, break the loop
            elif abs(ast) == abs(stack[-1]):
                stack.pop()
            # Mark the incoming asteroid as exploded and break out of the loop
            exploded = True
            break
        # If the asteroid did not explode during collisions, push it onto the stack
        if not exploded:
            stack.append(ast)
    # Return the surviving asteroids
    return stack


# Example usage:
# print(asteroidCollision([5,10,-5])) # Output: [5,10]
```

#739 MEDIUM

Daily Temperatures

LeetCrack · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Stack · Monotonic Stack

Lists: Grind 169

Problem:
Given an array of integers temperatures represents the daily temperatures, return an array answer such that answer[i] is the number of days you have to wait after the ith day to get a warmer temperature. If there is no future day for which this is possible, keep answer[i] == 0 instead.

Example 1:

```
Input: temperatures = [73,74,75,71,69,72,76,73]
Output: [1,1,4,2,1,1,0,0]
```

Example 2:

```
Input: temperatures = [30,40,50,60]
Output: [1,1,1,0]
```

Example 3:


```
Input: temperatures = [30,60,90]
Output: [1,1,0]
```

Constraints:

- 1 <= temperatures.length <= 105
- 30 <= temperatures[i] <= 100

>> Brute Force [Stack] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def dailyTemperatures(self, temperatures):
        # Brute Force O(n^2) - for each day scan forward for warmer day
        n = len(temperatures)
        res = [0] * n
        for i in range(n):
            for j in range(i + 1, n):
                if temperatures[j] > temperatures[i]:
                    res[i] = j - i
                    break
        return res
```

Python

Python

```python
def dailyTemperatures(temperatures):
    n = len(temperatures)
    # Initialize answer list with zeros
    answer = [0] * n
    # Stack to keep indices of temperatures with unresolved next warmer days
    stack = []
    # Iterate through the temperature list
    for i, temp in enumerate(temperatures):
        # Check if the current temperature resolves any previous indices
        while stack and temperatures[stack[-1]] < temp:
            prev_index = stack.pop() # Get the index of the previous colder day
            answer[prev_index] = i - prev_index # Update the number of days waited
        # Push the current index onto the stack
        stack.append(i)
    return answer

# Example usage:
if __name__ == "__main__":
    temps = [73,74,75,71,69,72,76,73]
    print(dailyTemperatures(temps))
```

Python


```python
class Solution:
    def dailyTemperatures(self, temperatures: list[int]) -> list[int]:
        ans = [0] * len(temperatures)
        stack = [] # a decreasing stack

        for i, temperature in enumerate(temperatures):
          while stack and temperature > temperatures[stack[-1]]:
            index = stack.pop()
            ans[index] = i - index
          stack.append(i)

        return ans
```

Python

```python
class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        stk = []
        n = len(temperatures)
        ans = [0] * n
        for i in range(n - 1, -1, -1):
            while stk and temperatures[stk[-1]] <= temperatures[i]:
                stk.pop()
            if stk:
                ans[i] = stk[-1] - i
            stk.append(i)
        return ans
```

Python

```python
class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        ans = [0] * len(temperatures)
        stk = []
        for i, t in enumerate(temperatures):
            while stk and temperatures[stk[-1]] < t:
                j = stk.pop()
                ans[j] = i - j
            stk.append(i)
        return ans
```

[*] Stack — Pattern Solution (matched from community answers)

Python

```python
def dailyTemperatures(temperatures):
    n = len(temperatures)
    # Initialize answer list with zeros
    answer = [0] * n
    # Stack to keep indices of temperatures with unresolved next warmer days
    stack = []
    # Iterate through the temperature list
    for i, temp in enumerate(temperatures):
        # Check if the current temperature resolves any previous indices
        while stack and temperatures[stack[-1]] < temp:
            prev_index = stack.pop() # Get the index of the previous colder day
            answer[prev_index] = i - prev_index # Update the number of days waited
        # Push the current index onto the stack
        stack.append(i)
    return answer

# Example usage:
if __name__ == "__main__":
    temps = [73,74,75,71,69,72,76,73]
    print(dailyTemperatures(temps))
```

#962 MEDIUM

Maximum Width Ramp

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Stack · Monotonic Stack

Lists: Denny Zhang

Problem:

A ramp in an integer array nums is a pair (i, j) for which i < j and nums[i] <= nums[j]. The width of such a ramp is j − i.

Given an integer array nums, return the maximum width of a ramp in nums. If there is no ramp in nums, return 0.

Example 1:

```
Input: nums = [6,0,8,2,1,5]
Output: 4
Explanation: The maximum width ramp is achieved at (i, j) = (1, 5): nums[1] = 0 and nums[5] =
5.
```

Example 2:

```
Input: nums = [9,8,1,0,1,9,4,0,4,1]
Output: 7
Explanation: The maximum width ramp is achieved at (i, j) = (2, 9): nums[2] = 1 and nums[9] =
1.
```

Constraints:

- 2 <= nums.length <= 5 * 104
- 0 <= nums[i] <= 5 * 104

>> Brute Force [Stack] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def maxWidthRamp(self, nums):
        # Brute Force O(n^2) - check every pair (i,j) where i<j and nums[i]<=nums[j]
        res = 0
        for i in range(len(nums)):
            for j in range(i, len(nums)):
                if nums[j] >= nums[i]:
                    res = max(res, j - i)
        return res
```

Python

Python

```python
# Python solution for Maximum Width Ramp

def maxWidthRamp(nums):
    n = len(nums)
    stack = []
    # Build a stack of indices where array values are in decreasing order
    for i in range(n):
        if not stack or nums[i] < nums[stack[-1]]:
            stack.append(i)
    max_width = 0
    # Iterate backwards to find the maximum width ramp
    for j in range(n - 1, -1, -1):
        while stack and nums[j] >= nums[stack[-1]]:
            max_width = max(max_width, j - stack.pop())
    return max_width

# Example usage:
nums = [6, 0, 8, 2, 1, 5]
print(maxWidthRamp(nums)) # Output: 4
```

Python

```python
class Solution:
  def maxWidthRamp(self, nums: list[int]) -> int:
    ans = 0
    stack = []

    for i, num in enumerate(nums):
      if stack == [] or num <= nums[stack[-1]]:
        stack.append(i)

    for i, num in reversed(list(enumerate(nums))):
      while stack and num >= nums[stack[-1]]:
        ans = max(ans, i - stack.pop())

    return ans
```

Python


```python
class Solution:
    def maxWidthRamp(self, nums: List[int]) -> int:
        stk = []
        for i, v in enumerate(nums):
            if not stk or nums[stk[-1]] > v:
                stk.append(i)
        ans = 0
        for i in range(len(nums) - 1, -1, -1):
            while stk and nums[stk[-1]] >= nums[i]:
                ans = max(ans, i - stk.pop())
            if not stk:
                break
        return ans
```

Python

```python
class Solution:
    def maxWidthRamp(self, nums: List[int]) -> int:
        stk = []
        for i, v in enumerate(nums):
            if not stk or nums[stk[-1]] > v:
                stk.append(i)
        ans = 0
        for i in range(len(nums) - 1, -1, -1):
            while stk and nums[stk[-1]] >= nums[i]:
                ans = max(ans, i - stk.pop())
            if not stk:
                break
        return ans
```

[*] Stack — Pattern Solution (matched from community answers)

Python

```python
# Python solution for Maximum Width Ramp

def maxWidthRamp(nums):
    n = len(nums)
    stack = []
    # Build a stack of indices where array values are in decreasing order
    for i in range(n):
        if not stack or nums[i] < nums[stack[-1]]:
            stack.append(i)
    max_width = 0
    # Iterate backwards to find the maximum width ramp
    for j in range(n - 1, -1, -1):
        while stack and nums[j] >= nums[stack[-1]]:
            max_width = max(max_width, j - stack.pop())
    return max_width

# Example usage:
nums = [6, 0, 8, 2, 1, 5]
print(maxWidthRamp(nums)) # Output: 4
```

#1214 MEDIUM

Two Sum BSTs

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

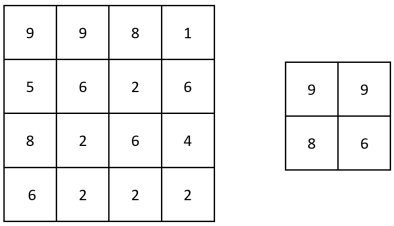
Two Pointers · Binary Search · Stack · Tree · DFS · BST · BFS

Lists: Premium 98

Problem:

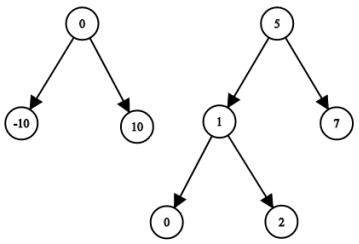
Given the roots of two binary search trees, root1 and root2, return true if and only if there is a node in the first tree and a node in the second tree whose values sum up to a given integer target.

Example 1:




```
Input: root1 = [2,1,4], root2 = [1,0,3], target = 5
Output: true
Explanation: 2 and 3 sum up to 5.
```

Example 2:




```
Input: root1 = [0,-10,10], root2 = [5,1,7,0,2], target = 18
Output: false
```

Constraints:

- The number of nodes in each tree is in the range [1, 5000].
- -109 <= Node.val, target <= 109

Python

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def twoSumBSTs(root1: TreeNode, root2: TreeNode, target: int) -> bool:
    # Create a set to store values from the first BST.
    values_set = set()

    # Helper function to perform DFS and add node values to the set.
    def dfs_collect(node):
        if not node:
            return
        values_set.add(node.val) # Add current node value
        dfs_collect(node.left) # Traverse left subtree
        dfs_collect(node.right) # Traverse right subtree

    # Collect all values from first tree.
    dfs_collect(root1)

    # Helper function to traverse second tree and check for complement.
    def dfs_check(node):
        if not node:
            return False
        # If the complement exists in the first tree values, return True.
        if target - node.val in values_set:
            return True
        # Recursively check left and right children.
        return dfs_check(node.left) or dfs_check(node.right)

    # Check the second tree.
    return dfs_check(root2)

# Example usage:
# Construct trees and call twoSumBSTs as needed.
```

Python

```python
class BSTIterator:
    def __init__(self, root: TreeNode | None, leftToRight: bool):
      self.stack = []
      self.leftToRight = leftToRight
      self._pushUntilNone(root)

    def hasNext(self) -> bool:
      return len(self.stack) > 0

    def next(self) -> int:
      node = self.stack.pop()
      if self.leftToRight:
        self._pushUntilNone(node.right)
      else:
        self._pushUntilNone(node.left)
      return node.val

    def _pushUntilNone(self, root: TreeNode | None):
      while root:
        self.stack.append(root)
        root = root.left if self.leftToRight else root.right


class Solution:
    def twoSumBSTs(
```

```python
        self,
        root1: TreeNode | None,
        root2: TreeNode | None,
        target: int,
    ) -> bool:
      bst1 = BSTIterator(root1, True)
      bst2 = BSTIterator(root2, False)

      l = bst1.next()
      r = bst2.next()
      while True:
        summ = l + r
        if summ == target:
          return True
        if summ < target:
          if not bst1.hasNext():
            return False
          l = bst1.next()
        else:
          if not bst2.hasNext():
            return False
          r = bst2.next()
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def twoSumBSTs(
        self, root1: Optional[TreeNode], root2: Optional[TreeNode], target: int
    ) -> bool:
        def dfs(root: Optional[TreeNode], i: int):
            if root is None:
                return
            dfs(root.left, i)
            nums[i].append(root.val)
            dfs(root.right, i)

        nums = [[], []]
        dfs(root1, 0)
        dfs(root2, 1)
        i, j = 0, len(nums[1]) - 1
        while i < len(nums[0]) and ~j:
            x = nums[0][i] + nums[1][j]
            if x == target:
                return True
```

```python
            if x < target:
                i += 1
            else:
                j -= 1
        return False
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def twoSumBSTs(
        self, root1: Optional[TreeNode], root2: Optional[TreeNode], target: int
    ) -> bool:
        def dfs(root: Optional[TreeNode], i: int):
            if root is None:
                return
            dfs(root.left, i)
            nums[i].append(root.val)
            dfs(root.right, i)

        nums = [[], []]
        dfs(root1, 0)
        dfs(root2, 1)
        i, j = 0, len(nums[1]) - 1
        while i < len(nums[0]) and ~j:
            x = nums[0][i] + nums[1][j]
            if x == target:
                return True
```



```python
            if x < target:
                i += 1
            else:
                j -= 1
        return False
```

[*] Stack — Pattern Solution (matched from community answers)

Python

```python
class BSTIterator:
    def __init__(self, root: TreeNode | None, leftToRight: bool):
      self.stack = []
      self.leftToRight = leftToRight
      self._pushUntilNone(root)

    def hasNext(self) -> bool:
      return len(self.stack) > 0

    def next(self) -> int:
      node = self.stack.pop()
      if self.leftToRight:
        self._pushUntilNone(node.right)
      else:
        self._pushUntilNone(node.left)
      return node.val

    def _pushUntilNone(self, root: TreeNode | None):
      while root:
        self.stack.append(root)
        root = root.left if self.leftToRight else root.right


class Solution:
    def twoSumBSTs(
```

```python
        self,
        root1: TreeNode | None,
        root2: TreeNode | None,
        target: int,
    ) -> bool:
      bst1 = BSTIterator(root1, True)
      bst2 = BSTIterator(root2, False)

      l = bst1.next()
      r = bst2.next()
      while True:
        summ = l + r
        if summ == target:
          return True
        if summ < target:
          if not bst1.hasNext():
            return False
          l = bst1.next()
        else:
          if not bst2.hasNext():
            return False
          r = bst2.next()
```


#1762 MEDIUM

Buildings With an Ocean View

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Stack · Monotonic Stack

Lists: Premium 98

Problem:

There are n buildings in a line. You are given an integer array heights of size n that represents the heights of the buildings in the line.

The ocean is to the right of the buildings. A building has an ocean view if the building can see the ocean without obstructions. Formally, a building has an ocean view if all the buildings to its right have a smaller height.

Return a list of indices (0-indexed) of buildings that have an ocean view, sorted in increasing order.

Example 1:

```
Input: heights = [4,2,3,1]
Output: [0,2,3]
Explanation: Building 1 (0-indexed) does not have an ocean view because building 2 is taller.
```

Example 2:

```
Input: heights = [4,3,2,1]
Output: [0,1,2,3]
Explanation: All the buildings have an ocean view.
```

Example 3:

```
Input: heights = [1,3,2,4]
Output: [3]
Explanation: Only building 3 has an ocean view.
```

Constraints:

- 1 <= heights.length <= 105
- 1 <= heights[i] <= 109

>> Brute Force [Stack] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findBuildingsWithOceanView(self, heights):
        # Brute Force O(n^2) - simulate without stack via inner loop
        n = len(heights)
        result = [-1] * n
        for i in range(n):
            for j in range(i + 1, n):
                if heights[j] > heights[i]:
                    result[i] = heights[j]
                    break
        return result
```

Python

Python


```python
def findBuildingsWithOceanView(heights):
    n = len(heights)
    result = []
    max_height = 0
    # Iterate from the rightmost building to the leftmost building.
    for i in range(n - 1, -1, -1):
        if heights[i] > max_height:
            result.append(i)
            max_height = heights[i]
    # Reverse the result to return indices in increasing order.
    result.reverse()
    return result

# Example usage:
# heights = [4,2,3,1]
# print(findBuildingsWithOceanView(heights)) # Output: [0,2,3]
```

Python

```python
class Solution:
    def findBuildings(self, heights: list[int]) -> list[int]:
      stack = []

      for i, height in enumerate(heights):
        while stack and heights[stack[-1]] <= height:
          stack.pop()
        stack.append(i)

      return stack
```

Python

```python
class Solution:
    def findBuildings(self, heights: List[int]) -> List[int]:
        ans = []
        mx = 0
        for i in range(len(heights) - 1, -1, -1):
            if heights[i] > mx:
                ans.append(i)
                mx = heights[i]
        return ans[::-1]
```

Python

```python
class Solution:
    def findBuildings(self, heights: List[int]) -> List[int]:
        ans = []
        mx = 0
        for i in range(len(heights) - 1, -1, -1):
            if heights[i] > mx:
                ans.append(i)
                mx = heights[i]
        return ans[::-1]
```

[*] Stack — Pattern Solution (matched from community answers)

Python


```python
class Solution:
    def findBuildings(self, heights: list[int]) -> list[int]:
      stack = []

      for i, height in enumerate(heights):
        while stack and heights[stack[-1]] <= height:
          stack.pop()
        stack.append(i)

      return stack
```

#32 HARD

Longest Valid Parentheses

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Dynamic Programming · Stack

Lists: Grind 169

Problem:

Given a string containing just the characters '(' and ')', return the length of the longest valid (well-formed) parentheses substring.

Example 1:

```
Input: s = "(()"
Output: 2
Explanation: The longest valid parentheses substring is "()".
```

Example 2:

```
Input: s = ")()())"
Output: 4
Explanation: The longest valid parentheses substring is "()()".
```

Example 3:

```
Input: s = ""
Output: 0
```

Constraints:

- 0 <= s.length <= 3 * 104
- s[i] is '(', or ')'.

Python

Python

```python
# Python solution using a stack
def longestValidParentheses(s: str) -> int:
    # Initialize the stack with the base index -1
    stack = [-1]
    max_length = 0

    # Iterate over the string with index
    for i, char in enumerate(s):
        if char == '(':
            # Push the index of '(' onto the stack
            stack.append(i)
        else:
            # Pop the last index for ')'
            stack.pop()
            if not stack:
                # If stack is empty, push current index as the new base
                stack.append(i)
            else:
                # Calculate the length of the current valid substring
                current_length = i - stack[-1]
                max_length = max(max_length, current_length)

    return max_length

# Example usage:

print(longestValidParentheses(")()())")) # Output: 4
```

Python

```python
class Solution:
   def longestValidParentheses(self, s: str) -> int:
      s2 = ')' + s
      # dp[i] := the length of the longest valid parentheses in the substring
      # s2[1:i+1]
      dp = [0] * len(s2)

      for i in range(1, len(s2)):
        if s2[i] == ')' and s2[i - dp[i - 1] - 1] == '(':
          dp[i] = dp[i - 1] + dp[i - dp[i - 1] - 2] + 2

      return max(dp)
```

Python

```python
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        n = len(s)
        f = [0] * (n + 1)
        for i, c in enumerate(s, 1):
            if c == ")":
                if i > 1 and s[i - 2] == "(":
                    f[i] = f[i - 2] + 2
                else:
                    j = i - f[i - 1] - 1
                    if j and s[j - 1] == "(":
                        f[i] = f[i - 1] + 2 + f[j - 1]
        return max(f)
```

Python


```python
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        stack = [-1]
        ans = 0
        for i in range(len(s)):
            if s[i] == '(':
                stack.append(i)
            else:
                stack.pop()
                if not stack:
                    stack.append(i)
                else:
                    ans = max(ans, i - stack[-1])
        return ans
```

Python

```python
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        left = right = 0
        res = 0
        for c in s: # from left to right, '(()' => will never hit left==right
            if c == '(':
                left += 1
            else:
                right += 1
            if left == right:
                res = max(res, 2 * left)
            if left < right:
                left = right = 0

        left = right = 0 # dont forget to reset
        for c in reversed(s): # from right to left, '())' => will never hit left==right
            if c == '(':
                left += 1
            else:
                right += 1
            if left == right:
                res = max(res, 2 * left)
            if left > right: # reverse '<' to '>'
                left = right = 0
```



```python
        return res

######

class Solution:
    def longestValidParentheses(self, s: str) -> int:
        n = len(s)
        if n < 2:
            return 0
        dp = [0] * n
        for i in range(1, n):
            if s[i] == ')':
                if s[i - 1] == '(':
                    dp[i] = 2 + (dp[i - 2] if i > 1 else 0)
                else:
                    j = i - dp[i - 1] - 1
                    if j >= 0 and s[j] == '(':
                        dp[i] = 2 + dp[i - 1] + dp[j - 1]
        return max(dp)

######

...
>>> s="abcdefg"
>>> [print(i,".",c) for i, c in enumerate(s, 1)]
```

```
1 . a
2 . b
3 . c
4 . d
5 . e
6 . f
7 . g
[None, None, None, None, None, None, None]
>>>
...
```

```python
class Solution:
    def longestValidParentheses(s: str) -> int:
        n = len(s)
        f = [0] * (n + 1)
        for i, c in enumerate(s, 1): # starting i from 1, not 0
            if c == ")":
                if i > 1 and s[i - 2] == "(":
                    f[i] = f[i - 2] + 2
                else:
                    j = i - f[i - 1] - 1
                    if j and s[j - 1] == "(":
                        f[i] = f[i - 1] + 2 + f[j - 1]
        return max(f)
```

[*] Stack — Pattern Solution (matched from community answers)

Python


```python
# Python solution using a stack
def longestValidParentheses(s: str) -> int:
    # Initialize the stack with the base index -1
    stack = [-1]
    max_length = 0

    # Iterate over the string with index
    for i, char in enumerate(s):
        if char == '(':
            # Push the index of '(' onto the stack
            stack.append(i)
        else:
            # Pop the last index for ')'
            stack.pop()
            if not stack:
                # If stack is empty, push current index as the new base
                stack.append(i)
            else:
                # Calculate the length of the current valid substring
                current_length = i - stack[-1]
                max_length = max(max_length, current_length)

    return max_length

# Example usage:

print(longestValidParentheses(")()())")) # Output: 4
```

#42 HARD

Trapping Rain Water

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Two Pointers · Dynamic Programming · Stack · Monotonic Stack

Lists: Grind 169

Problem:

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Example 1:

```
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The above elevation map (black section) is represented by array
[0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being
trapped.
```

Example 2:

```
Input: height = [4,2,0,3,2,5]
Output: 9
```

Constraints:

- n == height.length
- 1 <= n <= 2 * 104
- 0 <= height[i] <= 105

>> Brute Force [Stack] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def trap(self, height):
        # Brute Force O(n^2) - for each bar scan left/right for max heights
        n = len(height)
        res = 0
        for i in range(n):
            left_max = max(height[:i + 1])
            right_max = max(height[i:])
            res += min(left_max, right_max) - height[i]
        return res
```

Python

Python


```python
# Python solution using the two pointers approach
def trap(height):
    # Initialize left and right pointers and water accumulator
    left, right = 0, len(height) - 1
    left_max, right_max = 0, 0
    water = 0

    # Traverse the elevation map from both ends
    while left < right:
        # Update left max and right max as we encounter new bars
        if height[left] < height[right]:
            # left side is the determining side
            if height[left] >= left_max:
                left_max = height[left] # update left max since current height is higher
            else:
                water += left_max - height[left] # water trapped at left position
            left += 1 # move left pointer to the right
        else:
            # right side is the determining side
            if height[right] >= right_max:
                right_max = height[right] # update right max since current height is higher
            else:
                water += right_max - height[right] # water trapped at right position
            right -= 1 # move right pointer to the left
    return water


# Example usage:
print(trap([0,1,0,2,1,0,1,3,2,1,2,1])) # Expected output: 6
```

Python

```python
class Solution:
   def trap(self, height: list[int]) -> int:
      n = len(height)
      l = [0] * n # l[i] := max(height[0..i])
      r = [0] * n # r[i] := max(height[i..n])

      for i, h in enumerate(height):
        l[i] = h if i == 0 else max(h, l[i - 1])

      for i, h in reversed(list(enumerate(height))):
        r[i] = h if i == n - 1 else max(h, r[i + 1])

      return sum(min(l[i], r[i]) - h
                 for i, h in enumerate(height))
```

Python

```python
class Solution:
    def trap(self, height: list[int]) -> int:
        if not height:
            return 0

        ans = 0
        l = 0
        r = len(height) - 1
        maxL = height[l]
        maxR = height[r]

        while l < r:
            if maxL < maxR:
                ans += maxL - height[l]
                l += 1
                maxL = max(maxL, height[l])
            else:
                ans += maxR - height[r]
                r -= 1
                maxR = max(maxR, height[r])

        return ans
```

Python

```python
class Solution:
    def trap(self, height: List[int]) -> int:
        n = len(height)
        left = [height[0]] * n
        right = [height[-1]] * n
        for i in range(1, n):
            left[i] = max(left[i - 1], height[i])
            right[n - i - 1] = max(right[n - i], height[n - i - 1])
        return sum(min(l, r) - h for l, r, h in zip(left, right, height))
```

Python


```python
class Solution:
    def trap(self, height: List[int]) -> int:
        n = len(height)
        if n < 3:
            return 0

        lmx = [height[0]] * n
        rmx = [height[n - 1]] * n
        for i in range(1, n):
            lmx[i] = max(lmx[i - 1], height[i]) # i self compare
            rmx[n - 1 - i] = max(rmx[n - i], height[n - 1 - i])

        # no negative, worst is, min(left,right) is itself
        return sum( min(lmx[i], rmx[i]) - height[i] for i in range(n) )

######

class Solution: # zip
    def trap(self, height: List[int]) -> int:
        n = len(height)
        left = [height[0]] * n
        right = [height[-1]] * n
        for i in range(1, n):
            left[i] = max(left[i - 1], height[i])
            right[n - i - 1] = max(right[n - i], height[n - i - 1])

        return sum(min(l, r) - h for l, r, h in zip(left, right, height))

######

class Solution: # find the highest bar first
    def trap(self, height: List[int]) -> int:
        if not height:
            return 0

        total_water = 0
        max_index = 0
        max_height = float('-inf')

        # Find the index of the maximum height
        for i in range(len(height)):
            if height[i] > max_height:
                max_height = height[i]
                max_index = i

        # Calculate water trapped on the left side of the maximum height
        left_max = height[0]
        for i in range(1, max_index):
            if left_max < height[i]:
                left_max = height[i]
            total_water += left_max - height[i]
```



```python
        # Calculate water trapped on the right side of the maximum height
        right_max = height[-1]
        for i in range(len(height) - 2, max_index, -1):
            if right_max < height[i]:
                right_max = height[i]
            total_water += right_max - height[i]

        return total_water
```

[*] Stack — Pattern Solution (stored optimal)

Python

```python
# Python Solution using the two pointers approach
def trap(height):
    # Initialize left and right pointers and water accumulator
    left, right = 0, len(height) - 1
    left_max, right_max = 0, 0
    water = 0

    # Traverse the elevation map from both ends
    while left < right:
        # Update left_max and right_max as we encounter new bars
        if height[left] < height[right]:
            # left side is the determining side
            if height[left] >= left_max:
                left_max = height[left] # update left max since current height is higher
            else:
                water += left_max - height[left] # water trapped at left position
            left += 1 # move left pointer to the right
        else:
            # right side is the determining side
            if height[right] >= right_max:
                right_max = height[right] # update right max since current height is higher
            else:
                water += right_max - height[right] # water trapped at right position
            right -= 1 # move right pointer to the left
    return water

# Example usage:
print(trap([0,1,0,2,1,0,1,3,2,1,2,1])) # Expected output: 6
```

#84 HARD

Largest Rectangle in Histogram

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

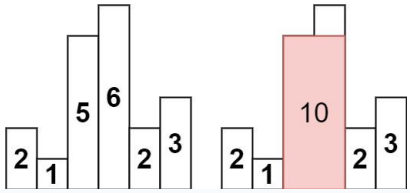
Array · Stack · Monotonic Stack

Lists: Grind 169

Problem:

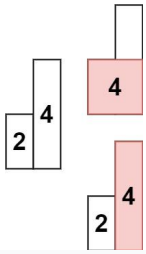
Given an array of integers heights representing the histogram's bar height where the width of each bar is 1, return the area of the largest rectangle in the histogram.

Example 1:




```
Input: heights = [2,1,5,6,2,3]
Output: 10
Explanation: The above is a histogram where width of each bar is 1.
The largest rectangle is shown in the red area, which has an area = 10 units.
```

Example 2:




```
Input: heights = [2,4]
Output: 4
```

Constraints:

- 1 <= heights.length <= 105

- 0 <= heights[i] <= 104

>> Brute Force [Stack] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def largestRectangleArea(self, heights):
        # Brute Force O(n^2) - fix left bar, expand right, track min height
        n = len(heights)
        res = 0
        for i in range(n):
            min_h = heights[i]
            for j in range(i, n):
                min_h = min(min_h, heights[j])
                res = max(res, min_h * (j - i + 1))
        return res
```

Python

Python

```python
# Python solution

def largestRectangleArea(heights):
    # Append a zero to flush out the remaining bars in the stack
    heights.append(0)
    stack = [] # Stack to store indices of bars
    max_area = 0

    # Iterate over each index and bar height
    for i, h in enumerate(heights):
        # Process bars that are taller than the current bar
        while stack and heights[stack[-1]] > h:
            height = heights[stack.pop()] # Height of the bar at the index popped from the stack
            # Calculate width: if stack is empty, width is the current index; otherwise, it's the
difference
            width = i if not stack else i - stack[-1] - 1
            max_area = max(max_area, height * width)
        stack.append(i)

    return max_area

# Example usage:
if __name__ == "__main__":
    histogram = [2, 1, 5, 6, 2, 3]
    print(largestRectangleArea(histogram))
```

Python


```python
class Solution:
    def largestRectangleArea(self, heights: list[int]) -> int:
        ans = 0
        stack = []

        for i in range(len(heights) + 1):
            while stack and (i == len(heights) or heights[stack[-1]] > heights[i]):
                h = heights[stack.pop()]
                w = i - stack[-1] - 1 if stack else i
                ans = max(ans, h * w)
            stack.append(i)

        return ans
```

Python

```python
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        n = len(heights)
        stk = []
        left = [-1] * n
        right = [n] * n
        for i, h in enumerate(heights):
            while stk and heights[stk[-1]] >= h:
                right[stk[-1]] = i
                stk.pop()
            if stk:
                left[i] = stk[-1]
            stk.append(i)
        return max(h * (right[i] - left[i] - 1) for i, h in enumerate(heights))
```

Python

```python
'''
□□□□□□□□□□□□ h □□□□□□□□□□□□□
```

```python
class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        if not heights:
            return 0
        heights.append(-1) # make for loop running
        stack = []
        ans = 0
        for i in range(0, len(heights)):
            while stack and heights[i] < heights[stack[-1]]:
                currentLowestBarIndex = stack.pop()
                h = heights[currentLowestBarIndex]
                # stack[-1] is after pop!). why not i-currentLowestBarIndex?
                # because not working for tie case in the stack,
                # eg. for input=[4,2,6,3,2,5], expected=6 but output=5
                # □□□□
```

Problem:
You are given an array of integers nums, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Example 1:

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:
Window position Max
--------------- -----
[1 3 -1] -3 5 3 6 7 3
 1 [3 -1 -3] 5 3 6 7 3
 1 3 [-1 -3 5] 3 6 7 5
 1 3 -1 [-3 5 3] 6 7 5
 1 3 -1 -3 [5 3 6] 7 6
 1 3 -1 -3 5 [3 6 7] 7
```

Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

Constraints:

- 1 <= nums.length <= 105
- -104 <= nums[i] <= 104
- 1 <= k <= nums.length

>> Brute Force [Stack] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def maxSlidingWindow(self, nums, k):
        # Brute Force O(n^2) - simulate without stack via inner loop
        n = len(nums)
        result = [-1] * n
        for i in range(n):
            for j in range(i + 1, n):
                if nums[j] > nums[i]:
                    result[i] = nums[j]
                    break
        return result
```

Python

Python


```python
from collections import deque

def maxSlidingWindow(nums, k):
    # Create a deque to store indices of elements in the current window
    deq = deque()
    result = []

    for i, num in enumerate(nums):
        # Remove indices from the front if they are out of the current window
        if deq and deq[0] == i - k:
            deq.popleft()

        # Remove elements from the back while the current number is larger
        # than the number at the indices stored in the deque
        while deq and nums[deq[-1]] < num:
            deq.pop()

        # Append the current index to the deque
        deq.append(i)

        # If we've hit the window size, append the current max to result
        if i >= k - 1:
            result.append(nums[deq[0]])

    return result


# Example usage:
# print(maxSlidingWindow([1,3,-1,-3,5,3,6,7], 3))
```

Python

```python
class Solution:
  def maxSlidingWindow(self, nums: list[int], k: int) -> list[int]:
    ans = []
    maxQ = collections.deque()

    for i, num in enumerate(nums):
      while maxQ and maxQ[-1] < num:
        maxQ.pop()
      maxQ.append(num)
      if i >= k and nums[i - k] == maxQ[0]: # out-of-bounds
        maxQ.popleft()
      if i >= k - 1:
        ans.append(maxQ[0])

    return ans
```

Python


```python
class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        q = [(-v, i) for i, v in enumerate(nums[: k - 1])]
        heapify(q)
        ans = []
        for i in range(k - 1, len(nums)):
            heappush(q, (-nums[i], i))
            while q[0][1] <= i - k:
                heappop(q)
            ans.append(-q[0][0])
        return ans
```

Python

```python
from collections import deque

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        q = deque()
        ans = []
        for i, v in enumerate(nums):
            if q and i - k + 1 > q[0]:
                q.popleft() # remove index if out of window left
            while q and nums[q[-1]] <= v: # '<=', not '<', to ensure the bigger index stored in deque
                q.pop()
            q.append(i)
            if i >= k - 1:
                ans.append(nums[q[0]])
        return ans

###########

class Solution(object):
  def maxSlidingWindow(self, nums, k):
    """
    :type nums: List[int]
    :type k: int
    :rtype: List[int]
    """
```



```python
    if k == 0:
      return []
    ans = [0 for _ in range(len(nums) - k + 1)]
    stack = collections.deque([])
    for i in range(0, k):
      while stack and nums[stack[-1]] < nums[i]:
        stack.pop()
      stack.append(i)
    ans[0] = nums[stack[0]]
    idx = 0
    for i in range(k, len(nums)):
      idx += 1
      if stack and stack[0] == i - k:
        stack.popleft()
      while stack and nums[stack[-1]] < nums[i]:
        stack.pop()
      stack.append(i)
      ans[idx] = nums[stack[0]]

    return ans
```

Python

```python
from collections import deque

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        q = deque()
        ans = []
        for i, v in enumerate(nums):
            if q and i - k + 1 > q[0]:
                q.popleft() # remove index if out of window left
            while q and nums[q[-1]] <= v: # '<=', not '<', to ensure the bigger index stored in deque
                q.pop()
            q.append(i)
            if i >= k - 1:
                ans.append(nums[q[0]])
        return ans

###########

class Solution(object):
  def maxSlidingWindow(self, nums, k):
    """
    :type nums: List[int]
    :type k: int
    :rtype: List[int]
    """
```

[*] Stack — Pattern Solution (matched from community answers)

Python


```python
    if k == 0:
      return []
    ans = [0 for _ in range(len(nums) - k + 1)]
    stack = collections.deque([])
    for i in range(0, k):
      while stack and nums[stack[-1]] < nums[i]:
        stack.pop()
      stack.append(i)
    ans[0] = nums[stack[0]]
    idx = 0
    for i in range(k, len(nums)):
      idx += 1
      if stack and stack[0] == i - k:
        stack.popleft()
      while stack and nums[stack[-1]] < nums[i]:
        stack.pop()
      stack.append(i)
      ans[idx] = nums[stack[0]]

    return ans
```

#716 HARD

Max Stack

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Stack · Design · Doubly-Linked List · Ordered Set

Lists: Denny Zhang

Problem:
Design a max stack data structure that supports the stack operations and supports finding the stack's maximum element.

Implement the MaxStack class:

- MaxStack() Initializes the stack object.
- void push(int x) Pushes element x onto the stack.
- int pop() Removes the element on top of the stack and returns it.
- int top() Gets the element on the top of the stack without removing it.
- int peekMax() Retrieves the maximum element in the stack without removing it.
- int popMax() Retrieves the maximum element in the stack and removes it. If there is more than one maximum element, only remove the top-most one.

You must come up with a solution that supports O(1) for each top call and O(logn) for each other call.

Example 1:

```
Input
["MaxStack", "push", "push", "push", "top", "popMax", "top", "peekMax", "pop", "top"]
[[], [5], [1], [5], [], [], [], [], [], []]
Output
[null, null, null, null, 5, 5, 1, 5, 1, 5]

Explanation
MaxStack stk = new MaxStack();
```



```
stk.push(5); // [5] the top of the stack and the maximum number is 5.
stk.push(1); // [5, 1] the top of the stack is 1, but the maximum is 5.
stk.push(5); // [5, 1, 5] the top of the stack is 5, which is also the maximum, because it is
the top most one.
stk.top(); // return 5, [5, 1, 5] the stack did not change.
stk.popMax(); // return 5, [5, 1] the stack is changed now, and the top is different from the
max.
stk.top(); // return 1, [5, 1] the stack did not change.
stk.peekMax(); // return 5, [5, 1] the stack did not change.
stk.pop(); // return 1, [5] the top of the stack and the max element is now 5.
stk.top(); // return 5, [5] the stack did not change.
```

Constraints:

- -107 <= x <= 107
- At most 105 calls will be made to push, pop, top, peekMax, and popMax.
- There will be at least one element in the stack when pop, top, peekMax, or popMax is called.

Python

Python

```python
# Python solution implementing the MaxStack data structure
# We use collections.OrderedDict for demonstration.
# However, in Python we do not have a built in balanced BST;
# one can use "sortedcontainers" or manually maintain a heap and dictionary.
# For clarity, we simulate an ordered map with a TreeMap-like structure using sorted list of keys.
# In production, one may use "SortedDict" from sortedcontainers.

from collections import defaultdict

class Node:
    def __init__(self, val):
        self.val = val
        self.prev = None
        self.next = None

class DoubleLinkedList:
    def __init__(self):
        # Create dummy head and tail nodes.
        self.head = Node(0)
        self.tail = Node(0)
        self.head.next = self.tail
        self.tail.prev = self.head

    def add_last(self, node):
        # Add node right before the tail.
```

```python
        node.prev = self.tail.prev
        node.next = self.tail
        self.tail.prev.next = node
        self.tail.prev = node

    def pop(self, node=None):
        # If no node is provided, pop from the end.
        if not node:
            node = self.tail.prev
        node.prev.next = node.next
        node.next.prev = node.prev
        return node

    def top(self):
        # Return the last node's value.
        return self.tail.prev.val

class MaxStack:
    def __init__(self):
        self.dll = DoubleLinkedList()
        # Dictionary mapping value to list of nodes
        self.valToNodes = defaultdict(list)
        # Sorted list to simulate ordered set of keys.
        self.sortedKeys = []

    def push(self, x: int) -> None:
        node = Node(x)
        self.dll.add_last(node)
        self.valToNodes[x].append(node)
        # Insert into sortedKeys if new, otherwise maintain count.
        if len(self.valToNodes[x]) == 1:
            # Binary insertion into sortedKeys
            lo, hi = 0, len(self.sortedKeys)
            while lo < hi:
                mid = (lo + hi) // 2
                if self.sortedKeys[mid] < x:
                    lo = mid + 1
                else:
                    hi = mid
            self.sortedKeys.insert(lo, x)

    def pop(self) -> int:
        node = self.dll.pop()
        x = node.val
        self.valToNodes[x].pop()
        if not self.valToNodes[x]:
            # Remove x from sortedKeys
            self.sortedKeys.remove(x)
        return x

    def top(self) -> int:
        return self.dll.top()

    def peekMax(self) -> int:
        # The maximum is the last element in sortedKeys.
```

Python


```python
class MaxStack(object):

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.stack = []
        self.max_stack = []
        # or the same trick in min-stack
        # self.max_stack = [-math.inf]

    def push(self, x):
        """
        :type x: int
        :rtype: void
        """
        self.stack.append(x)
        self.max_stack.append(max(x, self.max_stack[-1]) if self.max_stack else x)

    def pop(self):
        """
        :rtype: int
        """
        if len(self.stack) != 0:
            self.max_stack.pop(-1)

        return self.stack.pop(-1)

    def top(self):
        """
        :rtype: int
        """
        return self.stack[-1] if self.stack else None

    def peekMax(self):
        """
        :rtype: int
        """
        if len(self.max_stack) != 0:
            return self.max_stack[-1]

    def popMax(self):
        """
        :rtype: int
        """
        val = self.peekMax()
        buff = []
        while self.top() != val:
            # re-use pop(), ops on both max_stack and stack
            buff.append(self.pop())
        self.pop()
```



```python
        while len(buff) != 0:
            # re-use push(), no need to save buffer for max-stack
            self.push(buff.pop(-1))
        return val

##########

from sortedcontainers import SortedList

class Node:
    def __init__(self, val=0):
        self.val = val
        self.prev: Union[Node, None] = None
        self.next: Union[Node, None] = None


class DoubleLinkedList:
    def __init__(self):
        self.head = Node()
        self.tail = Node()
        self.head.next = self.tail
        self.tail.prev = self.head

    def append(self, val) -> Node:
        node = Node(val)

        node.next = self.tail
        node.prev = self.tail.prev
        self.tail.prev = node
        node.prev.next = node
        return node
```

[*] Stack — Pattern Solution (matched from community answers)

Python


```python
class MaxStack(object):

    def __init__(self):
        """
        initialize your data structure here.
        """
        self.stack = []
        self.max_stack = []
        # or the same trick in min-stack
        # self.max_stack = [-math.inf]

    def push(self, x):
        """
        :type x: int
        :rtype: void
        """
        self.stack.append(x)
        self.max_stack.append(max(x, self.max_stack[-1]) if self.max_stack else x)

    def pop(self):
        """
        :rtype: int
        """
        if len(self.stack) != 0:
            self.max_stack.pop(-1)

        return self.stack.pop(-1)

    def top(self):
        """
        :rtype: int
        """
        return self.stack[-1] if self.stack else None

    def peekMax(self):
        """
        :rtype: int
        """
        if len(self.max_stack) != 0:
            return self.max_stack[-1]

    def popMax(self):
        """
        :rtype: int
        """
        val = self.peekMax()
        buff = []
        while self.top() != val:
            # re-use pop(), ops on both max_stack and stack
            buff.append(self.pop())
        self.pop()
```



```python
        while len(buff) != 0:
            # re-use push(), no need to save buffer for max-stack
            self.push(buff.pop(-1))
        return val

##########

from sortedcontainers import SortedList

class Node:
    def __init__(self, val=0):
        self.val = val
        self.prev: Union[Node, None] = None
        self.next: Union[Node, None] = None


class DoubleLinkedList:
    def __init__(self):
        self.head = Node()
        self.tail = Node()
        self.head.next = self.tail
        self.tail.prev = self.head

    def append(self, val) -> Node:
        node = Node(val)

        node.next = self.tail
        node.prev = self.tail.prev
        self.tail.prev = node
        node.prev.next = node
        return node
```


#772 HARD

Basic Calculator III

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Math · String · Stack · Recursion

Lists: Premium 98

Problem:

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only non-negative integers, '+', '-', '*', '/' operators, and open '(' and closing parentheses ')'. The integer division should truncate toward zero.

You may assume that the given expression is always valid. All intermediate results will be in the range of [−231, 231 − 1].

Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as eval().

Example 1:

```
Input: s = "1+1"
Output: 2
```

Example 2:


```
Input: s = "6-4/2"
Output: 4
```

Example 3:

```
Input: s = "2*(5+5*2)/3+(6/2+8)"
Output: 21
```

Constraints:

- 1 <= s <= 104
- s consists of digits, '+', '-', '*', '/', '(', and ')'.
- s is a valid expression.

Python

Python

```python
def calculate(s: str) -> int:
    # Recursive helper function that processes the expression starting from index 'i'
    def helper(i):
        stack = []
        num = 0
        sign = '+'
        while i < len(s):
            char = s[i]
            # Build the current number if the character is a digit
            if char.isdigit():
                num = num * 10 + int(char)
            # If '(' is encountered, evaluate the subexpression recursively
            if char == '(':
                num, i = helper(i + 1)
            # When encountering an operator or closing parenthesis, process the sign and number
            if (not char.isdigit() and char != ' ') or i == len(s) - 1:
                if sign == '+':
                    stack.append(num)
                elif sign == '-':
                    stack.append(-num)
                elif sign == '*':
                    prev = stack.pop()
                    stack.append(prev * num)
                elif sign == '/':
                    prev = stack.pop()

                    stack.append(int(prev / num)) # Truncate toward zero
                sign = char
                num = 0
            # If a closing parenthesis is reached, end this level of recursion
            if char == ')':
                return sum(stack), i
            i += 1
        return sum(stack), i

    result, _ = helper(0)
    return result

# Example test
print(calculate("2*(5+5*2)/3+(6/2+8)"))
```

Python

```python
class Solution:
    def calculate(self, s: str) -> int:
        nums = []
        ops = []

        def calc():
            b = nums.pop()
            a = nums.pop()
            op = ops.pop()
            if op == '+':
                nums.append(a + b)
            elif op == '-':
                nums.append(a - b)
            elif op == '*':
                nums.append(a * b)
            else: # op == '/'
                nums.append(int(a / b))

        def precedes(prev: str, curr: str) -> bool:
            """
            Returns True if the previous character is a operator and the priority of
            the previous operator >= the priority of the current character (operator).
            """
            if prev == '(':
                return False

            return prev in '*/' or curr in '+-'

        i = 0
        hasPrevNum = False

        while i < len(s):
            c = s[i]
            if c.isdigit():
                num = int(c)
                while i + 1 < len(s) and s[i + 1].isdigit():
                    num = num * 10 + int(s[i + 1])
                    i += 1
                nums.append(num)
                hasPrevNum = True
            elif c == '(':
                ops.append('(')
                hasPrevNum = False
            elif c == ')':
                while ops[-1] != '(':
                    calc()
                ops.pop() # Pop '('
            elif c in '+-*/':
                if not hasPrevNum: # Handle input like "-1-(-1)"
                    num.append(0)
                while ops and precedes(ops[-1], c):
```



```python
                calc()
                ops.append(c)
            i += 1

        while ops:
            calc()

        return nums.pop()
```

Python

```python
class Solution:
    def calculate(self, s: str) -> int:
        def dfs(q):
            num, sign, stk = 0, "+", []
            while q:
                c = q.popleft()
                if c.isdigit():
                    num = num * 10 + int(c)
                if c == "(":
                    num = dfs(q)
                if c in "+-*/)" or not q:
                    match sign:
                        case "+":
                            stk.append(num)
                        case "-":
                            stk.append(-num)
                        case "*":
                            stk.append(stk.pop() * num)
                        case "/":
                            stk.append(int(stk.pop() / num))
                    num, sign = 0, c
                if c == ")":
                    break
            return sum(stk)

        return dfs(deque(s))
```

Python


```python
# good one, based on solution of Basic Calculator II
class Solution: # dfs recursion
    def calculate(self, s: str) -> int:
        stack = []
        num = 0
        op = "+"
        i = 0
        while i < len(s):
            c = s[i]
            if c.isdigit():
                num = num * 10 + int(c)
            elif c == "(":
                j = self.findClosing(s, i)
                num = self.calculate(s[i+1:j]) # j is ')'
                i = j # get if, i will +1 below
            if c in "+-*/" or i == len(s) - 1:
                if op == "+":
                    stack.append(num)
                elif op == "-":
                    stack.append(-num)
                elif op == "*":
                    stack[-1] += num
                else:
                    stack[-1] = int(stack[-1] / num)
                op = c

                num = 0
            i += 1
        return sum(stack)

    def findClosing(self, s: str, i: int) -> int:
        level = 0
        while i < len(s):
            if s[i] == "(":
                level += 1
            elif s[i] == ")":
                level -= 1
                if level == 0:
                    return i
            i += 1
        return i

############


class Solution: # iterative
    def calculate(self, s: str) -> int:
        def evaluate_expression(stack):
            res = stack.pop() if stack else 0
            # Evaluate the expression till we get '(' or empty stack
```



```python
            while stack and stack[-1] != '(':
                sign = stack.pop()
                if sign == '+':
                    res += stack.pop()
                elif sign == '-':
                    res -= stack.pop()
                elif sign == '*':
                    res *= stack.pop()
                elif sign == '/':
                    res //= stack.pop()
            return res

        stack = [] # mix of num and operator
        n = len(s)
        i = 0
        while i < n:
            if s[i].isdigit():
                # Get the complete number and push it to the stack
                j = i
                while j < n and s[j].isdigit():
                    j += 1
                stack.append(int(s[i:j]))
                i = j - 1
            elif s[i] in '+-*/(':
                stack.append(s[i])

            elif s[i] == ')':
                # Evaluate the expression within the parentheses
                res = evaluate_expression(stack)
                stack.pop() # Remove the opening '('
                stack.append(res)
```

[*] Stack — Pattern Solution (matched from community answers)

Python


```python
def calculate(s: str) -> int:
    # Recursive helper function that processes the expression starting from index 'i'
    def helper(i):
        stack = []
        num = 0
        sign = '+'
        while i < len(s):
            char = s[i]
            # Build the current number if the character is a digit
            if char.isdigit():
                num = num * 10 + int(char)
            # If '(' is encountered, evaluate the subexpression recursively
            if char == '(':
                num, i = helper(i + 1)
            # When encountering an operator or closing parenthesis, process the sign and number
            if (not char.isdigit() and char != ' ') or i == len(s) - 1:
                if sign == '+':
                    stack.append(num)
                elif sign == '-':
                    stack.append(-num)
                elif sign == '*':
                    prev = stack.pop()
                    stack.append(prev * num)
                elif sign == '/':
                    prev = stack.pop()

                    stack.append(int(prev / num)) # Truncate toward zero
                sign = char
                num = 0
            # If a closing parenthesis is reached, end this level of recursion
            if char == ')':
                return sum(stack), i
            i += 1
        return sum(stack), i

    result, _ = helper(0)
    return result

# Example test
print(calculate("2+(5+5*2)/3+(6/2+8)"))
```

#895 HARD

Maximum Frequency Stack

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · Stack · Design

Lists: Grind 169

Problem:

Design a stack-like data structure to push elements to the stack and pop the most frequent element from the stack.

Implement the FreqStack class:

- FreqStack() constructs an empty frequency stack.
- void push(int val) pushes an integer val onto the top of the stack.

- int pop() removes and returns the most frequent element in the stack. If there is a tie for the most frequent element, the element closest to the stack's top is removed and returned.

Example 1:

```
Input
["FreqStack", "push", "push", "push", "push", "push", "push", "pop", "pop", "pop", "pop"]
[[], [5], [7], [5], [7], [4], [5], [], [], [], []]
Output
[null, null, null, null, null, null, null, 5, 7, 5, 4]

Explanation
FreqStack freqStack = new FreqStack();
freqStack.push(5); // The stack is [5]
freqStack.push(7); // The stack is [5,7]
freqStack.push(5); // The stack is [5,7,5]
freqStack.push(7); // The stack is [5,7,5,7]
freqStack.push(4); // The stack is [5,7,5,7,4]
freqStack.push(5); // The stack is [5,7,5,7,4,5]
freqStack.pop(); // return 5, as 5 is the most frequent. The stack becomes [5,7,5,7,4].
freqStack.pop(); // return 7, as 5 and 7 is the most frequent, but 7 is closest to the top.
The stack becomes [5,7,5,4].
freqStack.pop(); // return 5, as 5 is the most frequent. The stack becomes [5,7,4].
freqStack.pop(); // return 4, as 4, 5 and 7 is the most frequent, but 4 is closest to the
top. The stack becomes [5,7].
```

Constraints:

- 0 <= val <= 109
- At most 2 * 104 calls will be made to push and pop.
- It is guaranteed that there will be at least one element in the stack before calling pop.

Python

Python

```python
# Python implementation of FreqStack

class FreqStack:
    def __init__(self):
        # Dictionary mapping value to its frequency
        self.freq = {}
        # Dictionary mapping frequency to a list (stack) of values
        self.group = {}
        # Variable tracking the current maximum frequency in the stack
        self.maxfreq = 0

    def push(self, val: int) -> None:
        # Update frequency count for the value
        f = self.freq.get(val, 0) + 1
        self.freq[val] = f
        # Update the maximum frequency if necessary
        if f > self.maxfreq:
            self.maxfreq = f
        # Append the value to the appropriate frequency group
        if f not in self.group:
            self.group[f] = []
        self.group[f].append(val)

    def pop(self) -> int:
        # Remove the value from the stack of the current maximum frequency

        val = self.group[self.maxfreq].pop()
        # Decrement the frequency count for the value
        self.freq[val] -= 1
        # If no more elements have the current max frequency, decrease maxfreq
        if not self.group[self.maxfreq]:
            self.maxfreq -= 1
        return val
```

Python

```python
class FreqStack:
  def __init__(self):
    self.maxFreq = 0
    self.count = collections.Counter()
    self.countToStack = collections.defaultdict(list)

  def push(self, val: int) -> None:
    self.count[val] += 1
    self.countToStack[self.count[val]].append(val)
    self.maxFreq = max(self.maxFreq, self.count[val])

  def pop(self) -> int:
    val = self.countToStack[self.maxFreq].pop()
    self.count[val] -= 1
    if not self.countToStack[self.maxFreq]:
      self.maxFreq -= 1
    return val
```

Python


```python
class FreqStack:
    def __init__(self):
        self.cnt = defaultdict(int)
        self.q = []
        self.ts = 0

    def push(self, val: int) -> None:
        self.ts += 1
        self.cnt[val] += 1
        heappush(self.q, (-self.cnt[val], -self.ts, val))

    def pop(self) -> int:
        val = heappop(self.q)[2]
        self.cnt[val] -= 1
        return val


# Your FreqStack object will be instantiated and called as such:
# obj = FreqStack()
# obj.push(val)
# param_2 = obj.pop()
```

Python

```python
class FreqStack:
    def __init__(self):
        self.cnt = defaultdict(int)
        self.d = defaultdict(list)
        self.mx = 0

    def push(self, val: int) -> None:
        self.cnt[val] += 1
        self.d[self.cnt[val]].append(val)
        self.mx = max(self.mx, self.cnt[val])

    def pop(self) -> int:
        val = self.d[self.mx].pop()
        self.cnt[val] -= 1
        if not self.d[self.mx]:
            self.mx -= 1
        return val


# Your FreqStack object will be instantiated and called as such:
# obj = FreqStack()
# obj.push(val)
# param_2 = obj.pop()
```

[*] Stack — Pattern Solution (stored optimal)

Python


```python
# Python implementation of FreqStack

class FreqStack:
    def __init__(self):
        # Dictionary mapping value to its frequency
        self.freq = {}
        # Dictionary mapping frequency to a list (stack) of values
        self.group = {}
        # Variable tracking the current maximum frequency in the stack
        self.maxfreq = 0

    def push(self, val: int) -> None:
        # Update frequency count for the value
        f = self.freq.get(val, 0) + 1
        self.freq[val] = f
        # Update the maximum frequency if necessary
        if f > self.maxfreq:
            self.maxfreq = f
        # Append the value to the appropriate frequency group
        if f not in self.group:
            self.group[f] = []
        self.group[f].append(val)

    def pop(self) -> int:
        # Remove the value from the stack of the current maximum frequency

        val = self.group[self.maxfreq].pop()
        # Decrement the frequency count for the value
        self.freq[val] -= 1
        # If no more elements have the current max frequency, decrease maxfreq
        if not self.group[self.maxfreq]:
            self.maxfreq -= 1
        return val
```


Quick Review — Stack 25 questions · insights · complexity · solution

#20 Valid Parentheses [Easy]

Key Insights

- The problem can be solved using a stack data structure.
- When encountering an opening bracket, push it onto the stack.
- When encountering a closing bracket, check if the top of the stack is its matching opening bracket.
- If the stack is empty before finding a match or if there is a mismatch, the string is invalid.
- At the end, the string is valid only if the stack is empty, meaning all brackets were properly matched.

Space & Time

Time Complexity: O(n), where n is the length of the string, since we process each character once.

Space Complexity: O(n) in the worst case for the stack when all characters are opening brackets.

Solution

We use a stack to check for valid parentheses. The algorithm iterates through the string, pushing opening brackets onto the stack and popping them when a corresponding closing bracket is encountered. A mapping of closing to opening brackets helps in validating the matching pairs. By ensuring that the stack is empty at the end, we confirm that every opening bracket was properly closed.

#232 Implement Queue using Stacks [Easy]

Key Insights

- Use two stacks:
- One (input stack) for enqueue operations.
- The other (output stack) for dequeue operations.
- When performing pop or peek, if the output stack is empty, transfer all elements from the input stack to the output stack. This reversal makes the oldest element accessible.
- Each element is moved at most once between the stacks, ensuring an amortized O(1) cost per operation.
- The space complexity is O(n) where n is the number of elements in the queue.

Space & Time

Time Complexity: Amortized O(1) per operation

Space Complexity: O(n)

Solution

We maintain two stacks: an input stack for pushing new elements and an output stack for popping or peeking the front element of the queue. When we need to access the front of the queue (either during a pop or peek), we check if the output stack is empty. If it is, we transfer all elements from the input stack to the output stack. This reversal of order allows us to simulate the FIFO nature of the queue. When performing a push, the element is simply added to the input stack.

#844 Backspace String Compare [Easy]

Key Insights

- '#' represents a backspace operation that deletes the previous character.
- A direct simulation using a stack can reconstruct the final string.
- A two-pointer approach allows comparison from the end of the strings while skipping backspaced characters.
- The two-pointer technique can achieve O(n) time and O(1) space by processing the strings in reverse.

Space & Time

Time Complexity: O(n) where n is the length of the longer string.

Space Complexity: O(1) using the two-pointer strategy (O(n) if using stacks).

Solution

The solution uses two pointers starting from the end of each string. For each string, maintain a skip counter to account for backspace characters. As you iterate backwards:

- If a '#' is encountered, increment the skip counter. - If a normal character is encountered while the skip counter is positive, skip that character by decrementing the counter. - Once a valid character is found, compare the characters from both strings. If all corresponding characters match and both pointers finish processing at the same time, the strings are considered equal.

#143 Reorder List [Medium]

Key Insights

- Find the middle of the linked list using the fast and slow pointer technique.
- Reverse the second half of the list.
- Merge the two halves, alternating nodes from the first half and the reversed second half.
- The challenge is done in-place, ensuring a linear time solution with constant extra space.

Space & Time

Time Complexity: O(n) - Each of the three main steps (finding the middle, reversing, merging) takes O(n) time.

Space Complexity: O(1) - The algorithm is done in-place with only a few pointers.

Solution

The solution involves three key steps:

- Use two pointers (slow and fast) to identify the middle of the list. - Reverse the second half of the list. This can be done iteratively by adjusting the next pointers. - Merge the two lists: One starting from the head and the other from the head of the reversed second half. Interleave the nodes by alternating between the two lists until complete. This approach uses in-place pointer manipulation, thus ensuring no extra space is used aside from a few temporary variables.

#150 Evaluate Reverse Polish Notation [Medium]

Key Insights

- Use a stack to process the tokens.
- Push numbers onto the stack.
- When an operator is encountered, pop the top two numbers, perform the operation, and push the result back.
- Handle division carefully to ensure it truncates toward zero.
- The final result remains as the only value in the stack.

Space & Time

Time Complexity: O(n), where n is the number of tokens (each token is processed once).

Space Complexity: O(n), in the worst-case scenario where all tokens are numbers and are stored in the stack.

Solution

We use a stack data structure to evaluate the RPN expression. Iterate through each token:

- If the token is a number, convert and push it onto the stack. - If the token is an operator, pop the top two numbers (be careful about order), perform the corresponding arithmetic operation, and push the result back onto the stack. - At the end of processing, the stack contains the result of the expression. Special attention is needed for division to ensure it truncates toward zero (using language-specific functions when necessary).

#155 Min Stack [Medium]

Key Insights

- Use an auxiliary data structure to track the current minimum element.

- A common approach is to use two stacks: one regular stack for all elements and a second stack to keep track of the minimum values.
- When pushing a new element, push it onto the main stack and also update the min stack if the element is less than or equal to the current minimum.
- When popping, if the popped element is equal to the top of the min stack, pop from the min stack as well.
- This design guarantees that all operations (push, pop, top, getMin) run in constant time, O(1).

Space & Time

Time Complexity: O(1) per operation (push, pop, top, getMin)

Space Complexity: O(n), where n is the number of elements in the stack, since an additional stack is used to keep track of minima.

Solution

The solution involves using two stacks:

- Main Stack: Stores all the elements. - Min Stack: Stores the minimum elements encountered so far.

When an element is pushed, we compare it with the top of the min stack. If it is smaller than or equal to the current minimum, we also push it onto the min stack. For the pop operation, if the element being popped is the same as the top of the min stack, we pop from the min stack as well. This approach ensures that the getMin operation always returns the top element of the min stack, which is the current minimum in constant time.

#173 Binary Search Tree Iterator [Medium]

Key Insights

- Use an iterative in-order traversal strategy with a stack.
- In-order traversal for BST yields sorted order.
- Pre-load the stack with all left descendants from the root.
- After returning a node, process its right subtree by pushing its left branch.
- This method maintains an average time complexity of O(1) per operation and uses O(h) memory, where h is the tree height.

Space & Time

Time Complexity: O(1) on average per operation (amortized)

Space Complexity: O(h), where h is the height of the tree

Solution

We use a stack to simulate an in-order traversal. The process involves initially pushing all left-child nodes from the root into the stack. For each call to next(), we pop the top element (the current smallest node) and then push all the left descendents of the node's right child. This ensures that the next smallest element is always at the top of the stack. The hasNext() function then simply checks if the stack is non-empty.

#227 Basic Calculator II [Medium]

Key Insights

- The expression can be processed in one pass while using a stack to handle operator precedence.
- When encountering '+' or '-' operators, the current number is directly added or subtracted (as a positive or negative value).
- For '*' and '/' operators, compute immediately with the last number from the stack and update it.
- Handle spaces and multi-digit numbers by building the number while iterating over the string.
- Division truncation toward zero must be carefully applied, which aligns with typical integer division in most languages.

Space & Time

Time Complexity: O(n) where n is the length of the expression, since the algorithm processes each character once.

Space Complexity: O(n) in the worst-case scenario when all numbers are stored in the stack.

Solution

We use a stack-based approach. Iterate over each character of the string while maintaining a variable for the current number and a previous operator (initialized to '+'). When an operator or the end of the string is reached, perform an action based on the previous operator:

- For '+' add the current number to the stack. - For '-' add the negative of the current number. - For '*' and '/' pop the last number from the stack, compute the result by performing multiplication or division, and then push the result back to the stack. After processing the entire string, sum the elements in the stack to get the final result. This approach ensures that multiplication and division are handled before addition and subtraction.

#255 Verify Preorder Sequence in Binary Search Tree [Medium]

Key Insights

- A BST has the property that every node's left subtree contains values less than the node, and every right subtree contains values greater.
- Preorder traversal processes nodes as: root, left subtree, then right subtree.
- Using a monotonic stack, we can simulate constructing the BST while tracking a lower bound for allowable node values.
- When moving to a right child, update the lower bound to ensure subsequent nodes are valid.

Space & Time

Time Complexity: O(n) - Each element is processed once.

Space Complexity: O(n) - In the worst-case an explicit stack is used; can be optimized to O(1) by reusing the input array.

Solution

We use a monotonic stack to simulate the BST construction from the preorder sequence. The process is:

- Initialize a variable lower_bound to -∞. - Iterate through each value in the preorder array: If a value is less than lower_bound, it violates BST properties and the sequence is invalid. While the stack is not empty and the current value is greater than the element at the top of the stack, pop from the stack and update lower_bound to the popped value. This simulates moving to the right subtree. Push the current value onto the stack. - If the sequence is processed without violations, return true.

This approach efficiently ensures that each value adheres to BST requirements during preorder traversal.

#394 Decode String [Medium]

Key Insights

- Use a stack or recursion to manage nested encoded patterns.
- Process the string character by character, accumulating digits for the repeat count.
- When encountering a '[', start a new context (recursively or via a new stack frame) for the encoded substring.
- When encountering a ']', finish the current context and repeat the accumulated substring as needed.
- Careful handling of multi-digit numbers is essential.

Space & Time

Time Complexity: O(n), where n is the length of the string.

Space Complexity: O(n), for the recursion stack or the auxiliary stack used.

Solution

We can solve this problem using a recursive approach. When we see a digit, we determine the full repeat count. Upon encountering a '[', we recursively read the encoded substring until the corresponding ']'. Each recursive call returns the decoded substring until it hits a ']', which is then repeated based on the previously computed count.

Alternatively, an iterative solution using a stack is also feasible. As we parse the string, push the current string and number onto the stack when encountering '[', then pop and combine when ']' is reached. This approach correctly handles nested patterns.

#439 Ternary Expression Parser [Medium]

Key Insights

- The ternary operator groups right-to-left. This property can be exploited by processing the string from the end to the beginning.
- Using a stack allows us to "collapse" nested ternary expressions as soon as their three components (condition, true branch, false branch) are available.
- An iterative approach is efficient and avoids building an explicit syntax tree for the expression.
- The condition is not stored on the stack; instead, it is encountered immediately preceding the '?' operator when traversing backwards.

Space & Time

Time Complexity: O(n), where n is the length of the expression, as we process each character exactly once.

Space Complexity: O(n), due to the use of a stack for storing characters during evaluation.

Solution

The solution uses an iterative approach with a stack. We traverse the expression string from right to left. For every character:

- If it is a digit, 'T', or 'F' (and not a '.' or part of a pending operator), we push it onto the stack. - When we encounter a '?' character, we know that the next available items on the stack represent the true and false results of the ternary expression. We then skip the ':' separator that comes between them. - We then use the condition (the character immediately preceding the '?') to choose which result to push back on the stack. This method "collapses" each ternary expression into a single result, and by the time we finish traversing the string, the stack contains the final evaluated result.

#484 Find Permutation [Medium]

Key Insights

- The answer is a permutation of numbers from 1 to n, where n = len(s) + 1.
- A common strategy is to use a stack to reverse the segments when a decrease ('D') is encountered.
- When an 'I' is encountered (or at the end), pop all numbers from the stack to output them in reverse order, ensuring the smallest lexicographical order.
- This greedy approach guarantees that as soon as a rising pattern is detected, we "flush" the pending decreases correctly.

Space & Time

Time Complexity: O(n) - Each element is pushed and popped exactly once.

Space Complexity: O(n) - For the stack and the output array.

Solution

We use a greedy approach with a stack. Iterate from 0 to n (where n = len(s) + 1):

- Push the current number (i+1) onto the stack. - If the current character is 'I' or we have reached the end of the string, pop all elements from the stack and append them to the result. By doing so, segments corresponding to series of 'D's are reversed, while 'I's trigger the flush, ensuring that the final permutation is the lexicographically smallest.

#735 Asteroid Collision [Medium]

Key Insights

- Use a stack to keep track of the surviving asteroids.
- Only a collision happens when a right-moving asteroid (positive) is on the stack and a left-moving asteroid (negative) is encountered.
- Compare the absolute values to decide if one or both asteroids explode.
- Continue processing collisions until no further collisions are possible with the current asteroid.

Space & Time

Time Complexity: O(n) where n is the number of asteroids, since each asteroid is pushed and popped at most once.

Space Complexity: O(n) in the worst case, when no collisions occur and all asteroids remain on the stack.

Solution

The solution uses a stack to simulate the asteroid collision process. For each asteroid in the input:

- If it is moving to the right or the stack is empty, simply add it to the stack. - If it is moving to the left, check the asteroid on top of the stack: If the top asteroid is moving to the right, they might collide. Compare sizes: If the left-moving asteroid is larger (in absolute terms), pop the stack and continue checking. If both are equal, pop the stack and do not add the current asteroid. If the right-moving asteroid is larger, the current asteroid explodes. - If no collision occurs for the current asteroid, push it onto the stack. Finally, the stack represents the surviving asteroids order.

#739 Daily Temperatures [Medium]

Key Insights

- Use a monotonic stack to keep track of indices of days with unresolved warmer temperatures.
- As you iterate through the temperatures, resolve previous days that have found a warmer temperature.
- The index differences yield the number of days waited until a warmer temperature.

Space & Time

Time Complexity: O(n), where n is the number of temperatures, because each index is pushed and popped from the stack at most once.

Space Complexity: O(n) for storing the stack and the answer array.

Solution

The solution uses a monotonic stack to track the indices whose next warmer temperature has not been found yet. As you iterate through the temperature list, compare the current temperature with the temperature at the index stored at the top of the stack. If the current temperature is higher, it means you've found a warmer day for the day represented by the index from the stack. Pop that index, calculate the difference between the current index and the popped index, and store the result. Push the current index onto the stack for future comparisons. This approach efficiently finds the answer without nested iterations.

#962 Maximum Width Ramp [Medium]

Key Insights

- We need to identify pairs (i, j) where i < j and nums[i] <= nums[j].
- A monotonic decreasing stack can be constructed to maintain candidate starting indices.
- By iterating from left to right, we push indices onto the stack only when they have a smaller value than the last.
- Then, a backward pass from the end of the array allows us to efficiently match each element with the smallest possible candidate index from the stack.

Space & Time

Time Complexity: O(n)

Space Complexity: O(n)

Solution

The solution is built in two phases:

- Build a decreasing stack of indices by iterating through the array (only push an index if its value is less than the value at the current top of the stack). - Iterate through the array backwards. For each index j, check and pop elements from the stack while nums[j] is greater than or equal to the nums value at the popped index. For each valid ramp found, compute the width (j - popped index) and update the maximum width accordingly. This approach guarantees we examine each element a constant number of times ensuring an efficient solution.

#1214 Two Sum BSTs [Medium]

Key Insights

- Traverse the first tree to collect all node values.
- Traverse the second tree to check for each node if the complement (target - current node value) exists in the first tree's values.
- Using a hash set for the first tree values allows efficient O(1) lookups.
- The BST property is not essential for the hash set solution, but can be exploited if using iterator techniques for a two-pointer approach.

Space & Time

Time Complexity: O(n + m) where n and m are the number of nodes in the first and second trees respectively.

Space Complexity: O(n) for storing the node values from the first tree (in worst-case scenario).

Solution

We perform a DFS (or any tree traversal) on the first BST to collect all its values in a hash set. Then, we traverse the second BST and for each node, we calculate the complement (target - node value) and check whether this complement exists in the hash set. If we find such a pair, we return true. If no such pair exists after traversing the second tree completely, we return false.

This approach is straightforward and leverages extra space (the set) for constant time lookups, yielding an overall linear time complexity relative to the number of nodes.

#1762 Buildings With an Ocean View [Medium]

Key Insights

- Processing the buildings from right to left simplifies the problem because the rightmost building always has an ocean view.
- Keep track of the highest building encountered while iterating from right to left.
- A building has an ocean view if its height is greater than the maximum height seen so far.
- Append indices meeting the criteria, then reverse the order of indices for the solution since they were collected in reverse order.

Space & Time

Time Complexity: O(n) where n is the number of buildings.

Space Complexity: O(n) for storing the result in the worst-case scenario.

Solution

Iterate over the "heights" array from right to left while maintaining a variable to store the maximum height encountered so far. For each building, if its height is greater than this maximum height, add its index to the results list and update the maximum height. Since indices are collected in reverse order, reverse the list before returning it. This approach ensures that each building is visited only once.

#32 Longest Valid Parentheses [Hard]

Key Insights

- Use a stack to keep track of indices where potential valid substrings begin.
- Start by pushing a dummy index (-1) to handle edge cases.
- For every '(', push its index onto the stack.
- For every ')', pop from the stack. If the stack becomes empty, push the current index as a new base. Otherwise, compute the length of the valid substring using the current index minus the top index of the stack.
- An alternative is the dynamic programming approach where dp[i] represents the length of the longest valid substring ending at index i.

Space & Time

Time Complexity: O(n) where n is the length of the string.

Space Complexity: O(n) for the stack data structure.

Solution

The stack-based solution works by maintaining indices for unmatched parentheses. Initially, a dummy index (-1) is pushed onto the stack to serve as the base for calculating subsequent valid substring lengths. As we iterate through the string:

- If the character is '(', we push its index onto the stack. - If the character is ')', we pop from the stack. If the stack is empty afterwards, that means there is no valid starting position, so we push the current index to reset the base. Otherwise, we calculate the length of the current valid substring by subtracting the current index with the new top index of the stack and update the maximum length accordingly.

#42 Trapping Rain Water [Hard]

Key Insights

- The water trapped at any index is determined by the minimum of the maximum height on its left and right minus the height at that index.
- A two-pointers approach can efficiently solve the problem in one pass by maintaining left and right boundaries.
- Alternative approaches include dynamic programming with pre-computed left max and right max arrays, or using a monotonic stack to determine boundaries.

Space & Time

Time Complexity: O(n)

Space Complexity: O(1) when using the two-pointer approach

Solution

We use a two-pointers strategy by initializing pointers at the beginning and the end of the array. Keep track of the maximum height seen so far from the left (left_max) and the right (right_max). At each step, move the pointer with the lesser height because the water trapped is determined by the shorter boundary. Compute the water at that position as the difference between the boundary (left_max or right_max) and the current height, accumulating it into the total. This approach ensures linear time scanning with constant extra space.

#84 Largest Rectangle in Histogram [Hard]

Key Insights

- Each bar can potentially extend left and right as long as the neighboring bars are not shorter.
- A monotonic (increasing) stack can be used to efficiently track indices of bars.
- When encountering a bar lower than the one on the top of the stack, pop the stack and calculate the area of the rectangle formed with the popped bar height.
- A dummy bar (of height 0) is appended at the end to ensure all bars are processed.

Space & Time

Time Complexity: O(n)

Space Complexity: O(n)

Solution

The solution employs a monotonic stack to traverse the histogram once. For each bar, while the current bar's height is less than the height at the top index of the stack, it pops from the stack and calculates the area using the popped height as the smallest bar. The width for the rectangle is determined by the difference between the current index and the index from the stack (or the current index if the stack is empty). This effectively finds the maximum rectangular area possible. A sentinel value (0) is added at the end of the histogram to flush any remaining bars from the stack.

#224 Basic Calculator [Hard]

Key Insights

- Use a stack to save previous results and signs when encountering parentheses.
- Process the string left-to-right by accumulating multi-digit numbers and applying the current sign.
- When encountering a '(', push the current result and sign onto the stack and reset them.

- When encountering a ')', complete the current sub-expression then pop the previous state to incorporate the computed value.
- Handle whitespaces by simply skipping them.

Space & Time

Time Complexity: O(n) where n is the length of the input string, since we process each character once.

Space Complexity: O(n) in the worst-case scenario due to the use of a stack for nested parentheses.

Solution

The solution uses an iterative approach with a stack:

- Initialize variables: result to keep the running total, sign to handle addition/subtraction, and index to iterate over the string. - As you iterate, build multi-digit numbers by checking for consecutive numerical characters. - Adjust the total by adding the product of the current number and its sign. - When encountering a '(', push the current result and sign onto the stack, then reset results and sign for the new sub-expression. - When encountering a ')', complete the sub-expression by popping the previous result and sign, then combine. - Skip over spaces. - The final result is the computed value after the end of the string.

This approach effectively simulates recursion and correctly handles nested expressions and negation.

#239 Sliding Window Maximum [Hard]

Key Insights

- Use a deque (double-ended queue) to maintain the indices of potential maximum elements in a decreasing order.
- Ensure that the deque always has the current window's indices by removing elements that fall outside the current window.
- Only add elements that are greater than the ones at the back of the deque, thus preserving a monotonic decreasing order.
- The maximum for the current window is at the front of the deque.

Space & Time

Time Complexity: O(n) where n is the number of elements in nums, since each element is processed at most twice.

Space Complexity: O(k), as the deque stores indices for at most k elements at any time.

Solution

The solution uses a monotonic deque to maintain indices of potential maximum elements in the current window. As the window slides, we:

- Remove indices from the front if they are out of the window. - Remove indices from the back if the current element is larger, because they cannot be the maximum if the current element is in the window. - Append the current index. - After the first k elements, the element at the front of the deque is the maximum for that window. This approach ensures that each element is pushed and popped from the deque only once, achieving an overall O(n) time complexity.

#716 Max Stack [Hard]

Key Insights

- Use a doubly-linked list to support fast insertion and removal from the stack.
- Maintain a balanced tree (or ordered dictionary/multiset) to quickly locate the maximum element in O(log n) time.
- Map each value to its corresponding node(s) in the doubly-linked list, so that when popMax is called, you can immediately remove the corresponding node from the list.
- The double-linking allows removal of arbitrary nodes in O(1), once you have the reference.

Space & Time

Time Complexity:

- push: O(log n) due to insertion in the ordered structure.
- pop: O(1) for removal from the doubly-linked list, plus O(log n) for updating the tree.
- top: O(1)

- peekMax: O(1) or O(log n) depending on the tree implementation.
- popMax: O(log n) for looking up and removal from the tree, O(1) for doubly-linked list deletion.

Space Complexity:

- O(n) for storing all nodes in the doubly-linked list and the tree structure mapping values to nodes.

Solution

We use two data structures:

- A doubly-linked list to store the elements in stack order. This supports fast push and pop operations and allows O(1) removal when given a reference. - An ordered structure (like a balanced BST, TreeMap in Java, or SortedList in Python) that maps each value to a list of nodes that hold that value in the linked list. This allows us to quickly determine the maximum element. When there are duplicates, we remove the one closest to the top by storing nodes in order. The trick is to keep these two structures synchronized. When an element is pushed, add its node to both the doubly-linked list and the ordered structure. When an element is removed (either via pop or popMax), remove its node from both structures.

#772 Basic Calculator III [Hard]

Key Insights

- Use recursion to handle nested expressions defined by parentheses.
- A stack can be used to correctly evaluate the expression and respect operator precedence.
- Multiplication and division have higher precedence than addition and subtraction.
- When encountering an open parenthesis, recursively compute its value until a closing parenthesis is found.
- Use immediate calculation for multiplication and division by updating the top of the stack before moving to the next operator.

Space & Time

Time Complexity: O(n) where n is the length of the string, as every character is processed.

Space Complexity: O(n) due to the recursion stack or auxiliary stack used for intermediate results.

Solution

We use a recursive approach combined with a stack to solve the problem. As we iterate through the string:

- Convert digits into numbers. - When encountering an operator or parenthesis, process the previously accumulated number based on the preceding operator. - For multiplication or division, pop the last number from the stack, apply the operation, and push the result back. - When a '(' is encountered, recursively evaluate the substring until the corresponding ')'. - After processing all tokens, sum the values in the stack to get the final result. This method correctly handles both operator precedence and nested expressions.

#895 Maximum Frequency Stack [Hard]

Key Insights

- Use a hash map to count the frequency of each element.
- Maintain another hash map that maps frequencies to stacks (lists) of elements.
- Track the current maximum frequency to enable O(1) retrieval of the most frequent element.
- When pushing an element, update its frequency and add it to the corresponding frequency stack.
- When popping an element, remove it from the stack corresponding to the maximum frequency and update the frequency count; if that stack becomes empty, decrease the current maximum frequency.

Space & Time

Time Complexity: O(1) per push and pop.

Space Complexity: O(n), where n is the number of elements pushed onto the stack.

Solution

The solution involves using two main data structures:

- A hash map (freq) to store the frequency count for each value. - A hash map (group) that maps a frequency to a stack (list) of values that have that frequency.

During a push:

- Increment the frequency count of the value. - Push the value onto the stack corresponding to its updated frequency. - Update the maximum frequency if necessary.

During a pop:

- Pop the top element from the stack corresponding to the current maximum frequency. - Decrement its frequency count in the freq map. - If the frequency stack becomes empty after popping, decrement the current maximum frequency. This design enables the retrieval of the most frequent element in constant time while dealing with frequency ties by returning the element closest to the top.

#21 Trees & BST — 37 questions · #21 of 21

#100 EASY

Same Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS · BFS

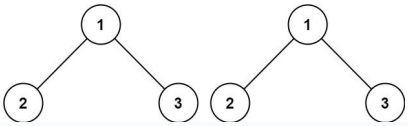
Lists: Grind 169

Problem:

Given the roots of two binary trees p and q, write a function to check if they are the same or not.

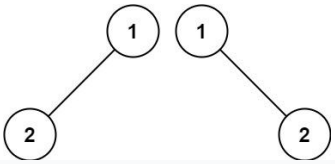
Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

Example 1:



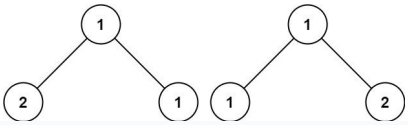

```
Input: p = [1,2,3], q = [1,2,3]
Output: true
```

Example 2:




```
Input: p = [1,2], q = [1,null,2]
Output: false
```

Example 3:




```
Input: p = [1,2,1], q = [1,1,2]
Output: false
```

Constraints:

- The number of nodes in both trees is in the range [0, 100].
- -104 <= Node.val <= 104

>> Brute Force [Trees & BST] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def isSameTree(self, p, q):
        # Brute force O(n) - recursive structural comparison
        if not p and not q: return True
        if not p or not q or p.val != q.val: return False
        return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

Python

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

def isSameTree(p, q):
    # If both nodes are None, trees are identical at this branch.
    if not p and not q:
        return True
    # If one of the nodes is None, trees are not identical.
    if not p or not q:
        return False
    # Check if current nodes have the same value.
    if p.val != q.val:
        return False
    # Recursively compare the left subtrees and the right subtrees.
    return isSameTree(p.left, q.left) and isSameTree(p.right, q.right)
```

Python


```python
class Solution:
    def isSameTree(self, p: TreeNode | None, q: TreeNode | None) -> bool:
        if not p or not q:
            return p == q
        return (p.val == q.val and
                self.isSameTree(p.left, q.left) and
                self.isSameTree(p.right, q.right))
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def isSameTree(self, p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
        if p == q:
            return True
        if p is None or q is None or p.val != q.val:
            return False
        return self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None

class Solution(object):
    def isSameTree(self, p, q):
        """
        :type p: TreeNode
        :type q: TreeNode
        :rtype: bool
        """
        if not p or not q:
            return p == q # covers: p=none and q!=none, p=none and p!=none, both none
        return p.val == q.val and self.isSameTree(p.left, q.left) and self.isSameTree(p.right, q.right)
```



```python
# iteration
class Solution:
    def isSameTree(self, p: TreeNode, q: TreeNode) -> bool:
        if p is None:
            return q is None

        if q is None:
            return p is None

        stack1 = [p]
        stack2 = [q]

        while stack1 and stack2:
            current1 = stack1.pop()
            current2 = stack2.pop()

            if current1 is None and current2 is None:
                continue
            elif current1 is None or current2 is None:
                return False
            elif current1.val != current2.val:
                return False

            stack1.append(current1.left)
            stack2.append(current2.left)

            stack1.append(current1.right)
            stack2.append(current2.right)

        return not stack1 and not stack2
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

def isSameTree(p, q):
    # If both nodes are None, trees are identical at this branch.
    if not p and not q:
        return True
    # If one of the nodes is None, trees are not identical.
    if not p or not q:
        return False
    # Check if current nodes have the same value.
    if p.val != q.val:
        return False
    # Recursively compare the left subtrees and the right subtrees.
    return isSameTree(p.left, q.left) and isSameTree(p.right, q.right)
```

#101 EASY

Symmetric Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

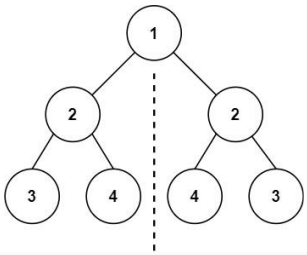
Tree · DFS · BFS

Lists: Grind 169

Problem:

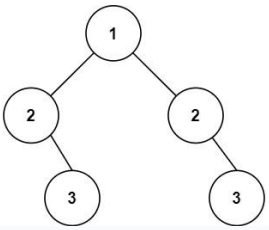
Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

Example 1:




```
Input: root = [1,2,2,3,4,4,3]
Output: true
```

Example 2:




```
Input: root = [1,2,2,null,3,null,3]
Output: false
```

Constraints:

- The number of nodes in the tree is in the range [1, 1000].
- -100 <= Node.val <= 100

Follow up: Could you solve it both recursively and iteratively?

Python

Python

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val # current node's value
        self.left = left # pointer to left child
        self.right = right # pointer to right child

class Solution:
    def isSymmetric(self, root: TreeNode) -> bool:
        # Helper function to check mirror symmetry between two trees.
        def isMirror(t1: TreeNode, t2: TreeNode) -> bool:
            # Both nodes are null, so they are mirrors.
            if t1 is None and t2 is None:
                return True
            # If one is null or the values do not match, symmetry fails.
            if t1 is None or t2 is None or t1.val != t2.val:
                return False
            # Recursively check mirror symmetry for children:
            # t1's left with t2's right and t1's right with t2's left
            return isMirror(t1.left, t2.right) and isMirror(t1.right, t2.left)

        # Start recursion comparing the tree with itself.
        return isMirror(root, root)
```


Python

```python
class Solution:
    def isSymmetric(self, root: TreeNode | None) -> bool:
        def isSymmetric(p: TreeNode | None, q: TreeNode | None) -> bool:
            if not p or not q:
                return p == q
            return (p.val == q.val and
                    isSymmetric(p.left, q.right) and
                    isSymmetric(p.right, q.left))

        return isSymmetric(root, root)
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        def dfs(root1: Optional[TreeNode], root2: Optional[TreeNode]) -> bool:
            if root1 == root2:
                return True
            if root1 is None or root2 is None or root1.val != root2.val:
                return False
            return dfs(root1.left, root2.right) and dfs(root1.right, root2.left)

        return dfs(root.left, root.right)
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        def dfs(root1, root2):
            if root1 is None and root2 is None:
                return True
            if root1 is None or root2 is None or root1.val != root2.val:
                return False
            return dfs(root1.left, root2.right) and dfs(root1.right, root2.left)

        return dfs(root, root)
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def isSymmetric(self, root: Optional[TreeNode]) -> bool:
        def dfs(root1: Optional[TreeNode], root2: Optional[TreeNode]) -> bool:
            if root1 == root2:
                return True
            if root1 is None or root2 is None or root1.val != root2.val:
                return False
            return dfs(root1.left, root2.right) and dfs(root1.right, root2.left)

        return dfs(root.left, root.right)
```

#104 EASY

Maximum Depth of Binary Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS · BFS

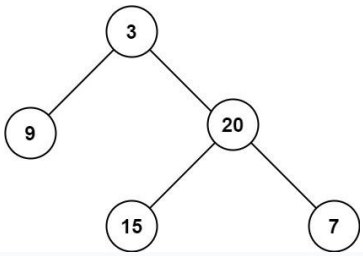
Lists: Grind 169

Problem:

Given the root of a binary tree, return its maximum depth.

A binary tree's maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Example 1:




```
Input: root = [3,9,20,null,null,15,7]
Output: 3
```

Example 2:

```
Input: root = [1,null,2]
Output: 2
```

Constraints:

- The number of nodes in the tree is in the range [0, 104].
- -100 <= Node.val <= 100

Python

Python


```python
# Python implementation using DFS

# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        # Base case: if the node is null, return 0.
        if not root:
            return 0

        # Recursively compute the maximum depth of the left subtree.
        left_depth = self.maxDepth(root.left)
        # Recursively compute the maximum depth of the right subtree.
        right_depth = self.maxDepth(root.right)

        # Return the greater depth plus one for the current node.
        return 1 + max(left_depth, right_depth)
```

Python

```python
class Solution:
    def maxDepth(self, root: TreeNode | None) -> int:
        if not root:
            return 0
        return 1 + max(self.maxDepth(root.left), self.maxDepth(root.right))
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if root is None:
            return 0
        l, r = self.maxDepth(root.left), self.maxDepth(root.right)
        return 1 + max(l, r)
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def maxDepth(self, root: TreeNode) -> int:
        if root is None:
            return 0
        l, r = self.maxDepth(root.left), self.maxDepth(root.right)
        return 1 + max(l, r)
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Python implementation using DFS

# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def maxDepth(self, root: Optional[TreeNode]) -> int:
        # Base case: if the node is null, return 0.
        if not root:
            return 0

        # Recursively compute the maximum depth of the left subtree.
        left_depth = self.maxDepth(root.left)
        # Recursively compute the maximum depth of the right subtree.
        right_depth = self.maxDepth(root.right)

        # Return the greater depth plus one for the current node.
        return 1 + max(left_depth, right_depth)
```

#108 EASY

Convert Sorted Array to Binary Search Tree

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

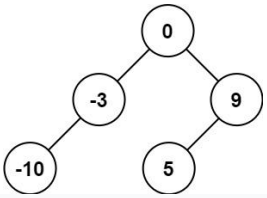
Array · Divide and Conquer · Tree

Lists: Grind 169

Problem:

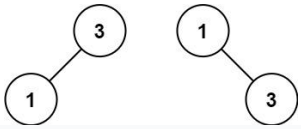
Given an integer array nums where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

Example 1:




```
Input: nums = [-10,-3,0,5,9]
Output: [0,-3,9,-10,null,5]
Explanation: [0,-10,5,null,-3,null,9] is also accepted:
```

Example 2:




```
Input: nums = [1,3]
Output: [3,1]
Explanation: [1,null,3] and [3,1] are both height-balanced BSTs.
```

Constraints:

- 1 <= nums.length <= 104
- -104 <= nums[i] <= 104
- nums is sorted in a strictly increasing order.

>> Brute Force [Trees & BST] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def sortedArrayToBST(self, nums):
        # Brute Force: collect inorder, then process sorted list
        vals = []
        def collect(node):
            if not node: return
            collect(node.left)
            vals.append(node.val)
            collect(node.right)
        collect(nums)
        return vals[0] if vals else 0
```

Python

Python

```python
# Definition for a binary tree node.
class TreeNode:
    # Initializes a tree node with a value and optional left/right children.
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def sortedArrayToBST(self, nums):
        # Helper function to build BST from subarray indices.
        def buildBST(left, right):
            # Base condition: if no elements to process, return None.
            if left > right:
                return None
            # Choose the middle element as the root for balance.
            mid = (left + right) // 2
            node = TreeNode(nums[mid])
            # Recursively build the left subtree.
            node.left = buildBST(left, mid - 1)
            # Recursively build the right subtree.
            node.right = buildBST(mid + 1, right)
            return node

        # Build and return the BST from the entire array.
        return buildBST(0, len(nums) - 1)
```

Python

```python
class Solution:
   def sortedArrayToBST(self, nums: list[int]) -> TreeNode | None:
      def build(l: int, r: int) -> TreeNode | None:
         if l > r:
            return None
         m = (l + r) // 2
         return TreeNode(nums[m],
                         build(l, m - 1),
                         build(m + 1, r))

      return build(0, len(nums) - 1)
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
        def dfs(l: int, r: int) -> Optional[TreeNode]:
            if l > r:
                return None
            mid = (l + r) >> 1
            return TreeNode(nums[mid], dfs(l, mid - 1), dfs(mid + 1, r))

        return dfs(0, len(nums) - 1)
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
        def dfs(l, r):
            if l > r:
                return None
            mid = (l + r) >> 1
            left = dfs(l, mid - 1)
            right = dfs(mid + 1, r)
            return TreeNode(nums[mid], left, right)

        return dfs(0, len(nums) - 1)

###########

# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None

# self.right = None

class Solution(object):
  def sortedArrayToBST(self, nums):
    """
    :type nums: List[int]
    :rtype: TreeNode
    """
    if nums:
      midPos = len(nums) / 2
      mid = nums[midPos]
      root = TreeNode(mid)
      root.left = self.sortedArrayToBST(nums[:midPos])
      root.right = self.sortedArrayToBST(nums[midPos + 1:])
      return root
```


[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def sortedArrayToBST(self, nums: List[int]) -> Optional[TreeNode]:
        def dfs(l, r):
            if l > r:
                return None
            mid = (l + r) >> 1
            left = dfs(l, mid - 1)
            right = dfs(mid + 1, r)
            return TreeNode(nums[mid], left, right)

        return dfs(0, len(nums) - 1)

###########

# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None

# self.right = None

class Solution(object):
  def sortedArrayToBST(self, nums):
    """
    :type nums: List[int]
    :rtype: TreeNode
    """
    if nums:
      midPos = len(nums) / 2
      mid = nums[midPos]
      root = TreeNode(mid)
      root.left = self.sortedArrayToBST(nums[:midPos])
      root.right = self.sortedArrayToBST(nums[midPos + 1:])
      return root
```

#110 EASY

Balanced Binary Tree

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

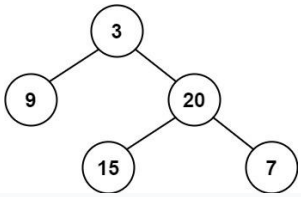
Tree · DFS

Lists: Grind 169

Problem:

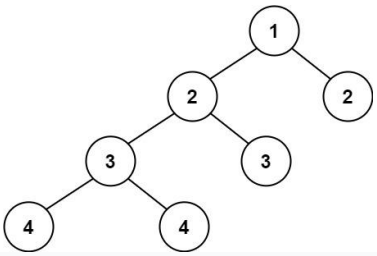
Given a binary tree, determine if it is height-balanced.

Example 1:




```
Input: root = [3,9,20,null,null,15,7]
Output: true
```

Example 2:




```
Input: root = [1,2,2,3,3,null,null,4,4]
Output: false
```

Example 3:

```
Input: root = []
Output: true
```

Constraints:

- The number of nodes in the tree is in the range [0, 5000].
- -104 <= Node.val <= 104

Python

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        # Helper function to return the height of the subtree or -1 if unbalanced.
        def checkHeight(node: TreeNode) -> int:
            # An empty tree has height 0 and is balanced.
            if not node:
                return 0

            # Recursively get the height of the left subtree.
            leftHeight = checkHeight(node.left)
            # If left subtree is unbalanced, return -1.
            if leftHeight == -1:
                return -1

            # Recursively get the height of the right subtree.
            rightHeight = checkHeight(node.right)
            # If right subtree is unbalanced, return -1.
            if rightHeight == -1:

                return -1

            # If the current node's subtrees differ in height by more than 1, it's unbalanced.
            if abs(leftHeight - rightHeight) > 1:
                return -1

            # Return the height of the current node.
            return max(leftHeight, rightHeight) + 1

        # The tree is balanced if checkHeight does not return -1.
        return checkHeight(root) != -1
```

Python

```python
class Solution:
    def isBalanced(self, root: TreeNode | None) -> bool:
        if not root:
            return True

        def maxDepth(root: TreeNode | None) -> int:
            if not root:
                return 0
            return 1 + max(maxDepth(root.left), maxDepth(root.right))

        return (abs(maxDepth(root.left) - maxDepth(root.right)) <= 1 and
                self.isBalanced(root.left) and
                self.isBalanced(root.right))
```


Python

```python
class Solution:
    def isBalanced(self, root: TreeNode | None) -> bool:
        ans = True

        def maxDepth(root: TreeNode | None) -> int:
            """Returns the height of root and sets ans to false if root unbalanced."""
            nonlocal ans
            if not root or not ans:
                return 0
            left = maxDepth(root.left)
            right = maxDepth(root.right)
            if abs(left - right) > 1:
                ans = False
            return max(left, right) + 1

        maxDepth(root)
        return ans
```

Python

```python
class Solution:
    def isBalanced(self, root: TreeNode | None) -> bool:
        def maxDepth(root: TreeNode | None) -> int:
            """Returns the height of root if root is balanced; otherwise, returns -1."""
            if not root:
                return 0

            left = maxDepth(root.left)
            if left == -1:
                return -1
            right = maxDepth(root.right)
            if right == -1:
                return -1
            if abs(left - right) > 1:
                return -1

            return 1 + max(maxDepth(root.left), maxDepth(root.right))

        return maxDepth(root) != -1
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        def height(root):
            if root is None:
                return 0
            l, r = height(root.left), height(root.right)
            if l == -1 or r == -1 or abs(l - r) > 1:
                return -1
            return 1 + max(l, r)

        return height(root) >= 0
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        def height(root):
            if root is None:
                return 0
            l, r = height(root.left), height(root.right)
            if l == -1 or r == -1 or abs(l - r) > 1:
                return -1
            return 1 + max(l, r)

        return height(root) >= 0
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def isBalanced(self, root: TreeNode) -> bool:
        # Helper function to return the height of the subtree or -1 if unbalanced.
        def checkHeight(node: TreeNode) -> int:
            # An empty tree has height 0 and is balanced.
            if not node:
                return 0

            # Recursively get the height of the left subtree.
            leftHeight = checkHeight(node.left)
            # If left subtree is unbalanced, return -1.
            if leftHeight == -1:
                return -1

            # Recursively get the height of the right subtree.
            rightHeight = checkHeight(node.right)
            # If right subtree is unbalanced, return -1.
            if rightHeight == -1:

                return -1

            # If the current node's subtrees differ in height by more than 1, it's unbalanced.
            if abs(leftHeight - rightHeight) > 1:
                return -1

            # Return the height of the current node.
            return max(leftHeight, rightHeight) + 1

        # The tree is balanced if checkHeight does not return -1.
        return checkHeight(root) != -1
```

#112 EASY

Path Sum

LeetCodeCx · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS · BFS

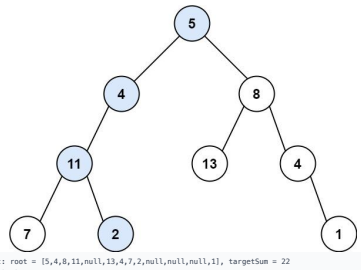
Lists: Denny Zhang

Problem:

Given the root of a binary tree and an integer targetSum, return true if the tree has a root-to-leaf path such that adding up all the values along the path equals targetSum.

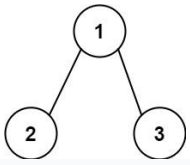
A leaf is a node with no children.

Example 1:




```
Input: root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22
Output: true
Explanation: The root-to-leaf path with the target sum is shown.
```

Example 2:




```
Input: root = [1,2,3], targetSum = 5
Output: false
Explanation: There are two root-to-leaf paths in the tree:
(1 --> 2): The sum is 3.
(1 --> 3): The sum is 4.
There is no root-to-leaf path with sum = 5.
```

Example 3:


```
Input: root = [], targetSum = 0
Output: false
Explanation: Since the tree is empty, there are no root-to-leaf paths.
```

Constraints:

- The number of nodes in the tree is in the range [0, 5000].
- -1000 <= Node.val <= 1000
- -1000 <= targetSum <= 1000

>> Brute Force [Trees & BST] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def hasPathSum(self, root, targetSum):
        # Brute force - collect inorder, then process sorted list
        vals = []
        def collect(node):
            if not node: return
            collect(node.left)
            vals.append(node.val)
            collect(node.right)
        collect(root)
        return vals[0] if vals else 0
```

Python

Python

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val # Node's value
        self.left = left # Pointer to left child
        self.right = right # Pointer to right child

def hasPathSum(root, targetSum):
    # If the current node is None, no valid path exists
    if not root:
        return False
    # Check if the current node is a leaf node
    if not root.left and not root.right:
        # Return True if the remaining targetSum equals the leaf value
        return targetSum == root.val
    # Recursively check the left and right subtrees with the updated targetSum
    return hasPathSum(root.left, targetSum - root.val) or hasPathSum(root.right, targetSum - root.val)

# Example usage:
# root = TreeNode(5, TreeNode(4, TreeNode(11, TreeNode(7), TreeNode(2))), TreeNode(8, TreeNode(13),
TreeNode(4, None, TreeNode(1))))
# print(hasPathSum(root, 22)) # Output: True
```

Python

```python
class Solution:
    def hasPathSum(self, root: TreeNode, summ: int) -> bool:
        if not root:
            return False
        if root.val == summ and not root.left and not root.right:
            return True
        return (self.hasPathSum(root.left, summ - root.val) or
                self.hasPathSum(root.right, summ - root.val))
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
        def dfs(root, s):
            if root is None:
                return False
            s += root.val
            if root.left is None and root.right is None and s == targetSum:
                return True
            return dfs(root.left, s) or dfs(root.right, s)

        return dfs(root, 0)
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
        def dfs(root, s):
            if root is None:
                return False
            s += root.val
            if root.left is None and root.right is None and s == targetSum:
                return True
            return dfs(root.left, s) or dfs(root.right, s)

        return dfs(root, 0)
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def hasPathSum(self, root: Optional[TreeNode], targetSum: int) -> bool:
        def dfs(root, s):
            if root is None:
                return False
            s += root.val
            if root.left is None and root.right is None and s == targetSum:
                return True
            return dfs(root.left, s) or dfs(root.right, s)

        return dfs(root, 0)
```

#226 EASY

Invert Binary Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

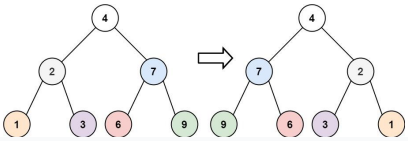
Tree · DFS · BFS

Lists: Grind 169

Problem:

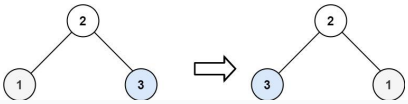
Given the root of a binary tree, invert the tree, and return its root.

Example 1:




```
Input: root = [4,2,7,1,3,6,9]
Output: [4,7,2,9,6,3,1]
```

Example 2:




```
Input: root = [2,1,3]
Output: [2,3,1]
```

Example 3:

```
Input: root = []
Output: []
```

Constraints:

- The number of nodes in the tree is in the range [0, 100].
- -100 <= Node.val <= 100

Python

Python

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val # Initialize node value
        self.left = left # Pointer to left child
        self.right = right # Pointer to right child

class Solution:
    def invertTree(self, root: TreeNode) -> TreeNode:
        # Base case: if the node is None, return None.
        if not root:
            return None
        # Recursively invert the left and right subtrees.
        left_inverted = self.invertTree(root.left)
        right_inverted = self.invertTree(root.right)
        # Swap the left and right children.
        root.left, root.right = right_inverted, left_inverted
        return root
```

Python

```python
class Solution:
    def invertTree(self, root: TreeNode | None) -> TreeNode | None:
        if not root:
            return None

        left = root.left
        right = root.right
        root.left = self.invertTree(right)
        root.right = self.invertTree(left)
        return root
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        if root is None:
            return None
        l, r = self.invertTree(root.left), self.invertTree(root.right)
        root.left, root.right = r, l
        return root
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        def dfs(root):
            if root is None:
                return
            root.left, root.right = root.right, root.left
            dfs(root.left)
            dfs(root.right)

        dfs(root)
        return root
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def invertTree(self, root: Optional[TreeNode]) -> Optional[TreeNode]:
        def dfs(root):
            if root is None:
                return
            root.left, root.right = root.right, root.left
            dfs(root.left)
            dfs(root.right)

        dfs(root)
        return root
```

#270 EASY

Closest Binary Search Tree Value

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

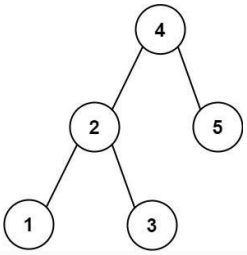
Binary Search · Tree · DFS · BST

Lists: Premium 98

Problem:

Given the root of a binary search tree and a target value, return the value in the BST that is closest to the target. If there are multiple answers, print the smallest.

Example 1:




```
Input: root = [4,2,5,1,3], target = 3.714286
Output: 4
```

Example 2:

```
Input: root = [1], target = 4.428571
Output: 1
```

Constraints:

- The number of nodes in the tree is in the range [1, 104].
- 0 <= Node.val <= 109
- -109 <= target <= 109

Python

Python


```python
# Python solution for Closest Binary Search Tree Value

# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def closestValue(self, root: TreeNode, target: float) -> int:
        # Initialize the best value with the value of the root
        best_value = root.val

        # Traverse the BST iteratively
        current = root
        while current:
            # Calculate the difference between current node's value and target
            current_diff = abs(current.val - target)
            best_diff = abs(best_value - target)

            # Update best value if a closer difference is found, or if equal and current value is smaller
            if current_diff < best_diff or (current_diff == best_diff and current.val < best_value):
                best_value = current.val

            # Decide the traversal direction based on the target comparison
            if target < current.val:
                current = current.left
            else:
                current = current.right

        return best_value
```

Python

```python
class Solution:
    def closestValue(self, root: TreeNode | None, target: float) -> int:
        # If target < root.val, search the left subtree.
        if target < root.val and root.left:
            left = self.closestValue(root.left, target)
            if abs(left - target) <= abs(root.val - target):
                return left

        # If target > root.val, search the right subtree.
        if target > root.val and root.right:
            right = self.closestValue(root.right, target)
            if abs(right - target) < abs(root.val - target):
                return right

        return root.val
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def closestValue(self, root: Optional[TreeNode], target: float) -> int:
        def dfs(node: Optional[TreeNode]):
            if node is None:
                return
            nxt = abs(target - node.val)
            nonlocal ans, diff
            if nxt < diff or (nxt == diff and node.val < ans):
                diff = nxt
                ans = node.val
            node = node.left if target < node.val else node.right
            dfs(node)

        ans = 0
        diff = inf
        dfs(root)
        return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None

class Solution(object):
  def closestValue(self, p, target):
    """
    :type root: TreeNode
    :type target: float
    :rtype: int
    """
    ans = p.val
    while p:
      if abs(target - p.val) < abs(ans - target):
        ans = p.val
      if target < p.val:
        p = p.left
      else:
        p = p.right
    return ans
```



```
###########

# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def closestValue(self, root: Optional[TreeNode], target: float) -> int:
        ans, mi = root.val, inf
        while root:
            t = abs(root.val - target)
            if t < mi:
                mi = t
                ans = root.val
            if root.val > target:
                root = root.left
            else:
                root = root.right
        return ans
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def closestValue(self, root: Optional[TreeNode], target: float) -> int:
        def dfs(node: Optional[TreeNode]):
            if node is None:
                return
            nxt = abs(target - node.val)
            nonlocal ans, diff
            if nxt < diff or (nxt == diff and node.val < ans):
                diff = nxt
                ans = node.val
            node = node.left if target < node.val else node.right
            dfs(node)

        ans = 0
        diff = inf
        dfs(root)
        return ans
```

#501 EASY

Find Mode in Binary Search Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS · BST

Lists: Denny Zhang

Problem:

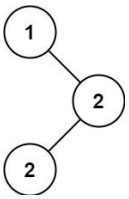
Given the root of a binary search tree (BST) with duplicates, return all the mode(s) (i.e., the most frequently occurred element) in it.

If the tree has more than one mode, return them in any order.

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than or equal to the node's key.
- The right subtree of a node contains only nodes with keys greater than or equal to the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:




```
Input: root = [1,null,2,2]
Output: [2]
```

Example 2:

```
Input: root = [0]
Output: [0]
```

Constraints:

- The number of nodes in the tree is in the range [1, 104].
- -105 <= Node.val <= 105

Follow up: Could you do that without using any extra space? (Assume that the implicit stack space incurred due to recursion does not count).

>> Brute Force [Trees & BST] Interview Prep

Python


```python
# Brute Force Approach
class Solution:
    def findMode(self, root):
        # Brute Force - collect inorder, then process sorted list
        vals = []
        def collect(node):
            if not node: return
            collect(node.left)
            vals.append(node.val)
            collect(node.right)
        collect(root)
        return vals[0] if vals else 0
```

Python

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def findMode(self, root):
        # Initialize state variables: prev value, current frequency count, maximum frequency, and modes
list.
        self.prev = None
        self.currentCount = 0
        self.maxCount = 0
        self.modes = []

        # First in-order traversal to find the maximum frequency (maxCount).
        def inorder_first(node):
            if not node:
                return
            inorder_first(node.left)
            # Update the count for the current node.
            if self.prev is None or node.val != self.prev:
                self.currentCount = 1
            else:
                self.currentCount += 1
```



```
            self.prev = node.val
            # Update maxCount if currentCount exceeds it.
            self.maxCount = max(self.maxCount, self.currentCount)
            inorder_first(node.right)

        inorder_first(root)

        # Second in-order traversal to collect all node values with frequency equal to maxCount.
        self.prev = None # Reset previous value.
        self.currentCount = 0 # Reset count.

        def inorder_second(node):
            if not node:
                return
            inorder_second(node.left)
            if self.prev is None or node.val != self.prev:
                self.currentCount = 1
            else:
                self.currentCount += 1
            self.prev = node.val
            if self.currentCount == self.maxCount:
                self.modes.append(node.val)
            inorder_second(node.right)

        inorder_second(root)

        return self.modes
```

Python

```python
class Solution:
  def findMode(self, root: TreeNode | None) -> list[int]:
    self.ans = []
    self.pred = None
    self.count = 0
    self.maxCount = 0

    def updateCount(root: TreeNode | None) -> None:
      if self.pred and self.pred.val == root.val:
        self.count += 1
      else:
        self.count = 1

      if self.count > self.maxCount:
        self.maxCount = self.count
        self.ans = [root.val]
      elif self.count == self.maxCount:
        self.ans.append(root.val)

      self.pred = root

    def inorder(root: TreeNode | None) -> None:
      if not root:
        return
```



```
        inorder(root.left)
        updateCount(root)
        inorder(root.right)

    inorder(root)
    return self.ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def findMode(self, root: TreeNode) -> List[int]:
        def dfs(root):
            if root is None:
                return
            nonlocal mx, prev, ans, cnt
            dfs(root.left)
            cnt = cnt + 1 if prev == root.val else 1
            if cnt > mx:
                ans = [root.val]
                mx = cnt
            elif cnt == mx:
                ans.append(root.val)
            prev = root.val
            dfs(root.right)

        prev = None
        mx = cnt = 0
        ans = []

        dfs(root)
        return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def findMode(self, root: TreeNode) -> List[int]:
        def dfs(root):
            if root is None:
                return
            nonlocal mx, prev, ans, cnt
            dfs(root.left)
            cnt = cnt + 1 if prev == root.val else 1
            if cnt > mx:
                ans = [root.val]
                mx = cnt
            elif cnt == mx:
                ans.append(root.val)
            prev = root.val
            dfs(root.right)

        prev = None
        mx = cnt = 0
        ans = []

        dfs(root)
        return ans
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def findMode(self, root):
        # Initialize state variables: prev value, current frequency count, maximum frequency, and modes
list.
        self.prev = None
        self.currentCount = 0
        self.maxCount = 0
        self.modes = []

        # First in-order traversal to find the maximum frequency (maxCount).
        def inorder_first(node):
            if not node:
                return
            inorder_first(node.left)
            # Update the count for the current node.
            if self.prev is None or node.val != self.prev:
                self.currentCount = 1
            else:
                self.currentCount += 1
```



```python
            self.prev = node.val
            # Update maxCount if currentCount exceeds it.
            self.maxCount = max(self.maxCount, self.currentCount)
            inorder_first(node.right)

        inorder_first(root)

        # Second in-order traversal to collect all node values with frequency equal to maxCount.
        self.prev = None # Reset previous value.
        self.currentCount = 0 # Reset count.

        def inorder_second(node):
            if not node:
                return
            inorder_second(node.left)
            if self.prev is None or node.val != self.prev:
                self.currentCount = 1
            else:
                self.currentCount += 1
            self.prev = node.val
            if self.currentCount == self.maxCount:
                self.modes.append(node.val)
            inorder_second(node.right)

        inorder_second(root)

        return self.modes
```

#543 EASY

Diameter of Binary Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS

Lists: Grind 169

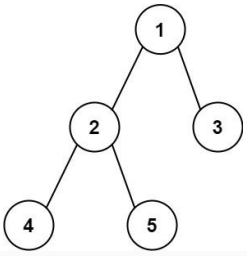
Problem:

Given the root of a binary tree, return the length of the diameter of the tree.

The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The length of a path between two nodes is represented by the number of edges between them.

Example 1:




```
Input: root = [1,2,3,4,5]
Output: 3
Explanation: 3 is the length of the path [4,2,1,3] or [5,2,1,3].
```

Example 2:

```
Input: root = [1,2]
Output: 1
```

Constraints:

- The number of nodes in the tree is in the range [1, 104].
- -100 <= Node.val <= 100

Python

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        # Global variable to track the maximum diameter found
        self.max_diameter = 0

        def dfs(node):
            # Base case: if node is None, depth is 0.
            if not node:
                return 0
            # Compute the depth of left and right subtrees.
            left_depth = dfs(node.left)
            right_depth = dfs(node.right)
            # Update the global diameter: the path through this node
            self.max_diameter = max(self.max_diameter, left_depth + right_depth)
            # Return the depth of the tree rooted at this node.
            return max(left_depth, right_depth) + 1

        dfs(root)

        return self.max_diameter
```

Python

```python
class Solution:
   def diameterOfBinaryTree(self, root: TreeNode | None) -> int:
      ans = 0

      def maxDepth(root: TreeNode | None) -> int:
        nonlocal ans
        if not root:
          return 0

        l = maxDepth(root.left)
        r = maxDepth(root.right)
        ans = max(ans, l + r)
        return 1 + max(l, r)

      maxDepth(root)
      return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        def dfs(root):
            if root is None:
                return 0
            nonlocal ans
            left, right = dfs(root.left), dfs(root.right)
            ans = max(ans, left + right)
            return 1 + max(left, right)

        ans = 0
        dfs(root)
        return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        def dfs(root):
            if root is None:
                return 0
            nonlocal ans
            left, right = dfs(root.left), dfs(root.right)
            ans = max(ans, left + right)
            return 1 + max(left, right)

        ans = 0
        dfs(root)
        return ans
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def diameterOfBinaryTree(self, root: TreeNode) -> int:
        # Global variable to track the maximum diameter found
        self.max_diameter = 0

        def dfs(node):
            # Base case: if node is None, depth is 0.
            if not node:
                return 0
            # Compute the depth of left and right subtrees.
            left_depth = dfs(node.left)
            right_depth = dfs(node.right)
            # Update the global diameter: the path through this node
            self.max_diameter = max(self.max_diameter, left_depth + right_depth)
            # Return the depth of the tree rooted at this node.
            return max(left_depth, right_depth) + 1

        dfs(root)

        return self.max_diameter
```

#572 EASY

Subtree of Another Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS · String Matching

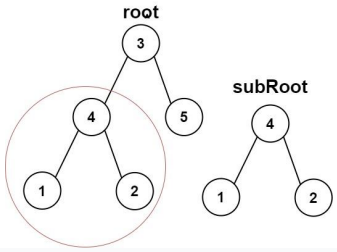
Lists: Grind 169

Problem:

Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.

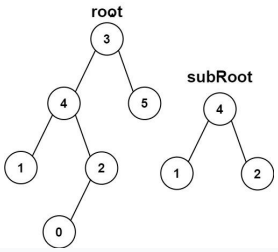
A subtree of a binary tree tree is a tree that consists of a node in tree and all of this node's descendants. The tree tree could also be considered as a subtree of itself.

Example 1:




```
Input: root = [3,4,5,1,2], subRoot = [4,1,2]
Output: true
```

Example 2:




```
Input: root = [3,4,5,1,2,null,null,null,null,0], subRoot = [4,1,2]
Output: false
```


Constraints:

- The number of nodes in the root tree is in the range [1, 2000].
- The number of nodes in the subRoot tree is in the range [1, 1000].
- -104 <= root.val <= 104
- -104 <= subRoot.val <= 104

Python

Python

```python
# Python solution for checking if subRoot is a subtree of root.

# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def isSubtree(self, root: TreeNode, subRoot: TreeNode) -> bool:
        # Helper function to check if two trees are identical.
        def isSameTree(s, t):
            # Both nodes are None, trees match.
            if not s and not t:
                return True
            # One node is None or values differ.
            if not s or not t or s.val != t.val:
                return False
            # Recurse on left and right subtrees.
            return isSameTree(s.left, t.left) and isSameTree(s.right, t.right)

        # Base case: if root is None, then no subtree exists.
        if not root:
            return False

        # Check if current subtree is identical to subRoot.
        if isSameTree(root, subRoot):
            return True
        # Recursively check left and right subtrees.
        return self.isSubtree(root.left, subRoot) or self.isSubtree(root.right, subRoot)
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def isSubtree(self, root: Optional[TreeNode], subRoot: Optional[TreeNode]) -> bool:
        def same(p: Optional[TreeNode], q: Optional[TreeNode]) -> bool:
            if p is None or q is None:
                return p is q
            return p.val == q.val and same(p.left, q.left) and same(p.right, q.right)

        if root is None:
            return False
        return (
            same(root, subRoot)
            or self.isSubtree(root.left, subRoot)
            or self.isSubtree(root.right, subRoot)
        )
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def isSubtree(self, root: TreeNode, subRoot: TreeNode) -> bool:
        def dfs(root1, root2):
            if root1 is None and root2 is None:
                return True
            if root1 is None or root2 is None:
                return False
            return (
                root1.val == root2.val
                and dfs(root1.left, root2.left)
                and dfs(root1.right, root2.right)
            )

        if root is None:
            return False
        return (
            dfs(root, subRoot)
            or self.isSubtree(root.left, subRoot)
            or self.isSubtree(root.right, subRoot)

        )
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def isSubtree(self, root: TreeNode, subRoot: TreeNode) -> bool:
        def dfs(root1, root2):
            if root1 is None and root2 is None:
                return True
            if root1 is None or root2 is None:
                return False
            return (
                root1.val == root2.val
                and dfs(root1.left, root2.left)
                and dfs(root1.right, root2.right)
            )

        if root is None:
            return False
        return (
            dfs(root, subRoot)
            or self.isSubtree(root.left, subRoot)
            or self.isSubtree(root.right, subRoot)

        )
```

#98 MEDIUM

Validate Binary Search Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS · BST

Lists: Grind 169

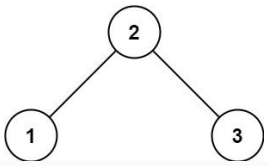
Problem:

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

A valid BST is defined as follows:

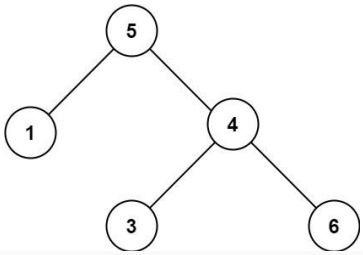
- The left subtree of a node contains only nodes with keys strictly less than the node's key.
- The right subtree of a node contains only nodes with keys strictly greater than the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:




```
Input: root = [2,1,3]
Output: true
```

Example 2:




```
Input: root = [5,1,4,null,null,3,6]
Output: false
Explanation: The root node's value is 5 but its right child's value is 4.
```

Constraints:

- The number of nodes in the tree is in the range [1, 104].
- -231 <= Node.val <= 231 - 1

Python

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val # Node's value
        self.left = left # Left child
        self.right = right # Right child

class Solution:
    def isValidBST(self, root: TreeNode) -> bool:
        # Helper function to recursively validate nodes
        def validate(node, low, high):
            if not node:
                return True # An empty node/subtree is valid
            # If current node's value is out of the allowed range, return False
            if not (low < node.val < high):
                return False
            # Recursively check the left subtree and right subtree
            return validate(node.left, low, node.val) and validate(node.right, node.val, high)

        return validate(root, float("-inf"), float("inf"))
```

Python

```python
class Solution:
  def isValidBST(self, root: TreeNode | None) -> bool:
    def isValidBST(root: TreeNode | None,
                   minNode: TreeNode | None, maxNode: TreeNode | None) -> bool:
      if not root:
        return True
      if minNode and root.val <= minNode.val:
        return False
      if maxNode and root.val >= maxNode.val:
        return False

      return (isValidBST(root.left, minNode, root) and
              isValidBST(root.right, root, maxNode))

    return isValidBST(root, None, None)
```

Python

```python
class Solution:
  def isValidBST(self, root: TreeNode | None) -> bool:
    stack = []
    pred = None

    while root or stack:
      while root:
        stack.append(root)
        root = root.left
      root = stack.pop()
      if pred and pred.val >= root.val:
        return False
      pred = root
      root = root.right

    return True
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        def dfs(root: Optional[TreeNode]) -> bool:
            if root is None:
                return True
            if not dfs(root.left):
                return False
            nonlocal prev
            if prev >= root.val:
                return False
            prev = root.val
            return dfs(root.right)

        prev = -inf
        return dfs(root)
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right


class Solution: # recursive
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        def dfs(root, mi, ma):
            if not root:
                return True
            return mi < root.val < ma and dfs(root.left, mi, root.val) and dfs(root.right, root.val, ma)
        return dfs(root, -math.inf, math.inf)


class Solution: # iterative
    def isValidBST(self, root: TreeNode) -> bool:
```



```python
        # store (node, lower, upper) tuples in a single queue
        queue = [(root, None, None)]

        # another option:
        # queue = [(root, -math.inf, math.inf)]

        while queue:
            node, lower, upper = queue.pop(0)

            if node is None:
                continue

            val = node.val
            if lower is not None and val <= lower:
                return False
            if upper is not None and val >= upper:
                return False

            queue.append((node.right, val, upper))
            queue.append((node.left, lower, val))

        return True

###########

class Solution:
  def isValidBST(self, root: Optional[TreeNode]) -> bool:
      def isValidBST(root: Optional[TreeNode],
                     minNode: Optional[TreeNode], maxNode: Optional[TreeNode]) -> bool:

        if not root:
          return True
        if minNode and root.val <= minNode.val:
          return False
        if maxNode and root.val >= maxNode.val:
          return False

        return isValidBST(root.left, minNode, root) and isValidBST(root.right, root, maxNode)

      return isValidBST(root, None, None)

###########

# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def isValidBST(self, root: TreeNode) -> bool:

        def dfs(root):
            nonlocal prev
            if root is None:
                return True
            if not dfs(root.left):
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def isValidBST(self, root: Optional[TreeNode]) -> bool:
        def dfs(root: Optional[TreeNode]) -> bool:
            if root is None:
                return True
            if not dfs(root.left):
                return False
            nonlocal prev
            if prev >= root.val:
                return False
            prev = root.val
            return dfs(root.right)

        prev = -inf
        return dfs(root)
```

#102 MEDIUM

Binary Tree Level Order Traversal

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

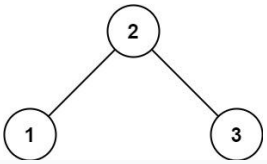
Tree · BFS

Lists: Grind 169

Problem:

Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

Example 1:




```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[9,20],[15,7]]
```


Example 2:

```
Input: root = [1]
Output: [[1]]
```

Example 3:

```
Input: root = []
Output: []
```

Constraints:

- The number of nodes in the tree is in the range [0, 2000].
- -1000 <= Node.val <= 1000

Python

Python

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

from collections import deque

class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        # Return an empty list if the tree is empty.
        if not root:
            return []

        result = [] # Final result list containing values of each level.
        queue = deque([root]) # Initialize the queue with the root node.

        # Process nodes level by level.
        while queue:
            level_size = len(queue) # Number of nodes at the current level.
            level_nodes = [] # List to hold node values at the current level.

            for _ in range(level_size):
                node = queue.popleft() # Dequeue the next node.

                level_nodes.append(node.val)
                # Enqueue left child if it exists.
                if node.left:
                    queue.append(node.left)
                # Enqueue right child if it exists.
                if node.right:
                    queue.append(node.right)
            # Append the current level's values to the result.
            result.append(level_nodes)
        return result
```

Python


```
class Solution:
    def levelOrder(self, root: TreeNode | None) -> list[list[int]]:
      if not root:
        return []

      ans = []
      q = collections.deque([root])

      while q:
        currLevel = []
        for _ in range(len(q)):
          node = q.popleft()
          currLevel.append(node.val)
          if node.left:
            q.append(node.left)
          if node.right:
            q.append(node.right)
        ans.append(currLevel)

      return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        ans = []
        if root is None:
            return ans
        q = deque([root])
        while q:
            t = []
            for _ in range(len(q)):
                node = q.popleft()
                t.append(node.val)
                if node.left:
                    q.append(node.left)
                if node.right:
                    q.append(node.right)
            ans.append(t)
        return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def levelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        ans = []
        if root is None:
            return ans
        q = deque([root])
        while q:
            t = []
            for _ in range(len(q)):
                node = q.popleft()
                t.append(node.val)
                if node.left:
                    q.append(node.left)
                if node.right:
                    q.append(node.right)
            ans.append(t)
        return ans
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

from collections import deque

class Solution:
    def levelOrder(self, root: TreeNode) -> List[List[int]]:
        # Return an empty list if the tree is empty.
        if not root:
            return []

        result = [] # Final result list containing values of each level.
        queue = deque([root]) # Initialize the queue with the root node.

        # Process nodes level by level.
        while queue:
            level_size = len(queue) # Number of nodes at the current level.
            level_nodes = [] # List to hold node values at the current level.

            for _ in range(level_size):
                node = queue.popleft() # Dequeue the next node.
```

```
            level_nodes.append(node.val)
            # Enqueue left child if it exists.
            if node.left:
                queue.append(node.left)
            # Enqueue right child if it exists.
            if node.right:
                queue.append(node.right)
        # Append the current level's values to the result.
        result.append(level_nodes)
    return result
```

#103 MEDIUM

Binary Tree Zigzag Level Order Traversal

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

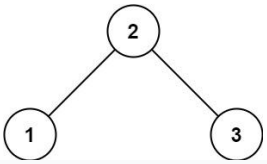
Tree · BFS

Lists: Grind 169

Problem:

Given the root of a binary tree, return the zigzag level order traversal of its nodes' values. (i.e., from left to right, then right to left for the next level and alternate between).

Example 1:




```
Input: root = [3,9,20,null,null,15,7]
Output: [[3],[20,9],[15,7]]
```

Example 2:

```
Input: root = [1]
Output: [[1]]
```

Example 3:

```
Input: root = []
Output: []
```

Constraints:

- The number of nodes in the tree is in the range [0, 2000].
- -100 <= Node.val <= 100

>> Brute Force [Trees & BST] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def zigzagLevelOrder(self, root):
        # Brute Force - collect inorder, then process sorted list
        vals = []
        def collect(node):
            if not node: return
            collect(node.left)
            vals.append(node.val)
            collect(node.right)
        collect(root)
        return vals[0] if vals else 0
```

Python

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

def zigzagLevelOrder(root):
    # If the tree is empty, return an empty list
    if not root:
        return []

    # Initialize the result list and the queue with the root node.
    result = []
    queue = [root]
    left_to_right = True # Flag to indicate the current direction of traversal

    # Process the tree level by level.
    while queue:
        level_size = len(queue)
        current_level = []

        # Process all nodes for the current level.
        for i in range(level_size):
            node = queue.pop(0) # Dequeue the node from the front of the queue
```



```
            current_level.append(node.val) # Append the node's value

            # Enqueue the left child if it exists.
            if node.left:
                queue.append(node.left)
            # Enqueue the right child if it exists.
            if node.right:
                queue.append(node.right)

        # If the current level's order is right-to-left, reverse the list.
        if not left_to_right:
            current_level.reverse()

        # Append the current level's values to the result.
        result.append(current_level)
        # Toggle the direction for the next level.
        left_to_right = not left_to_right

    return result
```

Python

```python
class Solution:
  def zigzagLevelOrder(self, root: TreeNode | None) -> list[list[int]]:
    if not root:
      return []

    ans = []
    dq = collections.deque([root])
    isLeftToRight = True

    while dq:
      currLevel = []
      for _ in range(len(dq)):
        if isLeftToRight:
          node = dq.popleft()
          currLevel.append(node.val)
          if node.left:
            dq.append(node.left)
          if node.right:
            dq.append(node.right)
        else:
          node = dq.pop()
          currLevel.append(node.val)
          if node.right:
            dq.appendleft(node.right)
          if node.left:
            dq.appendleft(node.left)
      ans.append(currLevel)
      isLeftToRight = not isLeftToRight

    return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def zigzagLevelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        ans = []
        if root is None:
            return ans
        q = deque([root])
        ans = []
        left = 1
        while q:
            t = []
            for _ in range(len(q)):
                node = q.popleft()
                t.append(node.val)
                if node.left:
                    q.append(node.left)
                if node.right:
                    q.append(node.right)
            ans.append(t if left else t[::-1])
            left ^= 1

        return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
...
can also use list.insert()

>>> my_list = [2, 3, 4]
>>> my_list.insert(0, 1) # Insert 1 at the head of the list
>>> print(my_list) # Output: [1, 2, 3, 4]


>>> a = deque([])
>>> a
deque([])
>>> a.append(1)
>>> a.append(2)
>>> a.append(3)
>>> a
deque([1, 2, 3])
>>>
>>> a.append(0, 555)
```



```
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: deque.append() takes exactly one argument (2 given)
>>> a.insert(0, 555)
>>> a
deque([555, 1, 2, 3])
'''

from collections import deque

class Solution:
    def zigzagLevelOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        ans = []
        if root is None:
            return ans
        q = deque([root])
        ans = []
        left = True
        while q:
            t = []
            for _ in range(len(q)):
                node = q.popleft()
                t.append(node.val)
                if node.left:
                    q.append(node.left)

                if node.right:
                    q.append(node.right)
            ans.append(t if left else t[::-1])
            left = (not left)
        return ans

###########

# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None
from collections import deque


class Solution(object):
    def zigzagLevelOrder(self, root):
        """
        :type root: TreeNode
        :rtype: List[List[int]]
        """
        stack = deque([root])
        ans = []

        odd = True
        while stack:
            level = []
            for k in range(0, len(stack)):
                top = stack.popleft()
```


[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

def zigzagLevelOrder(root):
    # If the tree is empty, return an empty list
    if not root:
        return []

    # Initialize the result list and the queue with the root node.
    result = []
    queue = [root]
    left_to_right = True # Flag to indicate the current direction of traversal

    # Process the tree level by level.
    while queue:
        level_size = len(queue)
        current_level = []

        # Process all nodes for the current level.
        for i in range(level_size):
            node = queue.pop(0) # Dequeue the node from the front of the queue
```

```python
            current_level.append(node.val) # Append the node's value

            # Enqueue the left child if it exists.
            if node.left:
                queue.append(node.left)
            # Enqueue the right child if it exists.
            if node.right:
                queue.append(node.right)

        # If the current level's order is right-to-left, reverse the list.
        if not left_to_right:
            current_level.reverse()

        # Append the current level's values to the result.
        result.append(current_level)
        # Toggle the direction for the next level.
        left_to_right = not left_to_right

    return result
```

#105 MEDIUM

Construct Binary Tree from Preorder and Inorder Traversal

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

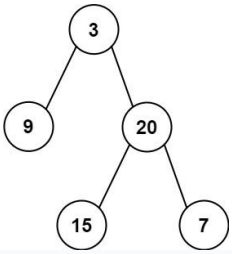
Array · Hash Table · Tree · DFS

Lists: Grind 169

Problem:

Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return the binary tree.

Example 1:




```
Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]
```

Example 2:

```
Input: preorder = [-1], inorder = [-1]
Output: [-1]
```

Constraints:

- 1 <= preorder.length <= 3000
- inorder.length == preorder.length
- -3000 <= preorder[i], inorder[i] <= 3000
- preorder and inorder consist of unique values.
- Each value of inorder also appears in preorder.
- preorder is guaranteed to be the preorder traversal of the tree.
- inorder is guaranteed to be the inorder traversal of the tree.

Python

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def buildTree(self, preorder: list[int], inorder: list[int]) -> TreeNode:
        # Create a hashmap to store value->index for inorder traversal
        inorder_index = {value: idx for idx, value in enumerate(inorder)}

        # Recursive helper function to build the tree
        def helper(pre_start, in_left, in_right):
            # Base case: if there are no elements to construct subtree
            if in_left > in_right:
                return None

            # Pick up pre_start element as a root
            root_val = preorder[pre_start]
            root = TreeNode(root_val)

            # Get the inorder index of the root
            in_index = inorder_index[root_val]

            # Number of nodes in left subtree
            left_subtree_size = in_index - in_left

            # Recursively build the left subtree
            root.left = helper(pre_start + 1, in_left, in_index - 1)
            # Recursively build the right subtree
            root.right = helper(pre_start + left_subtree_size + 1, in_index + 1, in_right)

            return root

        # Initiate recursion from the start of preorder and full inorder range
        return helper(0, 0, len(inorder) - 1)
```

Python


```python
class Solution:
    def buildTree(
        self,
        preorder: list[int],
        inorder: list[int],
    ) -> TreeNode | None:
        inToIndex = {num: i for i, num in enumerate(inorder)}

        def build(
            preStart: int,
            preEnd: int,
            inStart: int,
            inEnd: int,
        ) -> TreeNode | None:
            if preStart > preEnd:
                return None

            rootVal = preorder[preStart]
            rootInIndex = inToIndex[rootVal]
            leftSize = rootInIndex - inStart

            root = TreeNode(rootVal)
            root.left = build(preStart + 1, preStart + leftSize,
                              inStart, rootInIndex - 1)
            root.right = build(preStart + leftSize + 1,
                               preEnd, rootInIndex + 1, inEnd)
            return root

        return build(0, len(preorder) - 1, 0, len(inorder) - 1)
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        def dfs(i: int, j: int, n: int) -> Optional[TreeNode]:
            if n <= 0:
                return None
            v = preorder[i]
            k = d[v]
            l = dfs(i + 1, j, k - j)
            r = dfs(i + 1 + k - j, k + 1, n - k + j - 1)
            return TreeNode(v, l, r)

        d = {v: i for i, v in enumerate(inorder)}
        return dfs(0, 0, len(preorder))
```

Python


```python
class Solution:
    def getBinaryTrees(self, preOrder: List[int], inOrder: List[int]) -> List[TreeNode]:
        def dfs(i: int, j: int, n: int) -> List[TreeNode]:
            if n <= 0:
                return [None]
            v = preOrder[i]
            ans = []
            for k in d[v]:
                if j <= k < j + n:
                    for l in dfs(i + 1, j, k - j):
                        for r in dfs(i + 1 + k - j, k + 1, n - 1 - (k - j)):
                            ans.append(TreeNode(v, l, r))
            return ans

        d = defaultdict(list)
        for i, x in enumerate(inOrder):
            d[x].append(i)
        return dfs(0, 0, len(preOrder))
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

# best
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        if not preorder: # or, not inorder
            return None
        v = preorder[0]
        i = inorder.index(v)
        root = TreeNode(v)
        root.left = self.buildTree(preorder[1: i+1], inorder[:i])
        root.right = self.buildTree(preorder[i+1:], inorder[i+1:])
        return root


# inorder.index(preorder_val)
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        def dfs(pleft, pright, ileft, iright):
            if pleft > pright or ileft > iright:
```



```python
                return None
            k = inorder.index(preorder[pleft])
            root = TreeNode(preorder[pleft])
            root.left = dfs(pleft + 1, pleft + (k - ileft), ileft, k - 1)
            root.right = dfs(pleft + 1 + (k - ileft), pright, k + 1, iright)
            return root

        n = len(preorder)
        return dfs(0, n - 1, 0, n - 1)


# Build dict for val => index
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        def dfs(pleft, pright, ileft, iright):
            if pleft > pright or ileft > iright:
                return None
            v = preorder[pleft]
            k = d[v]
            root = TreeNode(v)
            root.left = dfs(pleft + 1, pleft + (k - ileft), ileft, k - 1)
            root.right = dfs(pleft + 1 + (k - ileft), pright, k + 1, iright)
            return root

        d = {v: i for i, v in enumerate(inorder)}
```

```
        n = len(preorder)
        return dfs(0, n - 1, 0, n - 1)

##########

...
search for index, also use .index(val)
  >>> a = [1,2,3,4,5]
  >>> a.index(3)
  2
...
class Solution(object):
    def buildTree(self, preorder, inorder):
        """
        :type preorder: List[int]
        :type inorder: List[int]
        :rtype: TreeNode
        """

        self.preindex = 0
        ind = {v: i for i, v in enumerate(inorder)}
        head = self.dc(0, len(preorder) - 1, preorder, inorder, ind)
        return head

    def dc(self, start, end, preorder, inorder, ind):
```

```
        if start <= end:
            mid = ind[preorder[self.preindex]]
            self.preindex += 1
            root = TreeNode(inorder[mid])
            root.left = self.dc(start, mid - 1, preorder, inorder, ind)
```


[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

# best
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        if not preorder: # or, not inorder
            return None
        v = preorder[0]
        i = inorder.index(v)
        root = TreeNode(v)
        root.left = self.buildTree(preorder[1: i+1], inorder[:i])
        root.right = self.buildTree(preorder[i+1:], inorder[i+1:])
        return root


# inorder.index(preorder_val)
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        def dfs(pleft, pright, ileft, iright):
            if pleft > pright or ileft > iright:
```


```
                return None
            k = inorder.index(preorder[pleft])
            root = TreeNode(preorder[pleft])
            root.left = dfs(pleft + 1, pleft + (k - ileft), ileft, k - 1)
            root.right = dfs(pleft + 1 + (k - ileft), pright, k + 1, iright)
            return root

        n = len(preorder)
        return dfs(0, n - 1, 0, n - 1)


# build dict for val => index
class Solution:
    def buildTree(self, preorder: List[int], inorder: List[int]) -> Optional[TreeNode]:
        def dfs(pleft, pright, ileft, iright):
            if pleft > pright or ileft > iright:
                return None
            v = preorder[pleft]
            k = d[v]
            root = TreeNode(v)
            root.left = dfs(pleft + 1, pleft + (k - ileft), ileft, k - 1)
            root.right = dfs(pleft + 1 + (k - ileft), pright, k + 1, iright)
            return root

        d = {v: i for i, v in enumerate(inorder)}
```

```python
        n = len(preorder)
        return dfs(0, n - 1, 0, n - 1)

###########

'''
search for index, also use .index(val)
  >>> a = [1,2,3,4,5]
  >>> a.index(3)
  2
'''
class Solution(object):
  def buildTree(self, preorder, inorder):
    """
    :type preorder: List[int]
    :type inorder: List[int]
    :rtype: TreeNode
    """

    self.preindex = 0
    ind = {v: i for i, v in enumerate(inorder)}
    head = self.dc(0, len(preorder) - 1, preorder, inorder, ind)
    return head

  def dc(self, start, end, preorder, inorder, ind):

    if start <= end:
      mid = ind[preorder[self.preindex]]
      self.preindex += 1
      root = TreeNode(inorder[mid])
      root.left = self.dc(start, mid - 1, preorder, inorder, ind)
```

#199 MEDIUM

Binary Tree Right Side View

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS · BFS

Lists: Grind 169

Problem:

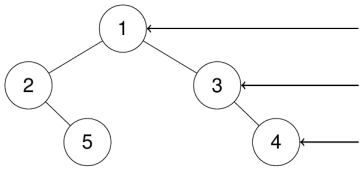
Given the root of a binary tree, imagine yourself standing on the right side of it, return the values of the nodes you can see ordered from top to bottom.

Example 1:

Input: root = [1,2,3,null,5,null,4]

Output: [1,3,4]

Explanation:

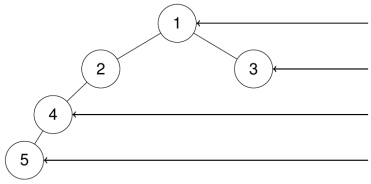


Example 2:

Input: root = [1,2,3,4,null,null,null,5]

Output: [1,3,4,5]

Explanation:



Example 3:

Input: root = [1,null,3]

Output: [1,3]

Example 4:

Input: root = []

Output: []

Constraints:

- The number of nodes in the tree is in the range [0, 100].
- -100 <= Node.val <= 100

>> Brute Force [Trees & BST] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def rightSideView(self, root):
        # Brute Force O(n) - BFS, take last node of each level
        from collections import deque
        if not root: return []
        result, queue = [], deque([root])
        while queue:
            level_size = len(queue)
            for i in range(level_size):
                node = queue.popleft()
                if i == level_size - 1: result.append(node.val)
                if node.left: queue.append(node.left)
                if node.right: queue.append(node.right)
        return result
```

Python

Python

```python
# Python solution using BFS
from collections import deque

class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val # Node value
        self.left = left # Left child
        self.right = right # Right child

def rightSideView(root):
    # If the tree is empty, return an empty list
    if not root:
        return []

    right_view = [] # List to store right side view
    queue = deque([root]) # Initialize queue with the root node

    while queue:
        level_size = len(queue) # Number of nodes in current level
        for i in range(level_size):
            node = queue.popleft() # Get the front node from the queue
            # If it's the last node in the current level, add it.
            if i == level_size - 1:
                right_view.append(node.val)
            # add left child if exists
```



```python
            if node.left:
                queue.append(node.left)
            # and right child if exists
            if node.right:
                queue.append(node.right)

    return right_view
```

Python

```python
class Solution:
  def rightSideView(self, root: TreeNode | None) -> list[int]:
    if not root:
      return []

    ans = []
    q = collections.deque([root])

    while q:
      size = len(q)
      for i in range(size):
        root = q.popleft()
        if i == size - 1:
          ans.append(root.val)
        if root.left:
          q.append(root.left)
        if root.right:
          q.append(root.right)

    return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        ans = []
        if root is None:
            return ans
        q = deque([root])
        while q:
            ans.append(q[0].val)
            for _ in range(len(q)):
                node = q.popleft()
                if node.right:
                    q.append(node.right)
                if node.left:
                    q.append(node.left)
        return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        ans = []
        if root is None:
            return ans
        q = deque([root])
        while q:
            ans.append(q[-1].val) # add last node of previous level traversal results
            for _ in range(len(q)):
                node = q.popleft()
                if node.left:
                    q.append(node.left)
                if node.right:
                    q.append(node.right)
        return ans

############

# Definition for a binary tree node.

# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None
from collections import deque


class Solution(object):
  def rightSideView(self, root):
    """
    :type root: TreeNode
    :rtype: List[int]
    """

    def dfs(root, h):
      if root:
        if h == len(ans):
          ans.append(root.val)
        # pre-order, all the way to the right
        dfs(root.right, h + 1)
        dfs(root.left, h + 1)

    ans = []
    dfs(root, 0)

    return ans
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def rightSideView(self, root: Optional[TreeNode]) -> List[int]:
        ans = []
        if root is None:
            return ans
        q = deque([root])
        while q:
            ans.append(q[-1].val) # add last node of previous level traversal results
            for _ in range(len(q)):
                node = q.popleft()
                if node.left:
                    q.append(node.left)
                if node.right:
                    q.append(node.right)
        return ans

############

# Definition for a binary tree node.

# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None
from collections import deque


class Solution(object):
  def rightSideView(self, root):
    """
    :type root: TreeNode
    :rtype: List[int]
    """

    def dfs(root, h):
      if root:
        if h == len(ans):
          ans.append(root.val)
        # pre-order, all the way to the right
        dfs(root.right, h + 1)
        dfs(root.left, h + 1)

    ans = []
    dfs(root, 0)

    return ans
```


#230 MEDIUM

Kth Smallest Element in a BST

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

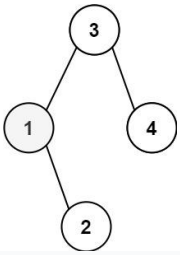
Tree · DFS · BST

Lists: Grind 169

Problem:

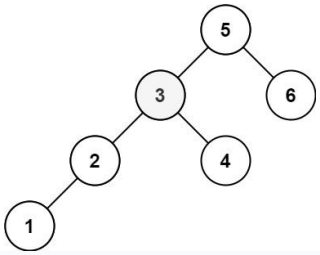
Given the root of a binary search tree, and an integer k, return the kth smallest value (1-indexed) of all the values of the nodes in the tree.

Example 1:




```
Input: root = [3,1,4,null,2], k = 1
Output: 1
```

Example 2:




```
Input: root = [5,3,6,2,4,null,null,1], k = 3
Output: 3
```

Constraints:

- The number of nodes in the tree is n.
- 1 <= k <= n <= 104
- 0 <= Node.val <= 104

Follow up: If the BST is modified often (i.e., we can do insert and delete operations) and you need to find the kth smallest frequently, how would you optimize?

Python

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def kthSmallest(self, root: TreeNode, k: int) -> int:
        # Initialize counter and result
        self.k = k
        self.ans = None

        # In-order traversal: left -> node -> right
        def inorder(node):
            if not node:
                return
            # Traverse left subtree
            inorder(node.left)
            # Process current node
            self.k -= 1
            if self.k == 0:
                self.ans = node.val
                return
            # Traverse right subtree if kth element not found

            inorder(node.right)

        inorder(root)
        return self.ans
```

Python

```python
class Solution:
    def kthSmallest(self, root: TreeNode | None, k: int) -> int:
        def countNodes(root: TreeNode | None) -> int:
            if not root:
                return 0
            return 1 + countNodes(root.left) + countNodes(root.right)

        leftCount = countNodes(root.left)

        if leftCount == k - 1:
            return root.val
        if leftCount >= k:
            return self.kthSmallest(root.left, k)
        return self.kthSmallest(root.right, k - 1 - leftCount) # leftCount < k
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        stk = []
        while root or stk:
            if root:
                stk.append(root)
                root = root.left
            else:
                root = stk.pop()
                k -= 1
                if k == 0:
                    return root.val
                root = root.right
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def kthSmallest(self, root: Optional[TreeNode], k: int) -> int:
        stk = []
        while root or stk:
            if root:
                stk.append(root)
                root = root.left
            else:
                root = stk.pop()
                k -= 1
                if k == 0:
                    return root.val
                root = root.right

###########

# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
```



```python
# self.val = val
# self.left = left
# self.right = right
class Solution_followup:
    def kthSmallest(self, root: TreeNode, k: int) -> int:
        node = self.build(root)
        return self.dfs(node, k)

    class MyTreeNode:
        def __init__(self, x):
            self.val = x
            self.count = 1 # default 1
            self.left = None
            self.right = None

    def build(self, root: TreeNode) -> MyTreeNode:
        if not root:
            return None
        node = self.MyTreeNode(root.val) # default count already is 1
        node.left = self.build(root.left)
        node.right = self.build(root.right)
        if node.left:
            node.count += node.left.count
        if node.right:
            node.count += node.right.count

        return node

    def dfs(self, node: MyTreeNode, k: int) -> int:
        if node.left:
            cnt = node.left.count
            if k < cnt + 1:
                return self.dfs(node.left, k)
            elif k > cnt + 1:
                return self.dfs(node.right, k - 1 - cnt)
            else: # k == cnt+1
                return node.val
        else:
            if k == 1: # cannot move to beginning of dfs()
                return node.val
            return self.dfs(node.right, k - 1)
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def kthSmallest(self, root: TreeNode, k: int) -> int:
        # Initialize counter and result
        self.k = k
        self.ans = None

        # In-order traversal: left -> node -> right
        def inorder(node):
            if not node:
                return
            # Traverse left subtree
            inorder(node.left)
            # Process current node
            self.k -= 1
            if self.k == 0:
                self.ans = node.val
                return
            # Traverse right subtree if kth element not found

            inorder(node.right)

        inorder(root)
        return self.ans
```

#235 MEDIUM

Lowest Common Ancestor of a Binary Search Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS · BST

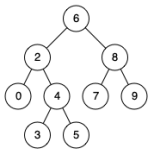
Lists: Grind 169

Problem:

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

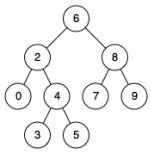
According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself)."

Example 1:




```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8
Output: 6
Explanation: The LCA of nodes 2 and 8 is 6.
```

Example 2:




```
Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 4
Output: 2
Explanation: The LCA of nodes 2 and 4 is 2, since a node can be a descendant of itself
according to the LCA definition.
```

Example 3:

```
Input: root = [2,1], p = 2, q = 1
Output: 2
```

Constraints:

- The number of nodes in the tree is in the range [2, 105].
- -109 <= Node.val <= 109
- All Node.val are unique.
- p != q
- p and q will exist in the BST.

>> Brute Force [Trees & BST] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def lowestCommonAncestor(self, root, p, q):
        # Brute Force O(n) - find root-to-node paths, find last common node
        def path_to(node, target):
            path = []
            while node:
                path.append(node)
                if target.val < node.val: node = node.left
                elif target.val > node.val: node = node.right
                else: break
            return path
        p_path = path_to(root, p)
        q_path = path_to(root, q)
        p_set = {n.val for n in p_path}
        for node in reversed(q_path):
            if node.val in p_set:
                return node
```

Python

Python

```python
def lowestCommonAncestor(root, p, q):
    # Iterate until we find the lowest common ancestor
    while root:
        # If both p and q are less than the current node, move to left subtree
        if p.val < root.val and q.val < root.val:
            root = root.left
        # If both are greater, move to right subtree
        elif p.val > root.val and q.val > root.val:
            root = root.right
        else:
            # Found the split point; return current node as LCA
            return root
```

Python

```python
class Solution:
  def lowestCommonAncestor(
      self,
      root: 'TreeNode',
      p: 'TreeNode',
      q: 'TreeNode',
  ) -> 'TreeNode':
    if root.val > max(p.val, q.val):
      return self.lowestCommonAncestor(root.left, p, q)
    if root.val < min(p.val, q.val):
      return self.lowestCommonAncestor(root.right, p, q)
    return root
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None


class Solution:
    def lowestCommonAncestor(
        self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode'
    ) -> 'TreeNode':
        while 1:
            if root.val < min(p.val, q.val):
                root = root.right
            elif root.val > max(p.val, q.val):
                root = root.left
            else:
                return root
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None

class Solution:
    def lowestCommonAncestor(
        self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode'
    ) -> 'TreeNode':
        while 1:
            if root.val < min(p.val, q.val): # no =, so root is p or q is in else block
                root = root.right
            elif root.val > max(p.val, q.val):
                root = root.left
            else:
                return root

###########

class Solution:
    def lowestCommonAncestor(self, root, p, q):
        if root is None:
            return root
```



```python
        if root.val > max(p.val, q.val):
            return self.lowestCommonAncestor(root.left, p, q)
        elif root.val < min(p.val, q.val):
            return self.lowestCommonAncestor(root.right, p, q)
        else:
            return root

###########

class Solution(object):
  def lowestCommonAncestor(self, root, p, q):
    """
    :type root: TreeNode
    :type p: TreeNode
    :type q: TreeNode
    :rtype: TreeNode
    """
    a, b = sorted((p.val, q.val))
    while not a <= root.val <= b:
      if a > root.val:
        root = root.right
      else:
        root = root.left
    return root
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
class Solution:
  def lowestCommonAncestor(
      self,
      root: 'TreeNode',
      p: 'TreeNode',
      q: 'TreeNode',
  ) -> 'TreeNode':
    if root.val > max(p.val, q.val):
      return self.lowestCommonAncestor(root.left, p, q)
    if root.val < min(p.val, q.val):
      return self.lowestCommonAncestor(root.right, p, q)
    return root
```

#236 MEDIUM

Lowest Common Ancestor of a Binary Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS

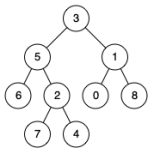
Lists: Grind 169

Problem:

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

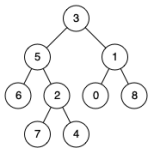
According to the definition of LCA on Wikipedia: "The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself)."

Example 1:




```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.
```

Example 2:




```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5, since a node can be a descendant of itself
according to the LCA definition.
```

Example 3:

```
Input: root = [1,2], p = 1, q = 2
Output: 1
```

Constraints:

- The number of nodes in the tree is in the range [2, 105].
- -109 <= Node.val <= 109
- All Node.val are unique.
- p != q

- p and q will exist in the tree.

Python

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
        # Base case: if the current node is None, return None.
        if root is None:
            return None

        # If the current node is either p or q, return the current node.
        if root == p or root == q:
            return root

        # Search for p and q in the left and right subtrees.
        left = self.lowestCommonAncestor(root.left, p, q)
        right = self.lowestCommonAncestor(root.right, p, q)

        # If both left and right are non-null, the current node is the LCA.
        if left and right:
            return root


        # Otherwise, return the non-null node.
        return left if left is not None else right
```

Python

```python
class Solution:
  def lowestCommonAncestor(
      self,
      root: 'TreeNode',
      p: 'TreeNode',
      q: 'TreeNode',
  ) -> 'TreeNode':
    if not root or root == p or root == q:
      return root
    left = self.lowestCommonAncestor(root.left, p, q)
    right = self.lowestCommonAncestor(root.right, p, q)
    if left and right:
      return root
    return left or right
```

Python

```python
class Solution:
   def lowestCommonAncestor(
       self,
       root: TreeNode | None,
       p: TreeNode | None,
       q: TreeNode | None,
   ) -> TreeNode | None:
       q_ = collections.deque([root])
       parent = {root: None}
       ancestors = set() # p's ancestors

       # Iterate until we find both p and q.
       while p not in parent or q not in parent:
         root = q_.popleft()
         if root.left:
           parent[root.left] = root
           q_.append(root.left)
         if root.right:
           parent[root.right] = root
           q_.append(root.right)

       # Insert all the p's ancestors.
       while p:
         ancestors.add(p)
         p = parent[p] #  p  becomes None in the end.

       # Go up from q until we meet any of p's ancestors.
       while q not in ancestors:
         q = parent[q]

       return q
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None


class Solution:
     def lowestCommonAncestor(
         self, root: "TreeNode", p: "TreeNode", q: "TreeNode"
     ) -> "TreeNode":
         if root in (None, p, q):
             return root
         left = self.lowestCommonAncestor(root.left, p, q)
         right = self.lowestCommonAncestor(root.right, p, q)
         return root if left and right else (left or right)
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None

class Solution:
     def lowestCommonAncestor(
         self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode'
     ) -> 'TreeNode':
         if root is None or root == p or root == q:
             return root
         left = self.lowestCommonAncestor(root.left, p, q)
         right = self.lowestCommonAncestor(root.right, p, q)
         return root if left and right else (left or right)

###########

# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None


class Solution(object):
   def lowestCommonAncestor(self, root, p, q):
     """
     :type root: TreeNode
     :type p: TreeNode
     :type q: TreeNode
     :rtype: TreeNode
     """

     if not root:
       return root

     left = self.lowestCommonAncestor(root.left, p, q)
     right = self.lowestCommonAncestor(root.right, p, q)

     if left and right:
       return root

     if root == p or root == q:
       return root

     if left:
       return left
     if right:

       return right
     return None
```

[⁴] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
     def lowestCommonAncestor(self, root: 'TreeNode', p: 'TreeNode', q: 'TreeNode') -> 'TreeNode':
         # Base cases: if the current node is None, return None.
         if root is None:
             return None

         # If the current node is either p or q, return the current node.
         if root == p or root == q:
             return root

         # Search for p and q in the left and right subtrees.
         left = self.lowestCommonAncestor(root.left, p, q)
         right = self.lowestCommonAncestor(root.right, p, q)

         # If both left and right are non-null, the current node is the LCA.
         if left and right:
             return root

         # Otherwise, return the non-null node.
         return left if left is not None else right
```

#250 MEDIUM

Count Univalue Subtrees

LeetCode↗ · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS

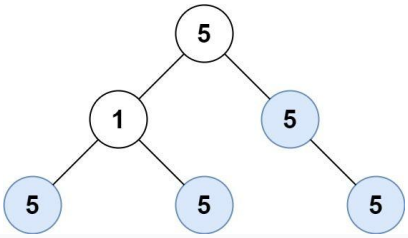
Lists: Premium 98

Problem:

Given the root of a binary tree, return the number of uni-value subtrees.

A uni-value subtree means all nodes of the subtree have the same value.

Example 1:




```
Input: root = [5,1,5,5,5,null,5]
Output: 4
```

Example 2:

```
Input: root = []
Output: 0
```

Example 3:

```
Input: root = [5,5,5,5,5,null,5]
Output: 6
```

Constraints:

- The number of the node in the tree will be in the range [0, 1000].
- -1000 <= Node.val <= 1000

Python

Python


```python
# Definition for a binary tree node.
class TreeNode:
     def __init__(self, val=0, left=None, right=None):
         self.val = val
         self.left = left
         self.right = right

class Solution:
     def countUnivalSubtrees(self, root: TreeNode) -> int:
         # Initialize count
         self.count = 0

         # Helper function performs postorder DFS
         def isUnival(node):
             # Base case: null nodes are considered univalue
             if node is None:
                 return True

             # Recursive calls for left and right subtrees
             leftUnival = isUnival(node.left)
             rightUnival = isUnival(node.right)

             # Check if left child exists and its value does not match current node's value
             if node.left and node.left.val != node.val:
                 leftUnival = False

             # Check if right child exists and its value does not match current node's value
             if node.right and node.right.val != node.val:
                 rightUnival = False

             # If either left or right subtree is not univalue, then the current subtree is not univalue
             if leftUnival and rightUnival:
                 # Increment the count if the subtree rooted at the current node is univalue
                 self.count += 1
                 return True
             else:
                 return False

         # Start recursion from root
         isUnival(root)
         return self.count

# Example usage:
# Construct the binary tree: [5,1,5,5,5,null,5]
# root = TreeNode(5)
# root.left = TreeNode(1, TreeNode(5), TreeNode(5))
# root.right = TreeNode(5, None, TreeNode(5))
# print(Solution().countUnivalSubtrees(root)) # Output: 4
```

Python


```python
class Solution:
   def countUnivalSubtrees(self, root: TreeNode | None) -> int:
     ans = 0

     def isUnival(root: TreeNode | None, val: int) -> bool:
       nonlocal ans
       if not root:
         return True

       if isUnival(root.left, root.val) & isUnival(root.right, root.val):
         ans += 1
         return root.val == val

       return False

     isUnival(root, math.inf)
     return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
     def countUnivalSubtrees(self, root: Optional[TreeNode]) -> int:
         def dfs(root):
             if root is None:
                 return True
             l, r = dfs(root.left), dfs(root.right)
             if not l or not r:
                 return False
             a = root.val if root.left is None else root.left.val
             b = root.val if root.right is None else root.right.val
             if a == b == root.val:
                 nonlocal ans
                 ans += 1
                 return True
             return False

         ans = 0
         dfs(root)
         return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def countUnivalSubtrees(self, root: TreeNode) -> int:
        count = 0

        def is_unival(node):
            nonlocal count
            if not node:
                return True

            left_unival = is_unival(node.left)
            right_unival = is_unival(node.right)

            if left_unival and right_unival:
                if node.left and node.val != node.left.val:
                    return False
                if node.right and node.val != node.right.val:
                    return False
                count += 1

                return True

            return False

        is_unival(root)
        return count


class Solution: # return 2 values
    def countUnivalSubtrees(self, root: TreeNode) -> int:
        def is_unival(node):
            if not node:
                return True, 0

            left_unival, left_count = is_unival(node.left)
            right_unival, right_count = is_unival(node.right)

            if left_unival and right_unival:
                if node.left and node.val != node.left.val:
                    return False, 0
                if node.right and node.val != node.right.val:
                    return False, 0
                return True, left_count + right_count + 1

            return False, 0


        _, count = is_unival(root)
        return count
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def countUnivalSubtrees(self, root: Optional[TreeNode]) -> int:
        def dfs(root):
            if root is None:
                return True
            l, r = dfs(root.left), dfs(root.right)
            if not l or not r:
                return False
            a = root.val if root.left is None else root.left.val
            b = root.val if root.right is None else root.right.val
            if a == b == root.val:
                nonlocal ans
                ans += 1
                return True
            return False

        ans = 0
        dfs(root)
        return ans
```

#285 MEDIUM

Inorder Successor in BST

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS · BST

Lists: Grind 169

Problem:

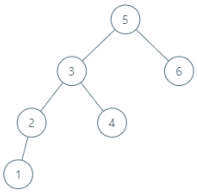
Given the root of a binary search tree and a node p in it, return the in-order successor of that node in the BST. If the given node has no in-order successor in the tree, return null.

The successor of a node p is the node with the smallest key greater than p.val.

Example 1:

```
Input: root = [2,1,3], p = 1
Output: 2
Explanation: 1's in-order successor node is 2. Note that both p and the return value is of
TreeNode type.
```


Example 2:




```
Input: root = [5,3,6,2,4,null,null,1], p = 6
Output: null
Explanation: There is no in-order successor of the current node, so the answer is null.
```

Constraints:

- The number of nodes in the tree is in the range [1, 104].
- -105 <= Node.val <= 105
- All Nodes will have unique values.

Python

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def inorderSuccessor(self, root: TreeNode, p: TreeNode) -> TreeNode:
        # If p has a right subtree, the successor is the left-most node in that subtree.
        if p.right:
            successor = p.right
            while successor.left:
                successor = successor.left
            return successor

        # Otherwise, start from the root and keep track of a potential successor.
        successor = None
        current = root
        while current:
            if p.val < current.val:
                # Current is a candidate, so record it and move left to find a smaller candidate
                successor = current
                current = current.left
            else:

                # If p.val is greater or equal, go right.
                current = current.right
        return successor
```

Python

```python
class Solution:
  def inorderSuccessor(
    self,
    root: TreeNode | None,
    p: TreeNode | None,
  ) -> TreeNode | None:
    if not root:
      return None
    if root.val <= p.val:
      return self.inorderSuccessor(root.right, p)
    return self.inorderSuccessor(root.left, p) or root
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None


class Solution:
    def inorderSuccessor(self, root: TreeNode, p: TreeNode) -> Optional[TreeNode]:
        ans = None
        while root:
            if root.val > p.val:
                ans = root
                root = root.left
            else:
                root = root.right
        return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None


class Solution:
    def inorderSuccessor(self, root: TreeNode, p: TreeNode) -> Optional[TreeNode]:
        ans = None
        while root:
            if root.val > p.val:
                ans = root
                root = root.left
            else:
                root = root.right
        return ans
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def inorderSuccessor(self, root: TreeNode, p: TreeNode) -> TreeNode:
        # If p has a right subtree, the successor is the left-most node in that subtree.
        if p.right:
            successor = p.right
            while successor.left:
                successor = successor.left
            return successor

        # Otherwise, start from the root and keep track of a potential successor.
        successor = None
        current = root
        while current:
            if p.val < current.val:
                # Current is a candidate, so record it and move left to find a smaller candidate
                successor = current
                current = current.left
            else:

                # If p.val is greater or equal, go right.
                current = current.right
        return successor
```


#298 MEDIUM

Binary Tree Longest Consecutive Sequence

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS

Lists: Premium 98

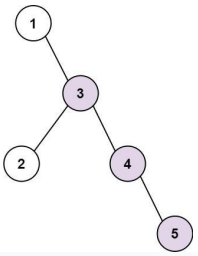
Problem:

Given the root of a binary tree, return the length of the longest consecutive sequence path.

A consecutive sequence path is a path where the values increase by one along the path.

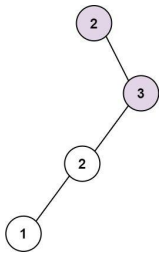
Note that the path can start at any node in the tree, and you cannot go from a node to its parent in the path.

Example 1:




```
Input: root = [1,null,3,2,4,null,null,null,5]
Output: 3
Explanation: Longest consecutive sequence path is 3-4-5, so return 3.
```

Example 2:




```
Input: root = [2,null,3,2,null,1]
Output: 2
Explanation: Longest consecutive sequence path is 2-3, not 3-2-1, so return 2.
```

Constraints:

- The number of nodes in the tree is in the range [1, 3 * 104].
- -3 * 104 <= Node.val <= 3 * 104

Python

Python

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def longestConsecutive(self, root: TreeNode) -> int:
        # Initialize a variable to keep track of the maximum sequence length.
        self.max_length = 0

        # Helper function to perform DFS traversal.
        def dfs(node, parent_val, current_length):
            if not node:
                return
            # Check if the current node's value is consecutive.
            if node.val == parent_val + 1:
                current_length += 1
            else:
                current_length = 1
            # Update the maximum sequence length found so far.
            self.max_length = max(self.max_length, current_length)
            # Recursively traverse the left and right children.
            dfs(node.left, node.val, current_length)

            dfs(node.right, node.val, current_length)

        # Start DFS with an initial condition that forces the first node to count.
        dfs(root, root.val - 1, 0)
        return self.max_length
```

Python


```python
class Solution:
    def longestConsecutive(self, root: TreeNode | None) -> int:
        if not root:
            return 0

        def dfs(root: TreeNode | None, target: int, length: int, maxLength: int) -> int:
            if not root:
                return maxLength
            if root.val == target:
                length += 1
                maxLength = max(maxLength, length)
            else:
                length = 1
            return max(dfs(root.left, root.val + 1, length, maxLength),
                       dfs(root.right, root.val + 1, length, maxLength))

        return dfs(root, root.val, 0, 0)
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def longestConsecutive(self, root: Optional[TreeNode]) -> int:
        def dfs(root: Optional[TreeNode]) -> int:
            if root is None:
                return 0
            l = dfs(root.left) + 1
            r = dfs(root.right) + 1
            if root.left and root.left.val - root.val != 1:
                l = 1
            if root.right and root.right.val - root.val != 1:
                r = 1
            t = max(l, r)
            nonlocal ans
            ans = max(ans, t)
            return t

        ans = 0
        dfs(root)
        return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

# dfs
class Solution:
    def longestConsecutive(self, root: TreeNode) -> int:
        maxLen = 0

        # curLen: consecutive length until parent node
        def dfs(root: TreeNode, lastVal: int, curLen: int) -> None:
            nonlocal maxLen
            if not root:
                return

            curLen = (curLen + 1) if (not lastVal) and root.val == lastVal + 1 else 1
            maxLen = max(maxLen, curLen)

            dfs(root.left, root.val, curLen)
            dfs(root.right, root.val, curLen)

        dfs(root, None, 0)

        return maxLen

###########

# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None

from collections import deque

# iterative, bfs
class Solution:
    def longestConsecutive(self, root: TreeNode) -> int:
        if not root:
            return 0

        max_length = 0
        # Queue elements are tuples: (current node, length of consecutive sequence)
        queue = deque([(root, 1)])

        while queue:
            current, length = queue.popleft()
```



```python
            max_length = max(max_length, length)

            for neighbor in [current.left, current.right]:
                if neighbor:
                    # Check if neighbor is consecutive
                    if neighbor.val == current.val + 1:
                        queue.append((neighbor, length + 1))
                    else:
                        queue.append((neighbor, 1))

        return max_length

###########

class Solution:
    def longestConsecutive(self, root: Optional[TreeNode]) -> int:
        def dfs(root: Optional[TreeNode]) -> int:
            if root is None:
                return 0
            l = dfs(root.left) + 1
            r = dfs(root.right) + 1
            if root.left and root.left.val - root.val != 1:
                l = 1
            if root.right and root.right.val - root.val != 1:
                r = 1

            t = max(l, r)
            nonlocal ans
            ans = max(ans, t)
            return t
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def longestConsecutive(self, root: TreeNode) -> int:
        # Initialize a variable to keep track of the maximum sequence length.
        self.max_length = 0

        # Helper function to perform DFS traversal.
        def dfs(node, parent_val, current_length):
            if not node:
                return
            # Check if the current node's value is consecutive.
            if node.val == parent_val + 1:
                current_length += 1
            else:
                current_length = 1
            # Update the maximum sequence length found so far.
            self.max_length = max(self.max_length, current_length)
            # Recursively traverse the left and right children.
            dfs(node.left, node.val, current_length)

            dfs(node.right, node.val, current_length)

        # Start DFS with an initial condition that forces the first node to count.
        dfs(root, root.val - 1, 0)
        return self.max_length
```

#314 MEDIUM

Binary Tree Vertical Order Traversal

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · Tree · BFS · Sorting

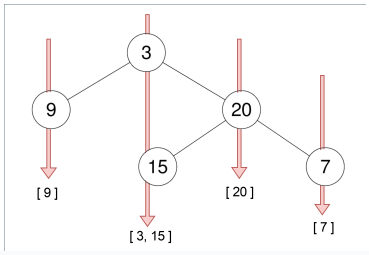
Lists: Premium 98

Problem:

Given the root of a binary tree, return the vertical order traversal of its nodes' values. (i.e., from top to bottom, column by column).

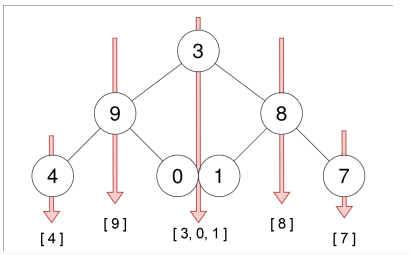
If two nodes are in the same row and column, the order should be from left to right.

Example 1:



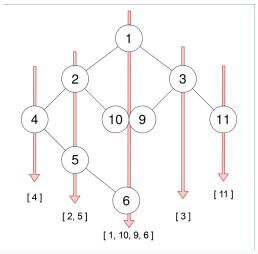

```
Input: root = [3,9,20,null,null,15,7]
Output: [[9],[3,15],[20],[7]]
```

Example 2:




```
Input: root = [3,9,8,4,0,1,7]
Output: [[4],[9],[3,0,1],[8],[7]]
```


Example 3:




```
Input: root = [1,2,3,4,10,9,11,null,5,null,null,null,null,null,null,null,6]
Output: [[4],[2,5],[1,10,9,6],[3],[11]]
```

Constraints:

- The number of nodes in the tree is in the range [0, 100].
- -100 <= Node.val <= 100

>> Brute Force [Trees & BST] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def verticalOrder(self, root: TreeNode):
        # Brute force - collect inorder, then process sorted list
        vals = []
        def collect(node):
            if not node: return
            collect(node.left)
            vals.append(node.val)
            collect(node.right)
        collect(root)
        return vals[0] if vals else 0
```

Python

Python


```python
from collections import deque, defaultdict

# Definition for a binary tree node.
class TreeNode:
    def __init__(self, x):
        self.val = x
        self.left = None
        self.right = None

class Solution:
    def verticalOrder(self, root: TreeNode):
        # Return empty list if the tree is empty
        if not root:
            return []

        # Dictionary to hold nodes for each column index
        col_table = defaultdict(list)
        # Use deque for efficient FIFO queue for BFS; store tuple (node, column)
        queue = deque([(root, 0)])

        while queue:
            node, col = queue.popleft()
            # Append current node value to the list of its column
            col_table[col].append(node.val)
            # If there is a left child, assign column col-1

            if node.left:
                queue.append((node.left, col - 1))
            # If there is a right child, assign column col+1
            if node.right:
                queue.append((node.right, col + 1))

        # Extract the columns in sorted order to build the final result
        sorted_cols = sorted(col_table.keys())
        result = [col_table[col] for col in sorted_cols]
        return result
```

Python


```python
class Solution:
  def verticalOrder(self, root: TreeNode | None) -> list[list[int]]:
    if not root:
      return []

    range_ = [0] * 2

    def getRange(root: TreeNode | None, x: int) -> None:
      if not root:
        return

      range_[0] = min(range_[0], x)
      range_[1] = max(range_[1], x)

      getRange(root.left, x - 1)
      getRange(root.right, x + 1)

    getRange(root, 0) # Get the leftmost and the rightmost x index.

    ans = [[] for _ in range(range_[1] - range_[0] + 1)]
    q = collections.deque([(root, -range_[0])]) # (TreeNode, x)

    while q:
      node, x = q.popleft()
      ans[x].append(node.val)

      if node.left:
        q.append((node.left, x - 1))
      if node.right:
        q.append((node.right, x + 1))

    return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def verticalOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        def dfs(root, depth, offset):
            if root is None:
                return
            d[offset].append((depth, root.val))
            dfs(root.left, depth + 1, offset - 1)
            dfs(root.right, depth + 1, offset + 1)

        d = defaultdict(list)
        dfs(root, 0, 0)
        ans = []
        for _, v in sorted(d.items()):
            v.sort(key=lambda x: x[0])
            ans.append([x[1] for x in v])
        return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

'''
>>> d = {1:'a', -1:'b', 10:'c', -100:'d'}
>>>
>>> d.items()
{1: 'a', 10: 'c', -100: 'd', -1: 'b'}

>>> ds = sorted(d.items())
>>> ds
[(-100, 'd'), (-1, 'b'), (1, 'a'), (10, 'c')]

>>> sorted(d.items(), key=lambda i: i[1])
[(1, 'a'), (-1, 'b'), (10, 'c'), (-100, 'd')]
>>> dict(sorted(d.items(), key=lambda i: i[1]))
{1: 'a', 10: 'c', -100: 'd', -1: 'b'}

>>> [v for i,v in sorted(d.items(), key=lambda i: i[0])]
['d', 'b', 'a', 'c']
'''


class Solution:
    def verticalOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        if root is None:
            return []
        q = deque([(root, 0)])
        d = defaultdict(list)
        while q:
            for _ in range(len(q)):
                root, offset = q.popleft()
                d[offset].append(root.val)
                if root.left:
                    q.append((root.left, offset - 1))
                if root.right:
                    q.append((root.right, offset + 1))
        return [v for _, v in sorted(d.items())]


##############

from sortedcontainers import SortedDict
from typing import List
from collections import deque

# similar to above, but use existing data structure
class Solution:
```



```python
    def verticalOrder(self, root: TreeNode) -> List[List[int]]:
        if not root:
            return []

        # Use SortedDict to automatically sort by column index
        sorted_dict = SortedDict()

        # Queue for breadth-first search; stores pairs of (node, column_index)
        queue = deque([(0, root)])

        while queue:
            node, column = queue.popleft()

            if column not in sorted_dict:
                sorted_dict[column] = [node.val]
            else:
                sorted_dict[column].append(node.val)

            # Add left and right children to the queue
            if node.left:
                queue.append((node.left, column - 1))
            if node.right:
                queue.append((node.right, column + 1))

        # Return the values from sorted_dict as a list of lists

        return list(sorted_dict.values())

##########

# another SortedDict, but different handling on None node, before enqueue vs after enqueue
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def verticalOrder(self, root: Optional[TreeNode]) -> List[List[int]]:
        def dfs(root, depth, offset):
            if root is None:
                return
            d[offset].append((depth, root.val))
            dfs(root.left, depth + 1, offset - 1)
            dfs(root.right, depth + 1, offset + 1)

        d = defaultdict(list)
        dfs(root, 0, 0)
        ans = []
        for _, v in sorted(d.items()):
            v.sort(key=lambda x: x[0])
            ans.append([x[1] for x in v])
        return ans
```

#333 MEDIUM

Largest BST Subtree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Dynamic Programming · Tree · DFS · BST

Lists: Premium 98

Problem:

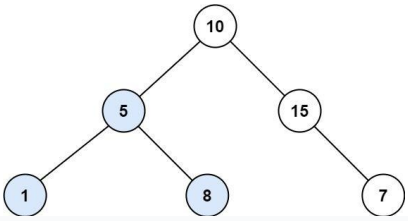
Given the root of a binary tree, find the largest subtree, which is also a Binary Search Tree (BST), where the largest means subtree has the largest number of nodes.

A Binary Search Tree (BST) is a tree in which all the nodes follow the below-mentioned properties:

- The left subtree values are less than the value of their parent (root) node's value.
- The right subtree values are greater than the value of their parent (root) node's value.

Note: A subtree must include all of its descendants.

Example 1:




```
Input: root = [10,5,15,1,8,null,7]
Output: 3
Explanation: The Largest BST Subtree in this case is the highlighted one. The return value is
the subtree's size, which is 3.
```

Example 2:

```
Input: root = [4,2,7,2,3,5,null,2,null,null,null,null,null,1]
Output: 2
```

Constraints:

- The number of nodes in the tree is in the range [0, 104].
- -104 <= Node.val <= 104

Follow up: Can you figure out ways to solve it with O(n) time complexity?

Python

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

class Solution:
    def largestBSTSubtree(self, root: TreeNode) -> int:
        # Global variable to store the size of the largest BST subtree found.
        self.max_bst_size = 0

        def helper(node):
            # Returns a tuple of: (isBST, size, min_val, max_val)
            if not node:
                # An empty node is considered a BST of size 0 with extreme min and max values.
                return (True, 0, float('inf'), float('-inf'))

            left_is_bst, left_size, left_min, left_max = helper(node.left)
            right_is_bst, right_size, right_min, right_max = helper(node.right)

            # Check if the current node forms a BST with its left and right subtrees.
            if left_is_bst and right_is_bst and node.val > left_max and node.val < right_min:
                current_size = left_size + right_size + 1
                self.max_bst_size = max(self.max_bst_size, current_size)

                current_min = min(left_min, node.val)
                current_max = max(right_max, node.val)
                return (True, current_size, current_min, current_max)
            else:
                # Not a BST: Return false and the size of the largest BST found so far in its subtrees.
                return (False, max(left_size, right_size), 0, 0)

        helper(root)
        return self.max_bst_size
```

Python


```python
from dataclasses import dataclass


@dataclass(frozen=True)
class T:
    mn: int # the minimum value in the subtree
    mx: int # the maximum value in the subtree
    size: int # the size of the subtree


class Solution:
    def largestBSTSubtree(self, root: TreeNode | None) -> int:
        def dfs(root: TreeNode | None) -> T:
            if not root:
                return T(math.inf, -math.inf, 0)

            l = dfs(root.left)
            r = dfs(root.right)

            if l.mx < root.val < r.mn:
                return T(min(l.mn, root.val), max(r.mx, root.val), 1 + l.size + r.size)

            # Mark one as invalid, but still record the size of children.
            # Return (-inf, inf) because no node will be > inf or < -inf.
            return T(-math.inf, math.inf, max(l.size, r.size))

        return dfs(root).size
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def largestBSTSubtree(self, root: Optional[TreeNode]) -> int:
        def dfs(root):
            if root is None:
                return inf, -inf, 0
            lmi, lmx, ln = dfs(root.left)
            rmi, rmx, rn = dfs(root.right)
            nonlocal ans
            if lmx < root.val < rmi:
                ans = max(ans, ln + rn + 1)
                return min(lmi, root.val), max(rmx, root.val), ln + rn + 1
            return -inf, inf, 0

        ans = 0
        dfs(root)
        return ans
```

Python


```
'''
>>> float("inf")
inf
>>> float("inf") + 1
inf
>>> import math
>>> float("inf") == math.inf
True
'''
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def largestBSTSubtree(self, root: Optional[TreeNode]) -> int:
        def dfs(root):
            if root is None:
                return inf, -inf, 0
            lmi, lmx, ln = dfs(root.left)
            rmi, rmx, rn = dfs(root.right)
            nonlocal ans
            # this if can check if left/right is a bst
            # eg. if left is not bst, the returned lmi=-inf and lmx=inf, so 'lmx < root.val < rmi' is false

            if lmx < root.val < rmi:
                ans = max(ans, ln + rn + 1)
                # lmi is guaranteed < root.val, except when left(right) is None
                return min(lmi, root.val), max(rmx, root.val), ln + rn + 1
            return -inf, inf, 0 # if this one not bst, then all parents will not be bst

        ans = 0
        dfs(root)
        return ans

###########

# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None

class Solution(object):
  def largestBSTSubtree(self, root):
    """
    :type root: TreeNode
    :rtype: int
    """
```



```
    def helper(root):
      if not root:
        return (0, 0, float("inf"), float("-inf")) # numBST, maxNumBST, min, max
      lnumBST, lmaxNumBST, lmin, lmax = helper(root.left)
      rnumBST, rmaxNumBST, rmin, rmax = helper(root.right)
      numBST = -1
      if lmax < root.val < rmin and lnumBST != -1 and rnumBST != -1:
        numBST = 1 + lnumBST + rnumBST
      maxNumBST = max(1, lmaxNumBST, rmaxNumBST, numBST)
      return numBST, maxNumBST, min(lmin, rmin, root.val), max(lmax, rmax, root.val)

    return helper(root)[1]
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
from dataclasses import dataclass


@dataclass(frozen=True)
class T:
    mn: int # the minimum value in the subtree
    mx: int # the maximum value in the subtree
    size: int # the size of the subtree


class Solution:
    def largestBSTSubtree(self, root: TreeNode | None) -> int:
        def dfs(root: TreeNode | None) -> T:
            if not root:
                return T(math.inf, -math.inf, 0)

            l = dfs(root.left)
            r = dfs(root.right)

            if l.mx < root.val < r.mn:
                return T(min(l.mn, root.val), max(r.mx, root.val), 1 + l.size + r.size)

            # Mark one as invalid, but still record the size of children.
            # Return (-inf, inf) because no node will be > inf or < -inf.
            return T(-math.inf, math.inf, max(l.size, r.size))

        return dfs(root).size
```

#366 MEDIUM

Find Leaves of Binary Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS · Sorting

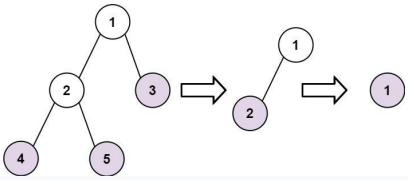
Lists: Premium 98

Problem:

Given the root of a binary tree, collect a tree's nodes as if you were doing this:

- Collect all the leaf nodes.
- Remove all the leaf nodes.
- Repeat until the tree is empty.

Example 1:




```
Input: root = [1,2,3,4,5]
Output: [[4,5,3],[2],[1]]
Explanation:
 [[3,5,4],[2],[1]] and [[3,4,5],[2],[1]] are also considered correct answers since per each
 level it does not matter the order on which elements are returned.
```

Example 2:

```
Input: root = [1]
Output: [[1]]
```

Constraints:

- The number of nodes in the tree is in the range [1, 100].
- -100 <= Node.val <= 100

>> Brute Force [Trees & BST] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def findLeaves(self, root: TreeNode):
        # Brute Force - collect inorder, then process sorted list
        vals = []
        def collect(node):
            if not node: return
            collect(node.left)
            vals.append(node.val)
            collect(node.right)
        collect(root)
        return vals[0] if vals else 0
```

Python

Python

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val # The value of the node.
        self.left = left # Left child node.
        self.right = right # Right child node.

class Solution:
    def findLeaves(self, root: TreeNode):
        result = [] # This will store lists of nodes at each removal step.

        def getHeight(node):
            # Base case: when the node is None, return 0.
            if not node:
                return 0
            # Recursively find the height of left and right subtrees.
            left_height = getHeight(node.left)
            right_height = getHeight(node.right)
            # Current node's height is max of left/right heights plus one.
            current_height = max(left_height, right_height) + 1
            # Ensure the result list has a sublist for the current height.
            if len(result) < current_height:
                result.append([])
            # Append the current node value to the corresponding height bucket.
            result[current_height - 1].append(node.val)

            return current_height

        getHeight(root) # Start recursion from the root.
        return result
```

Python

```python
class Solution:
   def findLeaves(self, root: TreeNode | None) -> list[list[int]]:
      ans = []

      def depth(root: TreeNode | None) -> int:
         """Returns the depth of the root (0-indexed)."""
         if not root:
            return -1

         l = depth(root.left)
         r = depth(root.right)
         h = 1 + max(l, r)

         if len(ans) == h: # Meet a leaf
            ans.append([])

         ans[h].append(root.val)
         return h

      depth(root)
      return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def findLeaves(self, root: Optional[TreeNode]) -> List[List[int]]:
        def dfs(root: Optional[TreeNode]) -> int:
            if root is None:
                return 0
            l, r = dfs(root.left), dfs(root.right)
            h = max(l, r)
            if len(ans) == h:
                ans.append([])
            ans[h].append(root.val)
            return h + 1

        ans = []
        dfs(root)
        return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def findLeaves(self, root: TreeNode) -> List[List[int]]:
        def dfs(root, prev, t):
            if root is None:
                return
            if root.left is None and root.right is None:
                t.append(root.val)
                if prev.left == root:
                    prev.left = None
                else:
                    prev.right = None
            dfs(root.left, root, t)
            dfs(root.right, root, t)

        res = []
        prev = TreeNode(left=root)
        while prev.left:
            t = []
            dfs(prev.left, prev, t)

            res.append(t)
        return res
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def findLeaves(self, root: TreeNode):
        result = [] # This will store lists of nodes at each removal step.

        def getHeight(node):
            # Base case: when the node is None, return 0.
            if not node:
                return 0
            # Recursively find the height of left and right subtrees.
            left_height = getHeight(node.left)
            right_height = getHeight(node.right)
            # Current node's height is max of left/right heights plus one.
            current_height = max(left_height, right_height) + 1
            # Ensure the result list has a sublist for the current height.
            if len(result) < current_height:
                result.append([])
            # Append the current node value to the corresponding height bucket.
            result[current_height - 1].append(node.val)

            return current_height

        getHeight(root) # Start recursion from the root.
        return result
```

#437 MEDIUM

Path Sum III

LeetDocs · SimplyLent · WalkCC · LeetCode · Editorial

Tree · DFS · Prefix Sum

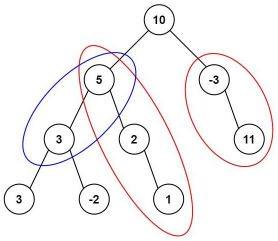
Lists: Grind 169

Problem:

Given the root of a binary tree and an integer targetSum, return the number of paths where the sum of the values along the path equals targetSum.

The path does not need to start or end at the root or a leaf, but it must go downwards (i.e., traveling only from parent nodes to child nodes).

Example 1:




```
Input: root = [10,5,-3,3,2,null,11,3,-2,null,1], targetSum = 8
Output: 3
Explanation: The paths that sum to 8 are shown.
```

Example 2:

```
Input: root = [5,4,8,11,null,13,4,7,2,null,null,5,1], targetSum = 22
Output: 3
```

Constraints:

- The number of nodes in the tree is in the range [0, 1000].
- -109 <= Node.val <= 109
- -1000 <= targetSum <= 1000

Python

Python


```python
# Python solution for Path Sum III
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val # value of the node
        self.left = left # left child
        self.right = right # right child

class Solution:
    def pathSum(self, root: TreeNode, targetSum: int) -> int:
        # Helper function to perform DFS traversal
        def dfs(node, current_sum, prefix_sums):
            if not node:
                return 0

            # Update current sum by including the current node's value
            current_sum += node.val
            # Check the number of times (current_sum - targetSum) has occurred
            num_paths = prefix_sums.get(current_sum - targetSum, 0)

            # Update the prefix sum hash map with the current sum
            prefix_sums[current_sum] = prefix_sums.get(current_sum, 0) + 1

            # Recursively search in the left and right subtree
            num_paths += dfs(node.left, current_sum, prefix_sums)
            num_paths += dfs(node.right, current_sum, prefix_sums)


            # Backtrack: remove the current sum count before returning to the parent node
            prefix_sums[current_sum] -= 1
            return num_paths

        # Initial prefix sum hashmap with 0 sum seen once (empty path)
        prefix_sums = {0: 1}
        return dfs(root, 0, prefix_sums)
```

Python

```python
class Solution:
   def pathSum(self, root: TreeNode | None, summ: int) -> int:
      if not root:
         return 0

      def dfs(root: TreeNode, summ: int) -> int:
         if not root:
            return 0
         return (int(summ == root.val) +
                 dfs(root.left, summ - root.val) +
                 dfs(root.right, summ - root.val))

      return (dfs(root, summ) +
              self.pathSum(root.left, summ) +
              self.pathSum(root.right, summ))
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> int:
        def dfs(node, s):
            if node is None:
                return 0
            s += node.val
            ans = cnt[s - targetSum]
            cnt[s] += 1
            ans += dfs(node.left, s)
            ans += dfs(node.right, s)
            cnt[s] -= 1
            return ans

        cnt = Counter({0: 1})
        return dfs(root, 0)
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def pathSum(self, root: Optional[TreeNode], targetSum: int) -> int:
        def dfs(node, s):
            if node is None:
                return 0
            s += node.val
            ans = cnt[s - targetSum]
            cnt[s] += 1
            ans += dfs(node.left, s)
            ans += dfs(node.right, s)
            cnt[s] -= 1
            return ans

        cnt = Counter({0: 1})
        return dfs(root, 0)
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Python solution for Path Sum III
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val # value of the node
        self.left = left # left child
        self.right = right # right child

class Solution:
    def pathSum(self, root: TreeNode, targetSum: int) -> int:
        # Helper function to perform DFS traversal
        def dfs(node, current_sum, prefix_sums):
            if not node:
                return 0

            # Update current sum by including the current node's value
            current_sum += node.val
            # Check the number of times (current_sum - targetSum) has occurred
            num_paths = prefix_sums.get(current_sum - targetSum, 0)

            # Update the prefix sum hash map with the current sum
            prefix_sums[current_sum] = prefix_sums.get(current_sum, 0) + 1

            # Recursively search in the left and right children
            num_paths += dfs(node.left, current_sum, prefix_sums)
            num_paths += dfs(node.right, current_sum, prefix_sums)

            # Backtrack: remove the current sum count before returning to the parent node
            prefix_sums[current_sum] -= 1
            return num_paths

        # Initial prefix sum hashmap with 0 sum seen once (empty path)
        prefix_sums = {0: 1}
        return dfs(root, 0, prefix_sums)
```

#545 MEDIUM

Boundary of Binary Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS

Lists: Premium 98

Problem:

The boundary of a binary tree is the concatenation of the root, the left boundary, the leaves ordered from left-to-right, and the reverse order of the right boundary.

The left boundary is the set of nodes defined by the following:

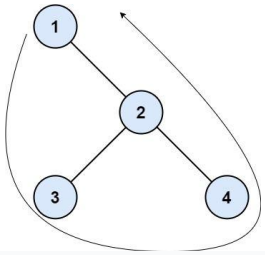
- The root node's left child is in the left boundary. If the root does not have a left child, then the left boundary is empty.
- If a node is in the left boundary and has a left child, then the left child is in the left boundary.
- If a node is in the left boundary, has no left child, but has a right child, then the right child is in the left boundary.
- The leftmost leaf is not in the left boundary.

The right boundary is similar to the left boundary, except it is the right side of the root's right subtree. Again, the leaf is not part of the right boundary, and the right boundary is empty if the root does not have a right child.

The leaves are nodes that do not have any children. For this problem, the root is not a leaf.

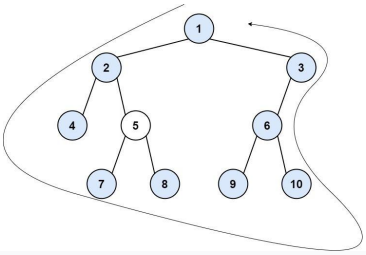
Given the root of a binary tree, return the values of its boundary.

Example 1:




```
Input: root = [1,null,2,3,4]
Output: [1,3,4,2]
Explanation:
- The left boundary is empty because the root does not have a left child.
- The right boundary follows the path starting from the root's right child 2 -> 4.
4 is a leaf, so the right boundary is [2].
- The leaves from left to right are [3,4].
Concatenating everything results in [1] + [] + [3,4] + [2] = [1,3,4,2].
```

Example 2:




```
Input: root = [1,2,3,4,5,6,null,null,null,7,8,9,10]
Output: [1,2,4,7,8,9,10,6,3]
Explanation:
- The left boundary follows the path starting from the root's left child 2 -> 4.
4 is a leaf, so the left boundary is [2].
- The right boundary follows the path starting from the root's right child 3 -> 6 -> 10.
10 is a leaf, so the right boundary is [3,6], and in reverse order is [6,3].
- The leaves from left to right are [4,7,8,9,10].
Concatenating everything results in [1] + [2] + [4,7,8,9,10] + [6,3] = [1,2,4,7,8,9,10,6,3].
```

Constraints:

- The number of nodes in the tree is in the range [1, 104].
- -1000 <= Node.val <= 1000

Python

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def boundaryOfBinaryTree(self, root: TreeNode) -> list:
        if not root:
            return []

        # Check if a node is a leaf.
        def isLeaf(node):
            return node and not node.left and not node.right

        # Add left boundary nodes (excluding leaves).
        def addLeftBoundary(node, boundary):
            while node:
                if not isLeaf(node):
                    boundary.append(node.val)
                # Prefer left child, if not present then right child.
                if node.left:
                    node = node.left
                else:
                    node = node.right

        # Add leaf nodes using DFS.
        def addLeaves(node, leaves):
            if not node:
                return
            if isLeaf(node):
                leaves.append(node.val)
            else:
                addLeaves(node.left, leaves)
                addLeaves(node.right, leaves)

        # Add right boundary nodes in reverse order (excluding leaves).
        def addRightBoundary(node, boundary):
            temp = []
            while node:
                if not isLeaf(node):
                    temp.append(node.val)
                if node.right:
                    node = node.right
                else:
                    node = node.left
            # Append in reverse order.
            while temp:
                boundary.append(temp.pop())
```



```python
        boundary = []
        # Add root value if it is not a leaf.
        if not isLeaf(root):
            boundary.append(root.val)

        addLeftBoundary(root.left, boundary)
        addLeaves(root, boundary)
        addRightBoundary(root.right, boundary)

        return boundary

# Example usage:
# Construct a binary tree and call boundaryOfBinaryTree(root)
```

Python

```python
class Solution:
    def boundaryOfBinaryTree(self, root: TreeNode | None) -> list[int]:
        if not root:
            return []

        ans = [root.val]

        def dfs(root: TreeNode | None, lb: bool, rb: bool):
            """
            1. root.left is left boundary if root is left boundary.
               root.right if left boundary if root.left is None.
            2. Same applys for right boundary.
            3. If root is left boundary, add it before 2 children - preorder.
               If root is right boundary, add it after 2 children - postorder.
            4. A leaf that is neighter left/right boundary belongs to the bottom.
            """
            if not root:
                return
            if lb:
                ans.append(root.val)
            if not lb and not rb and not root.left and not root.right:
                ans.append(root.val)

            dfs(root.left, lb, rb and not root.right)
            dfs(root.right, lb and not root.left, rb)

            if rb:
                ans.append(root.val)

        dfs(root.left, True, False)
        dfs(root.right, False, True)
        return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def boundaryOfBinaryTree(self, root: Optional[TreeNode]) -> List[int]:
        def dfs(nums: List[int], root: Optional[TreeNode], i: int):
            if root is None:
                return
            if i == 0:
                if root.left != root.right:
                    nums.append(root.val)
                    if root.left:
                        dfs(nums, root.left, i)
                    else:
                        dfs(nums, root.right, i)
            elif i == 1:
                if root.left == root.right:
                    nums.append(root.val)
                else:
                    dfs(nums, root.left, i)
                    dfs(nums, root.right, i)
            else:
                if root.left != root.right:
                    nums.append(root.val)
                    if root.right:
                        dfs(nums, root.right, i)
                    else:
                        dfs(nums, root.left, i)

        ans = [root.val]
        if root.left == root.right:
            return ans
        left, leaves, right = [], [], []
        dfs(left, root.left, 0)
        dfs(leaves, root, 1)
        dfs(right, root.right, 2)
        ans += left + leaves + right[::-1]
        return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def boundaryOfBinaryTree(self, root: TreeNode) -> List[int]:
        self.res = []
        if not root:
            return self.res
        # root
        if not self.is_leaf(root):
            self.res.append(root.val)

        # left boundary
        t = root.left
        while t:
            if not self.is_leaf(t):
                self.res.append(t.val)
            t = t.left if t.left else t.right

        # leaves
        self.add_leaves(root)

        # right boundary(reverse order)
        s = []
        t = root.right
        while t:
            if not self.is_leaf(t):
                s.append(t.val)
            t = t.right if t.right else t.left
        while s:
            self.res.append(s.pop())

        # output
        return self.res

    def add_leaves(self, root):
        if self.is_leaf(root):
            self.res.append(root.val)
            return
        if root.left:
            self.add_leaves(root.left)
        if root.right:
            self.add_leaves(root.right)

    def is_leaf(self, node) -> bool:
        return node and node.left is None and node.right is None
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
class Solution:
    def boundaryOfBinaryTree(self, root: TreeNode | None) -> list[int]:
        if not root:
            return []

        ans = [root.val]

        def dfs(root: TreeNode | None, lb: bool, rb: bool):
            """
            1. root.left is left boundary if root is left boundary.
               root.right if left boundary if root.left is None.
            2. Same applys for right boundary.
            3. If root is left boundary, add it before 2 children — preorder.
               If root is right boundary, add it after 2 children — postorder.
            4. A leaf that is neighter left/right boundary belongs to the bottom.
            """
            if not root:
                return
            if lb:
                ans.append(root.val)
            if not lb and not rb and not root.left and not root.right:
                ans.append(root.val)

            dfs(root.left, lb, rb and not root.right)
            dfs(root.right, lb and not root.left, rb)

            if rb:
                ans.append(root.val)

        dfs(root.left, True, False)
        dfs(root.right, False, True)
        return ans
```

#549 MEDIUM

Binary Tree Longest Consecutive Sequence II

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS

Lists: Premium 98

Problem:

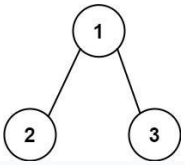
Given the root of a binary tree, return the length of the longest consecutive path in the tree.

A consecutive path is a path where the values of the consecutive nodes in the path differ by one. This path can be either increasing or decreasing.

- For example, [1,2,3,4] and [4,3,2,1] are both considered valid, but the path [1,2,4,3] is not valid.

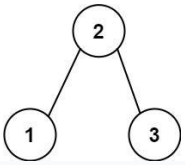
On the other hand, the path can be in the child-Parent-child order, where not necessarily be parent-child order.

Example 1:




```
Input: root = [1,2,3]
Output: 2
Explanation: The longest consecutive path is [1, 2] or [2, 1].
```

Example 2:




```
Input: root = [2,1,3]
Output: 3
Explanation: The longest consecutive path is [1, 2, 3] or [3, 2, 1].
```

Constraints:

- The number of nodes in the tree is in the range [1, 3 * 104].
- -3 * 104 <= Node.val <= 3 * 104

Python

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def longestConsecutive(self, root: TreeNode) -> int:
        # Initialize global maximum path length variable
        self.max_length = 0

        def dfs(node):
            if not node:
                return (0, 0) # (increasing, decreasing)

            inc = dec = 1 # Each node counts as length 1

            # Process left child
            if node.left:
                left_inc, left_dec = dfs(node.left)
                # If left child's value is consecutive increasing with respect to node
                if node.left.val == node.val + 1:
                    inc = max(inc, left_inc + 1)
                # If left child's value is consecutive decreasing with respect to node
                elif node.left.val == node.val - 1:
                    dec = max(dec, left_dec + 1)

            # Process right child
            if node.right:
                right_inc, right_dec = dfs(node.right)
                # If right child's value is consecutive increasing with respect to node
                if node.right.val == node.val + 1:
                    inc = max(inc, right_inc + 1)
                # If right child's value is consecutive decreasing with respect to node
                elif node.right.val == node.val - 1:
                    dec = max(dec, right_dec + 1)

            # Update the global maximum path by combining increasing and decreasing paths
            self.max_length = max(self.max_length, inc + dec - 1)
            return (inc, dec)

        dfs(root)
        return self.max_length
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def longestConsecutive(self, root: TreeNode) -> int:
        def dfs(root):
            if root is None:
                return [0, 0]
            nonlocal ans
            incr = decr = 1
            i1, d1 = dfs(root.left)
            i2, d2 = dfs(root.right)
            if root.left:
                if root.left.val + 1 == root.val:
                    incr = i1 + 1
                if root.left.val - 1 == root.val:
                    decr = d1 + 1
            if root.right:
                if root.right.val + 1 == root.val:
                    incr = max(incr, i2 + 1)
                if root.right.val - 1 == root.val:
                    decr = max(decr, d2 + 1)

            ans = max(ans, incr + decr - 1)
            return [incr, decr]

        ans = 0
        dfs(root)
        return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None

'''
>>> float('-inf')+1
-inf
>>> float('-inf')-1
-inf
'''
class Solution(object):
    def longestConsecutive(self, root):
        """
        :type root: TreeNode
        :rtype: int
        """

        def dfs(root):
            if not root:
                # val, increasing length, decreasing length, max length
                return float('-inf'), 0, 0, 0
            # inc/dec starting from root

            inc = dec = 1
            left, leftInc, leftDec, leftMax = dfs(root.left)
            right, rightInc, rightDec, rightMax = dfs(root.right)
            if root.val + 1 == left:
                inc = max(leftInc + 1, inc)
            if root.val - 1 == left:
                dec = max(leftDec + 1, dec)
            if root.val + 1 == right:
                inc = max(rightInc + 1, inc)
            if root.val - 1 == right:
                dec = max(rightDec + 1, dec)
            # return max val, max of children subtree, not current root's tree
            return root.val, inc, dec, max(inc + dec - 1, leftMax, rightMax, inc, dec)
            # -1 because root is counted twice, even left/right both null

        return dfs(root)[3]

###########

# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
```

```python
class Solution:
    def longestConsecutive(self, root: TreeNode) -> int:
        def dfs(root):
            if root is None:
                return [0, 0]
            nonlocal ans
            incr = decr = 1
            i1, d1 = dfs(root.left)
            i2, d2 = dfs(root.right)
            if root.left:
                if root.left.val + 1 == root.val:
                    incr = i1 + 1
                if root.left.val - 1 == root.val:
                    decr = d1 + 1
            if root.right:
                if root.right.val + 1 == root.val:
                    incr = max(incr, i2 + 1) # incr1 and incr2 must be in different tree branches, eg. left child cannot be both increasing and decreasing. Same as decr1 and decr2
                if root.right.val - 1 == root.val:
                    decr = max(decr, d2 + 1)
            ans = max(ans, incr + decr - 1) # -1 because root is counted twice, even left/right both null
            return [incr, decr]

        ans = 0
        dfs(root)
        return ans

##########

class Solution: # issue with this solution, duplicated dfs() search
    def longestConsecutive(self, root: TreeNode) -> int:
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def longestConsecutive(self, root: TreeNode) -> int:
        # Initialize global maximum path length variable
        self.max_length = 0

        def dfs(node):
            if not node:
                return (0, 0) # (increasing, decreasing)

            inc = dec = 1 # Each node counts as length 1

            # Process left child
            if node.left:
                left_inc, left_dec = dfs(node.left)
                # If left child's value is consecutive increasing with respect to node
                if node.left.val == node.val + 1:
                    inc = max(inc, left_inc + 1)
                # If left child's value is consecutive decreasing with respect to node

                elif node.left.val == node.val - 1:
                    dec = max(dec, left_dec + 1)

            # Process right child
            if node.right:
                right_inc, right_dec = dfs(node.right)
                # If right child's value is consecutive increasing with respect to node
                if node.right.val == node.val + 1:
                    inc = max(inc, right_inc + 1)
                # If right child's value is consecutive decreasing with respect to node
                elif node.right.val == node.val - 1:
                    dec = max(dec, right_dec + 1)

            # Update the global maximum path by combining increasing and decreasing paths
            self.max_length = max(self.max_length, inc + dec - 1)
            return (inc, dec)

        dfs(root)
        return self.max_length
```

#582 MEDIUM

Kill Process

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Array · Hash Table · Tree · DFS · BFS

Lists: Premium 98

Problem:

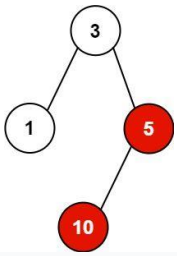
You have n processes forming a rooted tree structure. You are given two integer arrays pid and ppid, where pid[i] is the ID of the ith process and ppid[i] is the ID of the ith process's parent process.

Each process has only one parent process but may have multiple children processes. Only one process has ppid[i] = 0, which means this process has no parent process (the root of the tree).

When a process is killed, all of its children processes will also be killed.

Given an integer kill representing the ID of a process you want to kill, return a list of the IDs of the processes that will be killed. You may return the answer in any order.

Example 1:




```
Input: pid = [1,3,10,5], ppid = [3,0,5,3], kill = 5
Output: [5,10]
Explanation: The processes colored in red are the processes that should be killed.
```

Example 2:

```
Input: pid = [1], ppid = [0], kill = 1
Output: [1]
```

Constraints:

- n == pid.length
- n == ppid.length
- 1 <= n <= 5 * 104
- 1 <= pid[i] <= 5 * 104
- 0 <= ppid[i] <= 5 * 104
- Only one process has no parent.
- All the values of pid are unique.
- kill is guaranteed to be in pid.

>> Brute Force [Trees & BST] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def killProcess(self, pid, ppid, kill):
        # Brute Force - collect inorder, then process sorted list
        vals = []
        def collect(node):
            if not node: return
            collect(node.left)
            vals.append(node.val)
            collect(node.right)
        collect(pid)
        return vals[0] if vals else 0
```

Python

Python

```python
# Python solution using BFS

from collections import deque

def killProcess(pid, ppid, kill):
    # Build a mapping from parent process id to list of child process ids
    tree = {}
    for child, parent in zip(pid, ppid):
        if parent not in tree:
            tree[parent] = []
        tree[parent].append(child)

    # Use a deque for BFS traversal
    result = []
    queue = deque([kill])

    while queue:
        current = queue.popleft() # Get process from the left of the queue
        result.append(current) # Add the current process to the result
        if current in tree:
            # Enqueue all children of the current process
            for child in tree[current]:
                queue.append(child)

    return result

# Example usage:
# pid = [1, 3, 10, 5]
# ppid = [3, 0, 5, 3]
# kill = 5
# Expected output: [5, 10]
```

Python


```python
class Solution:
    def killProcess(
        self,
        pid: list[int],
        ppid: list[int],
        kill: int,
    ) -> list[int]:
        ans = []
        tree = collections.defaultdict(list)

        for v, u in zip(pid, ppid):
          if u == 0:
            continue
          tree[u].append(v)

        def dfs(u: int) -> None:
          ans.append(u)
          for v in tree.get(u, []):
            dfs(v)

        dfs(kill)
        return ans
```

Python

```python
class Solution:
    def killProcess(self, pid: List[int], ppid: List[int], kill: int) -> List[int]:
        def dfs(i: int):
            ans.append(i)
            for j in g[i]:
                dfs(j)

        g = defaultdict(list)
        for i, p in zip(pid, ppid):
            g[p].append(i)
        ans = []
        dfs(kill)
        return ans
```

Python

```python
class Solution:
    def killProcess(self, pid: List[int], ppid: List[int], kill: int) -> List[int]:
        def dfs(i: int):
            ans.append(i)
            for j in g[i]:
                dfs(j)

        g = defaultdict(list)
        for i, p in zip(pid, ppid):
            g[p].append(i)
        ans = []
        dfs(kill)
        return ans
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
class Solution:
    def killProcess(
        self,
        pid: list[int],
        ppid: list[int],
        kill: int,
    ) -> list[int]:
        ans = []
        tree = collections.defaultdict(list)

        for v, u in zip(pid, ppid):
          if u == 0:
            continue
          tree[u].append(v)

        def dfs(u: int) -> None:
          ans.append(u)
          for v in tree.get(u, []):
            dfs(v)

        dfs(kill)
        return ans
```

#662 MEDIUM

Maximum Width of Binary Tree

LeetDocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · BFS

Lists: Grind 169

Problem:

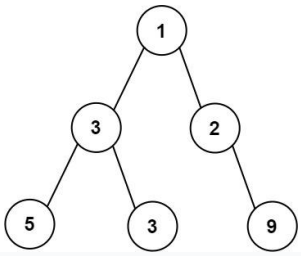
Given the root of a binary tree, return the maximum width of the given tree.

The maximum width of a tree is the maximum width among all levels.

The width of one level is defined as the length between the end-nodes (the leftmost and rightmost non-null nodes), where the null nodes between the end-nodes that would be present in a complete binary tree extending down to that level are also counted into the length calculation.

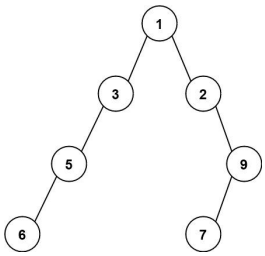
It is guaranteed that the answer will in the range of a 32-bit signed integer.

Example 1:



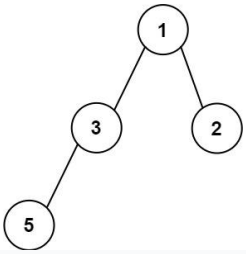

```
Input: root = [1,3,2,5,3,null,9]
Output: 4
Explanation: The maximum width exists in the third level with length 4 (5,3,null,9).
```

Example 2:




```
Input: root = [1,3,2,5,null,null,9,6,null,7]
Output: 7
Explanation: The maximum width exists in the fourth level with length 7
(6,null,null,null,null,null,7).
```

Example 3:




```
Input: root = [1,3,2,5]
Output: 2
Explanation: The maximum width exists in the second level with length 2 (3,2).
```

Constraints:

- The number of nodes in the tree is in the range [1, 3000].
- -100 <= Node.val <= 100

Python

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

from collections import deque

class Solution:
    def widthOfBinaryTree(self, root: TreeNode) -> int:
        if not root:
            return 0

        max_width = 0
        # Initialize queue with a tuple of (node, index)
        queue = deque([(root, 1)])

        while queue:
            level_length = len(queue)
            # Get the index of first node at current level
            _, first_index = queue[0]
            # Process nodes at current level
            for i in range(level_length):
                node, index = queue.popleft()

                # Check and add left child with index 2 * index
                if node.left:
                    queue.append((node.left, 2 * index))
                # Check and add right child with index 2 * index + 1
                if node.right:
                    queue.append((node.right, 2 * index + 1))
                # For the last node, calculate width of the level
                if i == level_length - 1:
                    current_width = index - first_index + 1
                    max_width = max(max_width, current_width)
        return max_width
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        ans = 0
        q = deque([(root, 1)])
        while q:
            ans = max(ans, q[-1][1] - q[0][1] + 1)
            for _ in range(len(q)):
                root, i = q.popleft()
                if root.left:
                    q.append((root.left, i << 1))
                if root.right:
                    q.append((root.right, i << 1 | 1))
        return ans
```


Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def widthOfBinaryTree(self, root: Optional[TreeNode]) -> int:
        ans = 0
        q = deque([(root, 1)])
        while q:
            ans = max(ans, q[-1][1] - q[0][1] + 1)
            for _ in range(len(q)):
                root, i = q.popleft()
                if root.left:
                    q.append((root.left, i << 1))
                if root.right:
                    q.append((root.right, i << 1 | 1))
        return ans
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right

from collections import deque

class Solution:
    def widthOfBinaryTree(self, root: TreeNode) -> int:
        if not root:
            return 0

        max_width = 0
        # Initialize queue with a tuple of (node, index)
        queue = deque([(root, 1)])

        while queue:
            level_length = len(queue)
            # Get the index of first node at current level
            _, first_index = queue[0]
            # Process nodes at current level
            for i in range(level_length):
                node, index = queue.popleft()
```



```python
                # Check and add left child with index 2 * index
                if node.left:
                    queue.append((node.left, 2 * index))
                # Check and add right child with index 2 * index + 1
                if node.right:
                    queue.append((node.right, 2 * index + 1))
                # For the last node, calculate width of the level
                if i == level_length - 1:
                    current_width = index - first_index + 1
                    max_width = max(max_width, current_width)
        return max_width
```

#863 MEDIUM

All Nodes Distance K in Binary Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · Tree · DFS · BFS

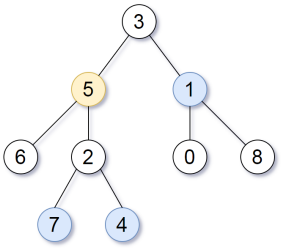
Lists: Grind 169

Problem:

Given the root of a binary tree, the value of a target node target, and an integer k, return an array of the values of all nodes that have a distance k from the target node.

You can return the answer in any order.

Example 1:




```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], target = 5, k = 2
Output: [7,4,1]
Explanation: The nodes that are a distance 2 from the target node (with value 5) have values
```



```
7, 4, and 1.
```

Example 2:

```
Input: root = [1], target = 1, k = 3
Output: []
```

Constraints:

- The number of nodes in the tree is in the range [1, 500].
- 0 <= Node.val <= 500
- All the values Node.val are unique.
- target is the value of one of the nodes in the tree.
- 0 <= k <= 1000

>> Brute Force [Trees & BST] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def distanceK(self, root: TreeNode, target: int, k: int):
        # Brute Force - collect inorder, then process sorted list
        vals = []
        def collect(node):
            if not node: return
            collect(node.left)
            vals.append(node.val)
            collect(node.right)
        collect(root)
        return vals[0] if vals else 0
```

Python

Python

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

def distanceK(root: TreeNode, target: int, k: int):
    from collections import defaultdict, deque

    # Build an adjacency list to represent the graph
    graph = defaultdict(list)

    # Helper function to build the graph using DFS
    def buildGraph(node, parent):
        if node is None:
            return
        if parent is not None:
            # Add bidirectional connection between node and parent
            graph[node.val].append(parent.val)
            graph[parent.val].append(node.val)
        buildGraph(node.left, node)
        buildGraph(node.right, node)

    buildGraph(root, None)


    # BFS from target node
    visited = set([target])
    queue = deque([target])
    current_distance = 0

    while queue:
        if current_distance == k:
            # Return all nodes at distance k
            return list(queue)
        size = len(queue)
        for _ in range(size):
            current = queue.popleft()
            for neighbor in graph[current]:
                if neighbor not in visited:
                    visited.add(neighbor)
                    queue.append(neighbor)
        current_distance += 1

    return [] # If no nodes at distance k

# Example usage:
# Construct tree or use a tree builder before calling distanceK function.
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None


class Solution:
    def distanceK(self, root: TreeNode, target: TreeNode, k: int) -> List[int]:
        def dfs(root, fa):
            if root is None:
                return
            g[root] = fa
            dfs(root.left, root)
            dfs(root.right, root)

        def dfs2(root, fa, k):
            if root is None:
                return
            if k == 0:
                ans.append(root.val)
                return
            for nxt in (root.left, root.right, g[root]):
                if nxt != fa:

                    dfs2(nxt, root, k - 1)

        g = {}
        dfs(root, None)
        ans = []
        dfs2(target, None, k)
        return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None


class Solution:
    def distanceK(self, root: TreeNode, target: TreeNode, k: int) -> List[int]:
        def parents(root, prev):
            nonlocal p
            if root is None:
                return
            p[root] = prev
            parents(root.left, root)
            parents(root.right, root)

        def dfs(root, k):
            nonlocal ans, vis
            if root is None or root.val in vis:
                return
            vis.add(root.val)
            if k == 0:
                ans.append(root.val)

                return
            dfs(root.left, k - 1)
            dfs(root.right, k - 1)
            dfs(p[root], k - 1)

        p = {}
        parents(root, None)
        ans = []
        vis = set()
        dfs(target, k)
        return ans
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None


class Solution:
    def distanceK(self, root: TreeNode, target: TreeNode, k: int) -> List[int]:
        def dfs(root, fa):
            if root is None:
                return
            g[root] = fa
            dfs(root.left, root)
            dfs(root.right, root)

        def dfs2(root, fa, k):
            if root is None:
                return
            if k == 0:
                ans.append(root.val)
                return
            for nxt in (root.left, root.right, g[root]):
                if nxt != fa:

                    dfs2(nxt, root, k - 1)

        g = {}
        dfs(root, None)
        ans = []
        dfs2(target, None, k)
        return ans
```

#1120 MEDIUM

Maximum Average Subtree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Tree · DFS

Lists: Premium 98

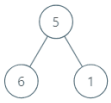
Problem:

Given the root of a binary tree, return the maximum average value of a subtree of that tree. Answers within 10-5 of the actual answer will be accepted.

A subtree of a tree is any node of that tree plus all its descendants.

The average value of a tree is the sum of its values, divided by the number of nodes.

Example 1:




```
Input: root = [5,6,1]
Output: 6.00000
Explanation:
For the node with value = 5 we have an average of (5 + 6 + 1) / 3 = 4.
For the node with value = 6 we have an average of 6 / 1 = 6.
For the node with value = 1 we have an average of 1 / 1 = 1.
So the answer is 6 which is the maximum.
```

Example 2:

```
Input: root = [0,null,1]
Output: 1.00000
```

Constraints:

- The number of nodes in the tree is in the range [1, 104].
- 0 <= Node.val <= 105

>> Brute Force [Trees & BST] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def maximumAverageSubtree(self, root):
        # Brute Force - collect inorder, then process sorted list
        vals = []
        def collect(node):
            if not node: return
            collect(node.left)
            vals.append(node.val)
            collect(node.right)
        collect(root)
        return vals[0] if vals else 0
```

Python

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val # value of the node
        self.left = left # left child
        self.right = right # right child

class Solution:
    def maximumAverageSubtree(self, root):
        # Initialize the maximum average to negative infinity
        self.maxAvg = float('-inf')

        # Recursive DFS function that returns a tuple (subtree sum, node count)
        def dfs(node):
            if not node:
                return (0, 0) # return sum 0 and count 0 if node is null
            leftSum, leftCount = dfs(node.left) # traverse left subtree
            rightSum, rightCount = dfs(node.right) # traverse right subtree
            currentSum = node.val + leftSum + rightSum # total sum at current node
            currentCount = 1 + leftCount + rightCount # total node count
            currentAvg = currentSum / currentCount # average value for current subtree
            self.maxAvg = max(self.maxAvg, currentAvg) # update maximum average if necessary
            return (currentSum, currentCount)

        dfs(root)

        return self.maxAvg
```

Python

```python
from dataclasses import dataclass


@dataclass(frozen=True)
class T:
  summ: int
  count: int
  maxAverage: int


class Solution:
  def maximumAverageSubtree(self, root: TreeNode | None) -> float:
    def maximumAverage(root: TreeNode | None) -> T:
      if not root:
        return T(0, 0, 0)

      left = maximumAverage(root.left)
      right = maximumAverage(root.right)

      summ = root.val + left.summ + right.summ
      count = 1 + left.count + right.count
      maxAverage = max(summ / count, left.maxAverage, right.maxAverage)
      return T(summ, count, maxAverage)

    return maximumAverage(root).maxAverage
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def maximumAverageSubtree(self, root: Optional[TreeNode]) -> float:
        def dfs(root):
            if root is None:
                return 0, 0
            ls, ln = dfs(root.left)
            rs, rn = dfs(root.right)
            s = root.val + ls + rs
            n = 1 + ln + rn
            nonlocal ans
            ans = max(ans, s / n)
            return s, n

        ans = 0
        dfs(root)
        return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def maximumAverageSubtree(self, root: Optional[TreeNode]) -> float:
        def dfs(root):
            if root is None:
                return 0, 0
            ls, ln = dfs(root.left)
            rs, rn = dfs(root.right)
            s = root.val + ls + rs
            n = 1 + ln + rn
            nonlocal ans
            ans = max(ans, s / n)
            return s, n

        ans = 0
        dfs(root)
        return ans
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val # value of the node
        self.left = left # left child
        self.right = right # right child

class Solution:
    def maximumAverageSubtree(self, root):
        # Initialize the maximum average to negative infinity
        self.maxAvg = float('-inf')

        # Recursive DFS function that returns a tuple (subtree sum, node count)
        def dfs(node):
            if not node:
                return (0, 0) # return sum 0 and count 0 if node is null
            leftSum, leftCount = dfs(node.left) # traverse left subtree
            rightSum, rightCount = dfs(node.right) # traverse right subtree
            currentSum = node.val + leftSum + rightSum # total sum at current node
            currentCount = 1 + leftCount + rightCount # total node count
            currentAvg = currentSum / currentCount # average value for current subtree
            self.maxAvg = max(self.maxAvg, currentAvg) # update maximum average if necessary
            return (currentSum, currentCount)

        dfs(root)

        return self.maxAvg
```

#1490 MEDIUM

Clone N-ary Tree

LeetDocs · SimplyLatet · WalkCC · LeetCode · Editorial

Hash Table · Tree · DFS · BFS

Lists: Premium 98

Problem:

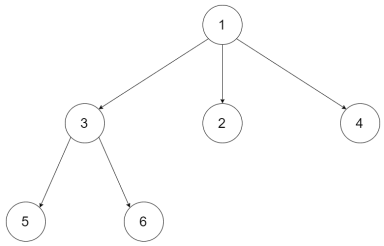
Given a root of an N-ary tree, return a deep copy (clone) of the tree.

Each node in the n-ary tree contains a val (int) and a list (List[Node]) of its children.

```
class Node {
public int val;
public List<Node> children;
}
```

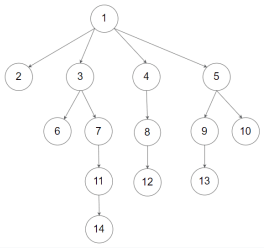
Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).

Example 1:




```
Input: root = [1,null,3,2,4,null,5,6]
Output: [1,null,3,2,4,null,5,6]
```

Example 2:




```
Input: root =
[1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]
```



```
Output:
[1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]
```

Constraints:

- The depth of the n-ary tree is less than or equal to 1000.
- The total number of nodes is between [0, 104].

Follow up: Can your solution work for the graph problem?

>> Brute Force [Trees & BST] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def cloneTree(self, root):
        # Brute force - collect inorder, then process sorted list
        vals = []
        def collect(node):
            if not node: return
            collect(node.left)
            vals.append(node.val)
            collect(node.right)
        collect(root)
        return vals[0] if vals else 0
```

Python

Python

```python
# Definition for a Node.
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []

def cloneTree(root):
    # Base case: if root is None, return None.
    if root is None:
        return None

    # Create a clone for the current node.
    cloned_node = Node(root.val)

    # Recursively clone each child and add it to the cloned_node's children list.
    for child in root.children:
        cloned_child = cloneTree(child)
        cloned_node.children.append(cloned_child)

    # Return the cloned node.
    return cloned_node

# Example usage:
# original_tree = Node(1, [Node(3, [Node(5), Node(6)]), Node(2), Node(4)])
# cloned_tree = cloneTree(original_tree)
```

Python


```
"""
# Definition for a Node.
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []
"""


class Solution:
    def cloneTree(self, root: 'Node') -> 'Node':
        if root is None:
            return None
        children = [self.cloneTree(child) for child in root.children]
        return Node(root.val, children)
```

Python


```
"""
# Definition for a Node.
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []
"""


class Solution:
    def cloneTree(self, root: 'Node') -> 'Node':
        if root is None:
            return None
        children = [self.cloneTree(child) for child in root.children]
        return Node(root.val, children)
```


[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a Node.
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []

def cloneTree(root):
    # Base case: if root is None, return None.
    if root is None:
        return None

    # Create a clone for the current node.
    cloned_node = Node(root.val)

    # Recursively clone each child and add it to the cloned_node's children list.
    for child in root.children:
        cloned_child = cloneTree(child)
        cloned_node.children.append(cloned_child)

    # Return the cloned node.
    return cloned_node

# Example usage:
# original_tree = Node(1, [Node(3, [Node(5), Node(6)]), Node(2), Node(4)])
# cloned_tree = cloneTree(original_tree)
```


#1522 MEDIUM

Diameter of N-Ary Tree

LeetDocs · SimplyLatet · WalkCC · LeetCode · Editorial

Tree · DFS

Lists: Premium 98

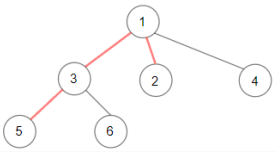
Problem:

Given a root of an N-ary tree, you need to compute the length of the diameter of the tree.

The diameter of an N-ary tree is the length of the longest path between any two nodes in the tree. This path may or may not pass through the root.

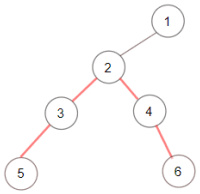
(Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value.)

Example 1:



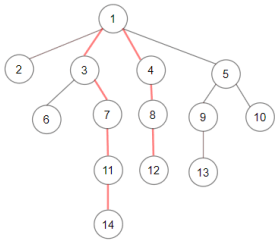

```
Input: root = [1,null,3,2,4,null,5,6]
Output: 3
Explanation: Diameter is shown in red color.
```

Example 2:




```
Input: root = [1,null,2,null,3,4,null,5,null,6]
Output: 4
```

Example 3:




```
Input: root =
[1,null,2,3,4,5,null,null,6,7,null,8,null,9,10,null,null,11,null,12,null,13,null,null,14]
Output: 7
```

Constraints:

- The depth of the n-ary tree is less than or equal to 1000.
- The total number of nodes is between [1, 104].

Python

Python


```python
# Definition for a Node.
# class Node:
# def __init__(self, val=None, children=None):
# self.val = val
# self.children = children if children is not None else []

class Solution:
    def diameter(self, root: 'Node') -> int:
        # Global variable to keep track of the maximum diameter found so far.
        self.max_diameter = 0

        def dfs(node):
            if not node:
                return 0 # Base case: if the node is None, return depth 0

            # Variables to hold the top two maximum depths from the children.
            max_depth1, max_depth2 = 0, 0

            # Iterate over each child to compute their maximum depths.
            for child in node.children:
                child_depth = dfs(child) # Recurse on the child
                # Update the two largest depths
                if child_depth > max_depth1:
                    max_depth2 = max_depth1
                    max_depth1 = child_depth

                elif child_depth > max_depth2:
                    max_depth2 = child_depth

            # Update the maximum diameter. The candidate is the sum of the two deepest paths.
            self.max_diameter = max(self.max_diameter, max_depth1 + max_depth2)

            # Return the maximum depth from this node.
            return max_depth1 + 1

        dfs(root)
        return self.max_diameter
```

Python


```python
class Solution:
    def diameter(self, root: 'Node') -> int:
        ans = 0

        def maxDepth(root: 'Node') -> int:
            """Returns the maximum depth of the subtree rooted at `root`."""
            nonlocal ans
            maxSubDepth1 = 0
            maxSubDepth2 = 0
            for child in root.children:
                maxSubDepth = maxDepth(child)
                if maxSubDepth > maxSubDepth1:
                    maxSubDepth2 = maxSubDepth1
                    maxSubDepth1 = maxSubDepth
                elif maxSubDepth > maxSubDepth2:
                    maxSubDepth2 = maxSubDepth
            ans = max(ans, maxSubDepth1 + maxSubDepth2)
            return 1 + maxSubDepth1

        maxDepth(root)
        return ans
```

Python

```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []
"""


class Solution:
    def diameter(self, root: 'Node') -> int:
        """
        :type root: 'Node'
        :rtype: int
        """

        def dfs(root):
            if root is None:
                return 0
            nonlocal ans
            m1 = m2 = 0
            for child in root.children:
                t = dfs(child)
                if t > m1:
                    m2, m1 = m1, t

                elif t > m2:
                    m2 = t
            ans = max(ans, m1 + m2)
            return 1 + m1

        ans = 0
        dfs(root)
        return ans
```

Python


```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val=None, children=None):
        self.val = val
        self.children = children if children is not None else []
"""


class Solution:
    def diameter(self, root: 'Node') -> int:
        """
        :type root: 'Node'
        :rtype: int
        """

        def dfs(root):
            if root is None:
                return 0
            nonlocal ans
            m1 = m2 = 0
            for child in root.children:
                t = dfs(child)
                if t > m1:
                    m2, m1 = m1, t

                elif t > m2:
                    m2 = t
            ans = max(ans, m1 + m2)
            return 1 + m1

        ans = 0
        dfs(root)
        return ans
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a Node.
# class Node:
# def __init__(self, val=None, children=None):
# self.val = val
# self.children = children if children is not None else []

class Solution:
    def diameter(self, root: 'Node') -> int:
        # Global variable to keep track of the maximum diameter found so far.
        self.max_diameter = 0

        def dfs(node):
            if not node:
                return 0 # Base case: if the node is None, return depth 0

            # Variables to hold the top two maximum depths from the children.
            max_depth1, max_depth2 = 0, 0

            # Iterate over each child to compute their maximum depths.
            for child in node.children:
                child_depth = dfs(child) # Recurse on the child
                # Update the two largest depths
                if child_depth > max_depth1:
                    max_depth2 = max_depth1
                    max_depth1 = child_depth

                elif child_depth > max_depth2:
                    max_depth2 = child_depth

            # Update the maximum diameter. The candidate is the sum of the two deepest paths.
            self.max_diameter = max(self.max_diameter, max_depth1 + max_depth2)

            # Return the maximum depth from this node.
            return max_depth1 + 1

        dfs(root)
        return self.max_diameter
```

#1650 MEDIUM

Lowest Common Ancestor of a Binary Tree III

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

Hash Table · Tree · Two Pointers

Lists: Premium 98

Problem:

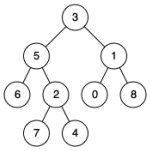
Given two nodes of a binary tree p and q, return their lowest common ancestor (LCA).

Each node will have a reference to its parent node. The definition for Node is below:

```
class Node {
public int val;
public Node left;
public Node right;
public Node parent;
}
```

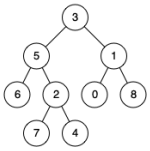
According to the definition of LCA on Wikipedia: "The lowest common ancestor of two nodes p and q in a tree T is the lowest node that has both p and q as descendants (where we allow a node to be a descendant of itself)."

Example 1:




```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.
```

Example 2:




```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5 since a node can be a descendant of itself
according to the LCA definition.
```

Example 3:

```
Input: root = [1,2], p = 1, q = 2
Output: 1
```

Constraints:

- The number of nodes in the tree is in the range [2, 105].

- -109 <= Node.val <= 109
- All Node.val are unique.
- p != q
- p and q exist in the tree.

>> Brute Force [Trees & BST] Interview Prep

Python

```python
# Brute Force Approach
class Solution:
    def lowestCommonAncestor(self, p, q):
        # Brute Force - collect inorder, then process sorted list
        vals = []
        def collect(node):
            if not node: return
            collect(node.left)
            vals.append(node.val)
            collect(node.right)
        collect(p)
        return vals[0] if vals else 0
```

Python

Python

```python
# Definition for a Node.
# class Node:
# def __init__(self, val=0, left=None, right=None, parent=None):
# self.val = val
# self.left = left
# self.right = right
# self.parent = parent

def lowestCommonAncestor(p, q):
    # Compute the depth for node p
    def getDepth(node):
        depth = 0
        while node:
            depth += 1
            node = node.parent
        return depth

    depthP = getDepth(p)
    depthQ = getDepth(q)

    # Bring both nodes to the same depth level
    while depthP > depthQ:
        p = p.parent
        depthP -= 1
    while depthQ > depthP:
        q = q.parent
        depthQ -= 1

    # Traverse up until we find the common ancestor
    while p != q:
        p = p.parent
        q = q.parent

    return p
```


Python

```python
class Solution:
  # Same as 160. Intersection of Two Linked Lists
  def lowestCommonAncestor(self, p: 'Node', q: 'Node') -> 'Node':
    a = p
    b = q

    while a != b:
      a = a.parent if a else q
      b = b.parent if b else p

    return a
```

Python

```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.parent = None
"""


class Solution:
    def lowestCommonAncestor(self, p: "Node", q: "Node") -> "Node":
        vis = set()
        node = p
        while node:
            vis.add(node)
            node = node.parent
        node = q
        while node not in vis:
            node = node.parent
        return node
```

Python


```python
"""
# Definition for a Node.
class Node:
    def __init__(self, val):
        self.val = val
        self.left = None
        self.right = None
        self.parent = None
"""


class Solution:
    def lowestCommonAncestor(self, p: 'Node', q: 'Node') -> 'Node':
        a, b = p, q
        while a != b:
            a = a.parent if a.parent else q
            b = b.parent if b.parent else p
        return a

############

# ref: https://iamagesk.medium.com/leetcode-1650-lowest-common-ancestor-of-a-binary-tree-iii-6d888b93376c

def lowestCommonAncestor(self, a: 'Node', b: 'Node') -> 'Node':
    ancestors = set()


    while a is not None:
        ancestors.add(a)
        a = a.parent

    while b is not None:
        if b in ancestors:
            return b
        b = b.parent

# or another one
def lowestCommonAncestor(self, a: 'Node', b: 'Node') -> 'Node':
    pointerA, pointerB = a, b

    while pointerA != pointerB:
        pointerA = pointerA.parent if pointerA else b
        pointerB = pointerB.parent if pointerA else a

    return pointerA
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a Node.
# class Node:
# def __init__(self, val=0, left=None, right=None, parent=None):
# self.val = val
# self.left = left
# self.right = right
# self.parent = parent

def lowestCommonAncestor(p, q):
    # Compute the depth for node p
    def getDepth(node):
        depth = 0
        while node:
            depth += 1
            node = node.parent
        return depth

    depthP = getDepth(p)
    depthQ = getDepth(q)

    # Bring both nodes to the same depth level
    while depthP > depthQ:
        p = p.parent
        depthP -= 1
    while depthQ > depthP:
        q = q.parent
        depthQ -= 1

    # Traverse up until we find the common ancestor
    while p != q:
        p = p.parent
        q = q.parent

    return p
```

#124 HARD

Binary Tree Maximum Path Sum

LeetDocs · Simplified · WalkCC · LeetCode · Editorial

Dynamic Programming · Tree · DFS

Lists: Grind 169

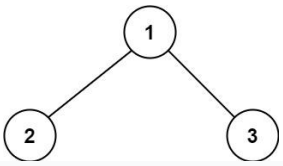
Problem:

A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence at most once. Note that the path does not need to pass through the root.

The path sum of a path is the sum of the node's values in the path.

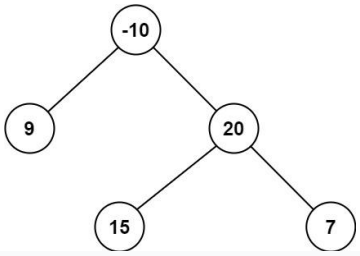
Given the root of a binary tree, return the maximum path sum of any non-empty path.

Example 1:




```
Input: root = [1,2,3]
Output: 6
Explanation: The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.
```

Example 2:




```
Input: root = [-10,9,20,null,null,15,7]
Output: 42
Explanation: The optimal path is 15 -> 20 -> 7 with a path sum of 15 + 20 + 7 = 42.
```

Constraints:

- The number of nodes in the tree is in the range [1, 3 * 104].
- -1000 <= Node.val <= 1000

Python

Python

```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def maxPathSum(self, root: TreeNode) -> int:
        # Initialize global maximum path sum with a very small number
        self.global_max = float('-inf')

        def dfs(node: TreeNode) -> int:
            if not node:
                return 0

            # Recursively calculate max gain from left and right children, discard negative gains
            left_gain = max(dfs(node.left), 0)
            right_gain = max(dfs(node.right), 0)

            # Current maximum including both left and right gains
            current_max = node.val + left_gain + right_gain

            # Update global maximum path sum if current_sum is higher
            self.global_max = max(self.global_max, current_max)

            # Return the node's value plus the max of one child gain (only one side can be chosen for parent path)
            return node.val + max(left_gain, right_gain)

        dfs(root)
        return self.global_max
```

Python

```python
class Solution:
  def maxPathSum(self, root: TreeNode | None) -> int:
    ans = -math.inf

    def maxPathSumDownFrom(root: TreeNode | None) -> int:
      """
      Returns the maximum path sum starting from the current root, where
      root.val is always included.
      """
      nonlocal ans
      if not root:
        return 0

      l = max(0, maxPathSumDownFrom(root.left))
      r = max(0, maxPathSumDownFrom(root.right))
      ans = max(ans, root.val + l + r)
      return root.val + max(l, r)

    maxPathSumDownFrom(root)
    return ans
```

Python


```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        def dfs(root: Optional[TreeNode]) -> int:
            if root is None:
                return 0
            left = max(0, dfs(root.left))
            right = max(0, dfs(root.right))
            nonlocal ans
            ans = max(ans, root.val + left + right)
            return root.val + max(left, right)

        ans = -inf
        dfs(root)
        return ans
```

Python

```python
# Definition for a binary tree node.
# class TreeNode:
# def __init__(self, val=0, left=None, right=None):
# self.val = val
# self.left = left
# self.right = right
class Solution:
    def maxPathSum(self, root: Optional[TreeNode]) -> int:
        def dfs(root: Optional[TreeNode]) -> int:
            if root is None:
                return 0
            left = max(0, dfs(root.left))
            right = max(0, dfs(root.right))
            nonlocal ans
            ans = max(ans, root.val + left + right)
            return root.val + max(left, right)

        ans = -inf
        dfs(root)
        return ans
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

class Solution:
    def maxPathSum(self, root: TreeNode) -> int:
        # Initialize global maximum path sum with a very small number
        self.global_max = float('-inf')

        def dfs(node: TreeNode) -> int:
            if not node:
                return 0

            # Recursively calculate max gain from left and right children, discard negative gains
            left_gain = max(dfs(node.left), 0)
            right_gain = max(dfs(node.right), 0)

            # Current maximum including both left and right gains
            current_max = node.val + left_gain + right_gain

            # Update global maximum path sum if current_sum is higher
            self.global_max = max(self.global_max, current_max)

            # Return the node's value plus the max of one child gain (only one side can be chosen for parent path)
            return node.val + max(left_gain, right_gain)

        dfs(root)
        return self.global_max
```


#297 HARD

Serialize and Deserialize Binary Tree

LeetDoocs · SimplyLeet · WalkCC · LeetCode · Editorial

String · Tree · DFS · BFS · Design

Lists: Grind 169

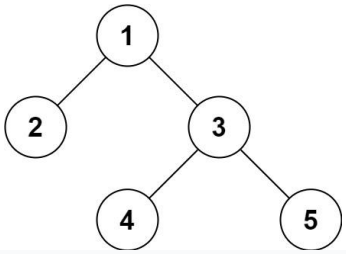
Problem:

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

Clarification: The input/output format is the same as how LeetCode serializes a binary tree. You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

Example 1:




```
Input: root = [1,2,3,null,null,4,5]
Output: [1,2,3,null,null,4,5]
```

Example 2:

```
Input: root = []
Output: []
```

Constraints:

- The number of nodes in the tree is in the range [0, 104].
- -1000 <= Node.val <= 1000

Python

Python


```python
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val # value of the node
        self.left = left # left child
        self.right = right # right child

class Codec:
    # Encodes a tree to a single string using BFS.
    def serialize(self, root):
        if not root:
            return "" # Return empty string for an empty tree
        result = []
        queue = [root] # Initialize the queue with the root node
        while queue:
            node = queue.pop(0)
            if node:
                result.append(str(node.val)) # Append the value of the node
                queue.append(node.left) # Enqueue left child
                queue.append(node.right) # Enqueue right child
            else:
                result.append("null") # Placeholder for null node
        return ",".join(result) # Join list with commas to form the string

    # Decodes the encoded data to reconstruct the tree using BFS.

    def deserialize(self, data):
        if not data:
            return None # Return None for empty string
        nodes = data.split(",")
        root = TreeNode(int(nodes[0])) # Create root node from first element
        queue = [root]
        i = 1 # Index for traversing nodes list
        while queue:
            current = queue.pop(0)
            if i < len(nodes) and nodes[i] != "null":
                current.left = TreeNode(int(nodes[i]))
                queue.append(current.left)
            i += 1
            if i < len(nodes) and nodes[i] != "null":
                current.right = TreeNode(int(nodes[i]))
                queue.append(current.right)
            i += 1
        return root
```

Python


```python
class Codec:
  def serialize(self, root: 'TreeNode') -> str:
    """Encodes a tree to a single string."""
    if not root:
      return ''

    s = ''
    q = collections.deque([root])

    while q:
      node = q.popleft()
      if node:
        s += str(node.val) + ' '
        q.append(node.left)
        q.append(node.right)
      else:
        s += 'n '

    return s

  def deserialize(self, data: str) -> 'TreeNode':
    """Decodes your encoded data to tree."""
    if not data:
      return None

    vals = data.split()
    root = TreeNode(vals[0])
    q = collections.deque([root])

    for i in range(1, len(vals), 2):
      node = q.popleft()
      if vals[i] != 'n':
        node.left = TreeNode(vals[i])
        q.append(node.left)
      if vals[i + 1] != 'n':
        node.right = TreeNode(vals[i + 1])
        q.append(node.right)

    return root
```

Python

```python
# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None


class Codec:
    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        if root is None:
            return ""
        q = deque([root])
        ans = []
        while q:
            node = q.popleft()
            if node:
                ans.append(str(node.val))
                q.append(node.left)
                q.append(node.right)

            else:
                ans.append("#")
        return ",".join(ans)

    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """
        if not data:
            return None
        vals = data.split(",")
        root = TreeNode(int(vals[0]))
        q = deque([root])
        i = 1
        while q:
            node = q.popleft()
            if vals[i] != "#":
                node.left = TreeNode(int(vals[i]))
                q.append(node.left)
            i += 1
            if vals[i] != "#":
                node.right = TreeNode(int(vals[i]))
                q.append(node.right)

            i += 1
        return root


# Your Codec object will be instantiated and called as such:
# codec = Codec()
# codec.deserialize(codec.serialize(root))
```


Python

```python
# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None


class Codec:
    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """

        if root is None:
            return ""
        res = []

        def preorder(root):
            if root is None:
                res.append("#,")
                return
            res.append(str(root.val) + ",")

            preorder(root.left)
            preorder(root.right)

        preorder(root)
        return ''.join(res)

    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """
        if not data:
            return None
        vals = data.split(",")

        def inner():
            first = vals.pop(0)
            if first == "#":
                return None
            return TreeNode(int(first), inner(), inner())
            # seems using a constructor __init__(val, left, right)

        return inner()
```



```python
# Your Codec object will be instantiated and called as such:
# ser = Codec()
# deser = Codec()
# ans = deser.deserialize(ser.serialize(root))

###########

from collections import deque

# bfs, each level based
class Codec:
    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        ret = []
        queue = deque([root])
        while queue:
            top = queue.popleft()
            if not top:
                ret.append("None")
                continue

            else:
                ret.append(str(top.val))
            queue.append(top.left)
            queue.append(top.right)
        return ",".join(ret)
```

[*] Trees & BST — Pattern Solution (matched from community answers)

Python


```python
# Definition for a binary tree node.
# class TreeNode(object):
# def __init__(self, x):
# self.val = x
# self.left = None
# self.right = None


class Codec:
    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """

        if root is None:
            return ""
        res = []

        def preorder(root):
            if root is None:
                res.append("#,")
                return
            res.append(str(root.val) + ",")

            preorder(root.left)
            preorder(root.right)

        preorder(root)
        return ''.join(res)

    def deserialize(self, data):
        """Decodes your encoded data to tree.

        :type data: str
        :rtype: TreeNode
        """
        if not data:
            return None
        vals = data.split(",")

        def inner():
            first = vals.pop(0)
            if first == "#":
                return None
            return TreeNode(int(first), inner(), inner())
            # seems using a constructor __init__(val, left, right)

        return inner()
```



```python
# Your Codec object will be instantiated and called as such:
# ser = Codec()
# deser = Codec()
# ans = deser.deserialize(ser.serialize(root))

###########

from collections import deque

# bfs, each level based
class Codec:
    def serialize(self, root):
        """Encodes a tree to a single string.

        :type root: TreeNode
        :rtype: str
        """
        ret = []
        queue = deque([root])
        while queue:
            top = queue.popleft()
            if not top:
                ret.append("None")
                continue

            else:
                ret.append(str(top.val))
            queue.append(top.left)
            queue.append(top.right)
        return ",".join(ret)
```


Quick Review — Trees & BST 37 questions · insights · complexity · solution

#100 Same Tree [Easy]

Key Insights

- Use recursion to compare nodes in both trees.
- If both nodes being compared are null, they are the same.
- If one node is null while the other is not, the trees differ.
- Check if current nodes' values match, then recursively verify the left and right subtrees.

Space & Time

Time Complexity: O(n), where n is the number of nodes in the trees, since every node is visited once.

Space Complexity: O(n) in the worst-case (unbalanced tree), due to recursion call stack usage. In a balanced tree, the space would be O(log n).

Solution

The problem can be solved using a recursive approach (Depth-First Search) where we compare the current nodes from both trees. If the current nodes are both null, they match; if one is null, then the trees are not identical. For non-null nodes, check if the values are equal. Then, recursively apply the same logic on the left and right children of the nodes. This approach ensures that both the structure and the node values are identical across the two trees.

#101 Symmetric Tree [Easy]

Key Insights

- Use recursion (or iteration) to compare pairs of nodes.
- For two trees (or two nodes) to be mirror images:
- Their root values must be the same.
- The left subtree of one must match the right subtree of the other.
- The right subtree of one must match the left subtree of the other.
- Alternatively, iterative approaches using a queue can be applied to compare the nodes in pairs.

Space & Time

Time Complexity: O(n), where n is the number of nodes in the tree, since we traverse each node once.

Space Complexity: O(h) for recursive call stack (worst-case O(n) for a skewed tree) or O(n) for the iterative approach using a queue.

Solution

We solve the problem using recursion by defining a helper function that takes two nodes and checks if they are mirrors of each other. The helper function:

- Returns true if both nodes are null. - Returns false if one node is null or if their values differ. - Recursively compares the left child of the first node with the right child of the second node, and vice-versa. This method ensures each comparison verifies symmetry at every level. A similar logic can be applied iteratively by using a queue to compare pairs of nodes.

#104 Maximum Depth of Binary Tree [Easy]

Key Insights

- Use a recursive strategy (DFS) to traverse the binary tree.
- At each node, compute the maximum depth of its left and right subtree.
- The maximum depth at the current node is 1 (current node) plus the maximum of the left and right subtree depths.
- Alternatively, an iterative approach using BFS (level-order traversal) can be used.

Space & Time

Time Complexity: O(n), where n is the number of nodes, since every node is visited once.

Space Complexity: O(h) for recursion stack in DFS, where h is the height of the tree (worst-case O(n) for a skewed tree); O(n) for BFS in the worst-case scenario if the tree is balanced.

Solution

The problem is approached using Depth-First Search (DFS) with recursion. The idea is to visit every node and calculate the maximum depth of the left and right subtrees, then add 1 for the current node. This method naturally handles the base case when a null node (empty tree) is reached by returning 0. Alternatively, a Breadth-First Search (BFS) approach can determine the tree level by level, with the total number of levels equaling the maximum depth. Key data structures include recursion call stack for DFS or a queue for BFS.

#108 Convert Sorted Array to Binary Search Tree [Easy]

Key Insights

- The array is sorted in ascending order, which allows the use of a divide and conquer strategy.
- Choosing the middle element of the array as the root ensures that the left and right subtrees are roughly equal in size, yielding a balanced BST.
- A recursive approach is natural for building the tree: the base case is when the subarray is empty.
- The algorithm splits the array into two halves to recursively build the left and right subtrees.

Space & Time

Time Complexity: O(n) - Every element is processed exactly once.

Space Complexity: O(n) - O(n) space is used to store the BST, and additional O(log n) space may be used for the recursion stack.

Solution

We use a divide and conquer approach with recursion. The main idea is to select the middle element of the current subarray as the root node so that the left subarray elements form the left subtree and the right subarray elements form the right subtree, ensuring the tree is balanced. The recursion continues until the subarray is empty, which serves as the base case. This strategy naturally maintains the BST property since all elements in the left subarray are smaller than the root, and all in the right subarray are larger.

#110 Balanced Binary Tree [Easy]

Key Insights

- Use a bottom-up DFS (Depth-First Search) traversal to compute the height of each subtree.
- If any subtree is found unbalanced, propagate an error indicator (e.g., -1) upward, allowing early termination.
- An empty tree is considered balanced.
- The key check is to ensure that for each node, the absolute difference between the heights of its left and right subtrees is no more than 1.

Space & Time

Time Complexity: O(n) where n is the number of nodes in the tree, since we visit each node once.

Space Complexity: O(h) where h is the height of the tree, which corresponds to the recursion stack (O(n) in the worst-case of a skewed tree).

Solution

The solution uses a recursive DFS approach. A helper function recursively computes the height of a subtree. If a subtree is found to be unbalanced (the height difference between its left and right subtrees exceeds 1), the helper returns a special value (e.g., -1) to indicate that the subtree is unbalanced. This value is then propagated up the recursion to terminate further unnecessary computations. The main function checks the result of the helper function and returns true if the tree is balanced (i.e., the helper did not return -1) and false otherwise.

#112 Path Sum [Easy]

Key Insights

- Use a depth-first search (DFS) strategy to traverse the tree from the root to the leaves.
- At each node, subtract the node's value from the targetSum.
- Check if the node is a leaf; if yes and the remaining targetSum equals the node's value then a valid path is found.
- If the tree is empty, return false as no path exists.

Space & Time

Time Complexity: O(N) where N is the number of nodes, since every node may need to be visited.

Space Complexity: O(H) where H is the height of the tree, accounting for the recursion stack (worst-case O(N) for a skewed tree).

Solution

The approach uses a recursive depth-first search. The algorithm subtracts the current node's value from targetSum as it moves down the tree. When it reaches a leaf, it checks if the modified targetSum is equal to the leaf's value. If yes, it returns true. The recursion continues until either a valid path is found or all paths are exhausted. The primary data structure used here is the call stack for recursion.

#226 Invert Binary Tree [Easy]

Key Insights

- The problem can be solved using recursion (depth-first search) by swapping the children of every node.
- Alternatively, an iterative approach (using a queue for breadth-first search) can be used.
- The recursion terminates when a null node is encountered.
- The operation is performed in-place, modifying the original tree structure.

Space & Time

Time Complexity: O(n) where n is the number of nodes, since each node is visited once.

Space Complexity: O(h) where h is the height of the tree, which is the recursion stack space in the recursive approach. In worst-case (skewed tree), this could be O(n).

Solution

We solve the problem using recursion (depth-first search). The approach is to:

- Check the base case: if the node is null, simply return null. - Recursively invert the left subtree and the right subtree. - Swap the left and right children of the current node. - Return the current node after the swap. The recursive approach elegantly handles all nodes and ensures the inversion process covers the complete tree.

#270 Closest Binary Search Tree Value [Easy]

Key Insights

- BST property allows us to traverse the tree efficiently by comparing the target with node values.
- We can iteratively traverse the tree, moving left if the target is smaller than the current node's value and moving right if greater.
- At each node, calculate the absolute difference between node's value and the target, and update the answer if the difference is smaller—or equal with a smaller candidate value.
- This approach minimizes the search area by leveraging the inherent ordering in BSTs.

Space & Time

Time Complexity: O(h) on average, where h is the height of the tree (O(log n) for a balanced tree, and O(n) in the worst-case scenario for a skewed tree).

Space Complexity: O(1) for an iterative solution (or O(h) if using recursion due to the call stack).

Solution

We solve the problem by performing an iterative traversal of the BST. Starting at the root, we maintain a variable to store the value closest to the target. For every node encountered, we:

- Compute the absolute difference between the node's value and the target. - Update the stored closest value if the computed difference is less than the current smallest difference; if the difference is the same, choose the smaller node value. - Depending on whether the target is less than or greater than the current node's value, move to the left or right child accordingly. This approach efficiently defines a search path that is consistent with the properties of a BST.

#501 Find Mode in Binary Search Tree [Easy]

Key Insights

- The BST in-order traversal yields a sorted order, which helps in counting consecutive duplicates.
- A two-pass in-order traversal can be used: one pass to determine the maximum frequency (maxCount) and a second pass to collect all values that match this frequency.
- The solution can be implemented without extra space (apart from recursion stack) by leveraging constant space counters and state variables.

Space & Time

Time Complexity: O(n), where n is the number of nodes, because each node is processed twice (once in each traversal).

Space Complexity: O(h) where h is the height of the tree due to the recursion stack. O(1) extra space is used aside from this.

Solution

We perform an in-order traversal of the BST twice. The first traversal determines the maximum frequency of any value in the BST by comparing the current node's value with the previously visited node. Since the in-order traversal processes nodes in sorted order, duplicate values are consecutive, making the counting simple. After calculating maxCount, we reset our state and perform a second in-order traversal to collect all node values with frequency equal to maxCount. The key trick is to leverage the BST property (sorted in-order traversal) to count frequencies in one pass without using additional data structures like hash maps.

#543 Diameter of Binary Tree [Easy]

Key Insights

- Use depth-first search (DFS) to explore the tree.
- The diameter at a node is the sum of the depths of its left and right subtrees.
- Track the maximum diameter found during the DFS traversal.
- Recursively calculating the depth of subtrees provides an efficient solution.

Space & Time

Time Complexity: O(n), where n is the number of nodes in the tree.

Space Complexity: O(n), in the worst-case scenario due to the recursion stack (skewed tree).

Solution

The solution employs a DFS traversal of the binary tree. At every node in the tree, compute the depth of the left and right subtrees. The sum of these depths gives the potential diameter (longest path through that node). Update a global maximum diameter variable if this sum exceeds the current maximum. Finally, return the maximum diameter after completing the DFS.

#572 Subtree of Another Tree [Easy]

Key Insights

- Use Depth-First Search (DFS) to visit each node in the main tree.
- At each node, check if the subtree starting from that node is identical to subRoot.
- A helper function is used to compare the structure and node values of two trees.
- Consider recursion as the primary approach to traverse and compare trees.
- Early exit when a mismatch is found can optimize average-case performance.

Space & Time

Time Complexity: O(m * n) in the worst case, where m is the number of nodes in the main tree and n is the number of nodes in the subRoot tree.

Space Complexity: O(max(depth of main tree, depth of subRoot)) due to recursive call stack.

Solution

We solve the problem by performing a DFS traversal on the main tree. For each visited node, a helper function checks whether the subtree starting at that node is identical to subRoot by comparing node values and recursively checking both left and right subtrees. This method leverages recursion effectively and provides early termination if mismatches occur, allowing for efficient traversal and comparison.

#98 Validate Binary Search Tree [Medium]

Key Insights

- Every node must adhere to a range: all nodes in the left subtree should be less than the current node, and all nodes in the right subtree should be greater.
- A recursive depth-first search approach works well by passing down the valid range (lower and upper bounds) for each node.
- Using the limits (negative and positive infinity) simplifies handling edge cases at the tree root.
- An in-order traversal would alternatively yield a sorted order for node values, but updating ranges directly is more intuitive for recursive implementation.

Space & Time

Time Complexity: O(N), where N is the number of nodes, as each node is visited once.

Space Complexity: O(H), where H is the height of the tree, due to the recursion stack.

Solution

We use a recursive approach to validate the BST property. We maintain two variables, low and high, representing the valid range for the current node's value. For the left child, the valid range becomes (low, node.val) and for the right child, it becomes (node.val, high). By checking if each node's value falls within its respective range, we can ensure that the tree satisfies the BST properties.

#102 Binary Tree Level Order Traversal [Medium]

Key Insights

- Use Breadth-First Search (BFS) to traverse the tree.
- A queue data structure is ideal for maintaining the order of nodes at each level.
- For every level, process all nodes currently in the queue and add their children for the next level.

Space & Time

Time Complexity: O(n), where n is the number of nodes, because each node is processed once.

Space Complexity: O(n), which accounts for the space used by the queue in the worst-case scenario.

Solution

This solution leverages a BFS approach using a queue. The algorithm starts by enqueuing the root node. Then, it enters a loop that continues until the queue is empty. For each iteration—representing one level of the tree—it computes the number of nodes at that level (levelSize), processes these nodes by dequeuing them, and collects their values. Their non-null children are then enqueued. The list of values for each level is added to the final result list. Key data structures include the queue (for managing nodes) and a list (to store level values).

#103 Binary Tree Zigzag Level Order Traversal [Medium]

Key Insights

- Use a breadth-first search (BFS) approach to traverse the tree level by level.
- Alternate the direction of traversal on each level to achieve the zigzag pattern.
- A flag or counter can be used to determine the order of nodes for the current level.

- Using a double-ended data structure or simply reversing the list at every alternate level is an effective approach.

Space & Time

Time Complexity: O(n), where n is the number of nodes, because each node is visited exactly once.

Space Complexity: O(n), as in the worst-case scenario (e.g., a complete binary tree) the queue used for BFS may hold O(n) nodes at once.

Solution

We can solve this problem using a breadth-first search (BFS). Start by initializing a queue to hold nodes for each level. For each level:

- Determine the number of nodes at the current level. - Iterate over these nodes, appending their children for the next level. - Depending on a flag (or level count), either append node values in the natural order (left to right) or reverse the order (right to left) before adding to the result. Key data structures:

- A queue for BFS traversal. - A list (or similar container) to store current level values. The main trick is toggling the order of insertion or reversing the list at alternate levels to achieve the "zigzag" pattern.

#105 Construct Binary Tree from Preorder and Inorder Traversal [Medium]

Key Insights

- Preorder traversal always starts with the root node.
- Inorder traversal splits the tree into left subtree and right subtree based on the root.
- A hash table (map/dictionary) can be used to quickly locate the index of the root in the inorder array.
- Use recursion to construct left and right subtrees.

Space & Time

Time Complexity: O(n) since we process every node exactly once.

Space Complexity: O(n) due to the recursion stack and the hash map used for inorder indices.

Solution

We solve the problem using a recursive divide-and-conquer approach. First, create a hash map to store each value's index in the inorder array for quick lookups. The preorder array always provides the current root. Use the inorder array to divide the tree into left and right subtrees. Recursively construct the subtrees by adjusting the preorder index and the boundaries of the inorder array. The algorithm leverages recursion and a hash map to achieve O(n) time complexity.

#199 Binary Tree Right Side View [Medium]

Key Insights

- The problem is essentially about capturing the last node at each level (or the rightmost node) in a binary tree.
- A Breadth-First Search (BFS) approach naturally works by processing nodes level by level.
- Alternatively, Depth-First Search (DFS) can be adapted by prioritizing the right subtree first, ensuring that the first node encountered at each depth is the visible one.
- The number of nodes is capped at 100, so both approaches are efficient.

Space & Time

Time Complexity: O(n) - we visit every node in the tree.

Space Complexity: O(n) - in the worst case, the queue (or call stack for DFS) may hold all nodes of the tree.

Solution

We can solve this problem using BFS (level-order traversal). The idea is to traverse the tree level by level using a queue. For each level, the last element added to the level represents the rightmost view from that level. We then append that element's value to our result list.

Alternatively, a DFS approach can be used where we visit the right subtree first. By keeping track of the current depth and checking if it's the first time we've encountered this depth, we can add the first node visited at that depth (which will be the rightmost one) to the result.

In both approaches, appropriate data structures (a queue for BFS or recursion with a list for DFS) are used to store nodes and track the traversal state.

#230 Kth Smallest Element in a BST [Medium]

Key Insights

- An in-order traversal of a BST visits nodes in increasing order.
- Use a counter to keep track of the number of nodes visited.
- As soon as the counter reaches k, the current node's value is the kth smallest.
- For frequent modifications (insertions/deletions) and queries, consider augmenting the BST with subtree counts.

Space & Time

Time Complexity: O(n) in the worst-case scenario (when k ≈ n), but on average the traversal may stop early.

Space Complexity: O(h) where h is the height of the tree (O(n) in the worst-case for a skewed tree).

Solution

We perform an in-order traversal of the BST which visits nodes in ascending order. During the traversal, we maintain a counter (or decrement k) to check when we have encountered the kth smallest element. When the counter reaches 0 (or k becomes 0), we capture the current node's value as our answer and return it. This approach is both simple and efficient for the given constraints.

#235 Lowest Common Ancestor of a Binary Search Tree [Medium]

Key Insights

- Utilize the BST property: for any node, values in the left subtree are less and values in the right subtree are greater.
- If both p and q have values less than the current node, the LCA must lie in the left subtree.
- If both p and q have values greater than the current node, the LCA must lie in the right subtree.
- When one node value is less than and the other is greater than (or equal to) the current node, the current node is the lowest common ancestor.
- An iterative approach is efficient and uses constant extra space.

Space & Time

Time Complexity: O(h), where h is the height of the tree (O(log n) for balanced trees and O(n) in the worst-case skewed tree).

Space Complexity: O(1) with an iterative approach (O(h) if using recursion due to call stack).

Solution

The solution leverages the BST properties. Algorithm:

- Start at the root node. - While the current node exists: If both p and q have values less than the current node's value, move to the left child. If both p and q have values greater than the current node's value, move to the right child. Otherwise, the current node is the split point where p and q diverge, so it is the lowest common ancestor. - Return the current node.

This approach efficiently finds the LCA using a single traversal from the root.

#236 Lowest Common Ancestor of a Binary Tree [Medium]

Key Insights

- Use a depth-first search (DFS) traversal to explore the tree.
- If the current node is either p or q, it could be the LCA.
- Recursively find p and q in the left and right subtrees.
- If both left and right recursive calls return a non-null value, then the current node is the LCA.
- If only one side returns a non-null value, propagate that value up the recursion.

Space & Time

Time Complexity: O(n), where n is the number of nodes in the tree since each node is visited once.

Space Complexity: O(h), where h is the height of the tree due to the recursion stack (O(n) in the worst case of a skewed tree).

Solution

We solve the problem using recursion with depth-first search. The algorithm checks if the current node matches p or q. It then recursively searches the left and right subtrees. If both subtrees contain one of p or q, the current node is the LCA. Otherwise, the algorithm forwards the non-null result up the recursion. This approach naturally handles cases where one node is an ancestor of the other.

#250 Count Univalue Subtrees [Medium]

Key Insights

- Perform a postorder traversal (left, right, root) to determine whether each subtree is univalue.
- Treat null nodes as univalue to ease the recursion.
- A node's subtree is univalue if both left and right subtrees are univalue and, if they exist, their values equal the current node's value.
- Use a global or external counter that increments every time a univalue subtree is confirmed.

Space & Time

Time Complexity: O(n), where n is the number of nodes, since each node is visited once.

Space Complexity: O(h), where h is the height of the tree, due to the recursion stack.

Solution

We use a recursive DFS approach (postorder traversal) to solve the problem. For each node, we recursively check its left and right subtrees to decide if they are univalue. By default, if a child is null, it is considered univalue. Then, we verify the current node's value against its non-null children. If both conditions hold (subtrees are univalue and matching values), we count the current node's subtree as univalue. This DFS checks in a bottom-up manner, ensuring that the properties required for a subtree to be univalue are maintained. The key trick is to return a boolean indicating if the subtree at each node is univalue and, while backtracking, increment the counter appropriately.

#285 Inorder Successor in BST [Medium]

Key Insights

- In a BST, an in-order traversal yields nodes in ascending order.
- If the node p has a right subtree, the in-order successor is the left-most node in that right subtree.
- If p has no right subtree, the successor is one of its ancestors; traverse from the root towards p while keeping track of the last node that is greater than p.
- An iterative approach is efficient and avoids recursion stack overhead.

Space & Time

Time Complexity: O(h), where h is the height of the tree. In the worst-case (skewed tree), it can be O(n).

Space Complexity: O(1) as only a constant amount of extra space is used.

Solution

We solve the problem using two cases:

- If node p has a right subtree, we simply find the left-most node in that right subtree. - If node p does not have a right subtree, initiate a search from the BST root. As we traverse: If the current node's value is greater than p.val, the current node is a candidate for the successor; move to the left subtree to look for a smaller candidate. Otherwise, move to the right subtree. By the end of the traversal, the candidate (if exists) is the in-order successor. This approach benefits from the BST property, limiting traversal to O(h) time.

#298 Binary Tree Longest Consecutive Sequence [Medium]

Key Insights

- Use a Depth-First Search (DFS) to traverse the binary tree.

- Pass the current node's value and the length of the current consecutive sequence in the recursive call.
- When a node's value is exactly one greater than its parent's value, increment the sequence length; otherwise, reset it to 1.
- Keep track of the maximum sequence length found during the traversal.

Space & Time

Time Complexity: O(N), where N is the number of nodes in the tree, since each node is visited once.

Space Complexity: O(H), where H is the height of the tree. In the worst-case of a skewed tree, the recursion stack can be O(N).

Solution

The solution involves using DFS to explore every node in the binary tree while keeping track of the length of the consecutive sequence. For each node, if the node's value is one greater than its parent's value, we increment the sequence length; otherwise, we reset the sequence length to 1. During the traversal, a global variable is updated if the current sequence length exceeds the previously recorded maximum. This approach ensures that every possible consecutive path is considered.

#314 Binary Tree Vertical Order Traversal [Medium]

Key Insights

- Use Breadth-First Search (BFS) to traverse the tree level by level.
- Track the column index for each node where root is column 0, left child is column-1, and right child is column+1.
- Group nodes by their column indices using a hash table (or dictionary).
- BFS traversal naturally handles left-to-right ordering within the same level.
- Finally, sort the keys (column indices) to generate results from leftmost column to rightmost column.

Space & Time

Time Complexity: O(n log n) in the worst-case due to sorting the column indices (n nodes in the worst-case if each node is on a unique column).

Space Complexity: O(n) for storing the node information and the resultant structure.

Solution

The solution uses BFS with a queue that stores nodes along with their corresponding column index. For each visited node, add its value to a dictionary keyed by its column index. When processing children, update the column index appropriately (left child: col-1, right child: col+1). After the traversal, sort the keys of the dictionary and compile the node values in order from smallest to largest column index. This approach guarantees that nodes at the same level will be processed in left-to-right order, matching the required output.

#333 Largest BST Subtree [Medium]

Key Insights

- Use a bottom-up DFS (post-order traversal) to process each node.
- For each node, determine whether its left and right subtrees are BSTs.
- Maintain additional data per subtree: size (number of nodes), minimum value, and maximum value.
- A subtree rooted at the current node is a BST if:
- Its left subtree is a BST.
- Its right subtree is a BST.
- The maximum value in the left subtree is less than the current node's value.
- The current node's value is less than the minimum value in the right subtree.
- Track the size of the largest BST identified during the traversal.

Space & Time

Time Complexity: O(n) (each node is visited once)

Space Complexity: O(h), where h is the height of the tree (due to recursion stack)

Solution

Perform a post-order traversal of the binary tree. For each node, return a tuple (or object in some languages) that provides:

- Whether the subtree rooted at this node is a BST. - The size of the subtree if it is a BST (or the size of the largest BST found within it). - The minimum value in the subtree. - The maximum value in the subtree.

At each node, if both left and right children form valid BSTs and the current node's value fits the BST condition (greater than left's max and less than right's min), then the subtree rooted at the node is also a BST. Calculate its size and update the current global maximum if necessary. If the condition fails, return the maximum BST size found in its subtrees.

#366 Find Leaves of Binary Tree [Medium]

Key Insights

- Use a postorder traversal (bottom-up approach) to process the tree.
- Determine the "height" of each node (distance from the deepest leaf) where leaves have a height of 1.
- Group nodes by their computed height to simulate the layer-wise removal of leaves.
- The same node may become a leaf after its children are removed.

Space & Time

Time Complexity: O(n) — Each node is visited exactly once.

Space Complexity: O(n) — Space used for the recursion stack and the result list.

Solution

The solution uses a recursive helper function that performs a postorder traversal of the tree. For each node, it computes its height as max(height of left, height of right) + 1. Nodes with the same height are collected together in a list. This idea mirrors the process of removing leaves level by level, where leaves (height 1) are removed first, then their parents become new leaves, and so on. Data Structures used:

- An array or list to maintain the groups of node values by their height. Algorithmic Approach: - Use Depth-First Search (DFS) via recursion. - Postorder traversal ensures that leaf nodes (base cases) are processed before their parents. - Append the node's value to the correct list based on the computed height.

#437 Path Sum III [Medium]

Key Insights

- A path can start and end at any nodes, as long as it goes downwards.
- Using DFS helps traverse the tree and explore all possible paths.
- A prefix sum technique with a hash map allows quick calculation of the sum of any subpath.
- Backtracking is used to remove a path's prefix sum when moving back up the recursion tree.
- Although a brute-force approach exists, it is inefficient compared to the prefix sum method.

Space & Time

Time Complexity: O(n) on average (where n is the number of nodes), though worst-case can be O(n^2) in a skewed tree.

Space Complexity: O(n) due to recursion stack and the hash map that stores prefix sums.

Solution

The solution employs a DFS traversal of the tree along with a hash map that records the frequency of prefix sums encountered so far. At each node, the current prefix sum is updated by adding the node's value. We then check the hash map for how many times (current prefix sum - targetSum) has occurred, since this value indicates the existence of a subpath summing to targetSum. We update the hash map with the current prefix sum, continue the DFS to traverse child nodes, and then backtrack by decrementing the current prefix sum's count before retreating to the parent node. This ensures that sums from other branches are not incorrectly combined.

#545 Boundary of Binary Tree [Medium]

Key Insights

- Break down the problem into three parts:
- Left boundary (excluding leaves).
- All leaves (do a DFS to find leaves).
- Right boundary (excluding leaves, then reverse the collected list).
- Special handling is needed when the tree doesn't have a left or right subtree.
- The root is always included and is never considered as a leaf, even if it has no children.
- Avoid duplication of leaf nodes in the left/right boundaries.

Space & Time

Time Complexity: O(n) - Every node is visited at most once.

Space Complexity: O(n) - O(n) space is used for the output list and recursion stack (in worst case tree is skewed).

Solution

The solution uses a Depth-First Search (DFS) strategy to traverse the tree and collect the boundary nodes in three steps:

- For the left boundary, start at root.left and traverse down, adding nodes (skipping leaves). - For the leaves, perform a DFS from the root and add any node that has no children. - For the right boundary, start at root.right and traverse down, adding nodes (skipping leaves). Since the right boundary must be added in reverse order, we reverse the collected list after traversal. - Finally, concatenate these lists in the order: root, left boundary, leaves, reversed right boundary.

The approach makes use of helper functions to clearly separate the logic for gathering left boundary, leaves, and right boundary. This modular approach simplifies reasoning and avoids edge case mistakes such as including duplicate leaf nodes.

#549 Binary Tree Longest Consecutive Sequence II [Medium]

Key Insights

- Use Depth-First Search (DFS) to visit every node.
- For each node, compute:
- The longest increasing consecutive path starting from that node.
- The longest decreasing consecutive path starting from that node.
- Combine the increasing and decreasing paths (subtracting one to avoid double counting the current node) to potentially form a longer consecutive sequence through the node.
- Update a global maximum to record the longest path seen.

Space & Time

Time Complexity: O(n), where n is the number of nodes since every node is visited once.

Space Complexity: O(h), where h is the height of the tree, due to the recursion stack.

Solution

The solution employs a DFS traversal. At every node, we calculate two values:

- inc - the length of the longest increasing consecutive sequence starting from that node. - dec - the length of the longest decreasing consecutive sequence starting from that node.

For each child of the current node:

- If the child's value is node.val + 1, update the increasing sequence. - If the child's value is node.val - 1, update the decreasing sequence.

Then, the longest path through the current node is inc + dec - 1 (subtracting one to avoid counting the current node twice). A global variable maintains the maximum length encountered. This process works regardless of the path direction because it combines both increasing and decreasing parts.

#582 Kill Process [Medium]

Key Insights

- Build a mapping (dictionary/hash table) from a parent process ID to its list of child process IDs.
- Use Depth-First Search (DFS) or Breadth-First Search (BFS) starting from the kill process to traverse the tree and collect all descendant processes.
- The problem requires handling a tree structure that may have multiple children per node.

Space & Time

Time Complexity: O(n), where n is the number of processes (both building the mapping and the tree traversal take linear time).

Space Complexity: O(n), for storing the mapping and the traversal queue/stack.

Solution

We first construct a mapping from each process's parent ID to its list of child IDs. This allows us to quickly access all children of any process. Once the mapping is built, we perform a tree traversal (either BFS or DFS) starting from the process to kill. During the traversal, we record every process encountered. This collected list represents all processes that will be terminated. The data structures used are a dictionary (for the mapping) and either a stack (for DFS) or a queue (for BFS), making the solution efficient and straightforward.

#662 Maximum Width of Binary Tree [Medium]

Key Insights

- Perform a level order traversal (BFS) of the tree.
- Assign an index to each node as if the tree were complete (e.g., root is index 1, left child is 2
- i, right child is 2
- i+1).
- For each level, compute the width as (last_index - first_index + 1).
- Using indices allows capturing gaps between nodes due to nulls in between.
- Edge cases include handling very skewed trees.

Space & Time

Time Complexity: O(N), where N is the number of nodes in the tree.

Space Complexity: O(N), due to the queue used for BFS which in the worst case holds nodes of the last level.

Solution

We use a Breadth-First Search (BFS) approach with a queue to traverse the tree level by level. At each level, we pair each node with an index representing its position assuming a complete binary tree. This allows us to compute the width of each level using the formula: width = last_index - first_index + 1. We then track the maximum width seen throughout the traversal. The use of indices handles cases where there are null gaps between nodes.

#863 All Nodes Distance K in Binary Tree [Medium]

Key Insights

- Convert the binary tree into an undirected graph so that we can easily traverse both down to the children and up to the parent.
- Use a DFS or simple recursive traversal to create an adjacency list representing the graph.
- Perform a BFS starting from the target node to explore nodes level-by-level until reaching distance k.
- Use a visited set (or equivalent) to avoid revisiting nodes and to prevent cycles.

Space & Time

Time Complexity: O(N), where N is the number of nodes in the tree. Each node is visited once when constructing the graph and once in the BFS.

Space Complexity: O(N), for storing the graph and the additional data structures used in BFS and DFS.

Solution

The solution involves two main steps:

- Build an undirected graph representation of the binary tree using a DFS. For each node, add the node's value as a key in an adjacency list and connect it with its children (if any). This creates bidirectional links between parent and child. - Use a BFS starting from the target node and explore nodes level-by-level. Keep a counter for the current distance, and once the counter reaches k, return all node values found at that level. To avoid processing nodes more than once, maintain a set of visited nodes.

#1120 Maximum Average Subtree [Medium]

Key Insights

- Use a depth-first search (DFS) to traverse every subtree.
- At each node, compute the sum and count of nodes in its subtree.
- Calculate the average for the current subtree and update a global maximum if needed.
- Processing every node exactly once yields an efficient solution.

Space & Time

Time Complexity: O(n), where n is the number of nodes (each node is visited once).

Space Complexity: O(h), where h is the height of the tree (space for recursion stack).

Solution

We perform a post-order DFS traversal, calculating at each node both the cumulative sum and the count of nodes in its subtree. This enables us to compute the average value in constant time for any subtree. We maintain a global variable to track the maximum average encountered during the traversal. The primary technique is recursive DFS combined with tuple (or pair) aggregation. The simplicity of the recursion and in-place aggregation ensures an efficient and clear solution.

#1490 Clone N-ary Tree [Medium]

Key Insights

- The tree is N-ary, meaning every node can have multiple children.
- A deep copy requires creating new nodes for each node in the tree.
- Both DFS (recursive) or BFS (iterative) traversals can be used to clone the tree.
- The overall time complexity is O(N) since each node is visited once.
- We can follow similar cloning techniques used in cloning graphs, using either recursion or an auxiliary data structure like a queue.

Space & Time

Time Complexity: O(N), where N is the number of nodes in the tree, since each node is visited exactly once.

Space Complexity: O(N), accounting for the recursive call stack (or an explicit queue in BFS) and the space needed to store the clone.

Solution

We perform a DFS traversal starting from the root node. For each node we encounter, we:

- Create a new node with the same value. - Recursively clone all its children and attach the cloned children to the new node. This approach ensures that every original node is copied into a new node and that the parent-child relationships are preserved. Edge cases such as an empty tree (null root) are also handled by returning null immediately.

#1522 Diameter of N-Ary Tree [Medium]

Key Insights

- The diameter of the tree can be found by considering the two largest depths from any node among its children.
- Use a depth-first search (DFS) to compute the maximum depth from each node.
- For each node, the potential diameter is the sum of the two longest depths among its children.
- Update a global variable tracking the maximum diameter as you recurse through the tree.

Space & Time

Time Complexity: O(n), where n is the number of nodes, since each node is visited once.

Space Complexity: O(h), where h is the height of the tree, due to the recursion stack (worst-case O(n) for a skewed tree).

Solution

Use a DFS approach to traverse the tree. At each node, calculate the longest and second longest depth among its children. The sum of these two depths gives the candidate diameter through that node. Maintain a global variable to track the maximum diameter found so far. Return the maximum depth from the node (i.e., one plus the largest depth among its children) to be used in parent's calculations.

#1650 Lowest Common Ancestor of a Binary Tree III [Medium]

Key Insights

- Each node has a parent pointer, so you can traverse from any node to the root.
- By determining the depth of each node, you can "align" the two nodes to the same depth.
- When both nodes are aligned, traverse upward simultaneously until they meet.
- This approach guarantees finding the LCA with O(h) time complexity, where h is the height of the tree.

Space & Time

Time Complexity: O(h), where h is the height of the tree.

Space Complexity: O(1)

Solution

We first compute the depth of both nodes by moving up their parent pointers. If one node is deeper, move it upward until both nodes are at the same level. Then, move both nodes upward one step at a time until they meet; the meeting point is the lowest common ancestor. This approach uses constant extra space and leverages the parent pointer for an efficient solution.

#124 Binary Tree Maximum Path Sum [Hard]

Key Insights

- Use Depth-First Search (DFS) to explore the tree in a post-order fashion.
- For each node, calculate the maximum gain including that node and optionally one of its subtrees.
- A node's best contribution to its parent can be either just its own value or its value plus the maximum gain from either the left or right child.
- Update a global maximum path sum that considers the possibility of combining both left and right gains with the current node.
- Handle negative values by comparing gains against 0 to avoid decreasing the path sum.

Space & Time

Time Complexity: O(n), where n is the number of nodes in the tree, since each node is visited exactly once.

Space Complexity: O(n) in the worst-case scenario due to the recursion stack (O(H) where H is the height of the tree, which can be O(n) for skewed trees).

Solution

We use a recursive DFS approach (post-order traversal) to compute the maximum gain from each subtree. For every node:

- Recursively compute the maximum gain from the left and right child, ensuring that negative gains are treated as 0.

- The maximum gain that can be provided to the node's parent is the node's value plus the larger gain from its left or right child. - Simultaneously, the potential maximum path sum including the current node as the highest (or root) node is the node's value plus both left and right gains. - Update a global variable that holds the maximum path sum found so far. - Return the maximum gain (node value plus one best child) to be used by the node's parent.

This strategy ensures that every possible path sum is considered while avoiding double-counting nodes.

#297 Serialize and Deserialize Binary Tree [Hard]

Key Insights

- The tree can be traversed using either depth-first search (DFS) or breadth-first search (BFS).
- Using BFS (level-order traversal) simplifies reconstruction since nodes are processed level-by-level.
- Placeholder markers (e.g., "null") are used to indicate missing children.
- Both serialization and deserialization processes should handle empty trees and edge cases gracefully.
- Ensuring a one-to-one mapping between the tree structure and the serialized string is critical.

Space & Time

Time Complexity: O(n) for both serialization and deserialization, where n is the number of nodes in the tree.

Space Complexity: O(n) due to the storage used for the serialized string and additional data structures (queue) during processing.

Solution

This solution uses level-order traversal (BFS) for both serialization and deserialization. During serialization, each node is processed sequentially with missing children recorded as "null". For deserialization, the string is split by commas, and a queue is used to rebuild the tree level by level. The root is created from the first token, then subsequent tokens are assigned as left and right children accordingly, ensuring an accurate reconstruction of the original tree.