

Algorithmic Adventures: Cracking Puzzles with Code

Ehsan Shah-Hosseini

2024

Algorithmic Adventures: Cracking Puzzles with Code

by Ehsan Shah Hosseini

© 2024 Ehsan Shah-Hosseini

All rights reserved.

No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage and retrieval system, without prior written permission from the author.

ISBN: 978-1-XXXXXX-XX-X

First Edition

Printed in the United States of America

Independently published by Ehsan Shah-Hosseini

Contents

<i>List of Problems</i>	7
<i>I Fundamental Data Structures</i>	13
<i>1 Arrays and Lists in Python</i>	15
<i> 1.1 Python Lists as Dynamic Arrays</i>	15
<i> 1.1.1 Creating a List</i>	15
<i> 1.1.2 Common List Operations</i>	16
<i> 1.1.3 List Comprehensions</i>	18
<i> 1.2 Time Complexity of List Operations</i>	18
<i> 1.3 Common Terms in Array and List Manipulation</i>	18
<i> 1.4 Key Considerations During Interviews</i>	19
<i> 1.5 Common Corner Cases to Test</i>	20
<i> 1.6 Techniques Applicable to Arrays and Strings</i>	21
<i> 1.7 Introduction to NumPy Arrays (Optional)</i>	21
<i> 1.7.1 Creating a NumPy Array</i>	22
<i> 1.7.2 Example of Vectorized Operations</i>	22

1.8	<i>Conclusion</i>	22
1.9	<i>Essential Array Questions: An Introduction</i>	27
1.9.1	<i>Merge Sorted Array</i>	37
2	<i>Linked Lists</i>	75
3	<i>Stacks and Queues</i>	107
4	<i>Stacks and LIFO</i>	109
4.1	<i>Introduction to Stacks</i>	109
4.2	<i>History of Stacks</i>	109
4.2.1	<i>Early Development</i>	110
4.2.2	<i>Stacks in Programming Languages</i>	110
4.3	<i>Fundamental Concepts of LIFO</i>	110
4.3.1	<i>Stack Operations</i>	110
4.3.2	<i>Stack Representation</i>	110
4.4	<i>Implementations of Stacks</i>	111
4.4.1	<i>Array Implementation</i>	111
4.4.2	<i>Linked List Implementation</i>	111
4.5	<i>Applications of Stacks</i>	112
4.5.1	<i>Expression Evaluation and Conversion</i>	112
4.5.2	<i>Function Call Management</i>	112
4.5.3	<i>Undo Mechanisms</i>	113
4.5.4	<i>Depth-First Search (DFS)</i>	113
4.5.5	<i>Memory Management</i>	113

4.6	<i>Conclusion</i>	113
5	<i>Queues</i>	135
5.1	<i>Introduction</i>	135
5.2	<i>Types of Queues</i>	135
5.3	<i>Basic Operations</i>	136
5.4	<i>Implementations of Queues</i>	137
5.5	<i>Complexities of Operations</i>	137
5.6	<i>Python Implementations</i>	137
5.7	<i>Applications of Queues</i>	140
5.8	<i>Important Queue Problems</i>	141
6	<i>Hash Tables</i>	153
7	<i>Hashing and Hash Tables</i>	155
7.1	<i>More Examples of Sliding Window Technique</i>	184
7.2	<i>Index as a Hash Key</i>	212
II	<i>Algorithmic Techniques</i>	219
8	<i>Sorting Algorithms</i>	221
8.1	<i>Sorting the Array</i>	221
9	<i>Two-Pointer Technique</i>	245
9.1	<i>Two Pointers Technique</i>	245

<i>10 Sliding Window Technique</i>	271
<i>10.1 Sliding Window Technique</i>	271
<i>10.2 More Examples of Sliding Window Technique</i>	280
<i>11 Binary Search</i>	309
<i>12 Recursion and Backtracking</i>	331
<i>13 Recursion</i>	333
<i>14 Backtracking</i>	337
<i>III Advanced Topics</i>	377
<i>15 Dynamic Programming</i>	379
<i>15.1 Kadane's Algorithm</i>	399
<i>15.2 Traversing from the Right</i>	437
<i>15.3 Intervals</i>	446
<i>15.3.1 Types of Intervals</i>	446
<i>15.3.2 Key Concepts</i>	446
<i>15.3.3 Important Considerations</i>	446
<i>15.3.4 Common Problems Involving Intervals</i>	446
<i>15.3.5 Techniques for Solving Interval Problems</i>	447
<i>15.3.6 Best Practices</i>	447

<i>15.4 Conclusion</i>	447
<i>15.5 Precomputation</i>	472

16 Trees and Graphs	489
----------------------------	-----

17 Trees	491
<i>17.1 Introduction to Trees</i>	491
<i>17.2 Properties of Trees</i>	491
<i>17.3 Binary Trees</i>	491
<i>17.3.1 Types of Binary Trees</i>	492
<i>17.4 Binary Search Trees (BSTs)</i>	492
<i>17.5 Tree Traversal</i>	492
<i>17.5.1 Depth-First Traversal</i>	492
<i>17.5.2 Breadth-First Traversal (Level Order)</i>	492
<i>17.6 Tree Operations</i>	493
<i>17.6.1 Insertion</i>	493
<i>17.6.2 Deletion</i>	493
<i>17.6.3 Searching</i>	493
<i>17.7 Applications of Trees</i>	493

18 Graphs	557
------------------	-----

<i>18.1 Number of Islands</i>	562
-------------------------------	-----

List of Problems

1.1	Contains Duplicate	23
1.2	Best Time to Buy and Sell Stock	27
1.3	Product of Array Except Self	30
1.4	Maximum Subarray	33
1.5	Two Pointers Technique for Merging Arrays	36
1.6	Maximum Product Subarray	43
1.7	Find Minimum in Rotated Sorted Array	46
1.8	Search in Rotated Sorted Array	49
1.9	Merge Two Sorted Lists	52
1.10	Remove Duplicates from a Sorted Array	57
1.11	Rotate Image	61
1.12	Spiral Matrix	65
1.13	Set Matrix Zeroes	70
2.1	Reverse a Linked List	78
2.2	Detect Cycle in a Linked List	82
2.3	Reorder List	87
2.4	Remove Nth Node From End of List	91
2.5	Add Two Numbers	96
2.6	Linked List Cycle	100
4.1	Valid Parentheses	114
4.2	Min Stack	115
4.3	Evaluate Reverse Polish Notation	119

4.4	Daily Temperatures	120
4.5	Next Greater Element	124
4.6	Asteroid Collision	125
4.7	Basic Calculator	127
4.8	Basic Calculator II	129
4.9	Largest Rectangle in Histogram	130
4.10	Trapping Rain Water	132
5.1	Implement Stack using Queues	141
5.2	Design Hit Counter	145
5.3	Number of Recent Calls	148
7.1	Two Sum	158
7.2	3Sum	161
7.3	Group Anagrams	165
7.4	Valid Anagram	168
7.5	Top K Frequent Elements	173
7.6	Longest Consecutive Sequence	178
7.7	Longest Substring Without Repeating Characters	184
7.8	Longest Palindromic Substring	187
7.9	First Missing Positive	191
7.10	Insert Delete GetRandom O(1)	194
7.11	Least Recently Used (LRU) Cache	197
7.12	All O(1) Data Structure	201
7.13	Anagram Detection	205
7.14	Ransom Note	209
7.15	First Missing Positive	213
7.16	Find All Pairs with a Given Target Sum	216
8.1	Merge Sort	224
8.2	Quick Sort	229
8.3	Sort Colors	235

8.4	Kth Largest Element in an Array	241
9.1	Find All Pairs with a Given Target Sum	245
9.2	3Sum	250
9.3	Container With Most Water	253
9.4	Remove Duplicates from a Sorted Array	256
9.5	Valid Palindrome	260
9.6	Trapping Rain Water	264
9.7	Longest Substring with At Most Two Distinct Characters	265
10.1	Maximum Sum Subarray of Size k	272
10.2	Longest Substring with At Most k Distinct Characters	275
10.3	Longest Substring Without Repeating Characters	280
10.4	Minimum Window Substring	283
10.5	Longest Repeating Character Replacement	287
10.6	Find All Anagrams in a String	290
10.7	Substring with Concatenation of All Words	293
10.8	Minimum Size Subarray Sum	300
10.9	Permutation in String	302
11.1	Binary Search in a Sorted Array	309
11.2	Search in Rotated Sorted Array	312
11.3	Find Minimum in Rotated Sorted Array	315
11.4	Search a 2D Matrix	318
11.5	Kth Smallest Element in a Sorted Matrix	321
11.6	Median of Two Sorted Arrays	325
14.1	N-Queens Problem	338
14.2	Generate Parentheses	339
14.3	Letter Combinations of a Phone Number	342
14.4	Permutations	345
14.5	Combinations	348
14.6	Subsets (Power Set)	351

14.7 Subsets with Duplicates	354
14.8 Solve Sudoku	357
14.9 Strobogrammatic Number II	361
14.10 Word Search	364
14.11 N-Queens Problem	369
14.12 Combination Sum	372
15.1 Climbing Stairs	383
15.2 Coin Change	386
15.3 House Robber	389
15.4 Longest Increasing Subsequence	393
15.5 Maximum Product Subarray	396
15.6 Maximum Subarray	402
15.7 Unique Paths	405
15.8 Edit Distance	408
15.9 Longest Common Subsequence	411
15.10 Decode Ways	416
15.11 Jump Game I	424
15.12 Palindromic Substrings	431
15.13 Merging Two Sorted Arrays	438
15.14 Daily Temperatures	439
15.15 Number of Visible People in a Queue	443
15.16 Meeting Rooms	447
15.17 Meeting Rooms II	448
15.18 Insert Interval	449
15.19 Merge Intervals	451
15.20 Non-overlapping Intervals	455
15.21 Reconstruct Queue by Height	459
15.22 Task Scheduler	464
15.23 Product of Array Except Self	473

15.24 Range Sum Query - Immutable	476
15.25 Range Sum Query 2D - Immutable	481
17.1 Binary Tree vs. Binary Search Tree	493
17.2 Same Tree	498
17.3 Symmetric Tree	500
17.4 Maximum Depth of Binary Tree	503
17.5 Balanced Binary Tree	506
17.6 Invert Binary Tree	510
17.7 Binary Tree Level Order Traversal	513
17.8 Binary Tree Right Side View	518
17.9 Lowest Common Ancestor of a Binary Search Tree	522
17.10 Validate Binary Search Tree	526
17.11 Kth Smallest Element in a Binary Search Tree	530
17.12 Lowest Common Ancestor of a Binary Tree	534
17.13 Binary Tree Maximum Path Sum	538
17.14 Subtree of Another Tree	542
17.15 Construct Binary Tree from Preorder and Inorder Traversal	546
17.16 Serialize and Deserialize Binary Tree	550
18.1 Clone Graph	570
18.2 Pacific Atlantic Water Flow	576
18.3 Graph Valid Tree	584
18.4 Course Schedule	591
18.5 Alien Dictionary	598
18.6 Word Ladder	601
18.7 Word Ladder II	609
18.8 Rotting Oranges	618
18.9 Heaps	626
18.10 Merge k Sorted Lists	633
18.11 Find Median from Data Stream	635

Part I

Fundamental Data Structures

Chapter 1

Arrays and Lists in Python

Arrays and lists are fundamental data structures that form the backbone of efficient data handling in many programming scenarios. In Python, lists are dynamic arrays that can hold items of varying data types, making them highly versatile and widely used.

Fundamental data structures

1.1 Python Lists as Dynamic Arrays

Python lists are mutable, ordered sequences of elements that can store items of different data types. They are implemented as dynamic arrays, which means they can automatically adjust their size when elements are added or removed.

Mutable and ordered
Dynamic resizing

1.1.1 Creating a List

Here's how you can create a list in Python to store account balances:

```
# Creating a list to store account balances
balance = [350, 420, 180]

print(balance) # Output: [350, 420, 180]
```

Listing 1.1: Creating a list to store account balances

Explanation: Lists are defined using square brackets, and elements are separated by commas. In this example, the `balance` list contains three integer values representing account balances.

Syntax for lists

1.1.2 Common List Operations

Python lists support a variety of operations that make data manipulation easy and efficient.

- **Accessing Elements:**

You can access elements in a list using their index:

```
print(balance[0]) # Output: 350
```

Listing 1.2: Accessing elements in a list

Explanation: Indexing starts at 0. Accessing elements by index is a constant Zero-based indexing time operation ($O(1)$).

- **Updating Elements:**

You can update elements in a list by assigning a new value to a specific index:

```
balance[1] = 275
print(balance) # Output: [350, 275, 180]
```

Listing 1.3: Updating elements in a list

Explanation: This operation replaces the element at index 1 with the new value 275.

- **Appending Elements:**

You can add elements to the end of the list using the `append()` method:

```
balance.append(490)
print(balance) # Output: [350, 275, 180, 490]
```

Listing 1.4: Appending elements to a list

Explanation: The `append()` method adds a new element to the end of the list. `append()` method This operation has an amortized time complexity of $O(1)$.

- **Inserting Elements:**

You can insert elements at a specific position using the `insert()` method:

```
balance.insert(1, 220)
print(balance) # Output: [350, 220, 275, 180, 490]
```

Listing 1.5: Inserting elements into a list

Explanation: This inserts the value 220 at index 1, shifting subsequent elements to the right. The time complexity is $O(n)$ due to the shifting of elements. Elements are shifted

- **Removing Elements:**

You can remove elements by value using the `remove()` method or by index using the `pop()` method:

```
balance.remove(180)
print(balance)  # Output: [350, 220, 275, 490]

removed_element = balance.pop(2)
print(removed_element)  # Output: 275
print(balance)  # Output: [350, 220, 490]
```

Listing 1.6: Removing elements from a list

Explanation: The `remove()` method deletes the first occurrence of the specified value. The `pop()` method removes and returns the element at the specified index. Removing elements may require shifting and has a time complexity of `pop()` method $O(n)$.

- **Slicing Lists:**

You can create sublists using slicing:

```
sub_balance = balance[0:2]
print(sub_balance)  # Output: [350, 220]
```

Listing 1.7: Slicing a list

Explanation: Slicing creates a new list containing elements from the start index up to, but not including, the end index. This operation has a time complexity of Start index inclusive, end index exclusive $O(k)$, where k is the number of elements in the slice.

- **Concatenating Lists:**

You can concatenate lists using the `+` operator:

```
additional_balances = [600, 700]
new_balance = balance + additional_balances
print(new_balance)  # Output: [350, 220, 490, 600, 700]
```

Listing 1.8: Concatenating lists

Explanation: Concatenation creates a new list by combining two lists. The time complexity is $O(n + m)$, where n and m are the lengths of the two lists.

- **Iterating Over a List:**

You can traverse a list using a loop:

```
for amount in balance:
    print(amount)
```

Listing 1.9: Iterating over a list

Explanation: This iterates over each element in the list, with a time complexity of $O(n)$.

1.1.3 List Comprehensions

List comprehensions provide a concise way to create lists:

```
# Creating a list of squares
squares = [x**2 for x in range(5)]
print(squares) # Output: [0, 1, 4, 9, 16]
```

Listing 1.10: Using list comprehensions

Explanation: This creates a new list by applying an expression to each element in a sequence.

List comprehension syntax

1.2 Time Complexity of List Operations

Understanding the time complexity of various list operations is crucial for writing efficient code. Here's a summary:

Algorithm efficiency

Operation	Time Complexity
Indexing	$O(1)$
Append	$O(1)$ amortized
Pop (end)	$O(1)$
Pop (middle)	$O(n)$
Insert	$O(n)$
Delete	$O(n)$
Iteration	$O(n)$
Get Slice	$O(k)$
Extend	$O(k)$
Sort	$O(n \log n)$
Reverse	$O(n)$

Table 1.1: Time Complexity of Common List Operations

Note: k represents the number of elements involved in the operation.

1.3 Common Terms in Array and List Manipulation

- **Subarray/Sublist:** A contiguous portion of an array or list.

```
numbers = [2, 3, 6, 1, 5, 4]
subarray = numbers[1:4] # [3, 6, 1]
```

Listing 1.11: Example of a sublist

- **Subsequence:** A sequence derived by selecting elements without changing their order, not necessarily contiguous.

```
numbers = [2, 3, 6, 1, 5, 4]
subsequence = [3, 1, 5]
```

Listing 1.12: Example of a subsequence

- **Prefix:** A subarray/sublist that starts at the first element.

```
prefix = numbers[:3] # [2, 3, 6]
```

Listing 1.13: Example of a prefix

- **Suffix:** A subarray/sublist that ends at the last element.

```
suffix = numbers[3:] # [1, 5, 4]
```

Listing 1.14: Example of a suffix

- **Sliding Window:** A technique where a window of a certain size moves through the array or list to process elements.

```
window_size = 3
for i in range(len(numbers) - window_size + 1):
    window = numbers[i:i+window_size]
    # Process the window
```

Listing 1.15: Using a sliding window

1.4 Key Considerations During Interviews

When solving array or list problems in interviews, it's important to clarify and consider several factors:

- **Input Constraints:**

- Size of the array or list (e.g., up to 10^5 elements). - Range of element values (e.g., Consider time complexity integers within $[-10^9, 10^9]$).

- **Duplicate Values:**

- Can the array contain duplicate values? - How should duplicates be handled?

- **Sorted vs. Unsorted:**

- Is the array sorted? - If not, can it be sorted, or must the original order be preserved?

- **Modification of Input:**

- Are you allowed to modify the input array or list? - Should the solution be in-place?

- **Time and Space Complexity Requirements:**

- Is there a specific time complexity target (e.g., $O(n)$, $O(n \log n)$)? - Are there space limitations (e.g., constant extra space)?

- **Edge Cases:**

- How should the solution handle empty arrays or lists? - What if the array contains only one element?

- **Return Requirements:**

- What should be returned (e.g., a value, an index, a modified array)? - Are multiple outputs required?

By addressing these considerations upfront, you can develop a solution that aligns with the problem's requirements and demonstrates your analytical skills.

Clarify requirements early

1.5 Common Corner Cases to Test

When testing your solutions, consider the following edge cases:

- **Empty Array or List:**

- Input: [] - Ensure your code handles this without errors.

- **Single Element:**

- Input: [5] - Verify that the code works when the array has only one element.

- **All Identical Elements:**

- Input: [2, 2, 2, 2] - Check for issues related to duplicate handling.

- **Already Sorted:**

- Input: [1, 2, 3, 4, 5] - Confirm that the solution works when the array is sorted.

- **Reverse Sorted:**

- Input: [5, 4, 3, 2, 1] - Test behavior with descending order.

- **Negative and Positive Integers:**

- Input: [-3, -1, 0, 2, 4] - Ensure correct handling of negative numbers.

- **Maximum/Minimum Integers:**

- Input: [2**31 - 1, -2**31] - Check for overflow or underflow issues.

- **Very Large Array:**

- Input: A list with 10^6 elements. - Test the efficiency and performance of your solution.

1.6 Techniques Applicable to Arrays and Strings

Many techniques used for arrays are also applicable to strings since strings are sequences of characters.

Strings as arrays of characters

- **Two-Pointer Technique:**

- Used for problems like reversing a string or checking for palindromes.

- **Sliding Window Technique:**

- Useful for finding substrings with certain properties, such as the longest substring without repeating characters.

- **Hashing and Frequency Counting:**

- Employed to count occurrences of elements or characters.

- **Dynamic Programming:**

- Applied to problems like the longest common subsequence or the edit distance between two strings.

- **Sorting:**

- Sorting characters or elements to detect anagrams or simplify comparisons.

Example: The two-pointer technique can be used to reverse both an array and a string in-place.

```
# Reversing a list
def reverse_list(lst):
    left, right = 0, len(lst) - 1
    while left < right:
        lst[left], lst[right] = lst[right], lst[left]
        left += 1
        right -= 1

# Reversing a string (strings are immutable, so we convert to a list)
def reverse_string(s):
    lst = list(s)
    reverse_list(lst)
    return ''.join(lst)
```

Listing 1.16: Reversing a list and a string using the two-pointer technique

1.7 Introduction to NumPy Arrays (Optional)

For numerical computations, the NumPy library provides the `ndarray` object, which offers efficient storage and operations for large arrays of homogeneous data types.

NumPy for numerical computing

1.7.1 Creating a NumPy Array

First, install NumPy if you haven't already:

```
pip install numpy
```

Listing 1.17: Installing NumPy

Then, you can create a NumPy array:

```
import numpy as np

# Creating a NumPy array
balance = np.array([350, 420, 180])

print(balance) # Output: [350 420 180]
```

Listing 1.18: Creating a NumPy array

Advantages of NumPy Arrays:

- **Performance:** Faster computations due to optimized C code under the hood.
- **Memory Efficiency:** Require less memory than Python lists for large datasets.
- **Vectorized Operations:** Allow element-wise operations without explicit loops.
- **Functionality:** Provide numerous mathematical functions for arrays.

1.7.2 Example of Vectorized Operations

```
# Adding 10 to each element
balance += 10
print(balance) # Output: [360 430 190]

# Element-wise multiplication
balance *= 2
print(balance) # Output: [720 860 380]
```

Listing 1.19: Vectorized operations with NumPy arrays

Explanation: Operations are applied to each element without the need for explicit iteration.

Avoid explicit loops

1.8 Conclusion

Arrays and lists are foundational data structures in Python, essential for efficient data manipulation and algorithm implementation. Understanding how to effectively use these structures is crucial for solving algorithmic puzzles.

Foundational concepts

tively use lists—as dynamic arrays—is crucial for writing optimized code in everyday programming and technical interviews.

By mastering list operations, recognizing time and space complexities, and being mindful of edge cases, you can develop robust solutions to a wide range of problems. Develop robust solutions Additionally, for numerical computations requiring high performance, NumPy arrays offer significant advantages.

As you continue through this book, you’ll encounter various problems that build upon these concepts, applying techniques like the two-pointer method, sliding window, and dynamic programming to both arrays and strings. Strengthening Building on fundamentals your grasp of these fundamentals will enhance your problem-solving skills and prepare you for more advanced topics ahead.

Practice Exercise

Implement a function that returns the maximum sum of any contiguous subarray of a given list of integers.

```
def max_subarray(nums):
    # Your code here
    pass

# Example usage:
print(max_subarray([-2,1,-3,4,-1,2,1,-5,4]))  # Output: 6
```

Listing 1.20: Practice exercise: Maximum subarray

Hint: Consider using Kadane’s Algorithm for an efficient solution.

Problem 1.1 Contains Duplicate

The “Contains Duplicate” problem is a fundamental challenge in array manipulation, testing one’s ability to efficiently detect duplicates in an array. While simple in its premise, this problem offers opportunities to explore trade-offs between time complexity, space complexity, and data structure selection.

Problem Statement

Given an integer array `nums`, return `true` if any value appears at least twice in the array, and `false` if every element is distinct.

Input: - An integer array `nums`.

Output: - A boolean indicating whether duplicates exist.

Example 1:

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

Example 2:

Input: `nums = [1,2,3,4]`
 Output: `false`

Example 3:

Input: `nums = [1,1,1,3,3,4,3,2,4,2]`
 Output: `true`

Algorithmic Approaches

Several methods can be used to solve this problem, each with its own trade-offs:

1. **Sorting:** Sort the array and check if any consecutive elements are equal. While this approach is simple and intuitive, it modifies the input array and has a time complexity of $O(n \log n)$ due to sorting. It requires $O(1)$ space if the sorting is done in place, but $O(n)$ if a copy of the array is made¹.
2. **Hash Set:** Use a hash set to store elements as you iterate through the array. If an element is already in the set, return `true`. Otherwise, add the element to the set. This approach has a time complexity of $O(n)$ and a space complexity of $O(n)$, making it optimal in terms of time but less space-efficient².
3. **Brute Force:** Check every pair of elements in the array to see if they are equal. This method has a time complexity of $O(n^2)$ and should be avoided for large input sizes³.

¹ Sorting is suitable when modifying the array is allowed and space efficiency is a priority

² Hash sets are ideal for quick lookups, offering $O(1)$ average-case complexity per operation

³ Brute force methods are only viable for small arrays where performance is not a concern

Python Implementation

The following is an implementation using a hash set for optimal time complexity:

```
class Solution:
    def containsDuplicate(self, nums: List[int]) -> bool:
        seen = set()

        for num in nums:
            if num in seen:
                return True
            seen.add(num)

        return False
```

Explanation:

- The `seen` set is used to track elements encountered in the array⁴.
- For each element in `nums`, the algorithm checks if it is already in the set. If yes, it immediately returns `true`.
- If no duplicates are found after processing all elements, it returns `false`.

⁴ Sets in Python provide $O(1)$ average-time complexity for insertions and lookups

Why This Approach?

The hash set approach is chosen for its balance of simplicity and efficiency. It avoids the $O(n \log n)$ overhead of sorting and the $O(n^2)$ inefficiency of brute force, making it the optimal choice for most scenarios.

Alternative Approaches

- **Sorting Method:**** Sort the array and check for consecutive duplicates. This approach avoids extra space but is slower than the hash set approach:

```
class Solution:
    def containsDuplicate(self, nums: List[int]) ->
        bool:
            nums.sort()
            for i in range(len(nums) - 1):
                if nums[i] == nums[i + 1]:
                    return True
            return False
```

- **Brute Force:**** Compare each pair of elements, which is straightforward but inefficient for large arrays:

```
class Solution:
    def containsDuplicate(self, nums: List[int]) ->
        bool:
            for i in range(len(nums)):
                for j in range(i + 1, len(nums)):
                    if nums[i] == nums[j]:
                        return True
            return False
```

Similar Problems to This One

- Contains Duplicate II:** Check for duplicates within a given index distance k .
- Contains Duplicate III:** Check for duplicates where the absolute difference between values is within a given range.

- **Intersection of Two Arrays:** Find common elements between two arrays.
- **Unique Elements in an Array:** Identify and return the unique elements in the array.

Things to Keep in Mind and Tricks

- Ensure the hash set approach is used only when extra space is permissible.
- Sorting modifies the input array, which may not be acceptable in all scenarios. Use a copy if the original array must remain intact.
- For small arrays, brute force may suffice due to its simplicity.
- Always test edge cases like empty arrays, arrays with one element, and arrays with all identical elements.

Complexities

- **Hash Set Approach:**
 - **Time Complexity:** $O(n)$, for iterating through the array.
 - **Space Complexity:** $O(n)$, for storing elements in the hash set.
- **Sorting Approach:**
 - **Time Complexity:** $O(n \log n)$, due to sorting.
 - **Space Complexity:** $O(1)$ (if sorting is done in place) or $O(n)$ (if using a copy of the array).
- **Brute Force:**
 - **Time Complexity:** $O(n^2)$, for comparing all pairs of elements.
 - **Space Complexity:** $O(1)$.

Conclusion

The **Contains Duplicate** problem is a straightforward yet essential challenge in array manipulation. The hash set approach provides the optimal balance of simplicity and efficiency, making it the preferred method in most cases. By understanding the trade-offs among the various approaches, you can tailor your solution to fit specific problem constraints and resource limitations.

1.9 Essential Array Questions: An Introduction

Arrays are among the most fundamental data structures in computer science, serving as the backbone for numerous algorithms and problem-solving strategies. Mastering essential array-related problems is crucial for excelling in technical interviews and understanding core algorithmic concepts. These problems often explore key topics such as subarray manipulation, prefix sums, hash keying, and sliding windows.

This section introduces a curated collection of essential array problems designed to cover a wide range of techniques and scenarios. Each problem emphasizes different aspects of array manipulation, optimization strategies, and algorithm design, ensuring comprehensive preparation. The following problems are presented with detailed explanations, algorithmic approaches, and Python implementations to solidify your understanding and equip you for success.

Problem 1.2 Best Time to Buy and Sell Stock

The "Best Time to Buy and Sell Stock" problem is a common interview question that assesses a candidate's ability to work with arrays and apply dynamic programming concepts. The problem requires finding the optimal buy and sell times to maximize profit from a series of daily stock prices.

Problem Statement

You are given an array `prices`, where `prices[i]` is the price of a given stock on the i th day. Your goal is to determine the maximum profit obtainable by buying on one day and selling on a later day. If no profit can be made, return 0.

Input: An array `prices`, with `prices[i]` representing the stock price on the i th day.

Output: The maximum profit that can be achieved.

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.
```

Example 2:

```
Input: prices = [7,6,4,3,1]
```



Figure 1.1: Best Time to Buy and Sell Stock.

Output: 0

Explanation: No transaction can yield a profit, hence the max profit = 0.

Algorithmic Approach

The common approach to solving this problem is to use a single pass. Iterate through the `prices` array while tracking the minimum price encountered so far. For each price, calculate the profit if the stock were sold at that price, updating the maximum profit as necessary.

Complexities

- **Time Complexity:** $O(n)$, where n is the number of days, since it requires a single pass through the array.
- **Space Complexity:** $O(1)$, as only a constant amount of extra space is needed for variables such as `min_price` and `max_profit`.

Python Implementation

The Python implementation uses the above algorithm to find the maximum profit with a single pass through the stock prices:

```

class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        min_price = float('inf') # Initialize to positive infinity
        max_profit = 0

        # Loop through all stock prices
        for price in prices:
            # Updating the minimum price seen so far
            if price < min_price:
                min_price = price
            # Computing potential profit and update max profit
            elif price - min_price > max_profit:
                max_profit = price - min_price
        return max_profit

```

This approach initializes the minimum price to infinity and the maximum profit to 0 and iterates over the `prices` array. For each stock price, it checks if the price is lower than the current minimum price. If it is, the minimum price is updated. Otherwise, it calculates the profit one would get if one sold at the current price and updates the maximum profit if the current profit is larger. After iterating through all the prices, the maximum profit is returned.

Why This Approach

This approach is chosen for its efficiency in both time and space. It navigates the array only once, ensuring a linear time complexity, which is optimal in this case because you have to examine each price at least once to determine the profit.

Alternative Approaches

Alternative methods might use more complex data structures, such as segment trees, to perform many queries on the range of days. However, this is overkill for the given problem and would not improve the best-case time complexity.

Similar Problems to This One

Similar problems include variations that allow for multiple transactions, incorporate a cooldown period between sales, or limit the number of transactions.

Things to Keep in Mind and Tricks

One trick for problems like this is to consider the running difference between the current and minimum prices as the "current profit" and to update the maximum profit if a higher current profit is found.

Corner and Special Cases to Test When Writing the Code

Test cases should include:

- Increasing and decreasing prices, ensuring profit is calculated correctly or confirmed as zero.
- Large input arrays to verify performance.
- Edge cases with the minimum input size (e.g., an array of just one price).

Explanation of Sliding Window Concept

In this problem, the sliding window is not explicitly represented by two pointers, but the concept is similar. The window dynamically adjusts as we iterate through the `prices` array, always keeping track of the lowest buying price and the highest potential selling price after that buying day. This ensures that each price is considered only once, resulting in a linear time complexity of $O(n)$ ⁵.

⁵ Linear time complexity

Problem 1.3 Product of Array Except Self

The "Product of Array Except Self" problem is a classic problem in the realm of algorithm and data structure problems often encountered on platforms like LeetCode.

Problem Statement

Given an array `nums` of n integers where $n > 1$, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

Given the constraints, it's guaranteed that the product of the array elements is within the range of a 32-bit signed integer.

Examples

Example 1:

Input: [1, 2, 3, 4]

Output: [24, 12, 8, 6]

Explanation: By calculating the product of all elements except for the one at the current index, you would get:

- For index 0: $(2 \times 3 \times 4) = 24$
- For index 1: $(1 \times 3 \times 4) = 12$
- For index 2: $(1 \times 2 \times 4) = 8$
- For index 3: $(1 \times 2 \times 3) = 6$

Note:

- You cannot use division in solving this problem.
- You should try to solve the problem in $O(n)$ time complexity.

Algorithmic Approach

To solve this problem, we utilize two arrays to store the prefix and suffix products of every element. We then multiply these prefix and suffix products to obtain the final output.

Complexities

- **Time Complexity:** The total time complexity of the solution is $O(n)$ as we are iterating through the array a constant number of times.
- **Space Complexity:** The space complexity is $O(n)$ due to the extra space taken up by the prefix and suffix arrays.

Python Implementation

Below is the complete Python code for solving the "Product of Array Except Self" problem without using division and with linear time complexity:

```
class Solution:
    def productExceptSelf(self, nums):
        length = len(nums)

        # Initialize arrays for left and right products
        left_products = [0]*length
        right_products = [0]*length
        output = [0]*length
```

```

# left_products[i] contains the product of all elements to the left of i
left_products[0] = 1
for i in range(1, length):
    left_products[i] = nums[i - 1] * left_products[i - 1]

# right_products[i] contains the product of all elements to the right of i
right_products[length - 1] = 1
for i in reversed(range(length - 1)):
    right_products[i] = nums[i + 1] * right_products[i + 1]

# Construct the output array
for i in range(length):
    output[i] = left_products[i] * right_products[i]

return output

```

By precalculating the product of elements to the left and right of every element, we can construct the output array without the use of division.

Why this approach

This approach is chosen because it adheres to the constraints of not using division and maintains a linear time complexity. By keeping track of the products to the left and right separately, we can multiply these values together to get the desired result for each index.

Alternative approaches

An alternative approach could have been directly calculating the total product and then dividing it by the current element, but the problem explicitly forbids the use of division. Additionally, using extra passes and division might have exceeded the desired linear time complexity.

Similar problems to this one

Similar problems may include other array transformation challenges that involve prefix sums or products, or problems that require the efficient computation of cumulative properties without using direct division, such as "Maximum Product Subarray" or "Trapping Rain Water."

Things to keep in mind and tricks

- Precomputation of cumulative quantities can lead to efficient algorithms.
- When division is not allowed, consider the use of prefix and suffix arrays to maintain state.
- Always consider the constraints and desired time complexity when designing a solution.
- Remember to check for edge cases, such as an array containing zeros or negative numbers.

Corner and special cases to test when writing the code

While testing, consider arrays with:

- A single zero, multiple zeros, and no zeros.
- Negative numbers, as they can affect the product sign.
- Large lengths to ensure that the time complexity is indeed linear.
- Edge cases, such as an empty array or an array with a single element.

Problem 1.4 Maximum Subarray

The **Maximum Subarray** problem is a cornerstone of algorithmic problem-solving, frequently used to introduce concepts like dynamic programming and divide-and-conquer techniques. Its simplicity and depth make it a classic challenge for both beginners and advanced programmers.

Problem Statement

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum, and return its sum⁶.

⁶ A subarray is defined as a contiguous part of the array, meaning the elements are adjacent and sequential

Example 1:

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

Example 2:

```
Input: nums = [1]
Output: 1
Explanation: The array contains only one element, which is the subarray.
```

Example 3:

```
Input: nums = [-1,-2,-3]
```

Output: -1

Explanation: The largest sum is the single element -1, as all numbers are negative.

Key Observations:

- An array with one element is a valid subarray⁷.
 - Negative values do not inherently prevent a subarray from being optimal; however, in some cases, starting a new subarray may yield better results.

⁷ Single-element arrays must be considered in edge cases

Algorithmic Approach

There are three primary approaches to solving this problem:

1. **Brute Force:** Examine every possible subarray and calculate their sums, maintaining the maximum encountered sum. This approach has $O(n^2)$ to $O(n^3)$ time complexity⁸.
 2. **Divide and Conquer:** Split the array into two halves, recursively find the maximum subarray sum for each half, and compute the maximum sum of a subarray that spans the midpoint. The time complexity is $O(n \log n)$ due to the recursive divisions⁹.
 3. **Dynamic Programming (Kadane's Algorithm):** Iteratively compute the maximum subarray sum ending at each index by comparing:

⁸ Avoid brute force unless explicitly required by constraints, as it is computationally expensive for large arrays

⁹ Divide and conquer provides a clear demonstration of recursive problem-solving but is less efficient than dynamic programming here

```
max_current = max(nums[i], max_current + nums[i])
```

Track the global maximum sum as:

`max_global = max(max_global, max_current)`

This approach has $O(n)$ time complexity and is the most efficient for this problem¹⁰.

¹⁰ Kadane's Algorithm is optimal because it processes the array in a single pass with constant space

Python Implementation

Below is the Python implementation of Kadane's Algorithm:

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        # Initialize current and global maximums to the first element
        max_current = max_global = nums[0]

        # Traverse the array from the second element onward
        for x in nums[1:]:
            max_current = max(x, max_current + x)  # Include current element or
            ↪ start new subarray
            if max_current > max_global:
                max_global = max_current

        return max_global
```

```

max_global = max(max_global, max_current) # Update global maximum if
    ↪ needed

return max_global

```

Explanation:

- The algorithm initializes both `max_current` and `max_global` to the first element of the array¹¹.
- For each element, it determines whether to include it in the current subarray or start a new subarray¹².
- `max_global` is updated whenever a larger subarray sum is encountered.
- The final value of `max_global` is returned as the result.

¹¹ This ensures that single-element arrays are handled naturally

¹² This decision is made using the ‘`max`’ function

Why This Approach?

Kadane’s Algorithm is chosen for its efficiency in both time and space. By maintaining running totals and a global maximum, it avoids the overhead of computing sums for all subarrays or managing recursion.

Alternative Approaches

The divide-and-conquer method is an elegant alternative that divides the problem into smaller subproblems. However, it is less efficient due to its higher time complexity of $O(n \log n)$.

Similar Problems to This One

- Maximum Product Subarray:** Find the subarray with the largest product instead of the largest sum.
- Best Time to Buy and Sell Stock:** Identify the best days to buy and sell stock for maximum profit.
- Longest Increasing Subarray:** Find the longest contiguous subarray with increasing elements.

Things to Keep in Mind and Tricks

- All-Negative Arrays:** When all numbers are negative, the largest sum is the single largest element. Kadane’s Algorithm naturally handles this case¹³.

¹³ No need for additional checks; the algorithm inherently accommodates negative numbers

- **Starting New Subarrays:** The decision to start a new subarray is pivotal. Always compare the current element with the sum of the current element and the existing subarray.
- **Edge Cases:** Consider empty arrays, single-element arrays, and arrays with alternating large positive and negative numbers.

Complexities

- **Time Complexity:** $O(n)$, as the algorithm processes each element exactly once.
- **Space Complexity:** $O(1)$, since it uses only a few variables for tracking sums.

Corner and Special Cases to Test

- **Empty Array:** Confirm the algorithm gracefully handles invalid input or returns a default value¹⁴.
- **Single-Element Array:** Ensure that the output is the element itself.
- **All-Negative Numbers:** Validate that the largest (least negative) number is returned.
- **Mixed Positive and Negative Numbers:** Test with arrays containing both large positive and negative numbers to ensure correct subarray selection.

¹⁴ Some implementations may raise exceptions for empty arrays

Conclusion

The **Maximum Subarray** problem exemplifies the power of dynamic programming in simplifying complex problems. Kadane's Algorithm is the optimal solution, offering both efficiency and elegance. By understanding the nuances of this problem, you can approach similar array challenges with confidence, leveraging dynamic programming concepts to solve them effectively.

Problem 1.5 Two Pointers Technique for Merging Arrays

When you are given two arrays to process, it is common to have one index per array (pointer) to traverse and compare both of them, incrementing one of the pointers when relevant. For example, we use this approach to merge two sorted arrays.

Merge two sorted arrays using two pointers.

Two Pointers Technique for Merging Arrays

When you are given two arrays to process, it is common to have one index per array (pointer) to traverse and compare both of them, incrementing one of the pointers

when relevant. For example, we use this approach to merge two sorted arrays.

1.9.1 Merge Sorted Array

The "Merge Sorted Array" problem is a common algorithmic challenge that focuses on efficiently merging two sorted arrays. The task is to merge the contents of `nums2` into `nums1`, ensuring that `nums1` remains sorted in non-decreasing order. This problem tests one's ability to manipulate arrays in-place while maintaining the integrity of the original data.

Problem Statement

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 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 .

Constraints:

- `nums1.length = m + n`
- `nums2.length = n`
- $0 \leq m, n \leq 200$
- $1 \leq m + n \leq 200$
- $-10^9 \leq \text{nums1}[i], \text{nums2}[i] \leq 10^9$
- `nums1` and `nums2` are sorted in non-decreasing order.

Input: Two sorted integer arrays `nums1` and `nums2`, and integers m and n .

Output: The merged and sorted array stored within `nums1`.

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]`

Example 2:

Input: `nums1 = [1], m = 1, nums2 = [], n = 0`
 Output: `[1]`

Example 3:

Input: `nums1 = [0], m = 0, nums2 = [1], n = 1`
 Output: `[1]`

Explanation: Since $m = 0$, there are no elements in `nums1`. The merged array is `[1]`.

Algorithmic Approach

To merge these two arrays efficiently, the two pointers technique is ideal. Instead of merging the arrays from the start, which could require extra space or unnecessary element shifts, we can start from the end of `nums1` and move backwards. This approach ensures that we overwrite the trailing zeroes in `nums1` while comparing the largest elements from both arrays.

Here's how the approach works:

- Initialize two pointers `p1` and `p2` to point at the last elements of the valid parts of `nums1` and `nums2`, respectively (i.e., `p1 = m - 1, p2 = n - 1`).
- Another pointer `p` starts at the last position of the combined array ($m + n - 1$) in `nums1`.
- Compare the elements at `p1` and `p2`. Place the larger element at position `p` in `nums1` and move the respective pointer.
- Decrement the `p` pointer and repeat until all elements are merged.
- If any elements remain in `nums2`, copy them over to `nums1`.

Detailed Walkthrough

Consider the example:

```
nums1 = [1,2,3,0,0,0], m = 3
nums2 = [2,5,6], n = 3
```

1. Set `p1 = 2` (points to 3 in `nums1`), `p2 = 2` (points to 6 in `nums2`), `p = 5`.
2. Compare `nums1[p1]` (3) and `nums2[p2]` (6). Since $6 > 3$, set `nums1[p] = 6`, decrement `p2` to 1, `p` to 4.
3. Compare `nums1[p1]` (3) and `nums2[p2]` (5). Since $5 > 3$, set `nums1[p] = 5`, decrement `p2` to 0, `p` to 3.

4. Compare `nums1[p1]` (3) and `nums2[p2]` (2). Since $3 > 2$, set `nums1[p] = 3`, decrement `p1` to 1, `p` to 2.
5. Compare `nums1[p1]` (2) and `nums2[p2]` (2). Since $2 == 2$, set `nums1[p] = 2`, decrement `p2` to -1, `p` to 1.
6. Since $p2 < 0$, copy remaining elements from `nums1` (if any). Here, set `nums1[p] = nums1[p1]` (2), decrement `p1` to 0, `p` to 0.
7. Set `nums1[p] = nums1[p1]` (1).

Final merged array: [1, 2, 2, 3, 5, 6].

Alternative Approaches

An alternative approach is to create a new array and merge `nums1` and `nums2` from the start, similar to the merge step in the Merge Sort algorithm. However, this approach requires additional space of $O(m + n)$ and extra work to copy back the merged array into `nums1`. The reverse two-pointer technique is more efficient in terms of space and time since it operates in-place and avoids shifting elements multiple times.

Complexities

- **Time Complexity:** The time complexity is $O(m + n)$ because each element in `nums1` and `nums2` is processed once. We iterate through both arrays starting from their ends and move backwards, ensuring that all elements are compared and placed correctly.
- **Space Complexity:** The space complexity is $O(1)$ since the merging is done in-place within `nums1`. We do not use any additional significant space that scales with the input size.

Python Implementation

Below is the Python code to implement the "Merge Sorted Array" problem using the two pointers technique:

```
from typing import List

def merge(nums1: List[int], m: int, nums2: List[int], n: int) -> None:
    """
    Merges nums2 into nums1 in-place, resulting in a single sorted array.

    Parameters:
        nums1 (List[int]): The target list where the merge will be performed.
        m (int): The number of elements present in nums1 before the merge.
        nums2 (List[int]): The list being merged into nums1.
        n (int): The number of elements present in nums2 before the merge.
    """
    # Initialize pointers for both arrays
    p1, p2, p = m - 1, n - 1, m + n - 1

    while p2 >= 0:
        if p1 < 0 or nums1[p1] < nums2[p2]:
            nums1[p] = nums2[p2]
            p2 -= 1
        else:
            nums1[p] = nums1[p1]
            p1 -= 1
        p -= 1
```

```

nums1 (List[int]): The first sorted array with a length of m + n,
where the first m elements denote the elements to merge,
and the last n elements are set to 0 and should be ignored.
m (int): Number of initialized elements in nums1.
nums2 (List[int]): The second sorted array.
n (int): Number of initialized elements in nums2.

Returns:
None: The function modifies nums1 in-place.
"""

p1, p2, p = m - 1, n - 1, m + n - 1

# While there are elements to compare in nums1 and nums2
while p1 >= 0 and p2 >= 0:
    if nums1[p1] > nums2[p2]:
        nums1[p] = nums1[p1]
        p1 -= 1
    else:
        nums1[p] = nums2[p2]
        p2 -= 1
    p -= 1

# If there are remaining elements in nums2, copy them
while p2 >= 0:
    nums1[p] = nums2[p2]
    p2 -= 1
    p -= 1

```

Example Usage and Test Cases

```

# Test case 1: General case
nums1 = [1,2,3,0,0,0]
m = 3
nums2 = [2,5,6]
n = 3
merge(nums1, m, nums2, n)
print(nums1)  # Output: [1, 2, 2, 3, 5, 6]

# Test case 2: nums2 is empty
nums1 = [1]
m = 1
nums2 = []
n = 0
merge(nums1, m, nums2, n)
print(nums1)  # Output: [1]

# Test case 3: nums1 is empty
nums1 = [0]
m = 0
nums2 = [1]

```

```

n = 1
merge(nums1, m, nums2, n)
print(nums1) # Output: [1]

# Test case 4: Negative numbers
nums1 = [-1,0,0,0]
m = 1
nums2 = [-3,-2,-1]
n = 3
merge(nums1, m, nums2, n)
print(nums1) # Output: [-3, -2, -1, -1]

```

Why This Approach

The reverse two-pointer technique is ideal for this problem because it avoids the need to move elements multiple times. By starting from the end of `nums1`, we place elements directly into their final positions without overwriting any unprocessed elements. This in-place approach is more efficient than merging from the start, which could require shifting elements forward to make space.

Starting from the end of the arrays ensures that we utilize the unused space at the end of `nums1` (the zeroes) to store the largest elements first. This method leverages the fact that we know the total number of elements and the arrays are sorted, allowing us to merge efficiently without extra space.

Similar Problems

Other problems that involve merging sorted data structures or using the two-pointer technique include:

- **Merge Two Sorted Lists:** Merge two sorted linked lists and return it as a new sorted list.
- **Merge k Sorted Lists:** Merge k sorted linked lists and return it as one sorted list.
- **Merge Intervals:** Merge all overlapping intervals in a list of intervals.
- **Two Sum II - Input Array Is Sorted:** Find two numbers such that they add up to a specific target number in a sorted array.

These problems also require careful handling of elements in sorted order, often leveraging two pointers or similar techniques.

Things to Keep in Mind and Tricks

- **Edge Cases:** Always consider edge cases such as empty arrays or arrays with one element. Ensure your algorithm handles these scenarios correctly.
- **Remaining Elements:** After the main loop, if there are remaining elements in `nums2`, they need to be copied over to `nums1`. If there are remaining elements in `nums1`, they are already in place.
- **Avoiding Shifts:** By starting from the end, we avoid shifting elements multiple times, which improves efficiency.
- **Optimization Tip:** If `nums2` is empty (`n == 0`), we can skip the merging process altogether.
- **Common Pitfall:** Do not forget to handle the case where `nums1` has no elements (`m == 0`). In this case, we need to copy all elements from `nums2` to `nums1`.

Exercises

1. **Descending Order Merge:** Modify the algorithm to merge two arrays sorted in non-increasing (descending) order.
2. **Merge Without Extra Space:** Suppose `nums1` has no extra space at the end (i.e., length is `m`). How would you merge `nums2` into `nums1` without using extra space?
3. **Non-Sorted Arrays:** Adapt the algorithm to merge two unsorted arrays into a sorted array without using built-in sorting functions.
4. **Alternative Languages:** Implement the merge function in another programming language, such as Java or C++, to practice language-specific syntax.

Questions for Reflection

- How would the algorithm change if the arrays were not sorted?
- Can this approach be extended to merge more than two arrays? How would you modify it?
- What are the trade-offs between in-place algorithms and those that use extra space?

References

LeetCode Problem: Merge Sorted Array

Problem 1.6 Maximum Product Subarray

The **Maximum Product Subarray** problem is a classic dynamic programming challenge that highlights the importance of tracking both the maximum and minimum values in a sequence. The problem's complexity arises from handling positive, negative, and zero values, which can significantly affect the product of subarrays.

Problem Statement

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest product, and return the product.

Example 1:

Input: `nums = [2,3,-2,4]`

Output: 6

Explanation: The subarray `[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 contiguous subarray.

Key Observations:

- Negative numbers can transform a large negative product into a large positive product when multiplied with another negative number¹⁵.
- Zeros break the continuity of a subarray's product, requiring a reset of the calculation¹⁶.

¹⁵ Tracking both the maximum and minimum products is crucial for this reason

¹⁶ Any subarray containing zero has a product of zero

Algorithmic Approach

The most efficient solution to this problem leverages dynamic programming to maintain:

- `max_product`: The maximum product of a subarray ending at the current index.

- `min_product`: The minimum product of a subarray ending at the current index¹⁷.

¹⁷ Tracking the minimum is essential to handle negative values correctly

At each step, update `max_product` and `min_product` using the current number and the products of the current number with the previous `max_product` and `min_product`. If the current number is negative, swap `max_product` and `min_product` before updating.

Algorithm:

1. Initialize `max_product`, `min_product`, and `result` to the first element of the array.
2. Iterate through the array starting from the second element:
 - If the current number is negative, swap `max_product` and `min_product`.
 - Update `max_product` as:

$$\text{max_product} = \max(\text{nums}[i], \text{max_product} \times \text{nums}[i])$$

- Update `min_product` as:

$$\text{min_product} = \min(\text{nums}[i], \text{min_product} \times \text{nums}[i])$$

- Update `result` as:

$$\text{result} = \max(\text{result}, \text{max_product})$$

3. Return `result`.

Complexities

- **Time Complexity:** $O(n)$, as the array is traversed only once.
- **Space Complexity:** $O(1)$, since only a constant amount of extra space is required.

Python Implementation

```
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        # Initialize max_product, min_product, and result with the first element
        max_product = min_product = result = nums[0]

        # Iterate through nums starting from the second element
        for i in range(1, len(nums)):
            # If the current element is negative, swap max_product and min_product
            if nums[i] < 0:
```

```

    max_product, min_product = min_product, max_product

    # Update max_product and min_product
    max_product = max(nums[i], max_product * nums[i])
    min_product = min(nums[i], min_product * nums[i])

    # Update the global result
    result = max(result, max_product)

return result

```

Why This Approach?

This approach efficiently calculates the maximum product of a subarray by maintaining local maxima and minima at each step, ensuring optimal performance. The $O(n)$ time complexity is achieved by avoiding recalculation of products for all possible subarrays, which would otherwise result in $O(n^2)$ complexity.

Alternative Approaches

- **Brute Force:** Calculate the product of every possible subarray. While straightforward, this approach has $O(n^2)$ time complexity and is impractical for large arrays.
- **Divide and Conquer:** Split the array into halves, recursively find the maximum product in each half, and compute the maximum product across the midpoint. This approach has $O(n \log n)$ time complexity but is less efficient than the dynamic programming solution.

Similar Problems

- **Maximum Subarray Sum:** Use Kadane's Algorithm to find the maximum sum of a contiguous subarray.
- **Circular Subarray Maximum Product:** Extend this problem to handle arrays with wraparound subarrays.

Things to Keep in Mind and Tricks

- **Negative Numbers:** Always track both maximum and minimum products to handle cases where negative values become positive when multiplied.
- **Zeros:** A zero resets the product, so consider restarting the calculation from the next element after a zero.

- **Edge Cases:** Test arrays with single elements, all positive numbers, all negative numbers, and arrays with zeros.

Corner and Special Cases to Test

- Arrays with one element ([3]): The product is the element itself.
- Arrays with all negative numbers ([-1, -2, -3]): The maximum product is the product of all elements if the count of negatives is even.
- Arrays with zeros ([0, -2, -3, 0, 4]): Ensure the algorithm handles resets correctly.
- Mixed positive and negative numbers ([2, 3, -2, 4, -1]): Check transitions between positive and negative subarrays.

Conclusion

The **Maximum Product Subarray** problem is a classic example of dynamic programming's power in handling complex array-based problems. By maintaining local maxima and minima, this approach elegantly solves the problem in linear time, ensuring efficiency even for large inputs. Mastering this problem builds a strong foundation for tackling similar challenges involving subarrays and dynamic optimization.

Problem 1.7 Find Minimum in Rotated Sorted Array

The **Find Minimum in Rotated Sorted Array** problem involves identifying the smallest element in a rotated sorted array. This problem demonstrates the efficient application of binary search techniques to leverage the sorted structure of the array while addressing the rotation.

—

Problem Statement

Given a rotated sorted array, locate its minimum element. A sorted array is considered rotated if some elements from the beginning are moved to the end, maintaining the overall ascending order. The solution should run in $O(\log n)$ time complexity.

—

Input: - `nums`: A list of integers representing the rotated sorted array.

Output: - An integer representing the minimum element in `nums`.

Example 1:

Input: `nums = [3, 4, 5, 1, 2]`
 Output: 1
 Explanation: The minimum value is 1.

Example 2:

Input: `nums = [4, 5, 6, 7, 0, 1, 2]`
 Output: 0
 Explanation: The minimum value is 0.

Example 3:

Input: `nums = [11, 13, 15, 17]`
 Output: 11
 Explanation: The array is not rotated, so the first element is the smallest.

—

Algorithmic Approach

To solve the problem efficiently: 1. Use binary search with two pointers, `left` and `right`, representing the bounds of the current search space. 2. Compute the middle index:

$$\text{mid} = \text{left} + (\text{right} - \text{left}) // 2$$

3. Compare `nums[mid]` with `nums[right]`: - If `nums[mid] > nums[right]`, the minimum must be in the right half. Update `left = mid + 1`. - Otherwise, the minimum is in the left half (including `mid`). Update `right = mid`. 4. Continue until `left == right`, at which point the minimum element is found at `nums[left]`.

—

Complexities

- **Time Complexity:** $O(\log n)$, since the search space is halved at each step.
- **Space Complexity:** $O(1)$, as no additional space is used beyond a few variables.

—

Python Implementation

```

class Solution:
    def findMin(self, nums: List[int]) -> int:
        left, right = 0, len(nums) - 1
        while left < right:
            mid = left + (right - left) // 2
            if nums[mid] > nums[right]:
                left = mid + 1
            else:
                right = mid
        return nums[left]

# Example usage:
nums = [4, 5, 6, 7, 0, 1, 2]
solution = Solution()
print(solution.findMin(nums))  # Output: 0

```

—

Why This Approach?

Binary search is ideal for this problem because the array is partially sorted. Instead of iterating through all elements ($O(n)$), binary search exploits the sorted structure to locate the minimum in $O(\log n)$ time.

—

Alternative Approaches

1. **Linear Search ($O(n)$):** Scan all elements to find the minimum. This approach is straightforward but inefficient for large arrays.
2. **Recursive Binary Search ($O(\log n)$):** The binary search logic can be implemented recursively, although it adds stack overhead.

—

Similar Problems

- **Search in Rotated Sorted Array:** Find the position of a target element in a rotated sorted array.
- **Find Peak Element:** Locate a peak element in an array.
- **Find Minimum in Rotated Sorted Array II:** Handles duplicates in the rotated array.

Corner Cases to Test

- Array is not rotated: `nums = [1, 2, 3, 4, 5]`.
 - Single element array: `nums = [1]`.
 - Array with two elements: `nums = [2, 1]`.
 - Minimum element is the last element: `nums = [3, 4, 5, 6, 1]`.
-

Conclusion

The "Find Minimum in Rotated Sorted Array" problem elegantly demonstrates the power of binary search in structured datasets. By efficiently narrowing down the search space, we achieve optimal performance while maintaining simplicity in implementation.

Problem 1.8 Search in Rotated Sorted Array

The **Search in Rotated Sorted Array** problem tests your ability to combine binary search with logic to handle shifted or rotated arrays. This is a frequent interview question that evaluates both problem-solving skills and the ability to optimize search operations.

Problem Statement

You are given a rotated sorted array `nums` of unique integers and a target integer `target`. Return the index of `target` if it exists in `nums`; otherwise, return `-1`.

The array is sorted in ascending order and then rotated at some unknown pivot index k . For example, $[0, 1, 2, 4, 5, 6, 7]$ might become $[4, 5, 6, 7, 0, 1, 2]$. Your task is to search for the `target` in $O(\log n)$ time.

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`

Algorithmic Approach

The problem is best solved using a modified binary search¹⁸.

¹⁸ Binary search ensures $O(\log n)$ time complexity, making it highly efficient for sorted or partially sorted arrays

Key Observations:

- One half of the array (either left or right) is always sorted¹⁹.
- The target can only lie in one of the two halves, depending on its value relative to the sorted half.

¹⁹ This property is a direct result of the array being rotated at a single pivot point

The algorithm:

1. Initialize two pointers, `left` at the start and `right` at the end of the array.

2. While `left ≤ right`, calculate the middle index:

$$\text{mid} = \text{left} + \frac{\text{right} - \text{left}}{2}$$

3. Check if the middle element is the target. If yes, return `mid`.

4. Determine whether the left half or the right half is sorted:

- If the left half is sorted:
 - Check if the target lies within this range²⁰.
 - If yes, adjust the `right` pointer to `mid - 1`; otherwise, adjust `left` to `mid + 1`.
- If the right half is sorted:
 - Check if the target lies within this range²¹.
 - If yes, adjust the `left` pointer to `mid + 1`; otherwise, adjust `right` to `mid - 1`.

²⁰ A sorted range is defined as `nums[left] ≤ target < nums[mid]`

²¹ Similarly, the range is defined as `nums[mid] < target ≤ nums[right]`

5. If the loop exits without finding the target, return `-1`.

Complexities

- **Time Complexity:** $O(\log n)$, as the array is halved in each iteration.
- **Space Complexity:** $O(1)$, as the algorithm operates in-place without additional memory allocation.

Python Implementation

```

class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1

        while left <= right:
            mid = left + (right - left) // 2

            # Check if the middle element is the target
            if nums[mid] == target:
                return mid

            # Determine which half is sorted
            if nums[left] <= nums[mid]:
                # Left half is sorted
                if nums[left] <= target < nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            else:
                # Right half is sorted
                if nums[mid] < target <= nums[right]:
                    left = mid + 1
                else:
                    right = mid - 1

        return -1

```

Why This Approach?

This approach leverages the sorted property of one half of the array in each iteration to eliminate half of the search space²². By dynamically adjusting the search range based on the target's position relative to the sorted half, the algorithm maintains $O(\log n)$ efficiency.

²² Binary search guarantees logarithmic time complexity by halving the search space repeatedly

Alternative Approaches

- **Linear Search:** Iterate through the array and check each element. This has $O(n)$ time complexity and is not suitable for large arrays.
- **Pivot Detection + Binary Search:** First, identify the pivot point (where the rotation occurs) using binary search, then perform binary search on the relevant segment. While still $O(\log n)$, this involves two binary search passes, making it less efficient.

Similar Problems

- **Find Minimum in Rotated Sorted Array:** Identify the smallest element in a rotated sorted array.
- **Search in Rotated Sorted Array II:** Handle duplicates while searching for the target.
- **Find Peak Element:** Locate a peak element in an array where adjacent elements are distinct.

Things to Keep in Mind and Tricks

- Always check if the middle element is the target before further processing²³.
- Handle edge cases like arrays with a single element, no rotation, or extreme rotations (pivot at the first or last element).
- Use integer division (`//`) for calculating `mid` to avoid potential overflow in some languages.

²³ This prevents unnecessary checks and guarantees correctness

Corner and Special Cases to Test

- **Single Element Array:** [1], target = 1 or 2.
- **No Rotation:** [1, 2, 3, 4], target = 3.
- **Full Rotation:** [1, 2, 3, 4], rotated back to [1, 2, 3, 4], target = 4.
- **Target Not in Array:** [4, 5, 6, 7, 0, 1, 2], target = 8.
- **Extreme Rotation:** [2, 3, 4, 5, 6, 7, 0, 1], target = 0.

Conclusion

The **Search in Rotated Sorted Array** problem demonstrates how binary search can be adapted to handle more complex scenarios like rotations. By exploiting the partially sorted structure of the array, this algorithm efficiently narrows down the search space, achieving optimal performance. Mastering this problem prepares you to tackle related challenges involving rotated or partially sorted data structures.

Problem 1.9 Merge Two Sorted Lists

The **Merge Two Sorted Lists** problem involves combining two sorted linked lists into a single, sorted linked list. The objective is to construct a new list by splicing together the nodes of the input lists in ascending order.

Efficiently combining two sorted lists is fundamental in data processing and algorithm design.

Commonly encountered in technical interviews and serves as a building block for more complex data structures.

Problem Statement

Given the heads of two sorted linked lists, `list1` and `list2`, merge them into one sorted linked list. The merged linked list should be composed by splicing together the nodes of the first two lists so that the resulting list is in ascending order.

[LeetCode Link]

[GeeksForGeeks Link]

[HackerRank Link]

[CodeSignal Link]

[InterviewBit Link]

[Educative Link]

[Codewars Link]

Algorithmic Approach

To efficiently merge two sorted lists, the **two-pointer technique** is utilized. This method allows simultaneous traversal of both lists, ensuring that the merged list maintains sorted order without the need for additional sorting.

1. Initialize Pointers:

- Create a dummy node to serve as the starting point of the merged list.
- Initialize a current pointer to track the end of the merged list.

The two-pointer technique is versatile and widely used in array and list manipulation problems.

2. Traverse and Compare:

- While neither `list1` nor `list2` is exhausted:
 - Compare the current nodes of both lists.
 - Append the node with the smaller value to the merged list.
 - Move the corresponding pointer forward.

3. Handle Remaining Elements:

- After one list is exhausted, append the remaining nodes from the other list to the merged list.

Complexities

- **Time Complexity:** $O(n + m)$, where n and m are the lengths of `list1` and `list2`, respectively. Each node is visited exactly once.
- **Space Complexity:** $O(1)$, as the merge is performed in-place without allocating additional space for the merged list.

Python Implementation

Below is the complete Python code for merging two sorted linked lists using the two-pointer technique:

Implementing the two-pointer technique ensures efficient runtime and optimal space usage.

```

# 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: ListNode, list2: ListNode) -> ListNode:
        # Create a dummy node to serve as the start of the merged list
        dummy = ListNode()
        current = dummy

        # Traverse both lists and append the smaller value to the merged list
        while list1 and list2:
            if list1.val < list2.val:
                current.next = list1 # Link to list1 node
                list1 = list1.next # Move list1 pointer
            else:
                current.next = list2 # Link to list2 node
                list2 = list2.next # Move list2 pointer
            current = current.next # Move merged list pointer

        # Append any remaining nodes from list1 or list2
        current.next = list1 if list1 else list2

        # Return the head of the merged list, skipping the dummy node
        return dummy.next

# Example Usage:
# Constructing the first sorted linked list: 1 -> 3 -> 5
list1 = ListNode(1, ListNode(3, ListNode(5)))

# Constructing the second sorted linked list: 2 -> 4 -> 6
list2 = ListNode(2, ListNode(4, ListNode(6)))

# Creating a Solution object and merging the lists
solution = Solution()
merged_head = solution.mergeTwoLists(list1, list2)

# Function to print the merged linked list
def print_linked_list(head):
    elems = []
    while head:
        elems.append(str(head.val))
        head = head.next
    print(" -> ".join(elems))

print_linked_list(merged_head) # Output: 1 -> 2 -> 3 -> 4 -> 5 -> 6

# Handling Edge Cases

```

```

print_linked_list(solution.mergeTwoLists(None, ListNode(1, ListNode(2, ListNode(3)
    ↵ )))) # Output: 1 -> 2 -> 3
print_linked_list(solution.mergeTwoLists(ListNode(1, ListNode(2, ListNode(3))),
    ↵ None)) # Output: 1 -> 2 -> 3
print_linked_list(solution.mergeTwoLists(ListNode(1, ListNode(3, ListNode(5))),
    ↵ ListNode(1, ListNode(3, ListNode(5))))) # Output: 1 -> 1 -> 3 -> 3 -> 5 ->
    ↵ 5

```

Explanation

The ‘mergeTwoLists‘ function efficiently merges two sorted linked lists by leveraging the **two-pointer technique**. Here’s a step-by-step breakdown of the implementation:

- **Initialization:**

- A dummy node is created to simplify edge cases, such as when one or both input lists are empty.
- A ‘current‘ pointer is initialized to track the end of the merged list.

- **Merging Process:**

- The function enters a loop that continues until either ‘list1‘ or ‘list2‘ is exhausted.
- Within the loop, the values of the current nodes of both lists are compared.
- The node with the smaller value is appended to the merged list by updating ‘current.next‘.
- The pointer of the list from which the node was taken is moved forward.
- The ‘current‘ pointer is then moved to the newly appended node.

- **Appending Remaining Nodes:**

- After the main loop, one of the lists may still contain nodes.
- The remaining nodes from the non-exhausted list are appended to the merged list.

- **Finalizing the Merged List:**

- The merged list is returned by skipping the dummy node (‘dummy.next‘).

Why This Approach

This method is chosen because it is both intuitive and efficient. By iterating through both lists simultaneously and always selecting the smaller current element, we ensure that the merged list remains sorted. Additionally, since the merge is done in-place, it optimizes space usage without the need for additional data structures.

In-place operations are crucial for optimizing memory usage, especially with large datasets.

Alternative Approaches

An alternative method involves using recursion to merge the two lists. In this recursive approach, the function compares the heads of both lists and recursively merges the remaining elements. While elegant, the recursive method may lead to increased space usage due to the call stack, especially with very long lists.

Recursive solutions can be more readable but may not always be the most space-efficient.

Similar Problems to This One

- Merge k Sorted Lists
- Merge Intervals
- Merge Sorted Arrays

Things to Keep in Mind and Tricks

- **Edge Cases:** Always consider scenarios where one or both input lists are empty.
- **Duplicate Elements:** Decide how to handle duplicate values; in this case, duplicates are allowed and included in the merged list.
- **In-Place Merging:** If allowed, modifying one of the input lists can save space.
- **Using Built-in Functions:** Python's built-in functions like `sorted()` can simplify merging but may not be as efficient for large lists.

Corner and Special Cases to Test When Writing the Code

- **Both Lists Empty:** `list1 = []`, `list2 = []`
- **One List Empty:** `list1 = []`, `list2 = [1, 2, 3]` and `list1 = [1, 2, 3]`, `list2 = []`
- **Different Lengths:** `list1 = [1, 4, 5]`, `list2 = [2, 3]`
- **All Elements from One List Smaller:** `list1 = [1, 2, 3]`, `list2 = [4, 5, 6]`
- **All Elements from One List Larger:** `list1 = [4, 5, 6]`, `list2 = [1, 2, 3]`
- **Duplicate Elements:** `list1 = [1, 3, 5]`, `list2 = [1, 3, 5]`

Problem 1.10 Remove Duplicates from a Sorted Array

Introduction

The problem of removing duplicates from a sorted array is a classic algorithmic challenge that emphasizes in-place array manipulation and efficient use of space. Leveraging the sorted nature of the array allows for optimal solutions that minimize time and space complexity. This problem not only reinforces fundamental array handling techniques but also serves as a foundational exercise for understanding more complex data manipulation tasks²⁴.

²⁴ For a comprehensive overview, refer to the LeetCode Remove Duplicates problem.

Problem Statement

Given a sorted array of integers, remove the duplicates in-place such that each element appears only once and return the new length. The relative order of the elements should be kept the same. Since it is impossible to change the length of the array in some programming languages, you must instead have the result placed in the first part of the array. More formally, if there are k elements after removing the duplicates, then the first k elements of the array should hold the final result. It does not matter what you leave beyond the first k elements.

Algorithmic Approach

The most efficient way to solve this problem is by using the **Two Pointers Technique**. Here's a step-by-step approach:

1. Initialize Two Pointers:

- **slow** pointer starts at index 0, representing the position of the last unique element found²⁵.
- **fast** pointer starts at index 1, traversing the array to find unique elements²⁶.

²⁵ This pointer tracks the position where the next unique element should be placed.

²⁶ The **fast** pointer scans through the array to identify unique elements.

2. Traverse the Array:

- While **fast** is less than the length of the array:
 - If the element at **fast** is not equal to the element at **slow**, it means a new unique element is found²⁷.
 - Increment **slow** and update the element at **slow** with the element at **fast**²⁸.
- Increment **fast** to continue traversing.

²⁷ Since the array is sorted, duplicates are adjacent, making it easy to detect new unique elements.

²⁸ This effectively moves the unique element to the front of the array.

3. Completion:

4. After traversal, the value of **slow** + 1 will represent the number of unique elements.

Python Implementation

```
def remove_duplicates(nums):
    """
    Removes duplicates from a sorted array in-place.

    Parameters:
    nums (List[int]): The input sorted array of integers.

    Returns:
    int: The number of unique elements after removing duplicates.
    """

    if not nums:
        return 0

    slow = 0
    for fast in range(1, len(nums)):
        if nums[fast] != nums[slow]:
            slow += 1
            nums[slow] = nums[fast]

    return slow + 1

# Example usage:
nums = [1, 1, 2]
k = remove_duplicates(nums)
print(k)          # Output: 2
print(nums[:k])  # Output: [1, 2]
```

Example Usage and Test Cases

```
# Test case 1: General case with duplicates
nums = [1, 1, 2]
k = remove_duplicates(nums)
print(k)          # Output: 2
print(nums[:k])  # Output: [1, 2]

# Test case 2: All elements are duplicates
nums = [2, 2, 2, 2]
k = remove_duplicates(nums)
print(k)          # Output: 1
print(nums[:k])  # Output: [2]

# Test case 3: No duplicates
nums = [1, 2, 3, 4, 5]
k = remove_duplicates(nums)
print(k)          # Output: 5
print(nums[:k])  # Output: [1, 2, 3, 4, 5]
```

```

# Test case 4: Empty array
nums = []
k = remove_duplicates(nums)
print(k)          # Output: 0
print(nums[:k])  # Output: []

# Test case 5: Single element array
nums = [1]
k = remove_duplicates(nums)
print(k)          # Output: 1
print(nums[:k])  # Output: [1]

```

Why This Approach

The **Two Pointers Technique** is particularly effective for this problem due to the following reasons:

- **In-Place Modification:** Eliminates the need for additional memory by modifying the array directly²⁹.
- **Linear Time Complexity:** Traverses the array only once, achieving $O(n)$ time complexity³⁰.
- **Utilizes Sorted Property:** The sorted nature of the array ensures that duplicates are adjacent, simplifying the detection and removal process³¹.
- **Simplicity:** The algorithm is straightforward, making it easy to implement and understand³².

²⁹ This is crucial for optimizing space usage, especially with large datasets.

³⁰ This ensures that the algorithm remains efficient even as the size of the input grows.

³¹ Sorting guarantees that all duplicates are clustered together, making it easier to identify unique elements.

³² Clear and concise logic reduces the likelihood of errors during implementation.

Complexity Analysis

- **Time Complexity:** $O(n)$, where n is the number of elements in the array. Each element is visited at most once³³.
- **Space Complexity:** $O(1)$, as the algorithm uses a constant amount of extra space³⁴.

³³ The **fast** pointer ensures a single pass through the array.

³⁴ No additional data structures are required beyond the input array itself.

Similar Problems

Other problems that can be efficiently solved using the two pointers technique include:

- **Two Sum II - Input Array Is Sorted:** Find two numbers that add up to a specific target number³⁵.

³⁵ This problem leverages the sorted property to efficiently locate the desired pair.

- **3Sum Problem:** Find all unique triplets in the array which give the sum of zero³⁶.
- **Container With Most Water:** Find two lines that together with the x-axis form a container that holds the most water³⁷.
- **Reverse a String or Array In-Place:** Reverse the elements by swapping from both ends³⁸.

³⁶ Extending the two pointers approach to handle three elements.

³⁷ Maximizing the area between two pointers.

³⁸ A fundamental application of the two pointers technique.

Things to Keep in Mind and Tricks

- **Sorted vs. Unsorted Arrays:** This approach relies on the array being sorted³⁹.
- **In-Place Modification Constraints:** Ensure that the environment or language allows in-place modifications of the data structure⁴⁰.
- **Handling Edge Cases:** Always consider edge cases such as empty arrays, single-element arrays, and arrays with all duplicates⁴¹.
- **Pointer Initialization:** Correctly initialize the pointers to avoid index out-of-bound errors⁴².
- **Understanding Return Values:** Depending on the problem's requirements, ensure that you return the correct value representing the number of unique elements⁴³.

³⁹ If the array is unsorted, consider sorting it first if the problem allows.

⁴⁰ Some languages may have immutable data structures, requiring alternative approaches.

⁴¹ Robustness against various input scenarios is crucial for algorithm reliability.

⁴² Proper starting points ensure that the algorithm functions as intended.

⁴³ Clarify what the function is expected to return to meet the problem's specifications.

Related Problems

1. **Two Sum II - Input Array Is Sorted:** Given a 1-indexed array of integers that is already sorted in non-decreasing order, find two numbers such that they add up to a specific target number⁴⁴.
2. **3Sum Problem:** Given an array $nums$ of n integers, are there elements a, b, c in $nums$ such that $a + b + c = 0$? Find all unique triplets in the array which gives the sum of zero⁴⁵.
3. **Move Zeroes:** Given an array $nums$, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements⁴⁶.
4. **Valid Palindrome II:** Given a string, determine if it can become a palindrome by removing at most one character⁴⁷.
5. **Minimum Size Subarray Sum:** Given an array of positive integers and a positive integer s , find the minimal length of a contiguous subarray for which the sum is at least s . If there isn't one, return 0 instead⁴⁸.

⁴⁴ Leverage the two pointers technique for an efficient solution.

⁴⁵ This problem extends the two pointers approach to three elements.

⁴⁶ This requires careful pointer management to preserve element order.

⁴⁷ Combining two pointers with conditional checks.

⁴⁸ This problem can be approached using a sliding window technique, often used alongside two pointers.

Questions for Reflection

- How does the two pointers technique optimize space compared to using additional data structures like hash sets?⁴⁹.
- In what scenarios might the two pointers technique not be applicable or efficient?⁵⁰.
- How can the two pointers technique be extended to handle more complex problems involving multiple conditions?⁵¹.
- Can the two pointers technique be combined with other algorithmic strategies, such as binary search or dynamic programming, to solve advanced problems?⁵².

⁴⁹ Consider scenarios where space complexity is a critical factor.

⁵⁰ Evaluate the limitations based on data structure properties.

⁵¹ Think about integrating additional logic or combining with other techniques.

⁵² Exploring hybrid approaches for enhanced problem-solving.

References

LeetCode Problem: ⁵³

⁵³ Remove Duplicates from Sorted Array

GeeksforGeeks Article: ⁵⁴

⁵⁴ Remove Duplicates from an Unsorted Linked List

HackerRank Problem: ⁵⁵

⁵⁵ Remove Duplicates

Tutorialspoint Article: ⁵⁶

⁵⁶ Python List Remove Duplicates

Conclusion

Removing duplicates from a sorted array is a fundamental problem that exemplifies the power of the Two Pointers Technique in optimizing both time and space complexities. By intelligently traversing the array with two pointers, the algorithm efficiently eliminates redundant elements without the need for extra storage⁵⁷. Mastery of this technique not only aids in solving similar array manipulation problems but also lays the groundwork for tackling more intricate algorithmic challenges in the realm of data structures and computational problem-solving.

⁵⁷ This approach is not only efficient but also elegant in its simplicity.

Problem 1.11 Rotate Image

The **Rotate Image** problem involves rotating a given $n \times n$ 2D matrix representing an image by 90 degrees (clockwise) **in-place**.

Rotating images is a common task in computer graphics and image processing.

Problem Statement

You are given an $n \times n$ 2D matrix representing an image, rotate the image by 90 degrees (clockwise).

This problem tests your ability to manipulate 2D arrays and understand matrix transformations.

****Note:**** 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.

58 59 60 61 62 63 64

⁵⁸ [LeetCode Link]

⁵⁹ [GeeksForGeeks Link]

⁶⁰ [HackerRank Link]

⁶¹ [CodeSignal Link]

⁶² [InterviewBit Link]

⁶³ [Educative Link]

⁶⁴ [Codewars Link]

Algorithmic Approach

To rotate the image by 90 degrees clockwise **“in-place”**, follow these steps:

1. Transpose the Matrix:

- Swap elements across the diagonal.
- This converts rows to columns.

2. Reverse Each Row:

- Reverse the elements in each row.
- This completes the 90-degree rotation.

Complexities

- **Time Complexity:** $O(n^2)$, where n is the number of rows or columns in the matrix. Each element is visited twice.
- **Space Complexity:** $O(1)$, as the rotation is performed in-place without using additional memory.

Python Implementation

Below is the complete Python code for rotating a matrix by 90 degrees clockwise using the in-place approach:

In-place operations optimize space usage, crucial for large matrices.

```
class Solution:
    def rotate(self, matrix: List[List[int]]) -> None:
        """
        Do not return anything, modify matrix in-place instead.
        """
        n = len(matrix)

        # Transpose the matrix
        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
for i in range(n):
    matrix[i].reverse()

# Example Usage:
solution = Solution()
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
solution.rotate(matrix)
print(matrix) # Output: [[7,4,1], [8,5,2], [9,6,3]]

# Handling Edge Cases:
# Empty Matrix
matrix = []
solution.rotate(matrix)
print(matrix) # Output: []

# Single Element Matrix
matrix = [[1]]
solution.rotate(matrix)
print(matrix) # Output: [[1]]

```

Explanation

The ‘rotate’ function performs a 90-degree clockwise rotation of the given $n \times n$ matrix **in-place**. Here’s a detailed breakdown:

- **Transposing the Matrix:**

- Iterate through each element above the diagonal (where $j \geq i$).
- Swap the elements $matrix[i][j]$ and $matrix[j][i]$.
- This converts the matrix’s rows into columns.

- **Reversing Each Row:**

- After transposition, each row of the matrix represents a column of the original matrix.
- Reverse each row to achieve the 90-degree rotation.

- **Final Result:**

- The matrix is now rotated by 90 degrees clockwise without using any additional space.

Why This Approach

This method is chosen because it:

- **Optimizes Space:** Performs the rotation in-place, requiring no extra memory.
- **Simplicity:** Breaks down the rotation into two clear, manageable steps—transposition and row reversal.
- **Efficiency:** Achieves the desired rotation with a time complexity of $O(n^2)$, which is optimal for this problem.

In-place algorithms are essential for applications where memory usage is a constraint.

Alternative Approaches

An alternative method involves rotating the matrix layer by layer, swapping elements in groups of four. While this approach also achieves an in-place rotation, it can be more complex to implement compared to the transpose-and-reverse method.

Layer-by-layer rotation involves more intricate index management but avoids transposition.

Similar Problems to This One

- Rotate Image Counter-Clockwise
- Spiral Matrix
- Set Matrix Zeros

Things to Keep in Mind and Tricks

- **Matrix Dimensions:** Ensure that the matrix is square ($n \times n$) before applying this in-place rotation method.
- **In-Place Operations:** Modifying the matrix directly helps in saving space but requires careful index management to avoid errors.
- **Edge Cases:** Handle special cases like empty matrices or single-element matrices to prevent runtime errors.
- **Using Python's 'reverse()':** Leveraging built-in functions like 'reverse()' can simplify the code and improve readability.

Corner and Special Cases to Test When Writing the Code

- **Empty Matrix:** `matrix = []`

- **Single Element Matrix:** `matrix = [[1]]`
- **Non-Square Matrices:** Ensure that the algorithm is only applied to square matrices, or modify it to handle non-square matrices appropriately.
- **Already Rotated Matrices:** Test matrices that are already rotated to ensure idempotency if applicable.
- **Large Matrices:** Verify performance and correctness with large $n \times n$ matrices.
- **Matrices with Duplicate Elements:** Ensure that duplicate values are correctly handled during rotation.

Problem 1.12 Spiral Matrix

The **Spiral Matrix** problem requires traversing a given matrix in a spiral order. This task demonstrates techniques for systematic traversal of 2D arrays and highlights careful boundary management during iteration.

Problem Statement

Given an $m \times n$ matrix, return all elements of the matrix in spiral order.

Input: - `matrix`: A list of lists representing an $m \times n$ integer matrix.

Output: - A list of integers containing the elements of `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]
```

Algorithmic Approach

The spiral order traversal follows these steps: 1. Start at the top-left corner and move right until reaching the boundary. 2. Move downward along the rightmost column. 3. Move left along the bottom row (if it hasn't been visited). 4. Move upward along the leftmost column (if it hasn't been visited). 5. Repeat the above steps while adjusting the boundaries for each traversal.

Complexities

1. **Time Complexity:** $O(m \times n)$, as every element is visited exactly once.
 2. **Space Complexity:** $O(1)$, excluding the space required for the output list.
-

Python Implementation

```
def spiralOrder(matrix):
    if not matrix or not matrix[0]:
        return []

    result = []
    top, bottom = 0, len(matrix) - 1
    left, right = 0, len(matrix[0]) - 1

    while top <= bottom and left <= right:
        # Traverse from left to right along the top row
        for col in range(left, right + 1):
            result.append(matrix[top][col])
        top += 1

        # Traverse from top to bottom along the right column
        for row in range(top, bottom + 1):
            result.append(matrix[row][right])
        right -= 1

        if top <= bottom:
            # Traverse from right to left along the bottom row
            for col in range(right, left - 1, -1):
                result.append(matrix[bottom][col])
            bottom -= 1

        if left <= right:
            # Traverse from bottom to top along the left column
            for row in range(bottom, top - 1, -1):
                result.append(matrix[row][left])
            left += 1
```

```

        for col in range(right, left - 1, -1):
            result.append(matrix[bottom][col])
        bottom -= 1

        if left <= right:
            # Traverse from bottom to top along the left column
            for row in range(bottom, top - 1, -1):
                result.append(matrix[row][left])
            left += 1

    return result

# Example usage:
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]
print(spiralOrder(matrix))  # Output: [1, 2, 3, 6, 9, 8, 7, 4, 5]

```

Why This Approach?

The approach ensures that all elements are visited exactly once, while carefully managing the boundaries of traversal. It avoids unnecessary computations and maintains a clear structure for moving in a spiral order.

Alternative Approaches

1. **Simulation-Based Traversal:** - Use a direction vector (e.g., right, down, left, up) to simulate the traversal and adjust boundaries dynamically. - This approach can be easier to extend but involves more bookkeeping.

Similar Problems

1. **Spiral Matrix II:** Generate an $n \times n$ matrix filled with integers from 1 to n^2 in spiral order.
2. **Diagonal Traversal of a Matrix:** Traverse a matrix diagonally.
3. **Matrix Rotation:** Rotate a matrix by 90 degrees clockwise.

Corner Cases to Test

1. Single row matrix: `matrix = [[1, 2, 3, 4]]`.
 2. Single column matrix: `matrix = [[1], [2], [3], [4]]`.
 3. Empty matrix: `matrix = []`.
 4. 1×1 matrix: `matrix = [[1]]`.
 5. Rectangular matrices where $m \neq n$.
-

Conclusion

The **Spiral Matrix** problem highlights the importance of systematic traversal and boundary management in 2D arrays. This solution is efficient, clear, and adaptable for various matrix traversal problems.

Creating and Manipulating Matrices

Creating an Empty $N \times M$ Matrix

For problems involving matrix traversal or dynamic programming, it is often necessary to create an empty matrix of the same size and dimensions as the given matrix. This new matrix is used to store visited states, dynamic programming values, or intermediate results. Familiarizing yourself with routines for creating and manipulating matrices in your programming language is essential.

In Python, an empty $N \times M$ matrix can be created easily in a single line:

```
# Assumes that the matrix is non-empty
zero_matrix = [[0 for _ in range(len(matrix[0]))] for _
    ↵ in range(len(matrix))]
```

Here, `zero_matrix` will be a matrix filled with zeros, matching the dimensions of the input `matrix`.

Copying a Matrix

If you need to create a duplicate of an existing matrix (e.g., to avoid modifying the original), you can use the following approach in Python:

```
copied_matrix = [row[:] for row in matrix]
```

This ensures that `copied_matrix` is a deep copy of the original matrix, meaning that changes to the copy will not affect the original.

Transposing a Matrix

The transpose of a matrix is obtained by interchanging its rows and columns. For a matrix A , the transpose A^T is defined such that:

$$A^T[i][j] = A[j][i]$$

In Python, transposing a matrix can be done succinctly using the `zip()` function:

```
transposed_matrix = list(zip(*matrix))
```

This creates a new matrix where the rows of the original matrix become columns in the transposed version.

Applications of Transposing Matrices

Transposing matrices has practical applications in many grid-based games and problems where you need to verify conditions both horizontally and vertically. For example:

- In **Tic-Tac-Toe**, you can check rows for a winning condition, transpose the matrix, and reuse the same logic to check columns.
- In **Connect 4**, you can efficiently verify horizontal and vertical alignments by leveraging transposition.
- In **Sudoku**, you can use transposition to verify constraints across rows and columns consistently.

By transposing a matrix, vertical cells become horizontal, allowing the reuse of horizontal verification logic for originally vertical conditions.

Example: Transposing a Tic-Tac-Toe Board

Consider a Tic-Tac-Toe board represented as a matrix:

$$\text{board} = \begin{bmatrix} X & O & X \\ O & X & O \\ X & X & O \end{bmatrix}$$

To verify rows and columns for a winning condition: 1. Check rows in the original matrix. 2. Transpose the matrix and reuse the same logic to check columns.

Python Implementation:

```
# Tic-Tac-Toe board
board = [
    ['X', 'O', 'X'],
    ['O', 'X', 'O'],
    ['X', 'X', 'O']
]

# Function to check if any row has the same value
def check_rows(matrix):
    for row in matrix:
        if len(set(row)) == 1 and row[0] != ' ':
            return True
    return False

# Check rows and columns
if check_rows(board) or check_rows(list(zip(*board))):
    print("Winning condition met!")
else:
    print("No winner yet.")
```

Output:

Winning condition met!

Conclusion

Creating, copying, and transposing matrices are fundamental operations in computational problems involving grids or tables. Mastering these techniques allows for efficient solutions to dynamic programming problems, grid-based games, and multidimensional data manipulations. Transposing a matrix, in particular, provides a versatile approach to reuse logic for horizontal and vertical verifications, simplifying implementation for many problems.

Problem 1.13 Set Matrix Zeroes

The **Set Matrix Zeroes** problem requires modifying a given $m \times n$ integer matrix in place such that if any element is zero, the entire row and column containing that element are set to zero. The challenge lies in performing this operation in-place, without using additional space proportional to the size of the matrix.

Problem Statement

Given an $m \times n$ matrix, set the entire row and column of any cell containing a zero to zeros. The transformation must be done in place.

Input: - `matrix`: A list of lists representing the $m \times n$ integer matrix.

Output: - Modify `matrix` directly without returning anything.

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]
]
```

Algorithmic Approach

The naive approach involves creating additional storage to track rows and columns that need to be zeroed, but this violates the in-place requirement. Instead, we use

the matrix itself as storage.

Steps for In-Place Solution

1. Use the first row and first column of the matrix to store information about which rows and columns need to be zeroed.
 2. Traverse the matrix:
 - If `matrix[i][j] = 0`, set the corresponding first-row and first-column elements (`matrix[0][j]` and `matrix[i][0]`) to 0.
 - 3. Update the rest of the matrix using the markers in the first row and column.
 - 4. Finally, handle the first row and column separately to zero them out if necessary.
-

Complexities

1. **Time Complexity:** $O(m \times n)$, as every cell is visited at least once.
 2. **Space Complexity:** $O(1)$, since no additional data structures are used.
-

Python Implementation

```
def setZeroes(matrix):
    rows, cols = len(matrix), len(matrix[0])
    first_row_zero = any(matrix[0][j] == 0 for j in range(cols))
    first_col_zero = any(matrix[i][0] == 0 for i in range(rows))

    # Use first row and column as markers
    for i in range(1, rows):
        for j in range(1, cols):
            if matrix[i][j] == 0:
                matrix[i][0] = 0
                matrix[0][j] = 0

    # Zero out cells based on markers
    for i in range(1, rows):
        for j in range(1, cols):
            if matrix[i][0] == 0 or matrix[0][j] == 0:
                matrix[i][j] = 0

    # Handle first row and first column
    if first_row_zero:
        for j in range(cols):
            matrix[0][j] = 0
    if first_col_zero:
```

```

for i in range(rows):
    matrix[i][0] = 0

# Example usage:
matrix = [
    [0, 1, 2, 0],
    [3, 4, 5, 2],
    [1, 3, 1, 5]
]
setZeroes(matrix)
print(matrix)
# Output: [
#      [0, 0, 0, 0],
#      [0, 4, 5, 0],
#      [0, 3, 1, 0]
# ]

```

Why This Approach?

The in-place strategy is chosen to minimize space complexity while leveraging the matrix itself to store metadata about rows and columns that need to be zeroed. This avoids the need for auxiliary data structures while adhering to the problem constraints.

Alternative Approaches

- **Using Auxiliary Space:** Create two arrays to track rows and columns with zeros. While simpler to implement, this approach uses $O(m + n)$ extra space.
- **Recursive Approach:** Traverse the matrix recursively, zeroing out rows and columns. However, this is inefficient and can lead to stack overflow for large matrices.

Similar Problems

- **Rotate Matrix:** Rotate an $N \times N$ matrix by 90 degrees in place.
- **Spiral Matrix:** Traverse a matrix in a spiral order.
- **Game of Life:** Update the state of a grid based on neighboring cells.

Corner Cases to Test

1. Entire matrix is zeros: `matrix = [[0, 0], [0, 0]]`.
2. Matrix with no zeros: `matrix = [[1, 2], [3, 4]]`.
3. Single row or column with zeros: `matrix = [[0, 1, 2, 3]]` or `matrix = [[1], [0], [3]]`.
4. Large matrices with sparse zeros.

—

Conclusion

The **Set Matrix Zeroes** problem highlights the importance of in-place data manipulation and efficient space usage. By utilizing the matrix itself as metadata storage, we achieve a balance between simplicity and performance, adhering to the problem's constraints.

Chapter 2

Linked Lists

Like arrays, a linked list is a fundamental data structure used to represent sequential data. Unlike arrays, however, linked lists do not require contiguous blocks of memory for storage. Instead, each element (known as a node) contains both data and a reference to the next node in the sequence. This unique design enables linked lists to dynamically grow and shrink, making them more flexible for certain use cases.

What is a Linked List?

A linked list is a linear collection of data elements, known as nodes, connected through links. Each node contains two components:

- **Data:** The value stored in the node.
- **Link:** A pointer or reference to the next node in the sequence.

Example:

```
[Data: 1] -> [Data: 2] -> [Data: 3] -> NULL
```

In this example, each node points to the next node, and the final node points to NULL, indicating the end of the list.

Types of Linked Lists

Linked lists come in various forms, each suited for different applications:

- **Singly Linked List:** Each node has a link to the next node. Traversal is one-way.

- **Doubly Linked List:** Each node has links to both the previous and the next nodes, enabling two-way traversal.
- **Circular Linked List:** The last node links back to the first node, forming a circular structure.

Advantages of Linked Lists

1. **Dynamic Size:** Unlike arrays, linked lists do not require a predefined size. They can grow or shrink dynamically as elements are added or removed. 2. **Efficient Insertion and Deletion:** Adding or removing a node (given its location) has a time complexity of $O(1)$, as no shifting of elements is needed. In arrays, insertion or deletion typically requires shifting all subsequent elements, leading to $O(n)$ complexity. 3. **Memory Utilization:** Linked lists allocate memory as needed, which can be more efficient than arrays that may preallocate excessive space.

Disadvantages of Linked Lists

1. **Linear Access Time:** Accessing an element in a linked list requires traversing the list from the start, resulting in $O(n)$ complexity. Arrays, in contrast, allow direct access via indexing (e.g., `arr[4]`). 2. **Increased Memory Overhead:** Each node requires extra memory to store the reference (or pointer) to the next node. 3. **Cache Unfriendliness:** Since linked list nodes are scattered across memory, they do not take full advantage of caching mechanisms, unlike arrays which store elements contiguously.

Basic Operations on Linked Lists

1. Traversal

To access elements of a linked list, traversal from the head (starting node) to the tail (last node) is necessary.

```
# Node definition
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

# Traversing a linked list
def traverse(head):
    current = head
    while current:
        print(current.data, end=" -> ")
```

```

        current = current.next
        print("NULL")

# Example usage
node1 = Node(1)
node2 = Node(2)
node3 = Node(3)
node1.next = node2
node2.next = node3

traverse(node1)  # Output: 1 -> 2 -> 3 -> NULL

```

2. Insertion

Insertion in a linked list can occur at:

- **The Head:** Add a new node at the beginning.
- **The Tail:** Add a new node at the end.
- **Middle Positions:** Insert a node at a specified position.

3. Deletion

Nodes can be removed from:

- **The Head:** Remove the first node.
- **The Tail:** Remove the last node.
- **Middle Positions:** Remove a node from a specific position.

4. Searching

Searching for a value in a linked list involves traversing the list until the value is found or the end is reached.

Applications of Linked Lists

- **Dynamic Data Structures:** Used to implement stacks, queues, and other abstract data types.
- **Memory-Efficient Data Representation:** Suitable for applications where frequent insertion and deletion of elements are required.
- **Graph Representation:** Adjacency lists for graphs are implemented using linked lists.
- **Undo Functionality:** Linked lists are used to store states for undo/redo operations.

Comparison with Arrays

Feature	**Array**	**Linked List**
Memory Allocation	Fixed (contiguous)	Dynamic (non-contiguous)
Access Time	$O(1)$ (direct access)	$O(n)$ (sequential)
Insertion/Deletion	$O(n)$	$O(1)$ (at head/tail)
Cache Performance	High	Low

Common Problems Involving Linked Lists

1. **Reverse a Linked List:** Reverse the order of nodes in a linked list.
2. **Detect a Cycle in a Linked List:** Check if the linked list has a cycle using Floyd's Tortoise and Hare algorithm.
3. **Merge Two Sorted Lists:** Combine two sorted linked lists into one sorted list.
4. **Remove Nth Node from End:** Remove the n -th node from the end of the list.
5. **Find Intersection:** Determine if two linked lists intersect and find the intersection point.

Conclusion

Linked lists are a versatile and essential data structure for representing sequential data. While they may have some limitations compared to arrays, their dynamic nature and efficient insertion/deletion operations make them a critical tool in computer science and software development. Understanding linked lists is foundational for mastering more advanced data structures and algorithms.

Problem 2.1 Reverse a Linked List

The **Reverse a Linked List** problem involves reversing the order of nodes in a singly linked list so that the head becomes the tail and vice versa. This task demonstrates the manipulation of pointers in a linked list and is a fundamental operation for mastering linked list problems.

—

Problem Statement

Given the head of a singly linked list, reverse the list and return its new head.

—

Input: - `head`: The head node of a singly linked list.

Output: - The head of the reversed linked 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: []

Algorithmic Approach

The solution involves manipulating pointers iteratively or recursively to reverse the direction of the list:

1. **Iterative Approach:** - Use three pointers: `prev`, `current`, and `next`. - Traverse the list while reassigning the `next` pointer of each node to point to its previous node. - Update the head to the last non-NULL node.
 2. **Recursive Approach:** - Reverse the rest of the list recursively. - Adjust the pointers as the recursion unwinds.
-

Complexities

1. **Time Complexity:** $O(n)$, where n is the number of nodes in the linked list.
 2. **Space Complexity:** - Iterative: $O(1)$ (in-place). - Recursive: $O(n)$ (due to recursive call stack).
-

Python Implementation

Iterative Approach

```

class ListNode:
    def __init__(self, value=0, next=None):
        self.val = value
        self.next = next

def reverseList(head):
    prev = None
    current = head

    while current:
        next_node = current.next # Save next node
        current.next = prev # Reverse the link
        prev = current # Move prev forward
        current = next_node # Move current forward

    return prev

# Example usage:
# Creating a linked list: 1 -> 2 -> 3 -> 4 -> 5
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
reversed_head = reverseList(head)

# Printing the reversed linked list: 5 -> 4 -> 3 -> 2 -> 1
current = reversed_head
while current:
    print(current.val, end=" -> ")
print("NULL")

```

Recursive Approach

```

def reverseListRecursive(head):
    # Base case: if the list is empty or only one node, return head
    if not head or not head.next:
        return head

    # Reverse the rest of the list
    reversed_head = reverseListRecursive(head.next)

    # Adjust pointers
    head.next.next = head
    head.next = None

    return reversed_head

```

```

# Example usage:
# Creating a linked list: 1 -> 2 -> 3 -> 4 -> 5
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
reversed_head = reverseListRecursive(head)

# Printing the reversed linked list: 5 -> 4 -> 3 -> 2 -> 1
current = reversed_head
while current:
    print(current.val, end=" -> ")
print("NULL")

```

Why These Approaches?

1. **Iterative:** The iterative solution is efficient in terms of both time and space. It directly manipulates the pointers in place, making it a preferred choice for most use cases.
2. **Recursive:** The recursive solution is elegant and simple, often preferred for its clarity, especially in interviews. However, it uses additional space due to the call stack.

Alternative Approaches

While the above approaches are optimal, another alternative involves using a stack to store nodes during traversal, then reconstructing the list in reversed order. This method, however, has $O(n)$ space complexity.

Similar Problems

1. **Reverse Linked List II:** Reverse a portion of the linked list between two given positions.
2. **Merge Two Sorted Lists:** Combine two sorted linked lists into one.
3. **Rotate List:** Rotate a linked list to the right by k places.

Corner Cases to Test

1. An empty list (`head = NULL`). 2. A list with a single node (`head = [1]`). 3. A list with multiple nodes (`head = [1, 2, 3, 4, 5]`).

Conclusion

Reversing a linked list is a foundational operation in data structures, demonstrating pointer manipulation and recursive thinking. Both iterative and recursive approaches offer efficient solutions, with the choice depending on the context and constraints of the problem.

Problem 2.2 Detect Cycle in a Linked List

The **Detect Cycle in a Linked List** problem involves determining whether a given singly linked list contains a cycle. A cycle occurs when a node's next pointer points to a previous node in the list, creating an infinite loop. This problem is fundamental in understanding linked list manipulations and pointer-based algorithms.

Problem Statement

Given the head of a singly linked list, determine if the linked list has a cycle in it. If there is a cycle, return `true`; otherwise, return `false`.

A cycle in a linked list occurs when a node's next pointer points to a previous node in the list, forming a loop.

Input: - `head`: The head node of a singly linked list.

Output: - A boolean value: `true` if there is a cycle in the linked list, `false` otherwise.

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 second node.

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 first node.

Example 3:

Input: head = [1], pos = -1

Output: false

Explanation: There is no cycle in the linked list.

—

Algorithmic Approach

Detecting a cycle in a linked list can be efficiently achieved using the **Floyd's Tortoise and Hare** algorithm, also known as the two-pointer technique. This approach uses two pointers that traverse the list at different speeds to determine if a cycle exists.

—

Floyd's Tortoise and Hare Algorithm

Key Idea: Use two pointers, `slow` and `fast`. `slow` moves one step at a time, while `fast` moves two steps at a time. If there is no cycle, `fast` will reach the end of the list. If there is a cycle, `fast` will eventually meet `slow` within the cycle.

Steps:

1. Initialize two pointers, `slow` and `fast`, both starting at the head of the linked list.
2. Traverse the linked list:
 - Move `slow` by one step.
 - Move `fast` by two steps.
3. At each step, check if `slow` and `fast` meet:
 - If they meet, a cycle exists; return `true`.
 - If `fast` reaches the end (`NULL`), no cycle exists; return `false`.

—

Alternative Approaches

- **Hash Table (Set) Approach:** Traverse the linked list and store each visited node in a hash table. If a node is revisited, a cycle exists. This method uses $O(n)$ time and $O(n)$ space.
 - **Marking Visited Nodes:** Modify the linked list by marking visited nodes (e.g., by changing node values or using flags). This approach can be risky as it alters the input data and may not be permissible.
-

Complexities

- **Time Complexity:** $O(n)$, where n is the number of nodes in the linked list. Both the hash table and two-pointer approaches traverse the list at most a constant number of times.
 - **Space Complexity:**
 - **Floyd's Tortoise and Hare:** $O(1)$, as it uses only two pointers.
 - **Hash Table Approach:** $O(n)$, due to storing visited nodes.
-

Python Implementation

Floyd's Tortoise and Hare (Two-Pointer) Approach

```
class ListNode:
    def __init__(self, x):
        self.val = x
        self.next = None

def hasCycle(head):
    if not head or not head.next:
        return False

    slow = head
    fast = head.next

    while slow != fast:
        if not fast or not fast.next:
            return False
        slow = slow.next
        fast = fast.next.next
```

```

    return True

# Example usage:
# Creating a cycle: 1 -> 2 -> 3 -> 4 -> 2 ...
node1 = ListNode(1)
node2 = ListNode(2)
node3 = ListNode(3)
node4 = ListNode(4)
node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node2 # Creates a cycle

print(hasCycle(node1)) # Output: True

```

Hash Table (Set) Approach

```

def hasCycleUsingSet(head):
    visited = set()
    current = head
    while current:
        if current in visited:
            return True
        visited.add(current)
        current = current.next
    return False

# Example usage:
# Creating a cycle: 1 -> 2 -> 3 -> 4 -> 2 ...
node1 = ListNode(1)
node2 = ListNode(2)
node3 = ListNode(3)
node4 = ListNode(4)
node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node2 # Creates a cycle

print(hasCycleUsingSet(node1)) # Output: True

```

Why These Approaches?

- **Floyd's Tortoise and Hare:** This approach is optimal in both time and space for detecting cycles in linked lists. It avoids the overhead of additional memory

structures, making it suitable for large datasets.

- **Hash Table Approach:** While not as space-efficient, this method is straightforward and easy to implement. It is useful when modifications to the linked list are not allowed or when simplicity is preferred over optimal space usage.
-

Alternative Approaches

- **Recursive Cycle Detection:** Implement cycle detection using recursion, keeping track of visited nodes through recursive calls. This method is generally not recommended due to the risk of stack overflow with large lists.
 - **Modifying Node Values:** Temporarily alter node values to mark them as visited. This approach is risky as it changes the input data and may not be permissible in certain contexts.
-

Similar Problems

- **Find the Starting Node of the Cycle:** Determine the node where the cycle begins.
 - **Linked List Intersection:** Identify if two linked lists intersect and find the intersection point.
 - **Remove Nth Node from End:** Remove the n -th node from the end of the list.
-

Corner Cases to Test

- **Empty List:** $\text{head} = \text{NULL}$.
 - **Single Node Without Cycle:** $\text{head} = [1]$.
 - **Single Node With Cycle:** $\text{head} = [1] \rightarrow [1]$.
 - **Multiple Nodes Without Cycle:** $\text{head} = [1, 2, 3, 4, 5]$.
 - **Multiple Nodes With Cycle:** $\text{head} = [1, 2, 3, 4, 5] \rightarrow 3$.
-

Conclusion

Detecting cycles in a linked list is a crucial problem that enhances understanding of pointer manipulation and algorithmic efficiency. Floyd's Tortoise and Hare algorithm provides an optimal solution with minimal space overhead, making it the preferred method for this task. Mastery of cycle detection not only aids in solving linked list problems but also contributes to a deeper comprehension of algorithmic design principles.

Problem 2.3 Reorder List

The **Reorder List** problem involves rearranging a given singly linked list in a specific order without altering the node values. The goal is to reorder the list such that it follows the pattern: first node, last node, second node, second last node, and so on.

Reordering linked lists is a common task in algorithm design, particularly useful in interview settings.

This problem tests your ability to manipulate linked lists efficiently, often requiring a combination of multiple techniques.

Problem Statement

Given the head of a singly linked list, reorder the list to follow a specific pattern:

$$L_0 \rightarrow L_n \rightarrow L_1 \rightarrow L_{n-1} \rightarrow L_2 \rightarrow L_{n-2} \rightarrow \dots$$

You must perform this rearrangement **“in-place”** without altering the node values.

[[LeetCode Link](#)]
[\[GeeksForGeeks Link\]](#)
[\[HackerRank Link\]](#)
[\[CodeSignal Link\]](#)
[\[InterviewBit Link\]](#)
[\[Educative Link\]](#)
[\[Codewars Link\]](#)

Algorithmic Approach

To efficiently reorder the linked list, the following **“three-step approach”** is employed:

1. Find the Middle of the List:

- Utilize the **“fast and slow pointer”** technique to locate the midpoint.
- This divides the list into two halves for subsequent processing.

2. Reverse the Second Half:

- Reverse the second half of the list to facilitate the interleaving process.

3. Merge the Two Halves:

- Alternately merge nodes from the first and the reversed second half to achieve the desired order.

Complexities

- **Time Complexity:** $O(n)$, where n is the number of nodes in the linked list.
Each step of the algorithm traverses the list linearly.
- **Space Complexity:** $O(1)$, as the rearrangement is performed in-place without using additional data structures.

Python Implementation

Below is the complete Python code for reordering a linked list using the outlined algorithmic approach:

```
# 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:
        """
        Do not return anything, modify head in-place instead.
        """
        if not head or not head.next:
            return

        # Step 1: Find the middle of the list
        slow, fast = head, head
        while fast.next and fast.next.next:
            slow = slow.next
            fast = fast.next.next

        # Step 2: Reverse the second half
        prev, curr = None, slow.next
        while curr:
            temp = curr.next
            curr.next = prev
            prev = curr
            curr = temp
        slow.next = None # Split the list into two halves

        # Step 3: Merge the two halves
        first, second = head, prev
        while second:
            temp1, temp2 = first.next, second.next
            first.next = second
            second.next = temp1
            first, second = temp1, temp2
```

Implementing the three-step approach ensures an efficient and clean solution with optimal space usage.

```

# Example Usage:
# Constructing the linked list: 1 -> 2 -> 3 -> 4
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4))))
solution = Solution()
solution.reorderList(head)

# Function to print the reordered linked list
def print_linked_list(head):
    elems = []
    while head:
        elems.append(str(head.val))
        head = head.next
    print(" -> ".join(elems))

print_linked_list(head)  # Output: 1 -> 4 -> 2 -> 3

# Handling Edge Cases:
# Single Node
head_single = ListNode(1)
solution.reorderList(head_single)
print_linked_list(head_single)  # Output: 1

# Two Nodes
head_two = ListNode(1, ListNode(2))
solution.reorderList(head_two)
print_linked_list(head_two)  # Output: 1 -> 2

# Empty List
head_empty = None
solution.reorderList(head_empty)
print_linked_list(head_empty)  # Output:

```

Explanation

The ‘reorderList’ function reorders a singly linked list in-place to follow the specific pattern required. Here’s a detailed breakdown of the implementation:

- **Initialization:**

- **Edge Case Handling:** If the list is empty or contains only one node, no reordering is necessary.
- **Pointers Setup:** Two pointers, ‘slow’ and ‘fast’, are initialized to traverse the list and find its middle.

- **Finding the Middle of the List:**

- **Fast and Slow Pointers:** ‘fast’ moves two steps at a time, while ‘slow’ moves one step.

- **Middle Detection:** When ‘fast’ reaches the end of the list, ‘slow’ will be at the midpoint.

- **Reversing the Second Half:**

- **Reversal Process:** Starting from ‘slow.next’, the second half of the list is reversed.
- **Pointer Manipulation:** Each node’s ‘next’ pointer is redirected to point to the previous node.
- **Splitting the List:** After reversal, ‘slow.next’ is set to ‘None’ to split the list into two separate halves.

- **Merging the Two Halves:**

- **Pointers Setup:** ‘first’ points to the head of the first half, and ‘second’ points to the head of the reversed second half.
- **Interleaving Nodes:** Alternately connect nodes from the first and second halves to achieve the desired order.
- **Termination:** The process continues until all nodes from the second half are merged.

- **Final Output:**

- The original list is now reordered in-place, following the pattern: first node, last node, second node, second last node, etc.

Why This Approach

This method is chosen due to its **efficiency** and **in-place** operation, which optimizes both time and space complexities. By breaking down the problem into three manageable steps—finding the middle, reversing the second half, and merging—the algorithm ensures a clear and systematic solution.

In-place algorithms are essential for optimizing memory usage, especially in environments with limited resources.

Alternative Approaches

An alternative method involves using additional data structures, such as arrays or stacks, to store the nodes’ values and then reconstruct the reordered list. While this approach can simplify the merging process, it increases the space complexity to $O(n)$, which is less optimal compared to the in-place method.

Using extra space can lead to higher memory usage, which may not be desirable in certain applications.

Similar Problems to This One

- Merge k Sorted Lists

- Reorder Array
- Reverse Linked List

Things to Keep in Mind and Tricks

- **Edge Cases:** Always consider scenarios where the linked list is empty, contains a single node, or has an even number of nodes.
- **Pointer Management:** Carefully manage pointers during list traversal and modification to prevent unintended cycles or loss of nodes.
- **In-Place Operations:** Strive to perform operations without using extra space to optimize memory usage.
- **Using Fast and Slow Pointers:** This technique is effective for finding the middle of the list and can be applied to other linked list problems.

Corner and Special Cases to Test When Writing the Code

- **Empty List:** head = None
- **Single Node:** head = ListNode(1)
- **Two Nodes:** head = ListNode(1, ListNode(2))
- **Odd Number of Nodes:** head = ListNode(1, ListNode(2, ListNode(3)))
- **Even Number of Nodes:** head = ListNode(1, ListNode(2, ListNode(3, ListNode(4))))
- **Duplicate Values:** head = ListNode(1, ListNode(1, ListNode(2, ListNode(2))))
- **Large Lists:** Test with a large number of nodes to ensure the algorithm handles scalability.

Problem 2.4 Remove Nth Node From End of List

The **Remove Nth Node From End of List** problem requires modifying a linked list by removing a specific node located a certain distance from the end.

Removing nodes from linked lists is a common operation in various algorithmic problems.

This problem tests your ability to manipulate linked lists efficiently using pointer techniques.

Problem Statement

Given the head of a linked list, the task is to remove the n -th node from the end of the list and return its head.

[[LeetCode Link](#)]
[GeeksForGeeks Link](#)
[HackerRank Link](#)
[CodeSignal Link](#)
[InterviewBit Link](#)
[Educative Link](#)
[Codewars Link](#)

Algorithmic Approach

The general approach for solving this problem is to use the **two-pointer technique**. Here is the step-by-step process:

1. Initialize Two Pointers:

- Create two pointers, ‘fast’ and ‘slow’, both initialized to the head of the list.

2. Advance the Fast Pointer:

- Move the ‘fast’ pointer forward by ‘n’ nodes. This creates a gap of ‘n’ nodes between ‘fast’ and ‘slow’.

3. Handle Edge Case:

- If the ‘fast’ pointer reaches the end after the initial advancement, it indicates that the node to be removed is the head. In this case, return ‘head.next’.

4. Move Both Pointers:

- Move both ‘fast’ and ‘slow’ pointers forward simultaneously until the ‘fast’ pointer reaches the last node.
- At this point, the ‘slow’ pointer will be just before the target node.

5. Remove the Target Node:

- Modify the ‘next’ pointer of the ‘slow’ node to skip the target node by pointing it to ‘slow.next.next’.

6. Return the Modified List:

- Return the head of the modified linked list.

Complexities

- **Time Complexity:** The time complexity of this algorithm is $O(L)$, where L is the length of the linked list, as we traverse the list at most twice.
- **Space Complexity:** The space complexity is $O(1)$ because only two extra pointers are used, regardless of the size of the input list.

Python Implementation

Below is the Python code that implements the aforementioned approach:

Implementing the two-pointer technique ensures an efficient solution with optimal space usage.

```

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def removeNthFromEnd(self, head: ListNode, n: int) -> ListNode:
        # Initialize two pointers
        fast = slow = head
        # Advance fast by n nodes
        for _ in range(n):
            fast = fast.next

        # Edge case: removing the first node
        if not fast:
            return head.next

        # Advance both pointers until fast reaches the last node
        while fast.next:
            fast = fast.next
            slow = slow.next

        # Skip the target node
        slow.next = slow.next.next
        return head

# Example Usage:
# Constructing the linked list: 1 -> 2 -> 3 -> 4 -> 5
head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(5)))))
solution = Solution()
new_head = solution.removeNthFromEnd(head, 2)

# Function to print the linked list
def print_linked_list(head):
    elems = []
    while head:
        elems.append(str(head.val))
        head = head.next
    print(" -> ".join(elems))

print_linked_list(new_head)  # Output: 1 -> 2 -> 3 -> 5

# Handling Edge Cases:
# Single node
head_single = ListNode(1)
new_head_single = solution.removeNthFromEnd(head_single, 1)
print_linked_list(new_head_single)  # Output:

# Two nodes
head_two = ListNode(1, ListNode(2))

```

```
new_head_two = solution.removeNthFromEnd(head_two, 1)
print_linked_list(new_head_two) # Output: 1
```

Explanation

The ‘removeNthFromEnd’ function efficiently removes the n -th node from the end of a singly linked list using the **two-pointer technique**. Here’s a detailed breakdown of the implementation:

- **Initialization:**

- **Fast and Slow Pointers:** Both ‘fast’ and ‘slow’ pointers are initialized to the head of the list.

- **Advancing the Fast Pointer:**

- **Offsetting Fast Pointer:** The ‘fast’ pointer is moved ‘ n ’ nodes ahead. This creates a gap of ‘ n ’ nodes between ‘fast’ and ‘slow’.

- **Handling Edge Cases:**

- **Removing the Head:** If the ‘fast’ pointer reaches ‘None’ after the initial advancement, it indicates that the node to remove is the head. In this case, the head is updated to ‘head.next’.

- **Moving Both Pointers:**

- **Simultaneous Traversal:** Both ‘fast’ and ‘slow’ pointers are moved one node at a time until ‘fast.next’ is ‘None’. At this point, ‘slow’ points to the node just before the target node.

- **Removing the Target Node:**

- **Skipping the Node:** The ‘next’ pointer of the ‘slow’ node is updated to skip the target node by pointing to ‘slow.next.next’.

- **Returning the Modified List:**

- **Final Head:** The function returns the head of the modified linked list, which now has the specified node removed.

Why This Approach

The two-pointer technique is chosen for this problem as it allows removing the n -th node from the end in a single pass through the list. Specifically, this approach eliminates the need to compute the length of the list beforehand, which would require an extra pass. This technique is both efficient and space-optimized, with $O(L)$ time complexity and $O(1)$ space complexity.

The two-pointer technique is versatile and can be applied to various linked list problems, such as detecting cycles or finding the middle node.

Alternative Approaches

An alternative method involves calculating the length of the linked list first and then removing the $L - n + 1$ -th node from the start. However, this requires two passes over the list: one to determine the length and another to locate and remove the target node. While straightforward, this approach is less efficient in terms of time compared to the two-pointer technique.

Using extra space can lead to higher memory usage, which may not be desirable in certain applications.

Similar Problems to This One

- Detect Cycle in a Linked List
- Find Middle of a Linked List
- Reverse Linked List

Things to Keep in Mind and Tricks

- **Edge Cases:** Always consider scenarios where the linked list has only one node, or where the node to be removed is the head itself.
- **Pointer Management:** Carefully manage pointer movements to avoid null reference errors or unintentional list breaks.
- **Two-Pointer Technique:** This technique is effective for finding the end of the list relative to the target node.
- **In-Place Modifications:** Strive to perform operations without using extra space to optimize memory usage.

Corner and Special Cases to Test When Writing the Code

- **Empty List:** `head = None, n = 1`
- **Single Node:** `head = [1], n = 1`
- **Two Nodes:** `head = [1, 2], n = 1`
- **n Equals List Length:** Removing the first node, e.g., `head = [1, 2, 3], n = 3`
- **n is Zero or Negative:** Invalid values for n , to test error handling (if applicable).
- **n Greater Than List Length:** To test the function's behavior when n exceeds the list length.
- **Multiple Removals:** Sequential removals to test the function's robustness.

Problem 2.5 Add Two Numbers

The **Add Two Numbers** problem is a classic example of manipulating linked lists in an algorithmic context. It combines the simplicity of arithmetic addition with the complexity of data structure traversal.

Adding numbers using linked lists is a fundamental algorithmic problem that combines arithmetic operations with data structure manipulation.

This problem is frequently encountered in technical interviews and serves as a foundation for more complex linked list operations.

Problem Statement

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.

Examples

Example 1:

[LeetCode Link]
 [GeeksForGeeks Link]
 [HackerRank Link]
 [CodeSignal Link]
 [InterviewBit Link]
 [Educative Link]
 [Codewars Link]

Input: 11 = [2,4,3], 12 = [5,6,4]
 Output: [7,0,8]
 Explanation: 342 + 465 = 807.

Example 2:

Input: 11 = [0], 12 = [0]
 Output: [0]

Example 3:

Input: 11 = [9,9,9,9,9,9,9], 12 = [9,9,9,9]
 Output: [8,9,9,9,0,0,0,1]

Algorithmic Approach

The solution to this problem is straightforward and simulates the addition process you would perform by hand.

This approach ensures that each digit is processed correctly, handling carries seamlessly.

1. Initialize Pointers and Variables:

- Create a dummy node to serve as the starting point of the resulting linked list.
- Initialize a ‘current’ pointer to track the end of the merged list.
- Initialize a ‘carry’ variable to handle sums exceeding 9.

2. Traverse Both Lists Simultaneously:

- While either of the linked lists has nodes left, or there is a carry value:
 - Extract the current values from both lists. If a list has been fully traversed, use 0 as its value.
 - Calculate the sum of the values along with the carry.
 - Update the carry for the next iteration using division and modulus operations.
 - Create a new node with the calculated digit and append it to the merged list.
 - Move the ‘current’ pointer forward.
 - Advance the pointers of both input lists if possible.

3. Handle Remaining Carry:

- After the main loop, if there is a remaining carry, create a new node with this value and append it to the merged list.

Complexities

- **Time Complexity:** $O(\max(n, m))$, where n and m are the lengths of the two linked lists. Each node is visited exactly once.
- **Space Complexity:** $O(\max(n, m))$, as the space required for the resulting linked list depends on the size of the larger input list.

Python Implementation

Below is the complete Python code that implements the aforementioned approach:

```
# 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: ListNode, l2: ListNode) -> ListNode:
        dummy = ListNode()
        current = dummy
        carry = 0

        while l1 or l2 or carry:
            val1 = (l1.val if l1 else 0)
            val2 = (l2.val if l2 else 0)
            carry, out = divmod(val1 + val2 + carry, 10)

            current.next = ListNode(out)
            current = current.next

            if l1:
                l1 = l1.next
            if l2:
                l2 = l2.next
```

Implementing the two-pointer technique ensures that the algorithm runs efficiently without unnecessary memory usage.

```

    current.next = ListNode(out)
    current = current.next

    l1 = (l1.next if l1 else None)
    l2 = (l2.next if l2 else None)

return dummy.next

```

This implementation starts by initializing a dummy node and a ‘current’ pointer. It then iterates through both linked lists, adding corresponding digits and managing the carry. If one list is longer than the other, the remaining digits are added along with the carry. Finally, any leftover carry is appended as a new node.

Explanation

The ‘addTwoNumbers’ function efficiently adds two numbers represented by linked lists by leveraging the **two-pointer technique**. Here’s a detailed breakdown of the implementation:

- **Initialization:**

- **Dummy Node:** A dummy node is created to simplify edge cases, such as when the resulting list is longer than both input lists.
- **Current Pointer:** The ‘current’ pointer tracks the end of the merged list.
- **Carry Variable:** The ‘carry’ variable holds any overflow value when the sum of two digits exceeds 9.

- **Adding Corresponding Digits:**

- **Value Extraction:** The values from the current nodes of both lists are extracted. If a list has been fully traversed, 0 is used as its value.
- **Sum Calculation:** The sum of the two values along with any existing carry is calculated.
- **Carry Update:** The ‘divmod’ function is used to determine the new carry and the digit to be placed in the current node.
- **Node Creation:** A new node with the calculated digit is created and appended to the merged list.
- **Pointer Advancement:** Both input list pointers and the ‘current’ pointer are moved forward.

- **Handling Remaining Carry:**

- After the main loop, if there is a remaining carry, a new node with this value is appended to the merged list.

- **Returning the Result:**

- The function returns the next node of the dummy node, effectively skipping the dummy node and providing the head of the merged linked list.

Why This Approach

The **two-pointer technique** is chosen for its efficiency and simplicity. By traversing both lists simultaneously and managing the carry during the traversal, the algorithm ensures that each digit is processed correctly in a single pass.

This approach avoids the need to first determine the lengths of the lists, thereby optimizing the runtime.

Alternative Approaches

An alternative approach involves first determining the lengths of both linked lists, aligning the pointers accordingly, and then performing the addition. However, this method requires two passes through the lists: one to calculate their lengths and another to perform the addition. This increases the time complexity compared to the two-pointer technique.

While feasible, this approach is less efficient and more complex to implement.

Similar Problems to This One

- Merge k Sorted Lists
- Linked List Cycle
- Reverse Linked List
- Detect Cycle in a Linked List

Things to Keep in Mind and Tricks

- **Edge Cases:** Always consider scenarios where one or both linked lists are empty, or where the resulting sum has an extra digit due to a final carry.
- **Pointer Management:** Carefully manage the advancement of pointers to prevent null reference errors or infinite loops.
- **Carry Handling:** Ensure that the carry is correctly propagated throughout the addition process, especially in cases where multiple consecutive digits result in a carry.
- **Dummy Node Usage:** Utilizing a dummy node simplifies the logic by avoiding the need to handle the head of the merged list separately.

Corner and Special Cases to Test When Implementing

- **Both Lists Empty:** `l1 = None, l2 = None`
- **One List Empty:** `l1 = [0], l2 = [1, 2, 3]`
- **Different Lengths:** `l1 = [2,4,3], l2 = [5,6]`
- **All Digits Result in Carry:** `l1 = [9,9,9], l2 = [1,1,1]`
- **Single Node Lists:** `l1 = [5], l2 = [5]`
- **Large Numbers:** Test with very long linked lists to ensure performance and correctness.
- **Multiple Carry Overlaps:** `l1 = [9,9,9,9], l2 = [9,9,9,9]`

Problem 2.6 Linked List Cycle

The **Linked List Cycle** problem is a common challenge that involves determining if a singly linked list contains a cycle—a situation where a node's 'next' pointer points to an earlier node, leading to an infinite loop when traversing the list. This problem is important in the field of computer science because cycles in linked lists can lead to bugs and inefficiencies in software applications. To solve this problem, an effective algorithm needs to be formulated that can detect the presence of a cycle without traversing the entire list repeatedly, which could cause an infinite loop if a cycle is present.

Detecting cycles in linked lists is crucial for preventing infinite loops and ensuring the integrity of data structures.

Efficient cycle detection is essential for optimizing algorithms that utilize linked lists, ensuring reliable performance and resource management.

Problem Statement

To determine if a cycle is present in a given linked list, the typical strategy is to use indicators or "pointers" to traverse the list. One widely used method to detect cycles is ["Floyd's Cycle Detection Algorithm"](#), also known as the ["Tortoise and Hare algorithm"](#). In this approach, two pointers are initialized at the head of the linked list:

- **Slow Pointer ('slow')**: Moves one step at a time.
- **Fast Pointer ('fast')**: Moves two steps at a time.

If there is a cycle in the list, the fast pointer will loop around and eventually meet the slow pointer within the cycle. If there is no cycle, the fast pointer will reach the end of the list without meeting the slow pointer.

The function signature for this problem, in Python, is as follows:

```

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        # Implementation goes here

```

Algorithmic Approach

Floyd's Cycle Detection Algorithm is the optimal solution to this problem because it requires only two pointers, satisfying the $O(1)$ memory constraint of the follow-up challenge. The implementation involves iterating through the list with the slow and fast pointers as described, checking for the condition where the fast pointer equals the slow pointer, which indicates a cycle. If the fast pointer reaches the end of the list (i.e., encounters a ‘None’ reference), the list does not contain a cycle, and the function returns ‘False’.

[LeetCode Link]
[GeeksForGeeks Link]
[HackerRank Link]
[CodeSignal Link]
[InterviewBit Link]
[Educative Link]
[Codewars Link]

This method is both time and space-efficient, making it suitable for large linked lists.

Complexities

- **Time Complexity:** The time complexity of Floyd's algorithm is $O(n)$, where n is the number of nodes in the linked list. In the worst case, this is the time it takes for the fast pointer to meet the slow pointer.
- **Space Complexity:** The algorithm achieves $O(1)$ space complexity since it only uses two pointers regardless of the size of the linked list.

Python Implementation

Below is the complete Python code for the ‘Solution‘ class, which implements the ‘hasCycle‘ function using Floyd’s Cycle Detection Algorithm to determine whether a linked list has a cycle:

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def hasCycle(self, head: ListNode) -> bool:
        if not head:
            return False

        slow = head
        fast = head.next

        while fast and fast.next:
            if slow == fast:
                return True
            slow = slow.next
            fast = fast.next.next

        return False

# Example Usage:
# Creating a linked list with a cycle: 1 -> 2 -> 3 -> 4 -> 2 ...
node1 = ListNode(1)
node2 = ListNode(2)
node3 = ListNode(3)
node4 = ListNode(4)
node1.next = node2
node2.next = node3
node3.next = node4
node4.next = node2 # Creates a cycle

solution = Solution()
print(solution.hasCycle(node1)) # Output: True

# Creating a linked list without a cycle: 1 -> 2 -> 3 -> 4 -> None
node1 = ListNode(1, ListNode(2, ListNode(3, ListNode(4))))
print(solution.hasCycle(node1)) # Output: False

# Edge Case: Empty List
print(solution.hasCycle(None)) # Output: False

# Edge Case: Single Node without Cycle
single_node = ListNode(1)
print(solution.hasCycle(single_node)) # Output: False
```

Implementing Floyd’s algorithm ensures efficient detection of cycles with minimal memory overhead.

```
# Edge Case: Single Node with Cycle
single_node_cycle = ListNode(1)
single_node_cycle.next = single_node_cycle
print(solution.hasCycle(single_node_cycle)) # Output: True
```

The code first checks for an empty list, which cannot have a cycle. It then initializes the ‘slow’ and ‘fast’ pointers and begins the iteration. If at any point, ‘slow’ is equal to ‘fast’, a cycle is detected, and the function returns ‘True’. If ‘fast’ or ‘fast.next’ becomes ‘None’, meaning the end of the list is reached, the function returns ‘False’, indicating no cycle.

Explanation

The ‘hasCycle’ function efficiently detects a cycle in a singly linked list by leveraging [**Floyd’s Cycle Detection Algorithm**](#). Here’s a detailed breakdown of the implementation:

- **Initialization:**

- **Edge Case Handling:** If the list is empty ('head' is 'None'), it cannot have a cycle, so the function returns 'False'.
- **Pointers Setup:** Two pointers, ‘slow’ and ‘fast’, are initialized. ‘slow’ starts at the head, while ‘fast’ starts at the second node ('head.next').

- **Cycle Detection Loop:**

- **Traversal:** The loop continues as long as ‘fast’ and ‘fast.next’ are not ‘None’. This ensures that we do not traverse beyond the end of the list.
- **Meeting Point Check:** If at any point ‘slow’ equals ‘fast’, a cycle is detected, and the function returns ‘True’.
- **Pointer Advancement:** If no cycle is detected at the current positions, ‘slow’ moves one step forward ('slow = slow.next'), and ‘fast’ moves two steps forward ('fast = fast.next.next').

- **Conclusion:**

- If the loop terminates without ‘slow’ meeting ‘fast’, it means the list has no cycle, and the function returns ‘False’.

Why This Approach

[**Floyd’s Cycle Detection Algorithm**](#) is chosen for its efficiency in both time and space. It does not require additional data structures, thereby adhering to the $O(1)$

space complexity constraint. This algorithm is also intuitive and reliable for detecting cycles in linked lists, making it a preferred choice in both academic and professional settings.

The two-pointer technique is versatile and can be applied to various linked list problems, such as finding the middle node or detecting cycles.

Alternative Approaches

An alternative approach involves using a **hash set** to keep track of visited nodes. Each node visited by the traversal would be added to the hash set. If a node is encountered that already exists in the hash set, a cycle is detected, and the function returns ‘True’. If the traversal reaches the end of the list (‘None’), the function returns ‘False’, indicating no cycle.

- **Pros:** Simple to implement and understand.
- **Cons:** Requires $O(n)$ additional space to store visited nodes, which is less optimal compared to Floyd’s algorithm.

While this method is straightforward, it does not satisfy the $O(1)$ space complexity requirement of the follow-up challenge, making Floyd’s Cycle Detection Algorithm a more optimal solution.

Using extra space for hash sets increases memory usage, which may not be ideal for large linked lists.

Similar Problems to This One

- Find Middle of Linked List
- Reverse Linked List
- Merge Two Sorted Lists
- Remove Nth Node From End of List

Things to Keep in Mind and Tricks

- **Edge Cases:** Always consider scenarios where the linked list is empty, contains only one node, or has a cycle starting at the head.
- **Pointer Management:** Carefully manage the advancement of pointers to prevent null reference errors or infinite loops.
- **Two-Pointer Technique:** This technique is effective not only for cycle detection but also for other linked list problems like finding the middle node or removing elements.
- **Early Termination:** If a cycle is detected early, the function can terminate without traversing the entire list.

Corner and Special Cases to Test When Implementing

- **Empty List:** head = None
- **Single Node without Cycle:** head = ListNode(1)
- **Single Node with Cycle:** head = ListNode(1, head)
- **Two Nodes with Cycle:** head = ListNode(1, ListNode(2, head))
- **Multiple Nodes without Cycle:** head = ListNode(1, ListNode(2, ListNode(3, ListNode(4))))
- **Cycle in the Middle:** head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(2)))))
- **Cycle at the End:** head = ListNode(1, ListNode(2, ListNode(3, ListNode(4, ListNode(3)))))
- **Large List with Cycle:** Test with a large number of nodes to ensure the algorithm handles scalability.

Chapter 3

Stacks and Queues

Chapter 4

Stacks and LIFO

Stacks are fundamental data structures in computer science, characterized by their Last-In, First-Out (LIFO) behavior. They play a crucial role in various algorithms and system functionalities, from expression evaluation to memory management. This chapter explores the history, fundamental concepts, implementations, and applications of stacks, providing a comprehensive understanding of their significance in computing.

4.1 Introduction to Stacks

A **stack** is a linear data structure that follows the **Last-In, First-Out (LIFO)** principle. This means the last element added to the stack will be the first one to be removed. Stacks allow two primary operations:

- **push:** Adds an element to the top of the stack.
- **pop:** Removes the element from the top of the stack.

Additionally, stacks often support operations such as `peek` (or `top`), which returns the top element without removing it, and `isEmpty`, which checks whether the stack is empty.

4.2 History of Stacks

The concept of stacks emerged in the context of mathematical logic and computing in the mid-20th century. Early computing pioneers recognized the need for a data structure that could manage nested function calls and expression evaluations.

Stacks are analogous to a stack of plates where the last plate placed on top is the first one to be removed.

The concept of stacks dates back to the early days of computing and mathematical logic.

4.2.1 Early Development

In the 1950s, the Polish mathematician Jan Łukasiewicz introduced **Polish notation**, a prefix notation for logical expressions without the need for parentheses. Later, **Reverse Polish Notation (RPN)** was developed, which placed operators after their operands. RPN is inherently stack-based and influenced the development of stack data structures.

Reverse Polish Notation is used in some calculators and programming languages for its simplicity in expression evaluation.

4.2.2 Stacks in Programming Languages

The introduction of recursive function calls in programming languages necessitated a mechanism to keep track of function calls and local variables. This led to the implementation of the **call stack** in the runtime environment of languages like ALGOL and later in C and its derivatives.

The call stack is essential for managing function calls, parameters, and return addresses during program execution.

4.3 Fundamental Concepts of LIFO

The LIFO property of stacks makes them suitable for scenarios where the most recently added data needs to be accessed first. This section delves into the fundamental concepts that underpin stacks and their operations.

4.3.1 Stack Operations

- **Push Operation:** Adds an element to the top of the stack.
- **Pop Operation:** Removes and returns the top element of the stack.
- **Peek (Top) Operation:** Returns the top element without removing it.
- **isEmpty Operation:** Checks if the stack has no elements.

Efficient stack operations typically run in constant $O(1)$ time.

4.3.2 Stack Representation

Stacks can be implemented using arrays (static stacks) or linked lists (dynamic stacks). The choice of implementation affects the flexibility and memory usage of the stack.

- **Array-Based Stacks:** Use a fixed-size array to store elements. Simple and efficient but have a fixed capacity.
- **Linked List-Based Stacks:** Use nodes connected via pointers. Dynamic in size, growing and shrinking as needed.

Array-based stacks are straightforward but may face overflow if capacity is exceeded.

Linked list stacks avoid overflow but have overhead due to node pointers.

4.4 Implementations of Stacks

Implementing a stack involves managing the addition and removal of elements while maintaining the LIFO order. Below are common ways to implement stacks in programming languages.

4.4.1 Array Implementation

An array-based stack uses an array and a variable to track the index of the top element.

```
class ArrayStack:
    def __init__(self, capacity):
        self.stack = [None] * capacity
        self.top = -1 # Initialize top at -1 to indicate an empty stack
        self.capacity = capacity

    def push(self, item):
        if self.top >= self.capacity - 1:
            raise Exception("Stack Overflow")
        self.top += 1
        self.stack[self.top] = item

    def pop(self):
        if self.top == -1:
            raise Exception("Stack Underflow")
        item = self.stack[self.top]
        self.top -= 1
        return item

    def peek(self):
        if self.top == -1:
            return None
        return self.stack[self.top]

    def isEmpty(self):
        return self.top == -1
```

Array stacks are efficient but require careful handling of overflow and underflow conditions.

4.4.2 Linked List Implementation

A linked list-based stack uses nodes where each node contains data and a reference to the next node.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

```

class LinkedListStack:
    def __init__(self):
        self.head = None # Initialize head to None

    def push(self, item):
        new_node = Node(item)
        new_node.next = self.head # New node points to the former head
        self.head = new_node # Head is now the new node

    def pop(self):
        if self.head is None:
            raise Exception("Stack Underflow")
        item = self.head.data
        self.head = self.head.next # Move head to the next node
        return item

    def peek(self):
        if self.head is None:
            return None
        return self.head.data

    def isEmpty(self):
        return self.head is None

```

4.5 Applications of Stacks

Stacks are employed in a wide array of applications due to their LIFO nature. Below are some common use cases.

4.5.1 Expression Evaluation and Conversion

Stacks are used to evaluate expressions in postfix (RPN) notation and to convert infix expressions to postfix or prefix notation.

- **Expression Evaluation:** Operands and operators are processed using stacks to compute the result of an expression.
- **Syntax Parsing:** Compilers use stacks to parse expressions and check for balanced parentheses.

Linked list stacks offer dynamic sizing but involve additional memory overhead per node.

Stacks are integral to many algorithms and system processes.

Stack-based evaluation avoids the need for parentheses in expressions.

4.5.2 Function Call Management

The call stack is a stack data structure that stores information about active subroutines or functions in a program.

- **Recursion Handling:** Each recursive call adds a new frame to the call stack. Excessive recursion can lead to stack overflow errors.
- **Local Variables and Return Addresses:** The call stack keeps track of local variables and return points for function calls.

4.5.3 Undo Mechanisms

Applications like text editors use stacks to implement undo and redo functionalities.

- **Undo Operation:** User actions are pushed onto a stack; undo pops the last action. Stacks enable reversal of actions in the order they occurred.
- **Redo Operation:** A separate stack can manage redo actions by storing undone operations.

4.5.4 Depth-First Search (DFS)

Stacks are utilized in graph algorithms, particularly in implementing non-recursive DFS.

- **Traversal Order:** Stacks ensure that the most recently discovered node is processed next.
- **Backtracking:** Stacks facilitate backtracking by keeping track of the path taken. DFS can be implemented recursively or iteratively using a stack.

4.5.5 Memory Management

Stacks play a role in memory allocation and deallocation processes.

- **Stack Memory:** Used for static memory allocation, managing function call frames.
- **Heap vs. Stack:** Understanding the difference is crucial for efficient memory usage. Stack memory is faster but limited in size compared to heap memory.

4.6 Conclusion

Stacks are a fundamental component of computer science, providing an efficient way to manage data that adheres to the LIFO principle. Their simplicity and versatility make them suitable for a wide range of applications, from expression evaluation to memory management. Understanding stacks is essential for developing efficient algorithms and solving complex computational problems.

In the following sections, we will explore various problems and algorithms that utilize stacks, delving deeper into their practical applications and implementation strategies.

Problem 4.1 Valid Parentheses

Problem Description: Given a string containing just the characters '(', ')', '[', ']', '{', and '}', determine if the input string is valid.

This classic problem uses stacks to check for balanced parentheses in expressions.

An input string is valid if:

- Open brackets must be closed by the same type of brackets.
- Open brackets must be closed in the correct order.

Solution Overview: Use a stack to keep track of opening brackets. Iterate through the string, and for each character:

- If it's an opening bracket ('(', '[', '{') push it onto the stack.
- If it's a closing bracket (')', ']', '}'), check if the stack is not empty and the top of the stack is the matching opening bracket. If so, pop the stack; otherwise, return False.

At the end, if the stack is empty, return True; otherwise, return False.

```
def isValid(s):
    stack = []
    mapping = {')': '(', ')': ')', '}': '{', ']': '['}
    for char in s:
        if char in mapping.values():
            stack.append(char)
        elif char in mapping:
            if not stack or mapping[char] != stack.pop():
                return False
        else:
            # Invalid character encountered
            return False
    return not stack

# Example usage:
print(isValid("()"))      # Output: True
print(isValid("()[]{}"))   # Output: True
print(isValid("(]"))       # Output: False
```

Problem 4.2 Min Stack

Problem Description: Design a stack that supports the following operations in constant time:

- `push(x)`: Push element x onto the stack.
- `pop()`: Removes the element on top of the stack.
- `top()`: Get the top element.
- `getMin()`: Retrieve the minimum element in the stack.

This problem introduces the concept of augmenting stack operations with auxiliary data to achieve additional functionality.

Notes:

- All operations must run in $O(1)$ time complexity.
- You may assume that all inputs are valid integers.

Solution Overview: To achieve constant-time retrieval of the minimum element, we can augment the stack to keep track of the current minimum at each level. There are several approaches:

Efficient retrieval of the minimum element enhances the utility of the stack in scenarios requiring dynamic minimum tracking.

1. Using an Auxiliary Stack:

- Maintain an additional stack, `minStack`, that stores the minimum value at each level.
- When pushing a new element, compare it with the current minimum and push the lesser of the two onto `minStack`.
- When popping, pop from both the main stack and `minStack`.
- `getMin()` simply returns the top of `minStack`.

2. Using a Single Stack with Value-Difference:

- Store the difference between the current value and the minimum value.
- Use a variable to keep track of the minimum.
- This method is more space-efficient but more complex.

We will focus on the first approach for its simplicity and clarity.

Implementation Details: Here's an implementation using two stacks in Python:

```

class MinStack:
    def __init__(self):
        self.stack = []
        self.minStack = []

    def push(self, x):
        self.stack.append(x)
        # If minStack is empty or x is smaller than the current minimum
        if not self.minStack or x <= self.minStack[-1]:
            self.minStack.append(x)
        else:
            # Repeat the current minimum
            self.minStack.append(self.minStack[-1])

    def pop(self):
        if not self.stack:
            raise Exception("Stack Underflow")
        self.stack.pop()
        self.minStack.pop()

    def top(self):
        if not self.stack:
            raise Exception("Stack is empty")
        return self.stack[-1]

    def getMin(self):
        if not self.minStack:
            raise Exception("Stack is empty")
        return self.minStack[-1]

# Example usage:
min_stack = MinStack()
min_stack.push(-2)
min_stack.push(0)
min_stack.push(-3)
print(min_stack.getMin())    # Output: -3
min_stack.pop()
print(min_stack.top())      # Output: 0
print(min_stack.getMin())    # Output: -2

```

Complexities:

- **Time Complexity:** All operations (push, pop, top, getMin) run in $O(1)$ time.
- **Space Complexity:** $O(n)$, where n is the number of elements in the stack, due to the additional `minStack`.

By mirroring the main stack's operations in `minStack`, we maintain synchronization of minimum values.

The extra space used is proportional to the number of elements but ensures constant-time operations.

Alternative Approach: Using a single stack with value-difference optimization:

- Keep a variable `min` to track the current minimum.
- When pushing, if the new element is less than or equal to `min`, push a special marker or encoded value.
- When popping, if the popped value indicates a change in `min`, update `min` accordingly.

Implementation Using Single Stack:

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.min = None

    def push(self, x):
        if not self.stack:
            self.stack.append(x)
            self.min = x
        elif x <= self.min:
            # Store previous min
            self.stack.append(2 * x - self.min)
            self.min = x
        else:
            self.stack.append(x)

    def pop(self):
        if not self.stack:
            raise Exception("Stack Underflow")
        top = self.stack.pop()
        if top < self.min:
            # Retrieve previous min
            self.min = 2 * self.min - top

    def top(self):
        if not self.stack:
            raise Exception("Stack is empty")
        top = self.stack[-1]
        if top < self.min:
            return self.min
        else:
            return top

    def getMin(self):
        if not self.stack:
            raise Exception("Stack is empty")
        return self.min
```

```
# Example usage:
min_stack = MinStack()
min_stack.push(-2)
min_stack.push(0)
min_stack.push(-3)
print(min_stack.getMin())    # Output: -3
min_stack.pop()
print(min_stack.top())      # Output: 0
print(min_stack.getMin())    # Output: -2
```

Complexities of Alternative Approach:

- **Time Complexity:** All operations run in $O(1)$ time.
- **Space Complexity:** $O(n)$, but potentially less than the two-stack approach in practice.

Why This Approach?

Using an auxiliary stack simplifies the implementation and makes the logic straightforward. It provides clear visibility into the minimum values at each stack level and is less error-prone compared to the single stack method.

Corner Cases to Test:

- **Empty Stack Operations:** Ensure that calling `pop`, `top`, or `getMin` on an empty stack is handled properly.
- **Duplicate Minimums:** Test scenarios where multiple elements have the same minimum value.
- **Negative Values:** Verify that the stack handles negative integers correctly.
- **Single Element Stack:** Confirm that operations work as expected when only one element is in the stack.

Similar Problems:

- **Max Stack:** Design a stack that supports retrieving the maximum element in constant time.
- **Stack with Increment Operation:** Implement a stack with an operation to increment the bottom k elements by a given value.
- **Design a Queue using Stacks:** Explore the reverse scenario of implementing a queue using stack operations.

This method saves space but adds complexity in handling encoded values.

Optimal for environments where space is at a premium and the added complexity is acceptable.

Thorough testing ensures robustness against edge cases and potential errors.

Conclusion:

The Min Stack problem exemplifies how data structures can be enhanced to provide additional functionality without compromising efficiency. By intelligently managing auxiliary data, we can extend the capabilities of a standard stack to meet specific requirements, such as constant-time retrieval of the minimum element. Mastery of such techniques is essential for designing efficient algorithms and solving complex computational problems.

Exploring variations of the problem deepens understanding of stack manipulation and augmentation.

Problem 4.3 Evaluate Reverse Polish Notation

Problem Description: Evaluate the value of an arithmetic expression in Reverse Polish Notation (RPN). Valid operators are '+', '-', '*', and '/'. Each operand may be an integer or another expression.

Augmenting data structures is a key skill in advanced algorithm design.

This problem demonstrates the use of stacks in evaluating expressions in postfix notation.

Notes:

- Division between two integers should truncate toward zero.
- The given RPN expression is always valid.
- The length of the tokens array will be at least 1.
- The answer and all intermediate calculations are guaranteed to be in the range of a 32-bit signed integer.

Solution Overview: Use a stack to store operands. Iterate through the tokens:

- If the token is an operand (number), push it onto the stack.
- If the token is an operator, pop the top two operands from the stack, apply the operator, and push the result back onto the stack.

At the end, the value on the top of the stack is the result of the expression.

```
def evalRPN(tokens):
    stack = []
    operators = {'+', '-', '*', '/'}
    for token in tokens:
        if token in operators:
            b = int(stack.pop())
            a = int(stack.pop())
            if token == '+':
                result = a + b
            elif token == '-':
                result = a - b
            elif token == '*':
                result = a * b
            else:
                result = a / b
            stack.append(result)
        else:
            stack.append(int(token))
    return stack[-1]
```

```

        result = a * b
    else: # token == '/'
        # Ensure truncation toward zero
        result = int(a / b)
        stack.append(result)
    else:
        stack.append(int(token))
return stack.pop()

# Example usage:
print(evalRPN(["2", "1", "+", "3", "*"]))      # Output: 9
print(evalRPN(["4", "13", "5", "/", "+"]))       # Output: 6
print(evalRPN([
    "10", "6", "9", "3", "+", "-11", "*",
    "/", "*", "17", "+", "5", "+"]))             # Output: 22

```

Problem 4.4 Daily Temperatures

The **Daily Temperatures** problem is a classic algorithmic challenge that involves predicting how many days one would have to wait until a warmer temperature. It is an excellent example to illustrate the effectiveness of stack-based solutions combined with the Two Pointers Technique, especially when processing the data in a specific order.

Problem Statement

Given a list of daily temperatures T , return a list such that, for each day in the input, tells you how many days you would have to wait until a warmer temperature. If there is no future day for which this is possible, put 0 instead.

Example:

Given the list $T = [73, 74, 75, 71, 69, 72, 76, 73]$,

Your output should be $[1, 1, 4, 2, 1, 1, 0, 0]$ ¹.

¹ This output indicates, for example, that after the first day with temperature 73, the next warmer temperature occurs in 1 day at temperature 74

Algorithmic Approach

The solution to the Daily Temperatures problem can be efficiently implemented by iterating through the list of temperatures from right to left². This approach leverages a stack to keep track of temperatures and their indices, enabling quick lookup of the next warmer day for each temperature.

² Processing from right to left allows us to maintain information about future warmer days that have already been processed.

1. **Initialize a Stack:** Create an empty stack to keep track of temperature indices³.

³ The stack will store indices of temperatures in a monotonically decreasing order, facilitating the search for the next warmer day.

2. **Initialize the Result List:** Create a result list filled with 0s, as a default for days with no warmer future temperature⁴.

3. **Iterate from Right to Left:** Start from the end of the temperature list and move towards the beginning.

- For each temperature $T[i]$:

- **Maintain Monotonic Stack:** While the stack is not empty and the current temperature $T[i]$ is greater than or equal to the temperature at the index on the top of the stack, pop the stack⁵.

- **Determine the Next Warmer Day:**

- * If the stack is not empty after the popping process, the next warmer day for $T[i]$ is the difference between the current index and the index at the top of the stack⁶.

- * If the stack is empty, there is no warmer day in the future, so the result remains 0⁷.

- **Push Current Index onto Stack:** Add the current index i to the stack⁸.

4. **Completion:** After iterating through all temperatures, the result list will contain the required number of days to wait for a warmer temperature for each day.

⁴ This list will be updated with the number of days to wait for a warmer temperature.

⁵ This ensures that the stack only contains indices of temperatures warmer than the current one.

⁶ This difference represents the number of days to wait for a warmer temperature.

⁷ This scenario occurs when no future day has a higher temperature than the current day.

⁸ This index may serve as the next warmer day for preceding temperatures.

Python Implementation

```
def dailyTemperatures(T):
    """
    Finds the number of days until a warmer temperature for each day.

    Parameters:
    T (List[int]): List of daily temperatures.

    Returns:
    List[int]: List indicating the number of days to wait for a warmer temperature
    ↵ .
    """

    n = len(T)
    res = [0] * n
    stack = []

    for i in range(n-1, -1, -1):
        # Remove temperatures that are less than or equal to current
        while stack and T[i] >= T[stack[-1]]:
            stack.pop()

        # If stack is not empty, the next warmer day is stack[-1] - i
        if stack:
            res[i] = stack[-1] - i

    # Push current index onto stack
```

```

        stack.append(i)

    return res

# Example usage:
T = [73, 74, 75, 71, 69, 72, 76, 73]
print(dailyTemperatures(T))  # Output: [1, 1, 4, 2, 1, 1, 0, 0]

```

Example Usage and Test Cases

```

# Test case 1: General case
T = [73, 74, 75, 71, 69, 72, 76, 73]
print(dailyTemperatures(T))  # Output: [1, 1, 4, 2, 1, 1,
                             ↪ 0, 0]

# Test case 2: Increasing temperatures
T = [30, 40, 50, 60]
print(dailyTemperatures(T))  # Output: [1, 1, 1, 0]

# Test case 3: Decreasing temperatures
T = [60, 50, 40, 30]
print(dailyTemperatures(T))  # Output: [0, 0, 0, 0]

# Test case 4: Mixed temperatures with duplicates
T = [30, 40, 40, 50, 30, 60]
print(dailyTemperatures(T))  # Output: [1, 1, 2, 1, 1, 0]

# Test case 5: Single element array
T = [30]
print(dailyTemperatures(T))  # Output: [0]

```

Why This Approach

The **Two Pointers Technique** combined with a stack-based approach is chosen for its **efficiency and optimal time complexity**⁹. By processing the temperature list from right to left, we can leverage the stack to keep track of indices of warmer temperatures⁹. This method ensures a single pass through the list with each element being pushed and popped at most once, resulting in a time complexity of $O(n)$ ¹⁰. Additionally, this approach maintains a space complexity of $O(n)$, primarily due to the stack, which is acceptable given the problem constraints.

⁹ This allows us to quickly determine how many days to wait for a warmer temperature by referring to the stack's top element

¹⁰ Linear time complexity is optimal for this problem, especially with large datasets

Complexity Analysis

- **Time Complexity:** $O(n)$ ¹¹.
- **Space Complexity:** $O(n)$ ¹².

¹¹ Each temperature is processed once, with push and pop operations on the stack occurring at most once per element

¹² The stack can potentially store all indices in the worst-case scenario

Similar Problems

Other problems that can be efficiently solved using the Two Pointers Technique and stack-based approaches include:

- **Next Greater Element:** Find the next greater element for each element¹³.
- **Stock Span Problem:** Calculate the span of stock's price for all days¹⁴.
- **Largest Rectangle in Histogram:** Find the largest rectangular area in a histogram¹⁵.
- **Maximum Depth of Binary Tree:** Determine the maximum depth of a binary tree¹⁶.

These problems often require maintaining information about previous elements or states, making stack-based or two pointers approaches highly effective¹⁷.

Things to Keep in Mind and Tricks

- **Processing Order:** Iterating from right to left allows you to have information about all future days¹⁸.
- **Handling Duplicates:** Ensure that duplicates are correctly handled to avoid redundant calculations¹⁹.
- **Stack Management:** Properly manage the stack by pushing and popping indices based on the current temperature²⁰.
- **Edge Cases:** Always consider edge cases such as single-element arrays or arrays with no warmer future days²¹.
- **Space Optimization:** While the stack requires additional space, it is necessary for achieving optimal time complexity²².

¹³ Utilizes a similar stack-based traversal to determine the next greater element.

¹⁴ Employs a stack to keep track of previous days' prices for span calculations.

¹⁵ Uses a stack to manage the indices of bars for efficient area computation.

¹⁶ Can be approached iteratively with the help of a stack for depth tracking.

¹⁷ Understanding these techniques provides a strong foundation for solving a variety of related computational challenges

¹⁸ This ensures that you can make informed decisions about the next warmer day based on already processed information

¹⁹ Skipping identical elements during traversal maintains the accuracy of the result

²⁰ This maintains a monotonically decreasing stack, essential for the algorithm's efficiency

²¹ Robust handling of these scenarios ensures the algorithm's reliability

²² Balancing space and time efficiency is crucial for effective algorithm design

Exercises

1. **Next Greater Element:** Given an array, find the next greater element for each element²³.
2. **Stock Span Problem:** Calculate the span of stock's price for all days²⁴.
3. **Largest Rectangle in Histogram:** Find the largest rectangular area in a histogram²⁵.
4. **Maximum Depth of Binary Tree:** Determine the maximum depth of a binary tree²⁶.
5. **Minimum Window Substring:** Given two strings, find the minimum window in the first string which will contain all the characters of the second string²⁷.

²³ Implement a stack-based solution similar to the Daily Temperatures problem

²⁴ Use a stack to keep track of previous days' prices and calculate spans accordingly

²⁵ Employ a stack to manage the indices of bars for efficient area computation

²⁶ Approach the problem iteratively using a stack to track depth levels

²⁷ Combine sliding window techniques with stack-based approaches for optimal solutions

Questions for Reflection

- Why is processing the temperature list from right to left more efficient than from left to right?²⁸.
- How does the stack help in keeping track of necessary information for determining the next warmer day?²⁹.
- Can the Two Pointers Technique be applied to other similar problems? Provide examples³⁰.
- What are the trade-offs between using a stack-based approach versus a brute-force approach in terms of time and space complexity?³¹.
- How can this approach be modified to handle different variations of the problem, such as finding the next colder day?³².

²⁸ Consider how information about future days is utilized in each approach

²⁹ Analyze the role of the stack in maintaining order and facilitating quick lookups

³⁰ Think about how the two pointers can manage and compare elements in different scenarios

³¹ Evaluate the benefits of optimized algorithms over naive implementations

³² Explore how reversing the conditions and maintaining the stack accordingly can adapt the solution

References

LeetCode Problem: ³³

³³ Daily Temperatures

GeeksforGeeks Article: ³⁴

³⁴ Two Pointers Technique

HackerRank Problem: ³⁵

³⁵ Two Sum

Conclusion

The Two Pointers Technique, when combined with a stack-based approach and processing the temperature list from right to left, offers an **efficient and optimal solution** to the Daily Temperatures problem³⁶. By leveraging the sorted nature of the stack and maintaining information about future warmer days, the algorithm efficiently determines the required wait times without unnecessary computations³⁷. Mastering this technique not only enhances problem-solving skills but also prepares you for tackling more complex algorithmic challenges effectively.

³⁶ This method ensures linear time complexity while effectively managing necessary information about future temperatures

³⁷ This approach demonstrates the power of combining multiple algorithmic strategies for enhanced performance

Problem 4.5 Next Greater Element

Problem Description:

Given an array of integers `nums`, for each element in the array, find the next greater element. The Next Greater Element of a number x is the first greater number to its right in the array. If it does not exist, output -1 for that number.

This problem utilizes monotonic stacks to find the next greater element for each item in a list.

Example 1:

- **Input:** `nums = [4, 5, 2, 25]`

- **Output:** [5, 25, 25, -1]

Example 2:

- **Input:** nums = [13, 7, 6, 12]
- **Output:** [-1, 12, 12, -1]

Solution Overview:

Use a stack to keep track of indices whose next greater element has not been found yet. Iterate through the array:

- For each element $nums[i]$:
 - While the stack is not empty and $nums[i] > nums[stack[-1]]$:
 - * Set the next greater element of $nums[stack[-1]]$ to $nums[i]$.
 - * Pop the index from the stack.
 - Push the current index i onto the stack.

At the end, set the next greater element of remaining indices in the stack to -1 .

```
def nextGreaterElement(nums):
    result = [-1] * len(nums)
    stack = []
    for i in range(len(nums)):
        while stack and nums[i] > nums[stack[-1]]:
            idx = stack.pop()
            result[idx] = nums[i]
        stack.append(i)
    return result

# Example usage:
print(nextGreaterElement([4, 5, 2, 25])) # Output:
#   ↪ [5, 25, 25, -1]
print(nextGreaterElement([13, 7, 6, 12])) # Output:
#   ↪ [-1, 12, 12, -1]
```

Problem 4.6 Asteroid Collision

Problem Description:

We are given an array `asteroids` of integers representing asteroids in a row.

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.

This problem involves simulating collisions using a stack.

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:** [5, 10, -5]
- **Output:** [5, 10]
- **Explanation:** The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

Example 2:

- **Input:** [8, -8]
- **Output:** []
- **Explanation:** The 8 and -8 collide exploding each other.

Example 3:

- **Input:** [10, 2, -5]
- **Output:** [10]
- **Explanation:** The 2 and -5 collide resulting in -5. The 10 and -5 collide resulting in 10.

Solution Overview:

Use a stack to simulate the collisions:

- Iterate through each asteroid in the array.
- For each asteroid:
 - If the stack is empty or the current asteroid is moving right (positive), push it onto the stack.
 - If the current asteroid is moving left (negative):
 - * Check for collisions with the top of the stack (which would be moving right).
 - * While there is a collision:
 - If the top asteroid is smaller, pop it and continue checking.
 - If they are equal in size, pop the top asteroid and stop checking.
 - If the top asteroid is larger, the current asteroid explodes; stop checking.

Code Implementation:

```

def asteroidCollision(asteroids):
    stack = []
    for a in asteroids:
        while stack and a < 0 < stack[-1]:
            if stack[-1] < -a:
                stack.pop()
                continue
            elif stack[-1] == -a:
                stack.pop()
                break
            else:
                break
        else:
            stack.append(a)
    return stack

# Example usage:
print(asteroidCollision([5, 10, -5]))    # Output: [5, 10]
print(asteroidCollision([8, -8]))          # Output: []
print(asteroidCollision([10, 2, -5]))      # Output: [10]

```

Problem 4.7 Basic Calculator

Problem Description: Implement a basic calculator to evaluate a simple expression string.

The expression string may contain open '(' and closing parentheses ')', the plus '+' or minus sign '-', non-negative integers, and empty spaces ' '. The expression string represents a valid arithmetic expression.

You may assume that the given expression is always valid.

Example 1:

- **Input:** "1 + 1"
- **Output:** 2

Example 2:

- **Input:** " 2-1 + 2 "
- **Output:** 3

Example 3:

This problem involves parsing and evaluating arithmetic expressions with parentheses using stacks.

- **Input:** "(1+(4+5+2)-3)+(6+8)"
- **Output:** 23

Solution Overview: Use a stack to evaluate the expression by handling numbers, signs, and parentheses. Iterate through the string character by character:

- If the character is a digit, build the current number.
- If the character is '+', add the current number to the result with the positive sign.
- If the character is '-', subtract the current number from the result.
- If the character is '(', push the current result and sign onto the stack and reset them.
- If the character is ')', compute the result inside the parentheses and combine it with the top values from the stack.

```
def calculate(s):
    stack = []
    result = 0
    number = 0
    sign = 1 # 1 for '+', -1 for '-'
    i = 0
    while i < len(s):
        char = s[i]
        if char.isdigit():
            number = number * 10 + int(char)
        elif char == '+':
            result += sign * number
            number = 0
            sign = 1
        elif char == '-':
            result += sign * number
            number = 0
            sign = -1
        elif char == '(':
            # Push current result and sign onto the stack
            stack.append(result)
            stack.append(sign)
            # Reset result and sign
            result = 0
            sign = 1
        elif char == ')':
            result += sign * number
            number = 0
            # Apply the sign before the parentheses
            result *= stack.pop()
            # Add to the result before the parentheses
            result += stack.pop()
```

```

    i += 1
    if number != 0:
        result += sign * number
    return result

# Example usage:
print(calculate("1 + 1"))           # Output: 2
print(calculate(" 2-1 + 2 "))       # Output: 3
print(calculate("(1+(4+5+2)-3)+(6+8)")) # Output: 23

```

Problem 4.8 Basic Calculator II

Problem Description: Implement a basic calculator to evaluate a simple expression string.

This problem extends the basic calculator to include multiplication and division.

The expression string contains only non-negative integers, '+', '-', '*', '/', and empty spaces ' '. The integer division should truncate toward zero.

You may assume that the given expression is always valid.

Example 1:

- **Input:** "3+2*2"
- **Output:** 7

Example 2:

- **Input:** " 3/2 "
- **Output:** 1

Example 3:

- **Input:** " 3+5 / 2 "
- **Output:** 5

Solution Overview: Use a stack to handle multiplication and division immediately, while addition and subtraction are deferred. Iterate through the string:

- Build the current number if the character is a digit.
- If an operator or the end of the string is reached:
 - If the previous operator is '+', push the number onto the stack.

- If ‘-’, push the negative number onto the stack.
- If ‘*’, pop the stack, multiply, and push the result.
- If ‘/’, pop the stack, divide, and push the result (truncate toward zero).
- Update the previous operator.
- Reset the current number.

At the end, sum all numbers in the stack for the final result.

```
def calculate(s):
    if not s:
        return 0
    s = s.replace(' ', '')
    stack = []
    num = 0
    prev_op = '+'
    i = 0
    while i < len(s):
        char = s[i]
        if char.isdigit():
            num = num * 10 + int(char)
        if char in '+-*/' or i == len(s) - 1:
            if prev_op == '+':
                stack.append(num)
            elif prev_op == '-':
                stack.append(-num)
            elif prev_op == '*':
                stack.append(stack.pop() * num)
            elif prev_op == '/':
                temp = stack.pop()
                if temp < 0:
                    stack.append(-(-temp // num))
                else:
                    stack.append(temp // num)
            prev_op = char
            num = 0
        i += 1
    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
```

Problem 4.9 Largest Rectangle in Histogram

Problem Description:

This problem uses a stack to find the largest rectangle in a histogram.

Given an array of integers `heights` representing the histogram's bar heights 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 largest rectangle has an area of 10 units, formed by the bars of heights 5 and 6.

Example 2:

- **Input:** `heights = [2, 4]`
- **Output:** 4

Solution Overview:

Use a stack to keep track of indices of bars in the histogram. Iterate through the array:

- For each bar at index i :
 - If the stack is empty or the current bar is taller than the bar at the stack's top index, push i onto the stack.
 - Otherwise, while the current bar is shorter than the bar at the stack's top index:
 - * Pop the top index from the stack.
 - * Calculate the area with the popped bar as the smallest bar.
 - * Update the maximum area if necessary.
 - Repeat until the stack is empty or the current bar is taller than the bar at the stack's top index.

After processing all bars, pop any remaining bars from the stack and calculate areas similarly.

Code Implementation:

```
def largestRectangleArea(heights):
    stack = [] # Stack to store indices
    max_area = 0
    i = 0
    while i <= len(heights):
        # Use 0 height for the bar beyond the last one
```

```

        h = 0 if i == len(heights) else heights[i]
        if not stack or h >= heights[stack[-1]]:
            stack.append(i)
            i += 1
        else:
            top = stack.pop()
            width = i if not stack else i - stack[-1] - 1
            area = heights[top] * width
            max_area = max(max_area, area)
    return max_area

# Example usage:
print(largestRectangleArea([2,1,5,6,2,3])) # Output: 10
print(largestRectangleArea([2,4]))          # Output: 4

```

Problem 4.10 Trapping Rain Water

Problem Description:

Given an array `height` 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

Example 2:

- **Input:** `height = [4,2,0,3,2,5]`
- **Output:** 9

Solution Overview:

Use a stack to keep track of the bars that are bounded by taller bars and hence can trap water. Iterate through the elevation map:

- For each bar at index i :
 - While the stack is not empty and $height[i] > height[stack[-1]]$:
 - * Pop the top of the stack as `bottom`.
 - * If the stack becomes empty, break.
 - * Calculate the distance between the current index and the new top of the stack minus one.

This problem calculates the total trapped rainwater using a stack-based approach.

- * Find the bounded height by taking the minimum of $height[i]$ and $height[stack[-1]]$ minus $height[bottom]$.
- * Add the trapped water to the total.
- Push i onto the stack.

Code Implementation:

```
def trap(height):
    stack = []
    water = 0
    i = 0
    while i < len(height):
        while stack and height[i] > height[stack[-1]]:
            bottom = stack.pop()
            if not stack:
                break
            distance = i - stack[-1] - 1
            bounded_height = min(height[i], height[stack
                ↪ [-1]]) - height[bottom]
            water += distance * bounded_height
        stack.append(i)
        i += 1
    return water

# Example usage:
print(trap([0,1,0,2,1,0,1,3,2,1,2,1])) # Output: 6
print(trap([4,2,0,3,2,5]))             # Output: 9
```


Chapter 5

Queues

A **queue** is a linear collection of elements that are maintained in a sequence and can be modified by the addition of elements at one end of the sequence (enqueue operation) and the removal of elements from the other end (dequeue operation). This First-In-First-Out (FIFO) property makes queues essential in various computing scenarios where order of processing is crucial.¹

¹ Understanding queues is fundamental for designing systems that require ordered task management and processing.

5.1 Introduction

Queues are fundamental data structures in computer science, analogous to real-world queues such as lines at a supermarket or tasks waiting to be processed. They provide an efficient way to manage ordered data, ensuring that elements are processed in the exact sequence they arrive. Understanding queues is essential for implementing algorithms that require ordered processing, such as breadth-first search (BFS) in graph traversal, task scheduling, and handling asynchronous data streams.²

² Queues help maintain order and fairness in processing tasks, which is critical in both software and hardware systems.

5.2 Types of Queues

Queues come in various forms, each tailored to specific use cases:

1. Simple Queue

A basic FIFO queue where elements are added at the rear (enqueue) and removed from the front (dequeue).

2. Circular Queue

A variation of the simple queue where the end of the queue wraps around to the beginning, effectively utilizing the available space and preventing overflow in fixed-size implementations.³

³ Circular queues optimize space usage by reusing vacant positions created by dequeued elements.

3. Priority Queue

A queue where each element has a priority assigned to it. Elements with higher priorities are dequeued before those with lower priorities, regardless of their insertion order.

4. Double-Ended Queue (Deque)

A queue that allows insertion and removal of elements from both ends, supporting both FIFO and LIFO (Last-In-First-Out) operations.⁴

⁴ Deques provide greater flexibility in managing elements, enabling more complex data manipulations.

5.3 Basic Operations

Queues support a set of fundamental operations:

1. Enqueue

Description: Add an element to the rear of the queue.

2. Dequeue

Description: Remove and return the element from the front of the queue.⁵

⁵ Dequeue operations must handle cases where the queue is empty to prevent errors.

3. Peek/Front

Description: Retrieve the front element without removing it from the queue.

4. IsEmpty

Description: Check whether the queue is empty.

5. Size

Description: Return the number of elements in the queue.

5.4 Implementations of Queues

Queues can be implemented using various underlying data structures, each with its own advantages and trade-offs.

1. Array-Based Implementation

Utilizes a fixed-size or dynamically resizing array to store elements. While offering $O(1)$ access time, it can suffer from inefficiencies due to shifting elements during dequeue operations unless implemented as a circular array.⁶

⁶ Circular arrays prevent the overhead of shifting by treating the array as a ring buffer.

2. Linked List-Based Implementation

Employs a singly or doubly linked list where each node points to the next (and possibly previous) node. This implementation allows for efficient $O(1)$ enqueue and dequeue operations without the need for shifting elements.

3. Using Built-In Data Structures

Many programming languages provide built-in data structures that can be leveraged to implement queues efficiently. For example, Python's ‘collections.deque’ offers an optimized double-ended queue with $O(1)$ time complexity for append and pop operations from both ends.⁷

⁷ Leveraging built-in structures can simplify implementation and improve performance due to optimized underlying code.

5.5 Complexities of Operations

Understanding the time and space complexities of queue operations is crucial for selecting the appropriate implementation based on the application's requirements.

Operation	Array-Based Queue	Linked List-Based Queue	Built-In (e.g., deque)
Enqueue	$O(1)$ amortized	$O(1)$	$O(1)$
Dequeue	$O(n)$ (unless circular)	$O(1)$	$O(1)$
Peek/Front	$O(1)$	$O(1)$	$O(1)$
IsEmpty	$O(1)$	$O(1)$	$O(1)$
Size	$O(1)$	$O(1)$	$O(1)$

5.6 Python Implementations

Below are examples of different ways to implement a queue in Python.

1. Using a List

A simple but less efficient approach due to $O(n)$ dequeue operations.

```
class Queue:
    def __init__(self):
        self.queue = []

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.isEmpty():
            return self.queue.pop(0)
        raise IndexError("Dequeue from empty queue")

    def peek(self):
        if not self.isEmpty():
            return self.queue[0]
        raise IndexError("Peek from empty queue")

    def isEmpty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)

# Example usage:
q = Queue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.dequeue()) # Output: 1
print(q.peek())   # Output: 2
```

8

⁸ Using a list for queues can lead to performance issues for large datasets due to the linear time complexity of dequeue operations.

2. Using a Linked List

Efficient $O(1)$ enqueue and dequeue operations.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class LinkedListQueue:
    def __init__(self):
        self.head = None
        self.tail = None
```

```

def enqueue(self, item):
    new_node = Node(item)
    if self.tail:
        self.tail.next = new_node
    self.tail = new_node
    if not self.head:
        self.head = new_node

def dequeue(self):
    if self.isEmpty():
        raise IndexError("Dequeue from empty queue")
    removed = self.head.data
    self.head = self.head.next
    if not self.head:
        self.tail = None
    return removed

def peek(self):
    if self.isEmpty():
        raise IndexError("Peek from empty queue")
    return self.head.data

def isEmpty(self):
    return self.head is None

def size(self):
    count = 0
    current = self.head
    while current:
        count +=1
        current = current.next
    return count

# Example usage:
llq = LinkedListQueue()
llq.enqueue(1)
llq.enqueue(2)
llq.enqueue(3)
print(llq.dequeue()) # Output: 1
print(llq.peek())   # Output: 2

```

9

⁹ Linked lists allow for dynamic memory usage, making them suitable for queues with unpredictable sizes.

3. Using collections.deque

Leveraging Python's optimized double-ended queue.

```
from collections import deque
```

```

class DequeQueue:
    def __init__(self):
        self.queue = deque()

    def enqueue(self, item):
        self.queue.append(item)

    def dequeue(self):
        if not self.isEmpty():
            return self.queue.popleft()
        raise IndexError("Dequeue from empty queue")

    def peek(self):
        if not self.isEmpty():
            return self.queue[0]
        raise IndexError("Peek from empty queue")

    def isEmpty(self):
        return len(self.queue) == 0

    def size(self):
        return len(self.queue)

# Example usage:
dq = DequeQueue()
dq.enqueue('a')
dq.enqueue('b')
dq.enqueue('c')
print(dq.dequeue()) # Output: 'a'
print(dq.peek())    # Output: 'b'

```

10

¹⁰ Using `deque` provides efficient $O(1)$ time complexity for both enqueue and dequeue operations, making it ideal for performance-critical applications.

5.7 Applications of Queues

Queues are employed in a variety of applications where order and sequence are paramount:

1. Breadth-First Search (BFS)

In graph and tree traversals, BFS uses a queue to explore nodes level by level, ensuring that nodes are processed in the order they are discovered.¹¹

¹¹ BFS is essential for finding the shortest path in unweighted graphs.

2. Task Scheduling

Operating systems use queues to manage tasks, processes, and threads, ensuring that tasks are handled in the order they arrive.¹²

¹² Efficient task scheduling improves system responsiveness and resource utilization.

3. Buffering

Queues are used in buffering data streams, such as in IO buffers, network packet handling, and print queues, to manage data flow efficiently.¹³

¹³ Buffers prevent data loss and manage data bursts by regulating the flow between producers and consumers.

4. Real-Time Data Processing

In scenarios like live data feeds, queues help manage incoming data in a controlled and sequential manner.¹⁴

¹⁴ Queues ensure that data is processed in the order it arrives, maintaining consistency and reliability.

5. Simulation Systems

Queues model real-world systems like customer service lines, traffic management, and event scheduling, allowing for analysis and optimization.¹⁵

¹⁵ Simulations using queues help in predicting system behavior under various conditions.

5.8 Important Queue Problems

Queues are not only fundamental in theory but also play a significant role in solving practical problems. Mastering these problems enhances one's ability to implement efficient algorithms and design robust systems. Below is a curated list of important queue-related problems, each accompanied by a detailed explanation and a high-level overview of the solution approach.

Problem 5.1 Implement Stack using Queues

Problem Description: Implement a last-in-first-out (LIFO) stack using only two first-in-first-out (FIFO) queues. The stack should support the following operations:

- `push(x)`: Push element `x` onto the stack.
- `pop()`: Removes the element on top of the stack and returns it.
- `top()`: Returns the element on top of the stack without removing it.
- `empty()`: Returns `true` if the stack is empty, `false` otherwise.

This problem demonstrates how to mimic stack behavior using queue operations, highlighting the differences between LIFO and FIFO structures.

Notes:

- You may use only standard queue operations (`enqueue`, `dequeue`, `isEmpty`, and `size`).
- All operations should be implemented using $O(n)$ time complexity, where n is the number of elements in the stack.
- You should not use any other data structures, such as arrays or lists.

Solution Overview: There are two primary approaches to implement a stack using queues:

Understanding how to implement one data structure using another deepens comprehension of their fundamental behaviors.

1. Making push Costly:

- Use two queues, `q1` and `q2`.
- When pushing an element, enqueue it into `q2`.
- Dequeue all elements from `q1` and enqueue them into `q2`.
- Swap the names of `q1` and `q2`.
- Now, `q1` has the new element at the front, maintaining stack order.

2. Making pop Costly:

- Use a single queue.
- For `pop` and `top`, rotate the queue elements until the last inserted element is at the front.

We will focus on the first approach, where the `push` operation is more time-consuming, but `pop` and `top` operations are efficient.

Implementation Details: Here's an implementation using two queues in Python:

```
from collections import deque

class MyStack:
    def __init__(self):
        self.q1 = deque()
        self.q2 = deque()

    def push(self, x):
        # Enqueue element to q2
        self.q2.append(x)
        # Move all elements from q1 to q2
        while self.q1:
            self.q2.append(self.q1.popleft())
        # Swap q1 and q2
        self.q1, self.q2 = self.q2, self.q1

    def pop(self):
```

```

    if self.empty():
        raise Exception("Stack Underflow")
    return self.q1.popleft()

def top(self):
    if self.empty():
        raise Exception("Stack is empty")
    return self.q1[0]

def empty(self):
    return not self.q1

# Example usage:
stack = MyStack()
stack.push(1)
stack.push(2)
print(stack.top())      # Output: 2
print(stack.pop())      # Output: 2
print(stack.empty())    # Output: False

```

Complexities:• **Time Complexity:**

- push: $O(n)$, due to moving all elements from q1 to q2.
- pop: $O(1)$, direct dequeue from q1.
- top: $O(1)$, accessing the front of q1.
- empty: $O(1)$, simple check on q1.

• **Space Complexity:** $O(n)$, where n is the number of elements in the stack.

Alternative Approach: Making the pop operation costly can be beneficial if push operations are more frequent.

Swapping queues after each push ensures the newest element is always at the front of q1.

Making push costly optimizes pop and top, which are often called more frequently.

Implementation with Costly pop:

```

class MyStack:
    def __init__(self):
        self.q = deque()

    def push(self, x):
        self.q.append(x)

    def pop(self):
        if self.empty():
            raise Exception("Stack Underflow")
        # Rotate the queue to bring the last element to the front
        size = len(self.q)
        for _ in range(size - 1):

```

```

        self.q.append(self.q.popleft())
    return self.q.popleft()

def top(self):
    if self.empty():
        raise Exception("Stack is empty")
    # Rotate the queue to get the last element
    size = len(self.q)
    for _ in range(size - 1):
        self.q.append(self.q.popleft())
    result = self.q[0]
    self.q.append(self.q.popleft()) # Restore the queue
    return result

def empty(self):
    return not self.q

# Example usage:
stack = MyStack()
stack.push(1)
stack.push(2)
print(stack.top())      # Output: 2
print(stack.pop())      # Output: 2
print(stack.empty())    # Output: False

```

Complexities:**• Time Complexity:**

- push: $O(1)$, straightforward enqueue.
- pop: $O(n)$, rotating elements to access the last one.
- top: $O(n)$, similar to pop, but with an extra step to restore the queue.
- empty: $O(1)$.

• Space Complexity: $O(n)$.**Why This Approach?**

The first approach is generally preferred when `pop` and `top` operations are expected to be called more frequently than `push`. It provides faster retrieval of the top element, which is critical in stack operations.

Corner Cases to Test:

- **Empty Stack Operations:** Ensure that calling `pop` or `top` on an empty stack raises appropriate exceptions or handles the case gracefully.
- **Single Element Stack:** Test the behavior when only one element is present in the stack.

This method keeps push operations at $O(1)$ time complexity.

Choose the approach based on the frequency of `push` vs. `pop` operations in your use case.

- **Sequence of Operations:** Perform a series of push, pop, and top operations to verify the stack maintains correct order.
- **Stress Test:** Test with a large number of elements to assess performance and correctness.

Similar Problems:

- **Implement Queue using Stacks:** Reverse of this problem, where you implement a queue using stack operations.
- **Design a Stack with Increment Operation:** Enhance the stack with an operation that increments the bottom k elements by a given value.
- **Min Stack:** Design a stack that supports retrieving the minimum element in constant time.

Testing various scenarios ensures robustness and reliability of the stack implementation.

Conclusion:

Implementing a stack using queues challenges the understanding of fundamental data structures and their operations. It requires manipulating the order of elements to emulate LIFO behavior using FIFO mechanisms. Mastery of such transformations enhances problem-solving skills and prepares one for more complex algorithmic challenges in computer science.

Exploring similar problems deepens understanding of data structure transformations.

Problem 5.2 Design Hit Counter

Problem Description: Design a hit counter that counts the number of hits received in the past 5 minutes (300 seconds). Implement the following operations:

- `hit(timestamp)`: Record a hit at the given timestamp.
- `getHits(timestamp)`: Return the number of hits in the past 5 minutes from the given timestamp.

Understanding these concepts is crucial for algorithm design and optimization.

Hit counters are vital for tracking user interactions and monitoring system metrics in real-time applications.

Notes:

- Each function call is guaranteed to be monotonically increasing in terms of timestamp (i.e., the timestamps are in non-decreasing order).
- Timestamps are in seconds.
- You may assume that the system will not receive more than 10^4 hits per second.

Solution Overview: An efficient approach uses a queue to store the timestamps of hits within the last 5 minutes. For each `hit` operation, the current timestamp is

Efficient time-based data structures are essential for handling high-frequency events in real-time systems.

enqueued. For each `getHits` operation, we dequeue all timestamps that are older than 300 seconds relative to the current timestamp. The number of elements remaining in the queue represents the number of hits in the past 5 minutes.

To optimize the space complexity, we can combine timestamps that are the same into a single entry with a count of hits at that timestamp. Alternatively, we can use an array or a fixed-size circular buffer to store counts per second.

Implementation Details: Here's an implementation using a queue (or deque) in Python:

```
from collections import deque

class HitCounter:
    def __init__(self):
        self.hits = deque()

    def hit(self, timestamp: int) -> None:
        self.hits.append(timestamp)

    def getHits(self, timestamp: int) -> int:
        while self.hits and self.hits[0] <= timestamp - 300:
            self.hits.popleft()
        return len(self.hits)

# Example usage:
# Initialize the hit counter
counter = HitCounter()
counter.hit(1)
counter.hit(2)
counter.hit(300)
print(counter.getHits(300))    # Output: 3
print(counter.getHits(301))    # Output: 2
```

Complexities:

- **Time Complexity:**

- `hit`: $O(1)$
- `getHits`: $O(n)$ in the worst case, where n is the number of hits in the past 5 minutes. However, since the maximum number of hits per second is limited, the operation is efficient on average.

- **Space Complexity:** $O(n)$, where n is the number of hits in the past 5 minutes.

Alternative Approach:

To achieve $O(1)$ time complexity for both `hit` and `getHits`, we can use a fixed-size array to store the counts per second.

Using a deque allows efficient addition and removal from both ends, which is ideal for sliding window problems.

By limiting the data stored to recent hits, we optimize memory usage and ensure scalability.

- Initialize an array (or list) of size 300 (since 5 minutes = 300 seconds).
- Each index in the array represents a timestamp modulo 300.
- For each `hit`, update the count at the index corresponding to $(\text{timestamp} \bmod 300)$.
- For each `getHits`, sum up the counts in the array where the timestamps are within the 5-minute window.

Implementation Using Fixed-Size Array:

```
class HitCounter:
    def __init__(self):
        self.times = [0] * 300
        self.counts = [0] * 300

    def hit(self, timestamp: int) -> None:
        idx = timestamp % 300
        if self.times[idx] != timestamp:
            self.times[idx] = timestamp
            self.counts[idx] = 1
        else:
            self.counts[idx] += 1

    def getHits(self, timestamp: int) -> int:
        total = 0
        for i in range(300):
            if timestamp - self.times[i] < 300:
                total += self.counts[i]
        return total

# Example usage:
# Initialize the hit counter
counter = HitCounter()
counter.hit(1)
counter.hit(2)
counter.hit(300)
print(counter.getHits(300))    # Output: 3
print(counter.getHits(301))    # Output: 3
```

Complexities of Alternative Approach:

Using a fixed-size array ensures constant-time operations and bounded space complexity.

- **Time Complexity:**
 - `hit`: $O(1)$
 - `getHits`: $O(1)$
- **Space Complexity:** $O(1)$, since the size of the arrays is fixed at 300.

Why This Approach?

Using a fixed-size array optimizes both time and space complexities. It eliminates the need to store every single hit timestamp, which can be memory-intensive if the hit rate is high. The modulo operation effectively cycles through the array, overwriting old data that is no longer within the 5-minute window.

Constant-time operations are ideal for high-throughput systems requiring minimal latency.

Corner Cases to Test:

- **No Hits:** Ensure that `getHits` returns 0 when there are no hits in the past 5 minutes.
- **Hits Exactly 5 Minutes Ago:** Verify that hits that occurred exactly 300 seconds ago are not counted.
- **High Frequency of Hits:** Test the system with the maximum allowed hits per second to assess performance.
- **Hits with Same Timestamps:** Ensure that multiple hits at the same timestamp are counted correctly.
- **Continuous Operation:** Simulate continuous operation over a long period to verify that the data structure handles timestamp wrapping correctly.

Conclusion:

Designing an efficient hit counter requires balancing time and space complexities while handling high-frequency data. By using appropriate data structures like queues or fixed-size arrays, we can achieve constant-time operations and bounded memory usage. Mastery of such problems enhances understanding of time-based data management, which is crucial in real-time analytics, monitoring systems, and rate-limiting applications.

Testing edge cases ensures the reliability and robustness of the hit counter implementation.

Problem 5.3 Number of Recent Calls

The **Number of Recent Calls** problem requires designing a class named `RecentCounter` to track the number of recent requests within a time frame of 3000 milliseconds. A request is represented by a single integer which indicates the time of the request in milliseconds. The class should support the following operations:

Efficient time-based algorithms are essential in scalable system designs.

- `RecentCounter()`: Initializes the `RecentCounter` object.
- `ping(int t)`: Records a new request at time `t` and returns the number of requests that occurred in the past 3000 milliseconds, including the request just made at time `t`.

Designing efficient data structures to handle real-time data is crucial in many applications, such as monitoring systems and network traffic analysis.

Problem Statement

LeetCode link: 933. Number of Recent Calls

[LeetCode Link]

[GeeksForGeeks Link]

[HackerRank Link]

[CodeSignal Link]

[InterviewBit Link]

[Educative Link]

[Codewars Link]

Algorithmic Approach

To keep track of the requests, we can use a **queue** to store the timestamps of all requests. When a new request comes in at time t , we add it to the queue. Then, we remove all requests from the front of the queue that are older than $t - 3000$ milliseconds, because we only need to consider requests within the last 3000 milliseconds. Finally, we return the size of the queue, which represents the number of recent requests within the required time frame.

Using a queue ensures that we only retain relevant requests, optimizing both time and space complexities.

Complexities

- **Time Complexity:** The time complexity of the `ping` operation is $O(1)$ amortized. Each request is added to the queue exactly once and removed at most once.
- **Space Complexity:** The space complexity is $O(n)$, where n is the maximum number of requests that can appear within the time frame of 3000 milliseconds at any given time.

Python Implementation

Below is the complete Python code implementation for the `RecentCounter` class:

```
from collections import deque

class RecentCounter:

    def __init__(self):
        self.requests = deque()

    def ping(self, t: int) -> int:
        self.requests.append(t)
        while self.requests and self.requests[0] < t - 3000:
            self.requests.popleft()
        return len(self.requests)

# Example Usage:
# obj = RecentCounter()
# param_1 = obj.ping(t)

# Test Cases:
# Example 1:
# obj = RecentCounter()
# print(obj.ping(1)) # Output: 1
# print(obj.ping(100)) # Output: 2
# print(obj.ping(3001))# Output: 3
# print(obj.ping(3002))# Output: 3

# Example 2:
# obj = RecentCounter()
# print(obj.ping(0)) # Output: 1
# print(obj.ping(3000))# Output: 2
# print(obj.ping(3001))# Output: 3
```

Implementing the `RecentCounter` class using a queue ensures efficient tracking of recent requests with optimal time and space usage.

The implementation utilizes Python's 'deque' from the 'collections' module to efficiently add and remove timestamps. The 'ping' method appends the current timestamp to the queue and then removes any timestamps that are older than ' $t - 3000$ ' milliseconds. Finally, it returns the number of timestamps remaining in the queue, representing the number of recent calls.

Explanation

The `RecentCounter` class maintains a queue of timestamps, each representing a request. The `ping` method performs the following steps:

1. **Append Current Request:**

- The current timestamp t is appended to the queue.

2. Remove Outdated Requests:

- Continuously remove requests from the front of the queue that are older than $t - 3000$ milliseconds.

3. Return Recent Request Count:

- The length of the queue after removals represents the number of recent requests within the last 3000 milliseconds.

Why This Approach

This approach is chosen due to its **efficiency** and **simplicity**. By using a queue, we ensure that only relevant requests (those within the last 3000 milliseconds) are stored, optimizing both time and space. The operations of adding to the queue and removing outdated requests are both efficient, with each operation taking constant time on average.

Alternative Approaches

An alternative approach could involve keeping an array or list of all requests and iterating over it every time to count how many fall within the last 3000 milliseconds. However, this would result in a higher time complexity for the `ping` method, as it would require iterating over potentially many more elements than necessary.

- **Pros:** Simpler to implement using basic list operations.
- **Cons:** Inefficient for large numbers of requests, leading to increased time complexity.

Another alternative could involve using a sliding window technique with two pointers to track the start and end of the relevant time frame. While this can also achieve similar time complexities, using a queue is more intuitive and straightforward in this context.

Sliding window techniques are powerful for handling range-based queries efficiently.

Similar Problems to This One

There are several other problems that involve processing requests over a sliding time window or maintaining counts within specific constraints, such as:

- Design Hit Counter

- Sliding Window Maximum
- Log System

Things to Keep in Mind and Tricks

- **Efficient Data Structures:** Utilizing appropriate data structures like queues can significantly optimize the performance of your solution.
- **Sliding Window Concept:** Understanding the sliding window concept helps in solving a variety of range-based and time-constrained problems.
- **Amortized Analysis:** Recognize that although some operations might take longer individually, the overall time complexity remains optimal when considering all operations together.
- **Edge Case Handling:** Always account for edge cases, such as the first request or requests that exactly hit the boundary of the time frame.

Corner and Special Cases to Test When Implementing

When implementing the `RecentCounter` class, it is crucial to test the following edge cases to ensure robustness:

- **First Ping:** Ensure that the queue correctly handles the first request.
- **Ping at Boundary Time:** Test a ping that is exactly at $t = 3000$ milliseconds to verify inclusivity.
- **Multiple Pings with Same Timestamp:** Although the problem states that each ping call has a strictly larger t , ensure that the implementation can handle pings with the same timestamp if extended.
- **Rapid Succession of Pings:** Simulate a scenario with a large number of pings in a short period to test the efficiency of the queue operations.
- **Long Duration Between Pings:** Test with pings that are spaced out by more than 3000 milliseconds to ensure old requests are properly removed.
- **Maximum Input Values:** Verify that the implementation can handle the maximum possible values of t as defined by the problem constraints.

Chapter 6

Hash Tables

Chapter 7

Hashing and Hash Tables

Hashing and hash tables are cornerstone concepts in computer science, providing a powerful and efficient way to store and retrieve data. Hashing maps data to fixed-size values, called hash codes, which can be used as indices in a hash table. This mechanism allows for fast lookups, insertions, and deletions in constant average time, $O(1)$. The versatility and efficiency of hashing make it a fundamental tool for solving a wide variety of problems, from data retrieval to detecting duplicates and implementing associative arrays.

In this chapter, we will explore the principles behind hashing, understand how hash tables work, and discuss their applications and limitations.

What is Hashing?

Hashing is the process of converting input data (such as a string, number, or object) into a fixed-size hash code using a hash function. The goal is to distribute the input data uniformly across a range of hash codes to minimize collisions¹.

Properties of a Good Hash Function

A good hash function should:

- **Be Deterministic:** The same input should always produce the same hash code.
- **Distribute Uniformly:** Hash codes should be spread evenly across the range to reduce collisions.
- **Be Efficient:** The computation of the hash code should be fast, ideally $O(1)$.
- **Minimize Collisions:** Although collisions are unavoidable, a good hash function reduces their likelihood².

¹ A collision occurs when two different inputs produce the same hash code. Managing collisions effectively is crucial for maintaining the efficiency of a hash table

² Perfect hash functions exist for certain datasets but are impractical for general use cases

What is a Hash Table?

A hash table is a data structure that maps keys to values using hashing. It is implemented as an array where the index for storing a key-value pair is determined by applying a hash function to the key.

Key Operations in a Hash Table

- **Insertion:** Compute the hash code for a key, map it to an index, and store the value at that index.
- **Search:** Compute the hash code for the key and retrieve the value at the corresponding index.
- **Deletion:** Compute the hash code for the key and remove the value at the corresponding index.

Collision Resolution Techniques

Collisions occur when two keys hash to the same index. Hash tables manage collisions using techniques like:

- **Chaining:** Store multiple key-value pairs at the same index using a linked list or a dynamic array³.
- **Open Addressing:** Probe sequentially or using a specific strategy (e.g., linear probing, quadratic probing, or double hashing) to find the next available slot in the table⁴.

³ Chaining is simple and allows for flexible resizing but requires extra memory for the additional structures

⁴ Open addressing is memory-efficient but can suffer from clustering

Applications of Hashing and Hash Tables

Hashing and hash tables are integral to solving a wide variety of problems, including:

- **Duplicate Detection:** Quickly determine if duplicate elements exist in a dataset.
- **Frequency Counting:** Count occurrences of elements in linear time.
- **Anagram Detection:** Use hash tables to compare character frequencies efficiently.
- **Caching:** Implement efficient caching systems using hash tables (e.g., Least Recently Used (LRU) Cache).
- **Hash-Based Data Structures:** Implement sets, dictionaries, and associative arrays.

Advantages of Hash Tables

- **Constant Time Operations:** On average, hash tables provide $O(1)$ time complexity for insertion, search, and deletion.
- **Flexibility:** Hash tables can handle a wide range of key types, including strings, numbers, and composite objects.
- **Dynamic Size:** Many hash table implementations (e.g., Python's dictionaries) dynamically resize to maintain efficiency.

Challenges and Limitations of Hash Tables

- **Collisions:** Poorly chosen hash functions can lead to frequent collisions, degrading performance to $O(n)$.
- **Memory Overhead:** Hash tables require extra space for unused slots or chaining structures.
- **Non-Ordered Data:** Hash tables do not maintain the order of elements, making them unsuitable for problems requiring sorted data.

Topics Covered in This Chapter

This chapter will delve into:

- The theory and implementation of hashing and hash tables.
- Advanced collision resolution techniques and their trade-offs.
- Applications of hash tables in solving real-world problems.
- Challenges and best practices for using hash tables in competitive programming and system design.

Common Problems Solved with Hashing

Examples of problems that leverage hashing include:

- **Two Sum:** Find two numbers in an array that add up to a target sum.
- **Longest Substring Without Repeating Characters:** Use hashing to track characters and efficiently manage the sliding window.
- **Find All Anagrams in a String:** Use frequency counts and hashing for efficient pattern matching.

- **Top K Frequent Elements:** Use a hash map to count frequencies and a heap to retrieve the top k elements.

Conclusion

Hashing and hash tables are indispensable tools in modern computing, enabling fast and efficient solutions to numerous problems. Mastery of these concepts equips you with the ability to tackle challenges in both theoretical computer science and practical applications, from data retrieval to algorithm optimization. In this chapter, we will build a strong foundation in hashing techniques and explore their vast range of applications.

Problem 7.1 Two Sum

The **Two Sum** problem is a classic algorithmic challenge that tests a candidate's ability to work with arrays and hash tables effectively. It focuses on finding two numbers in an array that add up to a specified target, with an emphasis on optimizing time and space complexity.

Problem Statement

Given an array of integers `nums` and an integer `target`, return the indices of the two numbers such that they add up to the target. Assume that there is exactly one solution, and you may not use the same element twice.

Input: - An array `nums` of integers. - An integer `target`.

Output: - A list of two indices $[i, j]$ such that $\text{nums}[i] + \text{nums}[j] = \text{target}$.

Example 1:

Input: `nums = [2, 7, 11, 15]`, `target = 9`

Output: `[0, 1]`

Explanation: $\text{nums}[0] + \text{nums}[1] = 2 + 7 = 9$

Example 2:

Input: `nums = [3, 2, 4]`, `target = 6`

Output: `[1, 2]`

Example 3:

Input: `nums = [3, 3]`, `target = 6`

Output: [0, 1]

Algorithmic Approach

The **Two Sum** problem can be efficiently solved using a hash map to store the complement of each number as we iterate through the array.

1. Initialize an empty hash map to store numbers and their indices.
2. Traverse the array:
 - For each number num, calculate its complement: complement = target – num.
 - Check if the complement exists in the hash map:
 - If yes, return the current index and the index of the complement.
 - Otherwise, add the current number and its index to the hash map.

Key Insight: The hash map allows for constant time $O(1)$ lookups for the complement, making the algorithm highly efficient.

Complexities

- **Time Complexity:** $O(n)$, where n is the length of the array. Each element is processed once, and hash map operations (insert and lookup) are $O(1)$ on average.
- **Space Complexity:** $O(n)$, for the hash map storing at most n elements.

Python Implementation

```
class Solution:
    def twoSum(self, nums: List[int], target: int) -> List[int]:
        num_map = {} # Dictionary to store numbers and their indices

        for i, num in enumerate(nums):
            complement = target - num
            if complement in num_map:
                return [num_map[complement], i]
            num_map[num] = i
```

This implementation initializes an empty dictionary to store numbers and their indices. It iterates through the array, calculating the complement of each number. If the complement is found in the dictionary, the function returns the indices. Otherwise, it adds the current number and its index to the dictionary.

Why This Approach

This approach is chosen for its efficiency, leveraging a hash map to achieve $O(n)$ time complexity. The alternative brute-force method, which involves checking all pairs of numbers, has $O(n^2)$ time complexity and is impractical for large input sizes.

Alternative Approaches

- **Brute-Force:** Iterate through all pairs of numbers and check if their sum equals the target. This method has $O(n^2)$ time complexity.
- **Two-Pointer Technique:** If the array is sorted, use two pointers to find the pair. This approach has $O(n)$ time complexity but requires sorting first, making the overall complexity $O(n \log n)$.

Similar Problems to This One

- **3Sum:** Find all unique triplets in an array that sum to zero.
- **4Sum:** Find all unique quadruplets in an array that sum to a target.
- **Two Sum II:** Given a sorted array, find the indices of two numbers that add up to a target.
- **Subarray Sum Equals K:** Find the number of subarrays that sum to k .

Things to Keep in Mind and Tricks

- Ensure that the problem constraints allow for the modification of the array or the use of additional space.
- Handle edge cases such as arrays with duplicate numbers or very small input sizes.
- Use a hash map to optimize time complexity when working with unsorted arrays.
- For sorted arrays, consider the two-pointer approach if additional space is restricted.

Corner and Special Cases to Test When Writing the Code

- **Duplicate Numbers:** Ensure the algorithm correctly identifies pairs involving duplicate values, e.g., [3, 3] with target 6.
- **Single Element Array:** Test cases with less than two elements, e.g., [1] or an empty array.

- **No Valid Pairs:** Arrays where no two numbers add up to the target, e.g., [1, 2, 3] with target 10.
- **Negative Numbers:** Ensure the algorithm works with arrays containing negative values.

Conclusion

The **Two Sum** problem is a cornerstone question in algorithm design, providing a foundation for mastering hash maps and efficient array traversal techniques. By understanding and implementing the hash map-based approach, you gain insights into optimizing time complexity and handling edge cases effectively. Mastering this problem prepares you to tackle more advanced variations and similar array challenges with confidence.

Problem 7.2 3Sum

The **3Sum** problem is a classic challenge in algorithmic tasks, involving finding all unique triplets in an array that sum up to zero.

Problem Statement

Given an array `nums` of n integers, determine whether there are elements a, b, c in `nums` such that $a + b + c = 0$ and find all unique triplets in the array that give the sum of zero⁵.

⁵ Refer to the LeetCode 3Sum Problem for more details.

Example:

Given array `nums` = $[-1, 0, 1, 2, -1, -4]$,

A solution set is:

$[-1, 0, 1],$
 $[-1, -1, 2]$

6

⁶ This example demonstrates how the algorithm identifies unique triplets that sum to zero, even in the presence of duplicate elements.

Algorithmic Approach

The solution to this problem can be approached by using a sorting-based approach along with the Two Pointers Technique⁷. First, sort the array to make it easier to navigate and to avoid duplicate solutions. Iterate through each element i in the array and for each, apply the Two Pointers Technique to find the other two elements that sum to the negative of i . Manage the two pointers to skip over duplicate values and find the unique triplets that satisfy the condition.

⁷ Sorting the array helps in efficiently managing duplicates and navigating the array with two pointers.

Python Implementation

```

class Solution:
    def threeSum(self, nums):
        """
        Finds all unique triplets in the array which gives the sum of zero.

        Parameters:
        nums (List[int]): The input array of integers.

        Returns:
        List[List[int]]: A list of unique triplets that sum up to zero.
        """
        res = []
        nums.sort()

        for i in range(len(nums)-2):
            if i > 0 and nums[i] == nums[i-1]:
                continue
            left, right = i+1, len(nums)-1
            while left < right:
                current_sum = nums[i] + nums[left] + nums[right]
                if current_sum < 0:
                    left += 1
                elif current_sum > 0:
                    right -= 1
                else:
                    res.append([nums[i], nums[left], nums[right]])
                    while left < right and nums[left] == nums[left+1]:
                        left += 1
                    while left < right and nums[right] == nums[right-1]:
                        right -= 1
                    left += 1
                    right -= 1
            return res

# Example usage:
nums = [-1, 0, 1, 2, -1, -4]
solution = Solution()
print(solution.threeSum(nums))  # Output: [[-1, -1, 2], [-1, 0, 1]]

```

Example Usage and Test Cases

```

# Test case 1: General case
nums = [-1, 0, 1, 2, -1, -4]
print(Solution().threeSum(nums))  # Output: [[-1, -1, 2],
                                ↪ [-1, 0, 1]]

# Test case 2: No triplet sums to zero

```

```

nums = [1, 2, 3, 4]
print(Solution().threeSum(nums)) # Output: []

# Test case 3: Multiple triplets with duplicates
nums = [-2, 0, 0, 2, 2]
print(Solution().threeSum(nums)) # Output: [[-2, 0, 2]]

# Test case 4: All elements are zero
nums = [0, 0, 0, 0]
print(Solution().threeSum(nums)) # Output: [[0, 0, 0]]

# Test case 5: Mixed positive and negative numbers
nums = [-4, -1, -1, 0, 1, 2]
print(Solution().threeSum(nums)) # Output: [[-1, -1, 2],
    ↪ [-1, 0, 1]]

```

Why This Approach

The Two Pointers Technique combined with a sorting-based approach is chosen for its **efficiency and simplicity**. By leveraging the sorted nature of the array, the algorithm avoids the need for nested loops, reducing the time complexity from $O(n^3)$ in a brute-force approach to $O(n^2)$ ⁸. Additionally, this method ensures that each element is processed only once, making it highly suitable for large datasets⁹.

⁸ Linearithmic sorting followed by quadratic traversal results in overall $O(n^2)$ time complexity

⁹ Single-pass algorithms are preferable for handling large inputs efficiently

Alternative Approaches

An alternative brute-force approach is to use three nested loops to check every triplet, but this would result in a time complexity of $O(n^3)$ and is not efficient for larger arrays¹⁰. Other approaches may involve using a hash set to check for complements¹¹, but care must be taken to still avoid duplicates.

¹⁰ Such approaches are impractical for large-scale data due to their high time complexity

¹¹ Hash-based solutions can offer linear time complexity but may require additional space and careful handling of duplicates

Similar Problems

Similar problems include:

- **2Sum:** Find pairs that sum up to a target value¹².
- **4Sum:** Find quadruplets that sum up to a target value¹³.
- **3Sum Closest:** Find the triplet with a sum closest to a target value¹⁴.
- **3Sum Smaller:** Count the number of triplets with a sum smaller than a target¹⁵.

¹² A simpler version of the problem focusing on pairs instead of triplets

¹³ An extension of the 3Sum problem requiring additional pointer management

¹⁴ Requires tracking the closest sum while iterating

¹⁵ Involves similar traversal logic with conditional counting

These problems can often be approached with similar sorting and two-pointer techniques, or hashing strategies¹⁶.

¹⁶ Understanding the core technique facilitates tackling a variety of related challenges

Things to Keep in Mind and Tricks

- **Handling Duplicates:** It's crucial to handle duplicates carefully to ensure that only unique triplets are included in the result¹⁷.
- **Sorted Array Advantage:** Sorting the array simplifies the process of finding triplets and managing pointers¹⁸.
- **Pointer Movement Logic:** Clearly define the conditions under which each pointer should move to ensure optimal traversal¹⁹.
- **Edge Cases:** Always consider edge cases such as arrays with all positive or all negative numbers, and arrays with insufficient elements²⁰.
- **Space Optimization:** The two pointers technique allows for solving the problem without additional space²¹.

¹⁷ Skipping over duplicate elements during traversal prevents redundant triplet entries

¹⁸ Sorted arrays allow for predictable pointer movements based on sum comparisons

¹⁹ Proper management of the left and right pointers is key to maintaining efficiency

²⁰ Robust handling of edge cases ensures the algorithm's reliability across diverse inputs

²¹ In-place algorithms are preferable for optimizing space usage, especially with large datasets

Corner and Special Cases to Test When Writing the Code

Special cases include:

- **All Zeroes:** Arrays where all elements are zero²².
- **No Valid Triplets:** Arrays where no three numbers sum to zero²³.
- **Multiple Duplicates:** Arrays with multiple duplicate elements²⁴.
- **Mixed Positive and Negative Numbers:** Arrays containing both positive and negative integers²⁵.
- **Single or Two Elements:** Arrays with fewer than three elements²⁶.

²² Ensures that the algorithm correctly identifies triplets of zeroes without duplication

²³ Checks the algorithm's ability to return an empty list appropriately

²⁴ Verifies that the algorithm skips duplicates correctly to avoid redundant triplets

²⁵ Tests the algorithm's capability to handle a diverse range of input values

²⁶ The algorithm should handle these gracefully, typically returning an empty list

References

LeetCode Problem: ²⁷

²⁷ 3Sum

GeeksforGeeks Article: ²⁸

²⁸ Two Pointers Technique

HackerRank Problem: ²⁹

²⁹ Two Sum

Conclusion

The Two Pointers Technique is an indispensable tool in the array and string manipulation arsenal. By leveraging the inherent order within data structures, it enables the development of efficient and elegant solutions to a wide range of problems³⁰. Mastering this technique not only enhances problem-solving skills but also prepares you for tackling more complex algorithmic challenges effectively.

³⁰ Its applicability spans numerous algorithmic challenges, making it a versatile strategy

Problem 7.3 Group Anagrams

The "Group Anagrams" problem is an interesting question that requires understanding of hash tables and string manipulation. It is a typical interview question that checks the candidate's ability to handle and classify data based on specific rules, in this case, anagrams.

Problem Statement

Given an array of strings `strs`, group the anagrams together. You may return the answer in any order.

Input: - An array of strings `strs`.

Output: - A list of lists, where each inner list contains strings that are anagrams of each other.

Example 1:

```
Input: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
Output: [["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]
```

Example 2:

```
Input: strs = []
Output: [[]]
```

Example 3:

```
Input: strs = ["a"]
Output: [["a"]]
```

Algorithmic Approach

Anagrams can be identified by their character composition. The two most common approaches involve using sorted strings or frequency counts as keys to group the anagrams.

Sorted String as Key

The sorted version of an anagram will always be identical for all its variants. For example, the sorted form of "eat", "tea", and "ate" is "aet".

Steps:

- Initialize a hash map to group strings by their sorted version.
- Iterate through `strs`, and for each string, sort its characters and use the sorted string as the key.
- Append the original string to the list corresponding to its sorted key.
- Return the values of the hash map as the grouped anagrams.

Complexity:

- **Time Complexity:** $O(n \cdot k \log k)$, where n is the number of strings and k is the average length of a string. Sorting each string takes $O(k \log k)$.
- **Space Complexity:** $O(n \cdot k)$, for the hash map and the grouped results.

Frequency Count as Key

Instead of sorting, use a frequency count of characters as the key. For example, the frequency counts of "eat" and "tea" are identical: [1, 0, 0, ..., 1, 1, 0, ...].

Steps:

- Initialize a hash map to group strings by their frequency count.
- Iterate through `strs`, and for each string, compute a frequency count for its characters.
- Use the frequency count tuple as the key in the hash map.
- Append the original string to the list corresponding to its frequency count key.
- Return the values of the hash map as the grouped anagrams.

Complexity:

- **Time Complexity:** $O(n \cdot k)$, where n is the number of strings and k is the average length of a string. Computing the frequency count takes $O(k)$.
- **Space Complexity:** $O(n \cdot k)$, for the hash map and the grouped results.

Python Implementation

Below is the implementation using the frequency count as the key:

```

from collections import defaultdict
from typing import List

def groupAnagrams(strs: List[str]) -> List[List[str]]:
    anagrams = defaultdict(list)

    for s in strs:
        # Compute character frequency count
        freq = [0] * 26 # For lowercase English letters
        for char in s:
            freq[ord(char) - ord('a')] += 1

        # Use tuple of frequency counts as the key
        anagrams[tuple(freq)].append(s)

    return list(anagrams.values())

```

Why This Approach?

Using a frequency count as the key is more efficient than sorting for strings with large lengths. The $O(k)$ time complexity for generating the frequency count ensures that this approach scales well with long strings. Additionally, hash maps make it easy to group and retrieve anagrams.

Alternative Approaches

- **Sorting-Based Approach:** While easier to implement, sorting each string increases the time complexity to $O(k \log k)$ for each string.
- **Prime Product Key:** Assign a unique prime number to each character and compute the product of these primes for each string. Use the product as the hash key. However, this method can encounter issues with integer overflow for long strings.

Similar Problems

- **Valid Anagram:** Check if two strings are anagrams by comparing their frequency counts or sorted forms.
- **Find All Anagrams in a String:** Locate all start indices of substrings in a string that are anagrams of another string.
- **Palindrome Permutation:** Determine if a string can be rearranged to form a palindrome.

Things to Keep in Mind and Tricks

- Use array-based frequency counts for fixed-size alphabets (e.g., lowercase English letters) and hash maps for larger character sets such as Unicode.
- Handle edge cases, such as empty strings or strings with only one character.
- Optimize for large inputs by avoiding unnecessary operations like sorting.

Corner and Special Cases to Test

- **Empty Strings:** Input: `strs = ["", ""]` (should group all empty strings together).
- **Single Character Strings:** Input: `strs = ["a", "b", "a"]` (should group "a" separately from "b").
- **Case Sensitivity:** Check if the solution handles lowercase and uppercase letters consistently.
- **Identical Strings:** Input: `strs = ["abc", "abc", "bca"]` (should group all strings together).

Conclusion

The **Group Anagrams** problem demonstrates the power of hash maps for categorizing strings based on their properties. By leveraging either sorted forms or frequency counts, this problem can be solved efficiently. Mastering this problem equips you with techniques for handling more advanced string grouping and classification challenges.

Problem 7.4 Valid Anagram

Problem Statement

Given two strings ‘s’ and ‘t’, write a function to determine if ‘t’ is an anagram of ‘s’. An Anagram is a word or phrase formed by rearranging the letters of another word or phrase, typically using all the original letters exactly once.

For example, the word “anagram” can be rearranged into “nag a ram”.

Note: Two empty strings are considered to be anagrams of each other.

An anagram involves rearranging the letters of a word to form a new word, testing your ability to manipulate and count character frequencies efficiently.

[LeetCode Link]
[GeeksForGeeks Link]
[HackerRank Link]
[CodeSignal Link]
[InterviewBit Link]
[Eduative Link]
[Codewars Link]

Examples

Example 1:

Input: s = "anagram", t = "nagaram"
 Output: true

Example 2:

Input: s = "rat", t = "car"
 Output: false

Algorithmic Approach

To determine if one string is an anagram of another, we can count the occurrences of each letter in both strings and then compare these counts. This can be accomplished with a hash table or an array of fixed size if the strings only consist of lowercase alphabetic characters.

Using fixed-size arrays for counting can optimize both time and space when dealing with a limited set of characters.

Complexities

- **Time Complexity:** The time complexity of the algorithm is $O(n)$, where n is the length of the input strings, because we have to count each letter in both strings.
- **Space Complexity:** The space complexity is $O(1)$ because the additional space used by the algorithm depends only on the size of the alphabet used in the strings and not on the length of the strings themselves.

Python Implementation

Below is the complete Python code for the ‘isAnagram’ function to check if one string is an anagram of another:

Implementing character frequency counting with fixed-size arrays ensures optimal performance for anagram detection.

```
class Solution:
    def isAnagram(self, s: str, t: str) -> bool:
        if len(s) != len(t):
            return False

        count = [0] * 26 # Since the problem statement mentions only lowercase
                         # ↪ alphabets

        for char in s:
            count[ord(char) - ord('a')] += 1
        for char in t:
            count[ord(char) - ord('a')] -= 1

        for c in count:
            if c != 0:
                return False

        return True

# Example Usage:
# solution = Solution()
# print(solution.isAnagram("anagram", "nagaram")) # Output: True
# print(solution.isAnagram("rat", "car"))           # Output: False
```

This implementation counts the frequency of each character in the string ‘s’ and ‘t’ by incrementing and decrementing the value at the respective indices in the ‘count’ array. After processing both strings, if all elements in the ‘count’ array are zeros, the strings are anagrams; otherwise, they aren’t.

Explanation

The ‘isAnagram’ function efficiently checks if two strings are anagrams by leveraging a fixed-size array to count character occurrences. Here’s a detailed breakdown of the implementation:

- **Initial Length Check:**
 - If the lengths of ‘s’ and ‘t’ are different, they cannot be anagrams. The function immediately returns ‘False’.
- **Character Frequency Counting:**

- Counting in ‘s’: Iterate through each character in ‘s’, and increment the corresponding index in the ‘count’ array.
- Counting in ‘t’: Iterate through each character in ‘t’, and decrement the corresponding index in the ‘count’ array.

- **Verification:**

- After processing both strings, iterate through the ‘count’ array. If any element is not zero, it indicates a mismatch in character frequencies, and the function returns ‘False’.
- If all elements are zero, the strings are anagrams, and the function returns ‘True’.

Why This Approach

This approach was chosen due to its efficiency in both time and space complexity. By utilizing an array of fixed size to count character occurrences instead of a hash table, we ensure constant time operations for updating counts, which significantly reduces the overhead and maintains linear time complexity with respect to the size of the input.

Alternative Approaches

An alternative approach involves sorting both strings and comparing them to check for equality. Although intuitive, this method has a higher time complexity of $O(n \log n)$, where n is the length of the strings due to the sorting operation.

- **Pros:** Simple to implement using built-in sorting functions.
- **Cons:** Less efficient for large strings due to the higher time complexity.

Another alternative could involve using a hash table (like Python’s ‘collections.Counter’) to count character frequencies. While this method is also $O(n)$ in time complexity, it may have slightly higher constant factors due to the overhead of hash table operations compared to fixed-size array indexing.

Using fixed-size arrays for counting is often faster than hash tables when dealing with a limited set of characters.

Similar Problems to This One

There are several other problems that involve checking for permutations, rearrangements, or matching character frequencies, such as:

- Find All Anagrams in a String

- Minimum Window Substring
- Permutation in String

Things to Keep in Mind and Tricks

- **Efficient Counting:** When dealing with a limited set of characters (like lower-case English letters), using fixed-size arrays for counting is more efficient than hash tables.
- **Early Termination:** If the lengths of the two strings differ, they cannot be anagrams. Perform this check upfront to save unnecessary computations.
- **ASCII Values:** Leveraging ASCII values allows for direct indexing into the counting array, enhancing performance.
- **Edge Case Handling:** Always consider and handle edge cases such as empty strings or strings with all identical characters.

Corner and Special Cases to Test When Writing the Code

When implementing the ‘isAnagram’ function, it is crucial to test the following edge cases to ensure robustness:

- **Empty Strings:** Both ‘s’ and ‘t’ are empty. They should be considered anagrams.
- **Different Lengths:** One string is longer than the other. The function should immediately return ‘False’.
- **Single Character Strings:** ‘s = ”a”’, ‘t = ”a”’ should return ‘True’, while ‘s = ”a”’, ‘t = ”b”’ should return ‘False’.
- **All Identical Characters:** ‘s = ”aaa”’, ‘t = ”aaa”’ should return ‘True’, whereas ‘s = ”aaa”’, ‘t = ”aaab”’ should return ‘False’.
- **Unicode Characters:** If the problem allows, test strings with Unicode or non-alphabetic characters to ensure the function handles them correctly.
- **Case Sensitivity:** Depending on the problem constraints, ensure that the function correctly handles uppercase and lowercase letters if they are considered distinct.
- **Strings with Spaces:** If spaces are allowed within the strings, verify how they are handled in the anagram check.
- **Large Strings:** Test with very long strings to ensure that the function performs efficiently without timing out.

Problem 7.5 Top K Frequent Elements

The **Top K Frequent Elements** problem involves finding the ‘ k ’ most frequent elements in a given array of integers. This problem tests your ability to efficiently count element frequencies and retrieve the top ‘ k ’ elements based on these frequencies.

Identifying the most frequent elements in a dataset is a common task in data analysis, machine learning, and software engineering.

Problem Statement

Given an integer array ‘`nums`’ and an integer ‘ k ’, return the ‘ k ’ most frequent elements. You may return the answer in any order.

Note:

- You may assume that ‘ k ’ is always valid, $1 \leq k \leq$ the number of unique elements in the array.
- Your algorithm’s time complexity must be better than $O(n \log n)$, where n is the array’s size.

Example 1:

Input: `nums = [1,1,1,2,2,3]`, $k = 2$

Output: `[1,2]`

Explanation: 1 appears three times, 2 appears twice, and 3 appears once. The top 2 frequent elements are

Example 2:

Input: `nums = [1]`, $k = 1$

Output: `[1]`

Explanation: 1 appears once. It is the top 1 frequent element.

LeetCode link: 347. Top K Frequent Elements

[\[LeetCode Link\]](#)

[\[GeeksForGeeks Link\]](#)

[\[HackerRank Link\]](#)

[\[CodeSignal Link\]](#)

[\[InterviewBit Link\]](#)

[\[Educative Link\]](#)

[\[Codewars Link\]](#)

Algorithmic Approach

To efficiently find the top ‘ k ’ frequent elements, we can follow these steps:

1. Frequency Counting:

- Traverse the array and count the frequency of each element using a hash map (dictionary).

2. Bucket Sort:

- Create an array of buckets where the index represents frequency, and each bucket at index ‘ i ’ contains elements that appear ‘ i ’ times.

3. Collect Top K Elements:

- Iterate through the buckets starting from the highest frequency, and collect elements until ‘ k ’ elements have been gathered.

This approach ensures that the time complexity remains $O(n)$, as both frequency counting and bucket sorting are linear operations.

Bucket sort is particularly effective here because the maximum frequency cannot exceed the length of the array, allowing us to use a fixed-size array for buckets.

Complexities

- **Time Complexity:** $O(n)$, where n is the number of elements in the array. Both the frequency counting and the bucket sort operations run in linear time.
- **Space Complexity:** $O(n)$, due to the additional space required for the frequency map and the bucket array.

Python Implementation

Below is the complete Python code for the ‘topKFrequent’ function to find the top ‘k’ frequent elements in an array:

Implementing the algorithm with bucket sort ensures optimal performance by avoiding unnecessary sorting operations.

```
from collections import defaultdict

class Solution:
    def topKFrequent(self, nums: List[int], k: int) -> List[int]:
        # Step 1: Frequency Counting
        frequency_map = defaultdict(int)
        for num in nums:
            frequency_map[num] += 1

        # Step 2: Bucket Sort
        max_freq = max(frequency_map.values())
        buckets = [[] for _ in range(max_freq + 1)]
        for num, freq in frequency_map.items():
            buckets[freq].append(num)

        # Step 3: Collect Top K Elements
        result = []
        for freq in range(max_freq, 0, -1):
            for num in buckets[freq]:
                result.append(num)
                if len(result) == k:
                    return result

# Example Usage:
# solution = Solution()
# print(solution.topKFrequent([1,1,1,2,2,3], 2)) # Output: [1,2]
# print(solution.topKFrequent([1], 1))           # Output: [1]
```

This implementation performs the following steps:

1. **Frequency Counting:** Uses a ‘defaultdict’ to count how many times each element appears in ‘nums’.
2. **Bucket Sort:** Creates buckets where each bucket at index ‘i’ contains elements that appear ‘i’ times.
3. **Collect Top K Elements:** Iterates through the buckets in reverse order (from highest to lowest frequency) and collects elements until ‘k’ elements have been gathered.

Explanation

The ‘topKFrequent’ function efficiently identifies the top ‘ k ’ most frequent elements in an array by leveraging a combination of frequency counting and bucket sort. Here’s a detailed breakdown of the implementation:

- **Frequency Counting:**

- Utilize a ‘defaultdict’ to map each unique element in ‘nums’ to its frequency.
- Traverse the ‘nums’ array, incrementing the count for each element.

- **Bucket Sort:**

- Determine the maximum frequency present in the frequency map.
- Initialize a list of empty lists (‘buckets’) where the index represents the frequency.
- Iterate through the frequency map and append each element to the corresponding bucket based on its frequency.

- **Collect Top K Elements:**

- Initialize an empty ‘result’ list to store the top ‘ k ’ elements.
- Iterate through the ‘buckets’ list in reverse order, starting from the highest frequency.
- For each bucket, append its elements to the ‘result’ list until ‘ k ’ elements have been collected.
- Return the ‘result’ list containing the top ‘ k ’ frequent elements.

Why This Approach

This approach was chosen due to its efficiency in both time and space complexity. By combining frequency counting with bucket sort, we avoid the need to sort the entire array, which would have resulted in a higher time complexity of $O(n \log n)$. Instead, this method maintains a linear time complexity of $O(n)$, making it highly suitable for large datasets.

Alternative Approaches

An alternative approach involves using a heap (priority queue) to keep track of the top ‘ k ’ frequent elements. Here’s how it works:

1. **Frequency Counting:** Similar to the primary approach, use a hash map to count frequencies.

2. **Min-Heap Construction:** Iterate through the frequency map and maintain a min-heap of size ‘k’. If the heap size exceeds ‘k’, remove the smallest frequency element.

3. **Extracting Results:** The heap will contain the top ‘k’ frequent elements.

- **Pros:** More intuitive for those familiar with heap operations; can be easier to implement in some languages.
- **Cons:** Slightly higher time complexity of $O(n \log k)$ compared to the primary approach’s $O(n)$.

While the heap approach is effective, the bucket sort method provides a better time complexity, making it more optimal for scenarios with large input sizes.

Heaps are powerful data structures for maintaining dynamic sets of elements with priority-based ordering.

Similar Problems to This One

There are several other problems that involve frequency counting and retrieving elements based on their counts, such as:

- Frequency Sort
- Find All Duplicates in an Array
- Group Anagrams
- Top K Frequent Words

Things to Keep in Mind and Tricks

- **Efficient Counting:** Utilize hash maps or fixed-size arrays for counting element frequencies to optimize performance.
- **Bucket Sort Advantage:** When the range of frequencies is limited, bucket sort can provide linear time solutions.
- **Heap Utilization:** Heaps can be useful for maintaining a dynamic set of top ‘k’ elements, especially when dealing with streams of data.
- **Edge Case Handling:** Always account for scenarios where multiple elements have the same frequency or where the array contains only one unique element.

Corner and Special Cases to Test When Writing the Code

When implementing the ‘topKFrequent’ function, it is crucial to test the following edge cases to ensure robustness:

- **Single Element Array:** ‘nums = [1]’, ‘k = 1’ should return ‘[1]’.
- **All Elements Same:** ‘nums = [1,1,1,1]’, ‘k = 1’ should return ‘[1]’.
- **Multiple Elements with Same Frequency:** ‘nums = [1,1,2,2,3,3]’, ‘k = 2’ could return ‘[1,2]’, ‘[1,3]’, or ‘[2,3]’.
- **Large Value of K:** ‘nums = [1,2,3,4,5,6,7,8,9,10]’, ‘k = 5’ should return the top 5 frequent elements, which could be any if all frequencies are equal.
- **Empty Array:** ‘nums = []’, ‘k = 0’ should handle gracefully, possibly returning an empty list.
- **Large Input Size:** Test with a very large array to ensure that the implementation performs efficiently without exceeding memory limits.
- **Negative Numbers:** ‘nums’ containing negative integers should be handled correctly.

Problem 7.6 Longest Consecutive Sequence

The **Longest Consecutive Sequence** problem requires finding the length of the longest sequence of consecutive numbers in an unsorted array. This must be achieved with a time complexity of $O(n)$, which rules out the possibility of sorting the array as a pre-processing step.

A classic array problem that demonstrates the power of hash-based data structures for achieving optimal time complexity.

Problem Statement

Given an unsorted array of integers ‘nums’, return the length of the longest consecutive elements sequence. It is essential that the algorithm operates in $O(n)$ time.

Examples:

- **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

Explanation: The longest consecutive elements sequence is [0, 1, 2, 3, 4, 5, 6, 7, 8]. Therefore its length is 9.

LeetCode link: [Longest Consecutive Sequence](#)

[LeetCode Link]

[GeeksForGeeks Link]

[HackerRank Link]

[CodeSignal Link]

[InterviewBit Link]

[Educative Link]

[Codewars Link]

Algorithmic Approach

The optimal solution uses a **hash set** to achieve $O(n)$ time complexity. The key insight is that we only need to check for sequence starts (numbers where num-1 doesn't exist in the set), eliminating redundant checks. For each sequence start, we can efficiently expand the sequence using the set's $O(1)$ lookup.

The hash set approach elegantly balances simplicity with optimal performance.

Complexities

- **Time Complexity:** $O(n)$, since each number in the array is processed once to insert into the set and at most once when finding a sequence.
- **Space Complexity:** $O(n)$, as a set is used to store the elements of the array.

Python Implementation

```

class Solution:
    def longestConsecutive(self, nums: List[int]) -> int:
        if not nums: # Handle empty input
            return 0

        num_set = set(nums)
        max_length = 1

        for num in num_set:
            # Only check sequences from their starting point
            if num - 1 not in num_set:
                current_length = 1
                current = num

                # Expand sequence as far as possible
                while current + 1 in num_set:
                    current_length += 1
                    current += 1

                max_length = max(max_length, current_length)

        return max_length

# Test cases demonstrating various scenarios:
def test_longest_consecutive():
    solution = Solution()
    assert solution.longestConsecutive([]) == 0
    assert solution.longestConsecutive([1]) == 1
    assert solution.longestConsecutive([1,2,3,5,6,7]) == 3
    assert solution.longestConsecutive([-1,-2,-3,0,1]) == 5

```

This implementation prioritizes readability while maintaining optimal performance.

This implementation uses a set to achieve constant-time look-up when checking for consecutive numbers. It starts a new count whenever a number is the beginning of a new sequence (i.e., the number just smaller is not in the set). The code efficiently avoids counting consecutive sequences multiple times by counting only from the beginning of such sequences.

Explanation

The ‘longestConsecutive’ function efficiently finds the longest consecutive sequence in an unsorted array by leveraging a set for constant-time look-ups. Here’s a detailed breakdown of the implementation:

- **Initialization:**

- **Set Creation:** Convert the input list ‘nums’ into a set ‘num-set’ to allow for $O(1)$ look-up times.
- **Variable Setup:** Initialize ‘longest-streak’ to keep track of the maximum sequence length found.

- **Iteration:**

- **Sequence Start Check:** For each number ‘num’ in ‘num-set’, check if ‘num - 1’ is not present. If ‘num - 1’ is absent, it means ‘num’ could be the start of a new sequence.
- **Streak Counting:** Initialize ‘current-num’ to ‘num’ and ‘current-streak’ to 1. Then, incrementally check if ‘current-num + 1’ exists in ‘num-set’. If it does, continue to extend the streak.
- **Maximum Streak Update:** After exiting the while loop, update ‘longest-streak’ if the current streak is longer than the previously recorded maximum.

- **Result:**

- After iterating through all elements, return ‘longest-streak’ as the length of the longest consecutive sequence.

Common Pitfalls and Tips

- **Initialization:** Don’t forget to handle empty input arrays explicitly.
- **Optimization:** Only start sequence checks from actual sequence beginnings.
- **Set vs List:** Using a list for lookups would degrade performance to $O(n^2)$.
- **Memory:** The set trades space for time - be aware of memory constraints with large inputs.

Alternative Approaches

An alternative method might involve sorting the array first; however, as mentioned, this would violate the $O(n)$ time complexity requirement. Another idea could be to use a **Union-Find** data structure to group consecutive elements, but that approach is generally more complex and doesn’t offer any significant advantages over the set-based method in this context.

- **Sorting Approach:**

- **Pros:** Intuitive and straightforward to implement.
- **Cons:** Higher time complexity of $O(n \log n)$, which is not optimal for large datasets.

- **Union-Find Approach:**

- **Pros:** Efficiently groups elements into sets.
- **Cons:** More complex to implement and doesn't improve upon the set-based approach's time complexity.

Understanding multiple approaches enhances problem-solving flexibility and depth.

Similar Problems to This One

There are several other problems that involve finding sequences or patterns within arrays, such as:

- Missing Ranges
- Summary Ranges
- Longest Substring Without Repeating Characters
- Maximum Subarray

Things to Keep in Mind and Tricks

- **Efficient Look-ups:** Utilizing a set for constant-time element presence checks is crucial for maintaining linear time complexity.
- **Sequence Start Identification:** Only initiate sequence counts from numbers that are potential sequence starters (i.e., ‘num - 1’ not in set).
- **Avoid Redundant Operations:** By starting counts only at sequence starts, redundant checks and counts are minimized.
- **Edge Case Handling:** Always consider edge cases such as empty arrays or arrays with all identical elements.

Corner and Special Cases to Test When Writing the Code

When implementing the ‘longestConsecutive’ function, it is crucial to test the following edge cases to ensure robustness:

- **Empty Array:** ‘nums = []’ should return ‘0’.
- **Single Element:** ‘nums = [1]’ should return ‘1’.
- **All Elements the Same:** ‘nums = [2,2,2,2]’ should return ‘1’.

- **Multiple Sequences of Same Length:** ‘nums = [1,2,3,10,11,12]‘ should return ‘3‘ as both sequences have the same length.
- **Negative Numbers:** ‘nums = [-1, -2, -3, 0, 1]‘ should return ‘5‘.
- **Large Input Size:** Test with a very large array to ensure that the implementation performs efficiently without exceeding memory limits.
- **Non-Consecutive Numbers with Gaps:** ‘nums = [10, 5, 12, 3, 55, 30, 4, 11, 2]‘ should return ‘4‘ for the sequence ‘[2, 3, 4, 5]‘.

7.1 More Examples of Sliding Window Technique

Here are more examples of the sliding window technique.

Problem 7.7 Longest Substring Without Repeating Characters

The "Longest Substring Without Repeating Characters" problem is a frequently addressed question in coding interviews, focusing on strings and the utilization of data structures to optimize search and retrieval operations. The task is to sift through the sequence of characters in a given string, identifying the lengthiest possible segment that consists purely of non-repeating characters.

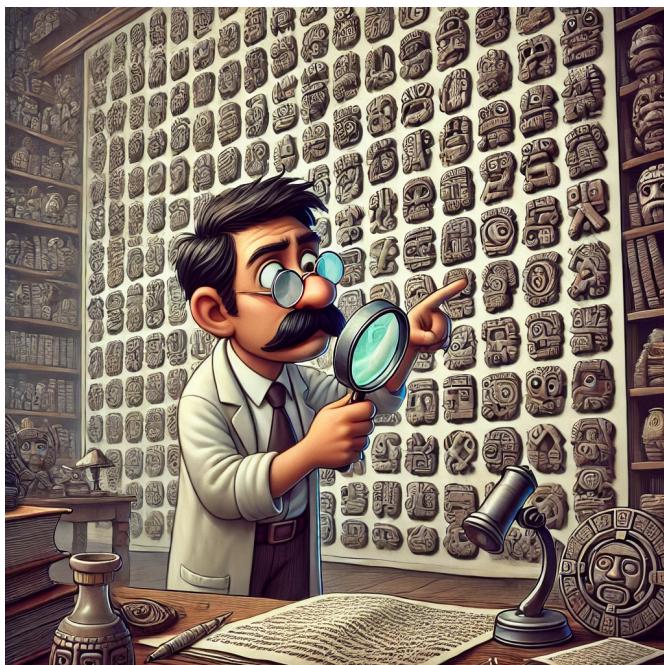


Figure 7.1: A scientist studying the Mayan alphabet to decode the longest substring without repeating characters.

Problem Statement Given a string s , the challenge lies in discerning the length of the longest substring devoid of recurrent characters. For example, within the string "abcabcbb", the longest substring that meets the criteria is "abc", which constitutes 3 characters. In contrast, for a string like "bbbbbb", the longest such substring is simply "b", reflecting a length of 1. The crux of the matter is to efficiently navigate through the string, implementing a strategy that maximizes the window of unique characters whilst conforming to the constraints set forth.

Algorithmic Approach To unravel this problem, a sliding window technique is employed, enhancing the algorithm's capability to dynamically adjust the extent of the

observed substring³¹. The primary objective is to expand this window by traversing the string and keeping a vigilant eye for duplicating elements. The moment a recurring character manifests, it necessitates a recalibration of the window's scope, excising the previously occurring instance of said character before progressing.

****Detailed Steps:****

1. ****Initialization:****

- Create a dictionary, `char_index_map`, to store the latest index of each character encountered.
- Initialize two pointers, `start` and `max_length`, to track the beginning of the current window and the maximum length found, respectively.

2. ****Traversal:****

- Iterate over the string using the `end` pointer.
- For each character, check if it exists in `char_index_map` and if its last occurrence is within the current window (i.e., $\text{char_index_map}[\text{char}] \geq \text{start}$).
- If so, move the `start` pointer to the position right after the last occurrence to ensure all characters in the window remain unique.
- Update the character's latest index in `char_index_map`.
- Calculate the current window size ($\text{end} - \text{start} + 1$) and update `max_length` if the current window is larger.

3. ****Termination:****

- After completing the traversal, `max_length` holds the length of the longest substring without repeating characters.

This method ensures that each character is processed only once, maintaining an overall time complexity of $O(n)$.

Complexities

- **Time Complexity:** The overarching time complexity of the solution pivots around $O(n)$, as the algorithm meticulously processes each character in the string a singular time — a testament to the efficiency of the sliding window methodology³².

³² Time complexity $O(n)$

- **Space Complexity:** The auxiliary space complexity, contingent upon the breadth of the character set employed in the hash map or set, invariably approaches $O(m)$, where m represents the size of the character alphabet (in the case of ASCII, potentially up to 128)³³.

³³ Space complexity $O(m)$

Python Implementation Below is the complete Python code for solving the "Longest Substring Without Repeating Characters" problem by employing a sliding window approach:

```
def length_of_longest_substring(s):
    """
    Finds the length of the longest substring without repeating characters.

    Parameters:
    s (str): The input string.

    Returns:
    int: The length of the longest substring without repeating characters.
    """
    char_index_map = {}
    max_length = 0
    start = 0

    for i, char in enumerate(s):
        if char in char_index_map and char_index_map[char] >= start:
            start = char_index_map[char] + 1
        char_index_map[char] = i
        max_length = max(max_length, i - start + 1)

    return max_length

# Example usage:
s = "abcabcbb"
print(length_of_longest_substring(s)) # Output: 3
```

Example Usage and Test Cases Here are some test cases to test the function:

```
# Test case 1: General case
s = "abcabcbb"
print(length_of_longest_substring(s)) # Output: 3

# Test case 2: All characters are the same
s = "bbbbbb"
print(length_of_longest_substring(s)) # Output: 1

# Test case 3: No repeating characters
s = "abcdefg"
print(length_of_longest_substring(s)) # Output: 7

# Test case 4: Empty string
s = ""
print(length_of_longest_substring(s)) # Output: 0

# Test case 5: String with special characters
s = "pwwkew"
```

```
print(length_of_longest_substring(s)) # Output: 3
```

Why This Approach The sliding window technique is favored in this context due to its real-time responsiveness to evolving conditions within the string. It streamlines operations by eliminating the need for redundant inspections of previously reviewed characters and concentrates computational efforts solely on the changing window edges, thereby enhancing the procedure's efficiency³⁴.

³⁴ Efficiency of sliding window

Alternative Approaches An alternative approach to the sliding window technique could involve brute force, which entails inspecting every possible substring for duplicates—a method that is computationally exhaustive and less feasible for longer strings. Another potential approach might utilize dynamic programming to keep track of the longest substring without repeating characters up to each index, but this is generally less efficient than the adopted sliding window strategy³⁵.

³⁵ Alternative approaches

Similar Problems to This One There are several problems akin to the "Longest Substring Without Repeating Characters," such as "Longest Substring with At Most Two Distinct Characters," "Longest Repeating Character Replacement," and "Substring with Concatenation of All Words." Each of these presents unique requisites that necessitate slight modifications to the basic sliding window construct³⁶.

³⁶ Similar problems

Things to Keep in Mind and Tricks When dealing with sliding window problems, an imperative trick is to adequately manage the window boundaries, especially when the window contracts. It's crucial to ensure that the starting boundary does not revert backwards, as this would lead to incorrect computations³⁷.

³⁷ Managing window boundaries

Corner and Special Cases to Test When Writing the Code One should be alert to scenarios involving extensive sequences of identical characters, strings consisting solely of unique characters, or strings with alternating patterns. Additionally, edge cases such as an empty string should be contemplated. It is advisable to conduct thorough testing against these circumstances to affirm the robustness of the algorithm³⁸.

³⁸ Corner cases

Problem 7.8 Longest Palindromic Substring

The **Longest Palindromic Substring** problem is a classic challenge in string manipulation that requires finding the longest contiguous substring in a given string that reads the same backward as forward. It serves as an excellent introduction to dynamic programming and center-expansion techniques for substring problems.

Problem Statement

Given a string s , return the longest palindromic substring in s .

Input: - A string s .

Output: - A string representing the longest palindromic substring in s .

Example 1:

Input: $s = "babad"$

Output: "bab"

Example 2:

Input: $s = "cbbd"$

Output: "bb"

Example 3:

Input: $s = "a"$

Output: "a"

Algorithmic Approaches

Several approaches can be used to solve the problem efficiently:

1. Dynamic Programming

Dynamic programming tracks whether substrings are palindromic and builds the solution iteratively. A 2D boolean array dp is used where $dp[i][j]$ is true if the substring $s[i:j+1]$ is a palindrome.

Steps:

- Initialize dp where $dp[i][i]$ is true for all i (single characters are palindromes).
- Fill dp for substrings of length 2 and greater using the relation:

$$dp[i][j] = (s[i] == s[j] \text{ and } dp[i+1][j-1])$$

- Track the maximum palindrome length and its starting index.

Complexity:

- **Time Complexity:** $O(n^2)$, where n is the length of the string.
- **Space Complexity:** $O(n^2)$, due to the 2D array.

2. Center Expansion (Optimal)

Palindromes can be expanded from their center. Each character (and between characters) is considered a potential center.

Steps:

- Iterate over each character and pair of characters as potential centers.
- Expand outward while the characters on both sides match.
- Track the longest palindrome found during expansion.

Complexity:

- **Time Complexity:** $O(n^2)$, as each character is a center and expansion takes $O(n)$ in total.
- **Space Complexity:** $O(1)$, as no additional data structures are needed.

Python Implementation

Below is the implementation of the center expansion approach:

```
def longestPalindrome(s: str) -> str:
    def expandAroundCenter(left: int, right: int) -> str:
        while left >= 0 and right < len(s) and s[left] == s[right]:
            left -= 1
            right += 1
        return s[left + 1:right]

    longest = ""
    for i in range(len(s)):
        # Odd-length palindromes
        palindrome1 = expandAroundCenter(i, i)
        # Even-length palindromes
        palindrome2 = expandAroundCenter(i, i + 1)
        # Update longest palindrome
        longest = max(longest, palindrome1, palindrome2, key=len)

    return longest

# Example usage:
s = "babad"
print(longestPalindrome(s))  # Output: "bab" or "aba"
```

Why This Approach?

The center expansion approach is preferred for its simplicity and $O(1)$ space complexity. By focusing only on expanding potential centers, it avoids the overhead of maintaining a DP table while achieving the same time complexity.

Alternative Approaches

- **Manacher's Algorithm:** An advanced algorithm that reduces the time complexity to $O(n)$ by transforming the string and calculating palindrome radii. However, it is more complex and rarely required for practical applications.
- **Brute Force:** Check all substrings to find the longest palindrome. This has $O(n^3)$ time complexity and is impractical for larger strings.

Similar Problems

- **Palindrome Partitioning:** Partition a string into the minimum number of palindromic substrings.
- **Count Substrings That Are Palindromes:** Count all substrings of a string that are palindromic.
- **Longest Palindromic Subsequence:** Find the longest subsequence (not necessarily contiguous) that is palindromic.

Things to Keep in Mind and Tricks

- Single characters are always palindromes.
- Even-length and odd-length palindromes must be treated separately in center expansion.
- Use the $O(1)$ space solution (center expansion) unless specifically required to use DP or a more complex method like Manacher's Algorithm.

Corner and Special Cases to Test

- **Empty String:** Input: $s = ""$, Output: $"$.
- **Single Character:** Input: $s = "a"$, Output: $"a"$.
- **All Characters Same:** Input: $s = "aaaa"$, Output: $"aaaa"$.
- **No Palindrome Longer Than One Character:** Input: $s = "abcd"$, Output: $"a"$ (or any single character).

Conclusion

The **Longest Palindromic Substring** problem highlights the power of efficient substring manipulation techniques such as dynamic programming and center expansion. Mastery of these methods enables you to solve a wide range of string-related problems efficiently and effectively.

Problem 7.9 First Missing Positive

The **First Missing Positive** problem is a challenging question that tests your understanding of in-place array manipulation and indexing. The goal is to identify the smallest positive integer that is missing from an unsorted array in $O(n)$ time and $O(1)$ space, excluding the space used for the input array.

Problem Statement

Given an unsorted integer array `nums`, return the smallest missing positive integer.

Input: - An array of integers `nums`.

Output: - An integer representing the smallest missing positive number.

Example 1:

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

Output: 3

Example 2:

Input: `nums = [3, 4, -1, 1]`

Output: 2

Example 3:

Input: `nums = [7, 8, 9, 11, 12]`

Output: 1

Algorithmic Approach

The problem can be efficiently solved by leveraging the array as a hash table using index manipulation. The key insight is that the first missing positive integer must be within the range $[1, n + 1]$, where n is the length of the array.

Steps:

- Iterate through the array and place each number x in its correct position if $1 \leq x \leq n^{39}$.
- After rearrangement, iterate through the array again to find the first index where the value does not match the expected value $i + 1$.
- Return $i + 1$ as the first missing positive. If no such index is found, return $n + 1$.

³⁹ This step reorders the array so that each index i ideally contains the number $i + 1$

Complexities

- **Time Complexity:** $O(n)$, since each number is placed in its correct position at most once.
- **Space Complexity:** $O(1)$, as the algorithm uses the input array for rearrangement without additional data structures.

Python Implementation

Below is the Python implementation of the above approach:

```
def firstMissingPositive(nums: List[int]) -> int:
    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]:
            nums[nums[i] - 1], nums[i] = nums[i], nums[nums[i] - 1]

    # Find the first missing positive
    for i in range(n):
        if nums[i] != i + 1:
            return i + 1

    return n + 1
```

Why This Approach?

This approach achieves optimal time and space complexity by leveraging in-place array manipulation. The reordering step ensures that numbers are placed in their "expected" positions, enabling efficient identification of the missing positive without additional storage.

Alternative Approaches

- **Hash Set Approach:** Use a hash set to store all positive integers in the array and iterate from 1 to $n + 1$ to find the first missing number. This approach is

simpler but requires $O(n)$ additional space⁴⁰.

⁴⁰ The hash set provides constant-time lookups but sacrifices space efficiency

- **Sorting Approach:** Sort the array and iterate to find the first missing positive. This has $O(n \log n)$ time complexity due to sorting and is less efficient than the optimal solution.

Similar Problems

- Missing Number:** Find the missing number from an array containing numbers 0 to n .
- Find All Duplicates in an Array:** Identify duplicate numbers using in-place manipulation.
- Find All Numbers Disappeared in an Array:** Find numbers missing from 1 to n in a given array.

Things to Keep in Mind and Tricks

- Numbers outside the range $[1, n]$ can be ignored as they do not affect the result.
- Use a while loop during reordering to ensure that each number is placed in its correct position⁴¹.
- Handle edge cases such as empty arrays or arrays with all negative numbers.

⁴¹ The while loop handles cases where swapping creates new misplaced elements

Corner and Special Cases to Test

- **Empty Array:** Input: `nums = []` (should return 1).
- **All Negative Numbers:** Input: `nums = [-1, -2, -3]` (should return 1).
- **All Positive Numbers in Range:** Input: `nums = [1, 2, 3]` (should return 4).
- **Unordered Numbers:** Input: `nums = [3, 4, -1, 1]` (should return 2).

Conclusion

The **First Missing Positive** problem is an excellent example of using in-place array manipulation to achieve optimal performance. Mastery of this problem demonstrates a strong understanding of indexing, boundary

Problem 7.10 Insert Delete GetRandom O(1)

The **Insert Delete GetRandom O(1)** problem is a classic design challenge that tests your ability to efficiently manage a dynamic collection of elements while performing insertions, deletions, and random retrievals in constant time. This problem requires designing a data structure that supports the following operations in $O(1)$ average time:

- **Insert:** Add an element to the collection if it does not already exist.
- **Delete:** Remove an element from the collection if it exists.
- **GetRandom:** Retrieve a random element from the collection, where each element has an equal probability of being chosen.

Problem Statement

Design a data structure that supports the operations `insert`, `delete`, and `getRandom` in constant average time.

Input: - A sequence of operations to be performed on the data structure.

Output: - The result of each operation, depending on the operation type.

Example 1:

Input:

```
["RandomizedSet", "insert", "insert", "getRandom", "remove", "getRandom"]
[], [1], [2], [], [1], []
```

Output:

```
null, true, true, 1 or 2, true, 2
```

Example 2:

Input:

```
["RandomizedSet", "insert", "insert", "insert", "getRandom"]
[], [1], [2], [3], []
```

Output:

```
null, true, true, true, 1 or 2 or 3
```

Algorithmic Approach

To achieve $O(1)$ time complexity for all operations, the data structure combines:

- A **hash map** to store elements and their indices in an array for efficient lookup and deletion⁴².

⁴² Hash maps provide constant-time lookup and updates

- A **dynamic array (list)** to store the actual elements, allowing $O(1)$ access by index and efficient appending.

Operations:

- **Insert:**

- Check if the element already exists in the hash map.
- If not, append it to the array and add its index to the hash map.

- **Delete:**

- Check if the element exists in the hash map.
- If it does, swap it with the last element in the array, update the hash map, and remove the last element.

- **GetRandom:**

- Use Python's `random.choice()` or a similar function to select a random element from the array.

Complexities

- **Time Complexity:** $O(1)$ for all operations, on average.
- **Space Complexity:** $O(n)$, where n is the number of elements in the data structure.

Python Implementation

Below is the Python implementation of the RandomizedSet data structure:

```
import random

class RandomizedSet:
    def __init__(self):
        self.data = [] # Dynamic array to store elements
        self.index_map = {} # Hash map to store element indices

    def insert(self, val: int) -> bool:
        if val in self.index_map:
            return False # Element already exists
        self.index_map[val] = len(self.data)
        self.data.append(val)
        return True

    def remove(self, val: int) -> bool:
        if val not in self.index_map:
```

```

        return False # Element does not exist
    # Swap the element with the last element in the array
    last_element = self.data[-1]
    idx_to_remove = self.index_map[val]
    self.data[idx_to_remove] = last_element
    self.index_map[last_element] = idx_to_remove
    # Remove the last element
    self.data.pop()
    del self.index_map[val]
    return True

def getRandom(self) -> int:
    return random.choice(self.data)

```

Why This Approach?

This approach leverages the hash map for fast element lookup and the array for efficient random access. By swapping the element to be removed with the last element in the array, the deletion operation avoids $O(n)$ complexity caused by shifting elements.

Alternative Approaches

- **Linked List with Hash Map:** Use a linked list for storage and a hash map to store node references. This allows for efficient insertions and deletions but complicates the `getRandom` operation.
- **Heap-Based Approach:** Maintain a heap for `getRandom` operation. However, the constant-time guarantee for insertion and deletion is lost.

Similar Problems

- Design a Data Structure That Supports Insert, Delete, and GetRandom**
Duplicates Allowed: Extend the data structure to handle duplicate elements.
- LRU Cache:** Use a hash map with a doubly linked list to efficiently manage cache replacement.
- Randomized Collection:** Handle duplicates while maintaining constant-time operations.

Things to Keep in Mind and Tricks

- Always ensure that the hash map and array are updated consistently during insert and delete operations.

- Random access is only efficient with arrays; avoid using linked lists for random retrieval.
- Edge cases such as inserting duplicate values or deleting non-existent elements should be handled carefully.

Corner and Special Cases to Test

- **Empty Data Structure:** Test `getRandom` when the structure is empty.
- **Single Element:** Test all operations with a single element in the structure.
- **Insert and Delete Same Element:** Insert an element, delete it, and try `getRandom`.
- **Large Input:** Stress test the structure with a large number of operations.

Conclusion

The **Insert Delete GetRandom O(1)** problem showcases the power of combining data structures like hash maps and arrays to achieve constant-time operations. Mastering this problem equips you with design strategies for solving dynamic data manipulation challenges efficiently.

Problem 7.11 Least Recently Used (LRU) Cache

The **Least Recently Used (LRU) Cache** is a classic design problem that tests your ability to implement an efficient cache system. The LRU Cache evicts the least recently used item when the cache reaches its capacity. It requires optimizing for fast access, insertion, and deletion, typically in $O(1)$ time for each operation.

Problem Statement

Design a data structure that implements the LRU cache with the following operations:

- **Get (key):** Return the value of the key if it exists in the cache, otherwise return `-1`.
- **Put (key, value):** Update or insert the value if the key is not already present. If the cache reaches its capacity, evict the least recently used item before inserting the new key-value pair.

Input: - A series of `get` and `put` operations.

Output: - The results of `get` operations and the state of the cache after each operation.

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]
```

Algorithmic Approach

The LRU Cache can be efficiently implemented using:

- A **hash map** for fast access to cache items.
- A **doubly linked list** to maintain the order of use, allowing $O(1)$ insertion and deletion at both ends⁴³.

Operations:

• Get (key):

- If the key exists in the hash map, retrieve the node, move it to the front of the doubly linked list, and return its value.
- If the key does not exist, return -1 .

• Put (key, value):

- If the key exists, update its value and move it to the front.
- If the key does not exist:
 - * Add it to the hash map and doubly linked list.
 - * If the cache exceeds its capacity, remove the least recently used item (from the back of the doubly linked list).

⁴³ The doubly linked list helps in quickly moving the most recently accessed item to the front and evicting the least recently used item from the back

Complexities

- **Time Complexity:** $O(1)$ for both `get` and `put`, as hash map and linked list operations are constant time.
- **Space Complexity:** $O(n)$, where n is the capacity of the cache, for storing n key-value pairs and their linked list nodes.

Python Implementation

Below is the Python implementation of the LRU Cache:

```

class Node:
    def __init__(self, key, value):
        self.key = key
        self.value = value
        self.prev = None
        self.next = None

class LRUCache:
    def __init__(self, capacity: int):
        self.capacity = capacity
        self.cache = {} # Hash map for storing key-node pairs
        self.head = Node(0, 0) # Dummy head of doubly linked list
        self.tail = Node(0, 0) # Dummy tail of doubly linked list
        self.head.next = self.tail
        self.tail.prev = self.head

    def _remove(self, node: Node):
        """Remove a node from the doubly linked list."""
        prev, nxt = node.prev, node.next
        prev.next, nxt.prev = nxt, prev

    def _add_to_front(self, node: Node):
        """Add a node to the front of the doubly linked list."""
        node.next = self.head.next
        node.prev = self.head
        self.head.next.prev = node
        self.head.next = node

    def get(self, key: int) -> int:
        if key in self.cache:
            node = self.cache[key]
            self._remove(node)
            self._add_to_front(node)
            return node.value
        return -1

    def put(self, key: int, value: int):
        if key in self.cache:
            self._remove(self.cache[key])
        node = Node(key, value)
        self.cache[key] = node
        self._add_to_front(node)
        if len(self.cache) > self.capacity:
            lru = self.tail.prev
            self._remove(lru)
            del self.cache[lru.key]
```

Why This Approach?

The combination of a hash map and a doubly linked list ensures that all operations are efficient. The hash map provides constant-time access to cache entries, while the doubly linked list allows constant-time insertion, deletion, and reordering of nodes to maintain the LRU order.

Alternative Approaches

- **Ordered Dictionary (Python Collections):** Use Python's `OrderedDict` to simplify the implementation. However, this may be less flexible for custom extensions⁴⁴.
- **Array-Based Approach:** Use an array for cache entries and reorder elements on each access. This approach has $O(n)$ time complexity for reordering, making it impractical for large caches.

⁴⁴ The `OrderedDict` internally maintains the order of insertion, making it suitable for LRU Cache implementation

Similar Problems

- **LFU Cache (Least Frequently Used):** A more advanced version of the cache problem that considers access frequency instead of recency.
- **Design a Data Structure with Expiry:** Implement a cache that supports expiration times for entries.
- **Priority Queue-Based Caches:** Explore heap-based designs for managing cache priorities.

Things to Keep in Mind and Tricks

- Use dummy head and tail nodes to simplify boundary conditions in the doubly linked list.
- Always update the cache size and handle capacity checks during put operations.
- Test edge cases like cache capacity of 1 or 0, and frequent repeated accesses to the same key.

Corner and Special Cases to Test

- **Empty Cache:** Test behavior when the cache has zero capacity.
- **Single Element Capacity:** Verify operations when the cache can only hold one element.
- **Overwriting Keys:** Insert a key-value pair that already exists in the cache.

- **Repeated Access:** Access the same key multiple times to ensure it remains in the cache.

Conclusion

The **Least Recently Used (LRU) Cache** problem demonstrates the importance of combining data structures to achieve efficient operations. By mastering this problem, you gain a deeper understanding of hash maps, doubly linked lists, and their applications in real-world scenarios like caching and memory management.

Problem 7.12 All O(1) Data Structure

The **All O(1) Data Structure** problem is a challenging design problem that requires implementing a data structure to manage key-value pairs while efficiently supporting operations to increment, decrement, retrieve the maximum key, and retrieve the minimum key. The goal is to ensure all operations run in $O(1)$ time on average.

Problem Statement

Design a data structure that supports the following operations:

- **Inc(key):** Increment the count of `key` by 1. If `key` does not exist, add it with a count of 1.
- **Dec(key):** Decrement the count of `key` by 1. If the count becomes 0, remove `key`.
- **GetMaxKey():** Retrieve a key with the maximum count. If no keys exist, return an empty string.
- **GetMinKey():** Retrieve a key with the minimum count. If no keys exist, return an empty string.

Input: - A sequence of operations on the data structure.

Output: - The results of `GetMaxKey()` and `GetMinKey()` operations.

Example 1:

Input:

```
["AllOne", "inc", "inc", "getMaxKey", "getMinKey", "dec", "getMaxKey", "getMinKey"]
[], ["hello"], ["hello"], [], [], ["hello"], [], []]
```

Output:

```
null, null, null, "hello", "hello", null, "", ""]
```

Algorithmic Approach

The problem is solved by combining a hash map for fast key lookups and a doubly linked list to manage keys grouped by their counts. Each node in the linked list represents a count, and it maintains a set of keys with that count.

Data Structure Design:

- **Hash Map (key_map):** Maps keys to their corresponding nodes in the linked list.
- **Doubly Linked List:** Maintains nodes representing counts in ascending order. Each node contains a set of keys with that count.
- **Helper Methods:**
 - Add a new count node to the list when needed.
 - Remove a count node if its set of keys becomes empty.

Operations:

- **Inc(key):**
 - If `key` exists, move it to the next count node.
 - If `key` does not exist, add it to the node with count 1.
 - Update the hash map and the doubly linked list as necessary.
- **Dec(key):**
 - If `key`'s count is 1, remove it entirely.
 - Otherwise, move it to the previous count node.
 - Update the hash map and the doubly linked list as necessary.
- **GetMaxKey():** Retrieve any key from the set of keys in the last node of the doubly linked list.
- **GetMinKey():** Retrieve any key from the set of keys in the first node of the doubly linked list.

Complexities

- **Time Complexity:** $O(1)$ for all operations, as the hash map and doubly linked list provide efficient lookups, insertions, and deletions.
- **Space Complexity:** $O(n)$, where n is the number of unique keys.

Python Implementation

Below is the Python implementation of the All O(1) Data Structure:

```

class Node:
    def __init__(self, count):
        self.count = count
        self.keys = set()
        self.prev = None
        self.next = None

class AllOne:
    def __init__(self):
        self.key_map = {} # Maps key to its count node
        self.head = Node(0) # Dummy head node
        self.tail = Node(0) # Dummy tail node
        self.head.next = self.tail
        self.tail.prev = self.head

    def _add_node(self, new_node, prev_node):
        """Add a new node after the given prev_node."""
        new_node.next = prev_node.next
        new_node.prev = prev_node
        prev_node.next.prev = new_node
        prev_node.next = new_node

    def _remove_node(self, node):
        """Remove a node from the doubly linked list."""
        node.prev.next = node.next
        node.next.prev = node.prev

    def inc(self, key: str):
        """Increment the count of a key."""
        if key in self.key_map:
            node = self.key_map[key]
            next_node = node.next
            if next_node.count != node.count + 1:
                next_node = Node(node.count + 1)
                self._add_node(next_node, node)
            next_node.keys.add(key)
            self.key_map[key] = next_node
            node.keys.remove(key)
            if not node.keys:
                self._remove_node(node)
        else:
            if self.head.next.count != 1:
                new_node = Node(1)
                self._add_node(new_node, self.head)
            self.head.next.keys.add(key)
            self.key_map[key] = self.head.next

```

```

def dec(self, key: str):
    """Decrement the count of a key."""
    if key in self.key_map:
        node = self.key_map[key]
        if node.count == 1:
            del self.key_map[key]
            node.keys.remove(key)
            if not node.keys:
                self._remove_node(node)
        else:
            prev_node = node.prev
            if prev_node.count != node.count - 1:
                prev_node = Node(node.count - 1)
                self._add_node(prev_node, node.prev)
            prev_node.keys.add(key)
            self.key_map[key] = prev_node
            node.keys.remove(key)
            if not node.keys:
                self._remove_node(node)

def getMaxKey(self) -> str:
    """Return any key with the maximum count."""
    return next(iter(self.tail.prev.keys)) if self.tail.prev != self.head else
        ""

def getMinKey(self) -> str:
    """Return any key with the minimum count."""
    return next(iter(self.head.next.keys)) if self.head.next != self.tail else
        ""

```

Why This Approach?

This approach achieves constant time complexity for all operations by combining a hash map for fast key lookups with a doubly linked list for maintaining count order. The design ensures that both maximum and minimum keys can be retrieved efficiently.

Alternative Approaches

- **Priority Queue:** Use a priority queue to manage counts. However, this approach has $O(\log n)$ complexity for insertion and deletion.
- **Balanced BST:** Use a balanced binary search tree to maintain counts. This also has $O(\log n)$ complexity and is less efficient than the hash map + linked list combination.

Similar Problems

- **LFU Cache (Least Frequently Used):** Manage a cache based on frequency counts.
- **Design a Data Structure with Expiry:** Implement a data structure to handle key-value pairs with expiration times.

Things to Keep in Mind and Tricks

- Ensure the doubly linked list is updated consistently during insertions and deletions to avoid dangling pointers.
- Handle edge cases such as empty data structures when retrieving maximum or minimum keys.
- Use dummy head and tail nodes to simplify boundary conditions in the linked list.

Corner and Special Cases to Test

- **Empty Data Structure:** Test `GetMaxKey()` and `GetMinKey()` when the data structure is empty.
- **Single Key:** Test all operations when there is only one key in the data structure.
- **Increment and Decrement:** Test incrementing and decrementing the same key multiple times.
- **Capacity Stress Test:** Test with a large number of keys and operations.

Conclusion

The **All O(1) Data Structure** problem demonstrates the power of combining hash maps and doubly linked lists to achieve constant time operations for complex

Problem 7.13 Anagram Detection

The **Anagram Detection** problem is a fundamental challenge that tests your ability to efficiently manipulate and compare strings. Two strings are considered anagrams if they can be rearranged to form each other. This means the strings must contain the same characters with the same frequencies, but in any order.

Problem Statement

Given two strings s_1 and s_2 , determine whether s_2 is an anagram of s_1 .

Input: - Two strings s_1 and s_2 .

Output: - A boolean indicating whether s_2 is an anagram of s_1 .

Example 1:

Input: $s_1 = \text{"listen"}$, $s_2 = \text{"silent"}$

Output: true

Example 2:

Input: $s_1 = \text{"hello"}$, $s_2 = \text{"world"}$

Output: false

Example 3:

Input: $s_1 = \text{"anagram"}$, $s_2 = \text{"nagaram"}$

Output: true

Algorithmic Approaches

Several approaches can be used to solve the anagram detection problem efficiently:

Sorting-Based Approach

Sort both strings and compare them. If the sorted versions of the strings are identical, they are anagrams.

Steps:

- Sort s_1 and s_2 .
- Compare the sorted versions.
- Return `true` if they are identical, otherwise `false`.

Complexity:

- **Time Complexity:** $O(n \log n)$, due to sorting.
- **Space Complexity:** $O(n)$, for storing the sorted strings.

Drawback: Sorting can be unnecessarily slow for large strings when faster methods are available.

Hash Map (Frequency Counting)

Count the frequency of each character in both strings using a hash map and compare the frequency tables.

Steps:

- Use a hash map to count character frequencies for $s1$.
- Subtract the frequency counts based on $s2$.
- If all counts return to zero, the strings are anagrams.

Complexity:

- **Time Complexity:** $O(n)$, for counting characters.
- **Space Complexity:** $O(1)$ (constant space for fixed alphabet size).

Array-Based Frequency Count

Use an array of size 26 (for lowercase English letters) to track character frequencies. This is more memory-efficient than a hash map for fixed alphabets.

Steps:

- Initialize an array of size 26 to zero.
- Increment counts based on $s1$ and decrement counts based on $s2$.
- Check if all values in the array are zero.

Complexity:

- **Time Complexity:** $O(n)$.
- **Space Complexity:** $O(1)$, for the array.

Python Implementation

Here is an implementation using the array-based frequency count:

```
def isAnagram(s1: str, s2: str) -> bool:
    if len(s1) != len(s2):
        return False

    # Frequency array for 26 lowercase English letters
    freq = [0] * 26
```

```

# Update frequency counts for both strings
for c1, c2 in zip(s1, s2):
    freq[ord(c1) - ord('a')] += 1
    freq[ord(c2) - ord('a')] -= 1

# Check if all frequencies are zero
return all(f == 0 for f in freq)

```

Why This Approach?

The array-based frequency count is chosen for its efficiency in both time and space. By leveraging a fixed-size array, it avoids the overhead of sorting or using a hash map. The constant-time updates to the frequency array ensure that the algorithm scales well with the input size.

Alternative Approaches

- **Hash Map Frequency Count:** Handles larger character sets, such as Unicode, at the cost of increased memory usage.
- **Prime Product Method:** Assign a unique prime number to each character and compute the product of these primes for both strings. If the products match, the strings are anagrams⁴⁵.

⁴⁵ This method is unique but can encounter issues with integer overflow for long strings

Similar Problems

- **Find All Anagrams in a String:** Locate all start indices of anagrams of one string in another.
- **Group Anagrams:** Group a list of strings into sets of anagrams.
- **Palindrome Permutation:** Check if a string can be rearranged into a palindrome.

Things to Keep in Mind and Tricks

- Ensure both strings are the same length before performing any checks.
- Use array-based methods for fixed character sets (e.g., English letters) and hash maps for larger alphabets or Unicode.
- Optimize for edge cases, such as empty strings or strings with only one character.

Corner and Special Cases to Test

- **Empty Strings:** Both `s1` and `s2` are empty (should return `true`).
- **Different Lengths:** Strings with different lengths (should return `false`).
- **Identical Strings:** Strings that are identical (should return `true`).
- **Case Sensitivity:** Strings with different cases (depends on problem constraints, e.g., "A" and "a").

Conclusion

The **Anagram Detection** problem highlights the importance of efficiently comparing strings while considering character frequencies. By leveraging sorting, hash maps, or frequency arrays, this problem can be solved with varying levels of efficiency. Mastering this challenge lays the groundwork for tackling more advanced string manipulation problems.

Problem 7.14 Ransom Note

The **Ransom Note** problem tests your ability to efficiently compare and manipulate character counts between two strings. The challenge revolves around determining whether one string (the ransom note) can be constructed entirely from the characters available in another string (the magazine).

Problem Statement

Given two strings `ransomNote` and `magazine`, determine if `ransomNote` can be constructed using only the characters from `magazine`. Each character in `magazine` can only be used once.

Input: - Two strings `ransomNote` and `magazine`.

Output: - A boolean indicating whether `ransomNote` can be constructed from `magazine`.

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

Algorithmic Approaches**Hash Map (Frequency Counting)**

Count the frequency of each character in both `ransomNote` and `magazine`. Compare the counts to determine if `magazine` has enough of each character to construct `ransomNote`.

Steps:

- Use a hash map to count the frequency of characters in `magazine`.
- Iterate through `ransomNote`, checking if each character exists in the hash map with sufficient count.
- Decrement the count for each character used. If a character is unavailable or the count is insufficient, return `false`.

Complexity:

- **Time Complexity:** $O(n + m)$, where n is the length of `ransomNote` and m is the length of `magazine`.
- **Space Complexity:** $O(k)$, where k is the size of the character set (e.g., 26 for lowercase English letters).

Array-Based Frequency Count

For fixed-size alphabets like lowercase English letters, use an array of size 26 to track character frequencies.

Steps:

- Initialize an array of size 26 to store frequencies of characters in `magazine`.
- Iterate through `magazine` and update the frequencies.
- Iterate through `ransomNote`, decrementing the frequency for each character. If a character's count drops below zero, return `false`.

Complexity:

- **Time Complexity:** $O(n + m)$.
- **Space Complexity:** $O(1)$, as the array size is constant.

Python Implementation

Below is the implementation using the array-based frequency count:

```
def canConstruct(ransomNote: str, magazine: str) -> bool:
    # Frequency array for 26 lowercase English letters
    freq = [0] * 26

    # Update frequencies for magazine
    for char in magazine:
        freq[ord(char) - ord('a')] += 1

    # Check if ransomNote can be constructed
    for char in ransomNote:
        freq[ord(char) - ord('a')] -= 1
        if freq[ord(char) - ord('a')] < 0:
            return False

    return True
```

Why This Approach?

The array-based frequency count is highly efficient for problems involving fixed alphabets like lowercase English letters. It provides constant space usage and linear time complexity, making it ideal for large inputs. The algorithm iterates over both strings once, ensuring optimal performance.

Alternative Approaches

- **Hash Map:** Useful for handling larger character sets, such as Unicode, at the cost of increased space complexity.
- **Sorting:** Sort both strings and compare character counts, but this approach has $O(n \log n + m \log m)$ time complexity, which is slower than counting-based methods.

Similar Problems

- **Valid Anagram:** Check if two strings are anagrams by comparing their character counts.

- **Find All Anagrams in a String:** Locate all anagrams of a string within another string using a sliding window and frequency counts.
- **Minimum Window Substring:** Find the smallest substring of a string containing all characters of another string.

Things to Keep in Mind and Tricks

- Use array-based methods for problems involving fixed-size alphabets for their simplicity and efficiency.
- Be cautious with edge cases, such as empty strings or when `ransomNote` contains characters not present in `magazine`.
- Ensure the frequency array or hash map is correctly updated to avoid logical errors.

Corner and Special Cases to Test

- **Empty Ransom Note:** `ransomNote = "", magazine = "anystring"` (should return `true`).
- **Empty Magazine:** `ransomNote = "a", magazine = ""` (should return `false`).
- **Insufficient Characters:** `ransomNote = "aa", magazine = "a"` (should return `false`).
- **Exact Match:** `ransomNote = "abc", magazine = "abc"` (should return `true`).

Conclusion

The **Ransom Note** problem is a classic example of leveraging frequency counts to efficiently compare two strings. By using hash maps or arrays, this problem can be solved in linear time with minimal space overhead. Mastery of this problem provides a strong foundation for tackling more complex string comparison challenges.

7.2 Index as a Hash Key

In scenarios where space complexity is a constraint, such as when the interviewer explicitly asks for $O(1)$ additional space, it is often possible to use the input array itself as a hash table. By manipulating the values at specific indices based on the array's structure, we can encode additional information without allocating extra memory⁴⁶.

⁴⁶ This approach leverages the input array's indices and values, effectively treating the array as both data and metadata

When to Use Index as a Hash Key

This technique is particularly effective when:

- The array contains values within a known range, such as 1 to N , where N is the length of the array.
- Modifying the array values directly does not violate the problem's constraints.
- Additional space is limited to $O(1)$, and hash maps or auxiliary arrays cannot be used.
- The goal is to identify properties like presence, frequency, or missing numbers within the array⁴⁷.

⁴⁷ This method is commonly employed in problems involving unique elements or specific numerical ranges

Common Applications of Index-as-Hash Techniques

1. **Presence Indication:** Negating the value at an index to indicate the presence of a number.
 - Example: For an array with values 1 to N , negate the value at index $\text{num} - 1$ when num is encountered⁴⁸.
2. **Counting or Tracking:** Incrementing values at indices to track frequency.
 - Example: Use modular arithmetic to encode additional counts into the array values.
3. **Identifying Missing or Duplicate Values:** Analyzing the final state of the array to determine missing or duplicate elements.
 - Example: If the value at index i is positive, the number $i + 1$ is missing⁴⁹.

⁴⁸ Negating a value marks the corresponding index as "visited" or "present"

⁴⁹ Positive values indicate indices that were not marked during traversal

Problem 7.15 First Missing Positive

The **First Missing Positive** problem exemplifies the use of the input array as a hash table.

Problem Statement: Given an unsorted integer array nums , find the smallest missing positive integer.

Input: $\text{nums} = [3, 4, -1, 1]$ **Output:** 2 **Explanation:** The numbers 1 and 3 are present, but 2 is missing.

Algorithmic Approach

To solve this problem in $O(n)$ time and $O(1)$ additional space, we can use the following approach:

1. **Normalize the Array:** Replace any negative numbers and zeros with a placeholder value outside the range (e.g., $N + 1$), where N is the length of the array⁵⁰. This ensures that only relevant numbers are processed in the subsequent steps
2. **Mark Indices:** For each number num in the array:
 - Calculate the target index $abs(num) - 1$.
 - Negate the value at the target index to mark it as "present" (if the value is within the range 1 to N)⁵¹.
3. **Identify the Missing Positive:** Traverse the array again. The first index i with a positive value indicates that the number $i + 1$ is missing⁵².

⁵¹ Using the absolute value ensures that previously marked indices are not skipped

⁵² Positive values at indices correspond to unmarked indices, revealing missing numbers

Python Implementation

```
def firstMissingPositive(nums: List[int]) -> int:
    """
    Finds the smallest missing positive integer in an unsorted array.

    Parameters:
    nums (List[int]): Input list of integers.

    Returns:
    int: The smallest missing positive integer.
    """
    n = len(nums)

    # Step 1: Normalize the array
    for i in range(n):
        if nums[i] <= 0 or nums[i] > n:
            nums[i] = n + 1

    # Step 2: Mark indices
    for num in nums:
        if 1 <= abs(num) <= n:
            idx = abs(num) - 1
            nums[idx] = -abs(nums[idx])

    # Step 3: Identify the missing positive
    for i in range(n):
        if nums[i] > 0:
            return i + 1

    # If all numbers from 1 to n are present
    return n + 1

# Example usage:
nums = [3, 4, -1, 1]
print(firstMissingPositive(nums))  # Output: 2
```

Why This Approach

This method achieves $O(n)$ time complexity by traversing the array multiple times and avoids additional memory allocation by reusing the input array. By marking indices through negation, we efficiently track the presence of elements within the range 1 to N without requiring auxiliary data structures.

Complexity Analysis

- **Time Complexity:** $O(n)$ ⁵³.
- **Space Complexity:** $O(1)$ ⁵⁴.

⁵³ The algorithm processes each element a constant number of times

⁵⁴ No additional data structures are used; the input array itself is modified in-place

Similar Problems

Other problems that can be solved using the index-as-hash technique include:

- **Find All Numbers Disappeared in an Array:** Identify all missing numbers in the range 1 to N .
- **Find the Duplicate Number:** Locate the duplicate element in an array of $N + 1$ integers where each integer is in the range 1 to N .
- **Cyclic Sort Problems:** Use the index-as-hash approach to sort an array of consecutive integers in $O(n)$ time.

Things to Keep in Mind

- Ensure that the array values remain within the range 1 to N for the technique to work correctly⁵⁵.
- Use absolute values when marking indices to avoid conflicts caused by previously negated values.
- Avoid using this technique if the array values cannot be modified, as this violates the problem constraints.
- Always check edge cases, such as empty arrays or arrays containing only negative numbers.

⁵⁵ Handle out-of-range values by replacing them with placeholders

Conclusion

Using the index of an array as a hash key is an elegant space-efficient technique that leverages the array's structure to store metadata alongside the data itself. This approach is particularly effective in problems involving presence, frequency, or range

queries. By mastering this technique, you can solve a wide range of problems with optimal time and space complexity, making it a valuable addition to your algorithmic toolkit.

Problem 7.16 Find All Pairs with a Given Target Sum

Problem Statement:

Given a **sorted** array of integers, find two numbers such that they add up to a specific target number. Return the indices of the two numbers (1-based index) as an integer array of size two, where ‘ $1 \leq \text{index1} < \text{index2} \leq \text{array.length}$ ’. You may assume that each input would have exactly one solution, and you may not use the same element twice.

Example:

- **Input:** numbers = [2, 7, 11, 15], target = 9
- **Output:** [1, 2]
- **Explanation:** The numbers at indices 1 and 2 ($2 + 7$) add up to the target 9.

Two Pointer Technique:

The **Two Pointer Technique** is an efficient method commonly used to solve problems involving arrays or lists, especially when dealing with sorted data. This technique uses two pointers moving through the data structure to find the desired elements without the need for additional storage or excessive computations.

Approach for the Pair Sum Problem:

1. **Initialize Two Pointers:** - **Left Pointer ('left'):** Start at the beginning of the array ('index 0'). - **Right Pointer ('right'):** Start at the end of the array ('index n-1').
 2. **Iterative Process:** - Calculate the sum of the elements pointed to by 'left' and 'right'. - **If the sum equals the target:** - Return the indices ('left + 1', 'right + 1') as the solution. - **If the sum is less than the target:** - Move the 'left' pointer one step to the right to increase the sum. - **If the sum is greater than the target:** - Move the 'right' pointer one step to the left to decrease the sum.
 3. **Termination:** - Continue the process until the 'left' pointer is no longer less than the 'right' pointer. - Since the problem guarantees exactly one solution, the loop will terminate once the solution is found.

Advantages of the Two Pointer Technique:

- **Time Efficiency:** Operates in linear time, $O(n)$, making it highly efficient for large datasets.
- **Space Efficiency:** Requires constant space, $O(1)$, as it uses

only a fixed number of additional variables. - **Simplicity:** Easy to implement and understand, reducing the likelihood of errors.

Sample Solution in Python:

```
def two_sum(numbers, target):
    left, right = 0, len(numbers) - 1
    while left < right:
        current_sum = numbers[left] + numbers[right]
        if current_sum == target:
            return [left + 1, right + 1]  # 1-based
            ↪ indexing
        elif current_sum < target:
            left += 1  # Move left pointer to the right
        else:
            right -= 1  # Move right pointer to the left
    return []  # If no solution is found
```

Listing 7.1: Two Pointer Solution for Pair Sum Problem

Explanation of the Code:

1. **Initialization:** - ‘left’ is set to the first index (‘0’). - ‘right’ is set to the last index (‘len(numbers) - 1’).
2. **Loop Condition:** - The loop continues as long as ‘left’ is less than ‘right’.
3. **Sum Calculation:** - `current_sum` is the sum of the elements at the `left` and `right` pointers.
4. **Comparison and Pointer Adjustment:** - **Equal to Target:** Return the 1-based indices. - **Less than Target:** Increment `left` to increase the sum. - **Greater than Target:** Decrement `right` to decrease the sum.
5. **Return Statement:** - If no valid pair is found (which shouldn’t happen as per problem constraints), return an empty list.

Complexity Analysis:

- **Time Complexity:** $O(n)$, where n is the number of elements in the array. Each pointer moves at most n steps. - **Space Complexity:** $O(1)$, as no additional space proportional to the input size is used.

Key Takeaways:

- The Two Pointer Technique is highly effective for solving problems involving sorted arrays or lists.
- It optimizes both time and space, making it preferable over brute-force methods in scenarios where efficiency is crucial.
- Proper initialization and careful movement of pointers based on conditional checks are essential for the correct application of this technique.

Part II

Algorithmic Techniques

Chapter 8

Sorting Algorithms

8.1 Sorting the Array

Sorting is a fundamental algorithmic technique that can significantly simplify the process of solving various computational problems. By arranging data in a particular order, sorting can enable more efficient searching, merging, and optimization, among other operations¹.

When to Sort the Array

Before deciding to sort an array, it's essential to assess the nature of the data and the specific requirements of the problem at hand. Consider the following key questions²:

1. **Is the array sorted or partially sorted?**³.
2. **Can you sort the array?**⁴.
3. **What is the impact of sorting on time and space complexity?**⁵.
4. **Are there constraints that prevent sorting?**⁶.

Algorithmic Approach

The decision to sort an array hinges on the problem's specific requirements and constraints. Below are the strategic considerations and steps involved in determining whether to sort an array as part of the solution:

1. **Assess Array Order:** Determine if the array is already sorted or partially sorted.
⁷.

¹ Sorting is often a preliminary step in many complex algorithms, providing a structured foundation for further processing

² Evaluating these aspects helps determine the appropriateness and benefits of sorting for a given problem

³ If the array is already sorted or nearly sorted, certain algorithms like binary search become applicable, offering significant performance improvements

⁴ Sorting the array first may simplify the problem considerably, but this approach is only feasible if the original order of elements does not need to be preserved

⁵ Sorting typically introduces an $O(n \log n)$ time complexity, which is acceptable for many applications but may be prohibitive for extremely large datasets

⁶ For example, if the problem requires maintaining the original order of elements, sorting may not be a viable option

⁷ Pre-sorted arrays allow for more efficient algorithms like binary search, which operates in $O(\log n)$ time

2. **Evaluate Binary Search Applicability:** If the array is sorted, consider using binary search for operations like searching for elements.⁸
3. **Decide on Sorting Necessity:** If the array is unsorted and the problem allows for sorting without violating constraints, proceed to sort.⁹
4. **Choose a Sorting Algorithm:** Select an appropriate sorting algorithm based on the array size and required time complexity.¹⁰
5. **Implement Sorting:** Apply the chosen sorting algorithm to arrange the array in the desired order.¹¹
6. **Proceed with Problem-Specific Logic:** Utilize the sorted array to implement the remaining parts of the solution, such as searching for elements, merging intervals, or eliminating duplicates.¹²

⁸ Binary search is only feasible on sorted arrays and provides a faster search mechanism compared to linear search

⁹ Sorting can transform an unsorted problem into a more manageable one by imposing order and enabling efficient processing

¹⁰ Common sorting algorithms include Quick Sort ($O(n \log n)$), Merge Sort ($O(n \log n)$), and Heap Sort ($O(n \log n)$)

¹¹ Ensure that the sorting step does not alter the original data in a way that affects subsequent operations

¹² A sorted array often reduces the complexity of these operations, enabling more efficient algorithms

Advantages of Sorting the Array

Sorting the array can offer several benefits, including:

- **Simplified Problem Solving:** Sorting often transforms a complex problem into a more straightforward one by providing a predictable order¹³.
- **Efficient Searching:** Sorted arrays facilitate faster search operations like binary search¹⁴.
- **Elimination of Duplicates:** Sorting brings duplicate elements together, making it easier to remove them¹⁵.
- **Optimized Merging:** When dealing with multiple datasets, sorted arrays enable efficient merging operations¹⁶.

¹³ For example, sorting intervals can simplify the process of merging overlapping ones

¹⁴ Binary search significantly reduces search time compared to linear search in unsorted arrays

¹⁵ This is particularly useful in problems requiring unique elements, such as the 3Sum problem

¹⁶ Merge Sort, for instance, relies on this principle to combine sorted subarrays effectively

Disadvantages of Sorting the Array

Despite its advantages, sorting may not always be the optimal choice:

- **Increased Time Complexity:** Sorting introduces an additional $O(n \log n)$ time complexity, which may be undesirable for time-sensitive applications¹⁷.
- **Space Overhead:** Some sorting algorithms require additional space, which may be a constraint in memory-limited environments¹⁸.
- **Altered Data Order:** Sorting changes the original order of elements, which may be problematic if the problem requires preserving the initial arrangement¹⁹.

¹⁷ For very large datasets, the sorting step can become a bottleneck

¹⁸ In-place sorting algorithms like Heap Sort mitigate this issue but may sacrifice other benefits

¹⁹ In such cases, alternative strategies that do not involve sorting must be considered

Python Implementation Example

Below is a Python example demonstrating how sorting can simplify the solution to the **Merge Intervals** problem²⁰:

²⁰ Merge Intervals is a classic problem where sorting is a crucial initial step

```
def merge(intervals):
    """
    Merges overlapping intervals.

    Parameters:
    intervals (List[List[int]]): A list of intervals represented as [start, end].

    Returns:
    List[List[int]]: A list of merged, non-overlapping intervals.
    """

    if not intervals:
        return []

    # Sort the intervals based on the start time
    intervals.sort(key=lambda x: x[0])

    merged = [intervals[0]]

    for current in intervals[1:]:
        prev = merged[-1]
        if current[0] <= prev[1]:
            # Overlapping intervals, merge them
            merged[-1][1] = max(prev[1], current[1])
        else:
            # Non-overlapping interval, add to the list
            merged.append(current)

    return merged

# Example usage:
intervals = [[1,3],[2,6],[8,10],[15,18]]
print(merge(intervals)) # Output: [[1,6],[8,10],[15,18]]
```

Conclusion

Sorting the array is a powerful technique that can transform complex problems into more manageable ones by imposing order and enabling the use of efficient algorithms²¹. However, it is essential to evaluate the necessity and impact of sorting on both the algorithm's performance and the problem's constraints²². Mastering the art of sorting and understanding when to apply it enhances problem-solving skills and prepares you for tackling more intricate algorithmic problems effectively.

²¹ This approach is instrumental in reducing time and space complexities for a wide range of computational challenges

²² Balancing the benefits of sorting against its overhead ensures the development of optimal solutions

Problem 8.1 Merge Sort

The **Merge Sort** problem involves implementing the Merge Sort algorithm to sort a given array of integers in ascending order. Merge Sort is renowned for its efficiency and reliability, especially with large datasets, due to its consistent $O(n \log n)$ time complexity.

Merge Sort is a fundamental divide-and-conquer algorithm that efficiently sorts arrays with a stable and predictable performance.

Problem Statement

Given an array of integers ‘nums’, sort the array in ascending order using the Merge Sort algorithm and return the sorted array.

Note:

- You must implement the Merge Sort algorithm; using built-in sorting functions is not allowed.
- Aim for a time complexity of $O(n \log n)$ and a space complexity of $O(n)$.

Example 1:

Input: `nums = [5,2,9,1,5,6]`

Output: `[1,2,5,5,6,9]`

Explanation: The array is sorted in ascending order.

Example 2:

Input: `nums = [3,0,2,5,-1,4,1]`

Output: `[-1,0,1,2,3,4,5]`

Explanation: The array is sorted in ascending order.

LeetCode link: Sort an Array

[\[LeetCode Link\]](#)

[\[GeeksForGeeks Link\]](#)

[\[HackerRank Link\]](#)

[\[CodeSignal Link\]](#)

[\[InterviewBit Link\]](#)

[\[Educative Link\]](#)

[\[Codewars Link\]](#)

Algorithmic Approach

Merge Sort is a classic divide-and-conquer algorithm that divides the array into smaller subarrays, sorts them, and then merges the sorted subarrays to produce the final sorted array. The approach can be broken down into the following steps:

1. Divide:

- Recursively split the array into two halves until each subarray contains only one element.

2. Conquer:

- Merge the subarrays by comparing their elements and arranging them in the correct order.

3. Combine:

- Continue merging the subarrays until the entire array is sorted.

This method ensures that the array is sorted efficiently by breaking down the problem into smaller, manageable parts and then combining them systematically.

Merge Sort's stable nature makes it suitable for scenarios where the relative order of equal elements needs to be preserved.

Complexities

- **Time Complexity:** $O(n \log n)$, where n is the number of elements in the array. This is due to the array being divided logarithmically and each division involving linear time operations during the merge process.
- **Space Complexity:** $O(n)$, as additional space is required to hold the temporary arrays during the merge process.

Python Implementation

Below is the complete Python code for the ‘mergeSort’ function to sort an array using the Merge Sort algorithm:

```
class Solution:
    def mergeSort(self, nums: List[int]) -> List[int]:
        if len(nums) <= 1:
            return nums

        mid = len(nums) // 2
        left_half = self.mergeSort(nums[:mid])
        right_half = self.mergeSort(nums[mid:])

        return self.merge(left_half, right_half)

    def merge(self, left: List[int], right: List[int]) -> List[int]:
        sorted_list = []
        i = j = 0

        # Merge the two halves while comparing their elements
        while i < len(left) and j < len(right):
            if left[i] < right[j]:
                sorted_list.append(left[i])
                i += 1
            else:
                sorted_list.append(right[j])
                j += 1

        # Append any remaining elements from the left half
        while i < len(left):
            sorted_list.append(left[i])
            i += 1

        # Append any remaining elements from the right half
        while j < len(right):
            sorted_list.append(right[j])
            j += 1

        return sorted_list

# Example Usage:
# solution = Solution()
# print(solution.mergeSort([5,2,9,1,5,6])) # Output: [1,2,5,5,6,9]
# print(solution.mergeSort([-3,0,2,5,-1,4,1])) # Output: [-1,0,1,2,3,4,5]
```

Implementing Merge Sort requires careful handling of array indices and merging logic to ensure efficiency and correctness.

This implementation follows the classic Merge Sort algorithm:

- 1. Recursive Division:** The ‘mergeSort’ function recursively divides the array into halves until subarrays of size one are reached.

2. **Merging:** The ‘merge‘ function takes two sorted subarrays (‘left‘ and ‘right‘) and merges them into a single sorted array by comparing their elements.
3. **Combining Results:** The sorted subarrays are combined step-by-step to form the final sorted array.

Explanation

The ‘mergeSort‘ function efficiently sorts an array by recursively dividing it into smaller subarrays and then merging those subarrays in a sorted manner. Here’s a detailed breakdown of the implementation:

- **Base Case:**
 - If the array ‘nums‘ has one or no elements, it is already sorted. The function returns ‘nums‘ as is.
- **Recursive Division:**
 - The array is divided into two halves using the midpoint ‘mid‘.
 - ‘lefthalf‘ recursively calls ‘mergeSort‘ on the first half of the array.
 - ‘right-half‘ recursively calls ‘mergeSort‘ on the second half of the array.
- **Merging Sorted Halves:**
 - The ‘merge‘ function combines ‘left-half‘ and ‘right-half‘ into a single sorted array.
 - It iterates through both halves, comparing elements and appending the smaller one to ‘sorted-list‘.
 - After one half is exhausted, the remaining elements from the other half are appended to ‘sorted-list‘.
- **Result:**
 - The merged and sorted array is returned up the recursive call stack.
 - Eventually, the original ‘mergeSort‘ call returns the fully sorted array.

Why This Approach

This approach is chosen due to its consistent $O(n \log n)$ time complexity and its ability to handle large datasets efficiently. Merge Sort’s divide-and-conquer strategy ensures that the algorithm remains efficient even as the size of the input array grows. Additionally, Merge Sort is stable, meaning that it preserves the relative order of equal elements, which can be beneficial in certain applications.

Alternative Approaches

An alternative approach to sorting arrays is the **“Quick Sort”** algorithm. Here’s a comparison between Merge Sort and Quick Sort:

- **Quick Sort:**

- **Pros:** Often faster in practice due to better cache performance; in-place sorting with $O(\log n)$ space complexity.
- **Cons:** Worst-case time complexity of $O(n^2)$ if not implemented with optimizations like random pivot selection.

- **Merge Sort:**

- **Pros:** Consistent $O(n \log n)$ time complexity; stable sort; well-suited for linked lists and external sorting.
- **Cons:** Requires additional space for merging; can be slower than Quick Sort in practice due to extra memory usage.

While Quick Sort is often preferred for in-memory sorting due to its in-place nature and average-case efficiency, Merge Sort remains a robust choice for scenarios where stability and predictable performance are paramount.

Choosing the right sorting algorithm depends on the specific requirements and constraints of the problem at hand.

Similar Problems to This One

There are several other problems that involve sorting or manipulating arrays based on sorting algorithms, such as:

- Counting Sort
- Quick Sort
- Heap Sort
- Radix Sort

Things to Keep in Mind and Tricks

- **Divide and Conquer:** Understanding how to break down problems into smaller subproblems can simplify complex algorithms like Merge Sort.
- **Handling Indices Carefully:** When dividing arrays, ensure that the indices are correctly calculated to prevent out-of-bounds errors.

- **Efficient Merging:** Optimizing the merge step can significantly impact the overall performance of the algorithm.
- **Recursive Thinking:** Merge Sort's recursive nature requires a clear understanding of base cases and how subproblems contribute to the final solution.

Corner and Special Cases to Test When Writing the Code

When implementing the ‘mergeSort’ function, it is crucial to test the following edge cases to ensure robustness:

- **Empty Array:** ‘nums = []’ should return an empty array ‘[]’.
- **Single Element:** ‘nums = [1]’ should return ‘[1]’.
- **Already Sorted Array:** ‘nums = [1,2,3,4,5]’ should return ‘[1,2,3,4,5]’.
- **Reverse Sorted Array:** ‘nums = [5,4,3,2,1]’ should return ‘[1,2,3,4,5]’.
- **Array with Duplicates:** ‘nums = [3,1,2,3,4,1]’ should return ‘[1,1,2,3,3,4]’.
- **Array with Negative Numbers:** ‘nums = [-1, -3, -2, 0, 2, 1]’ should return ‘[-3,-2,-1,0,1,2]’.
- **Large Input Size:** Test with a very large array to ensure that the implementation performs efficiently without exceeding memory limits.

Problem 8.2 Quick Sort

The **Quick Sort** problem involves implementing the Quick Sort algorithm to sort a given array of integers in ascending order. Quick Sort is favored for its average-case efficiency and in-place sorting capabilities, making it suitable for large datasets.

Quick Sort is a highly efficient sorting algorithm that employs a divide-and-conquer strategy to sort elements in-place with an average time complexity of $O(n \log n)$.

Problem Statement

Given an array of integers ‘nums’, sort the array in ascending order using the Quick Sort algorithm and return the sorted array.

Note:

- You must implement the Quick Sort algorithm; using built-in sorting functions is not allowed.
- Aim for a time complexity of $O(n \log n)$ on average and a space complexity of $O(\log n)$ due to recursion.

Example 1:

Input: nums = [3,6,8,10,1,2,1]

Output: [1,1,2,3,6,8,10]

Explanation: The array is sorted in ascending order.

Example 2:

Input: nums = [5,4,3,2,1]

Output: [1,2,3,4,5]

Explanation: The array is sorted in ascending order.

LeetCode link: Sort an Array

[\[LeetCode Link\]](#)

[\[GeeksForGeeks Link\]](#)

[\[HackerRank Link\]](#)

[\[CodeSignal Link\]](#)

[\[InterviewBit Link\]](#)

[\[Educative Link\]](#)

[\[Codewars Link\]](#)

Algorithmic Approach

Quick Sort is a **divide-and-conquer** algorithm that works by selecting a **pivot** element from the array and partitioning the other elements into two subarrays, according to whether they are less than or greater than the pivot. The subarrays are then recursively sorted.

1. Choose a Pivot:

- Select an element from the array as the pivot. Common strategies include choosing the first element, the last element, the middle element, or a random element.

2. Partitioning:

- Rearrange the array so that all elements less than the pivot are moved to its left, and all elements greater than the pivot are moved to its right.

3. Recursion:

- Recursively apply the above steps to the subarrays of elements with smaller values and separately to the subarrays of elements with greater values.

This approach ensures that the array is sorted efficiently by systematically breaking down the problem into smaller, manageable parts.

Choosing the right pivot can significantly impact the performance of Quick Sort, especially in avoiding the worst-case time complexity.

Complexities**• Time Complexity:**

- **Average Case:** $O(n \log n)$, where n is the number of elements in the array.
- **Worst Case:** $O(n^2)$, which occurs when the smallest or largest element is always chosen as the pivot.
- **Space Complexity:** $O(\log n)$, due to the space required for recursive function calls.

Python Implementation

Below is the complete Python code for the ‘quickSort’ function to sort an array using the Quick Sort algorithm:

Implementing Quick Sort requires careful handling of the pivot selection and partitioning logic to ensure efficiency and correctness.

```
class Solution:
    def quickSort(self, nums: List[int]) -> List[int]:
        def _quickSort(items, low, high):
            if low < high:
                # Partition the array
                pi = partition(items, low, high)

                # Recursively sort elements before and after partition
                _quickSort(items, low, pi - 1)
                _quickSort(items, pi + 1, high)

        def partition(items, low, high):
            # Choose the last element as pivot
            pivot = items[high]
            i = low - 1 # Index of smaller element

            for j in range(low, high):
                if items[j] < pivot:
                    i += 1
                    items[i], items[j] = items[j], items[i]

            # Swap the pivot element with the element at i+1
            items[i + 1], items[high] = items[high], items[i + 1]
            return i + 1

        _quickSort(nums, 0, len(nums) - 1)
        return nums

# Example Usage:
# solution = Solution()
# print(solution.quickSort([3,6,8,10,1,2,1])) # Output: [1,1,2,3,6,8,10]
# print(solution.quickSort([5,4,3,2,1]))       # Output: [1,2,3,4,5]
```

This implementation follows the classic Quick Sort algorithm:

- Recursive Division:** The ‘quickSort’ function initializes the recursive ‘_quickSort’ helper function, which sorts the array in place.
- Partitioning:** The ‘partition’ function selects the last element as the pivot and rearranges the array such that elements less than the pivot are on its left, and those greater are on its right.
- Recursion:** The array is recursively sorted by applying ‘_quickSort’ to the sub-arrays before and after the pivot.

Explanation

The ‘quickSort‘ function efficiently sorts an array by leveraging the Quick Sort algorithm’s divide-and-conquer strategy. Here’s a detailed breakdown of the implementation:

- **Helper Function ‘quickSort’:**

- **Base Case:** If the ‘low‘ index is not less than the ‘high‘ index, the function returns, as the subarray is already sorted.
- **Partitioning:** The ‘partition‘ function is called to partition the array around a pivot.
- **Recursive Calls:** After partitioning, ‘quickSort‘ is recursively called on the subarrays to the left and right of the pivot.

- **Partition Function ‘partition’:**

- **Pivot Selection:** The last element in the subarray is chosen as the pivot.
- **Rearrangement:** Iterate through the subarray, and if an element is less than the pivot, increment the smaller element index and swap the current element with the element at this index.
- **Pivot Placement:** After the iteration, swap the pivot element with the element at ‘ $i + 1$ ’, ensuring all elements to the left are less than the pivot and those to the right are greater.
- **Return Pivot Index:** The function returns the index of the pivot element after partitioning.

- **Result:**

- After all recursive calls complete, the original array ‘nums‘ is sorted in ascending order.

Why This Approach

This approach is chosen for its efficiency and in-place sorting capability. Quick Sort’s average-case time complexity of $O(n \log n)$ makes it highly efficient for large datasets. Additionally, by sorting the array in place, it minimizes the space usage compared to other sorting algorithms like Merge Sort.

Alternative Approaches

An alternative approach to sorting arrays is the **Merge Sort** algorithm. Here’s a comparison between Quick Sort and Merge Sort:

- **Quick Sort:**

- **Pros:**

- * Often faster in practice due to better cache performance.
- * In-place sorting with $O(\log n)$ space complexity.

- **Cons:**

- * Worst-case time complexity of $O(n^2)$, though this can be mitigated with optimizations like random pivot selection.
- * Not a stable sort.

- **Merge Sort:**

- **Pros:**

- * Consistent $O(n \log n)$ time complexity.
- * Stable sort; maintains the relative order of equal elements.
- * Well-suited for linked lists and external sorting.

- **Cons:**

- * Requires additional space for merging, leading to $O(n)$ space complexity.
- * Can be slower than Quick Sort in practice due to extra memory usage.

While Quick Sort is often preferred for in-memory sorting due to its in-place nature and average-case efficiency, Merge Sort remains a robust choice for scenarios where stability and predictable performance are paramount.

Choosing the right sorting algorithm depends on the specific requirements and constraints of the problem at hand.

Similar Problems to This One

There are several other problems that involve sorting or manipulating arrays based on sorting algorithms, such as:

- Counting Sort
- Heap Sort
- Radix Sort
- Top K Frequent Elements

Things to Keep in Mind and Tricks

- **Divide and Conquer:** Understanding how to break down problems into smaller subproblems can simplify complex algorithms like Quick Sort.
- **Choosing the Right Pivot:** Selecting an optimal pivot can prevent worst-case scenarios and ensure balanced partitions.

- **In-Place Sorting:** Quick Sort's ability to sort the array in place minimizes additional memory usage.
- **Handling Equal Elements:** Decide how to handle elements equal to the pivot to maintain desired properties (e.g., stability if needed).
- **Tail Recursion Optimization:** Implementing tail recursion can help reduce the space complexity by minimizing the depth of recursive calls.

Corner and Special Cases to Test When Writing the Code

When implementing the ‘quickSort’ function, it is crucial to test the following edge cases to ensure robustness:

- **Empty Array:** ‘nums = []’ should return an empty array ‘[]’.
- **Single Element:** ‘nums = [1]’ should return ‘[1]’.
- **Already Sorted Array:** ‘nums = [1,2,3,4,5]’ should return ‘[1,2,3,4,5]’.
- **Reverse Sorted Array:** ‘nums = [5,4,3,2,1]’ should return ‘[1,2,3,4,5]’.
- **Array with Duplicates:** ‘nums = [3,1,2,3,4,1]’ should return ‘[1,1,2,3,3,4]’.
- **Array with Negative Numbers:** ‘nums = [-1, -3, -2, 0, 2, 1]’ should return ‘[-3,-2,-1,0,1,2]’.
- **Large Input Size:** Test with a very large array to ensure that the implementation performs efficiently without exceeding memory limits.
- **All Elements the Same:** ‘nums = [2,2,2,2]’ should return ‘[2,2,2,2]’.

Problem 8.3 Sort Colors

The **Sort Colors** problem involves sorting an array containing only three distinct values in ascending order. This problem is a variant of the Dutch National Flag problem and tests your ability to implement an efficient in-place sorting algorithm with linear time complexity.

Sort Colors is a classic algorithm problem that requires arranging elements in a specific order with optimal time and space complexity.

Problem Statement

Given an array of integers ‘nums’ with ‘n’ elements where each element is either ‘0’, ‘1’, or ‘2’, sort the array in-place so that all ‘0’s come first, followed by all ‘1’s, and then all ‘2’s.

You must solve this problem without using the library's sort function and with a one-pass algorithm using constant extra space.

Examples:

- **Example 1:**

```
Input: nums = [2,0,2,1,1,0]
Output: [0,0,1,1,2,2]
Explanation: After sorting, the array becomes [0,0,1,1,2,2].
```

- **Example 2:**

```
Input: nums = [2,0,1]
Output: [0,1,2]
Explanation: After sorting, the array becomes [0,1,2].
```

- **Example 3:**

```
Input: nums = [0]
Output: [0]
Explanation: The array is already sorted.
```

LeetCode link: Sort Colors

[LeetCode Link]
[GeeksForGeeks Link]
[HackerRank Link]
[CodeSignal Link]
[InterviewBit Link]
[Educative Link]
[Codewars Link]

Algorithmic Approach

Quickly sorting an array with only three distinct values can be efficiently achieved using the [“Dutch National Flag algorithm”](#). This approach employs three pointers to partition the array into three sections: elements less than the pivot, equal to the pivot, and greater than the pivot.

1. Initialize Pointers:

- ‘low’ - the boundary for the next ‘0’.
- ‘mid’ - the current element under consideration.
- ‘high’ - the boundary for the next ‘2’.

2. Traverse the Array:

- If ‘nums[mid]’ is ‘0’:
 - Swap ‘nums[low]’ and ‘nums[mid]’.

- Increment both ‘low’ and ‘mid’.
- If ‘nums[mid]’ is ‘1’:
 - Increment ‘mid’.
- If ‘nums[mid]’ is ‘2’:
 - Swap ‘nums[mid]’ and ‘nums[high]’.
 - Decrement ‘high’.

3. Continue Until Mid Exceeds High:

- The algorithm terminates when ‘mid’ > ‘high’, ensuring all elements are correctly partitioned.

This method ensures a single-pass traversal with constant space, achieving the desired $O(n)$ time complexity.

The Dutch National Flag algorithm effectively handles multiple partitions in a single traversal, making it ideal for this problem.

Complexities

- **Time Complexity:** $O(n)$, where n is the number of elements in the array. The array is traversed only once.
- **Space Complexity:** $O(1)$, as the sorting is done in-place without requiring additional memory.

Python Implementation

Below is the complete Python code for the ‘sortColors’ function to sort an array using the Dutch National Flag algorithm:

Implementing the Dutch National Flag algorithm ensures an efficient in-place sort with linear time complexity.

```
class Solution:
    def sortColors(self, nums: List[int]) -> None:
        """
        Do not return anything, modify nums in-place instead.
        """
        low, mid, high = 0, 0, len(nums) - 1

        while mid <= high:
            if nums[mid] == 0:
                nums[low], nums[mid] = nums[mid], nums[low]
                low += 1
                mid += 1
            elif nums[mid] == 1:
                mid += 1
            else:
                nums[mid], nums[high] = nums[high], nums[mid]
                high -= 1

# Example Usage:
# solution = Solution()
# nums = [2,0,2,1,1,0]
# solution.sortColors(nums)
# print(nums) # Output: [0,0,1,1,2,2]
# nums = [2,0,1]
# solution.sortColors(nums)
# print(nums) # Output: [0,1,2]
```

This implementation follows the Dutch National Flag algorithm:

1. Initialization:

- ‘low’ points to the next position for ‘0’.
- ‘mid’ is the current element under consideration.
- ‘high’ points to the next position for ‘2’.

2. Traversal and Partitioning:

- Traverse the array with the ‘mid’ pointer.
- Swap elements based on their value relative to the pivot (‘1’ in this case).

3. Termination:

- The loop terminates when ‘mid’ exceeds ‘high’, ensuring all elements are sorted.

Explanation

The ‘sortColors‘ function efficiently sorts the array by categorizing elements into three distinct groups: ‘0’s, ‘1’s, and ‘2’s. Here’s a detailed breakdown of the implementation:

- **Initialization:**

- ‘low‘ is initialized to ‘0‘ and represents the boundary for the next ‘0‘.
- ‘mid‘ is initialized to ‘0‘ and is used to traverse the array.
- ‘high‘ is initialized to ‘len(nums) - 1‘ and represents the boundary for the next ‘2‘.

- **Traversal and Partitioning:**

- **Case 1:** If ‘nums[mid] == 0‘
 - * Swap ‘nums[low]‘ and ‘nums[mid]‘ to move the ‘0‘ to its correct position.
 - * Increment both ‘low‘ and ‘mid‘ to continue traversal.
- **Case 2:** If ‘nums[mid] == 1‘
 - * The ‘1‘ is already in the correct position; simply increment ‘mid‘.
- **Case 3:** If ‘nums[mid] == 2‘
 - * Swap ‘nums[mid]‘ and ‘nums[high]‘ to move the ‘2‘ to its correct position.
 - * Decrement ‘high‘ to reduce the range of unsorted elements.
 - * Do not increment ‘mid‘ here because the swapped element needs to be evaluated.

- **Termination:**

- The loop continues until ‘mid‘ surpasses ‘high‘, ensuring all elements are sorted.

Why This Approach

This approach is chosen for its efficiency and simplicity. By using three pointers (‘low‘, ‘mid‘, and ‘high‘), the algorithm effectively partitions the array into three sections in a single pass. This ensures a linear time complexity of $O(n)$ and constant space complexity of $O(1)$, making it highly optimal for large datasets.

Alternative Approaches

An alternative method to solve this problem is **Counting Sort**, which involves counting the occurrences of each element and then overwriting the original array based on these counts.

- **Counting Sort:**

- **Pros:**

- * Simple to implement.
- * Directly counts the frequency of each element.

- **Cons:**

- * Requires additional space proportional to the range of input values.
- * Less efficient in scenarios with a large range of elements.

While Counting Sort is effective for this problem due to the limited range of input values ('0', '1', '2'), the Dutch National Flag algorithm provides an in-place solution without requiring additional memory, making it more space-efficient.

Similar Problems to This One

There are several other problems that involve sorting or categorizing elements based on specific criteria, such as:

- Dutch National Flag Problem
- Partition Array
- Counting Sort
- Three Way Partitioning

Things to Keep in Mind and Tricks

- **Three-Pointer Technique:** Utilizing three pointers ('low', 'mid', 'high') allows for efficient in-place sorting without additional space.
- **Avoiding Unnecessary Swaps:** Only swap when necessary to minimize the number of operations and enhance performance.
- **Understanding Edge Cases:** Always consider arrays with all elements the same, already sorted arrays, and arrays with only one type of element.
- **In-Place Sorting:** Achieving the desired sort without using extra space is crucial for optimizing space complexity.
- **Pivot Selection in Related Problems:** While not directly applicable here, understanding pivot selection is beneficial for other sorting algorithms like Quick Sort.

Corner and Special Cases to Test When Writing the Code

When implementing the ‘sortColors‘ function, it is crucial to test the following edge cases to ensure robustness:

- **Empty Array:** ‘nums = []‘ should handle gracefully, possibly returning an empty array ‘[]‘.
- **Single Element:** ‘nums = [1]‘ should return ‘[1]‘.
- **All Elements the Same:** ‘nums = [2,2,2,2]‘ should return ‘[2,2,2,2]‘.
- **Already Sorted Array:** ‘nums = [0,1,2,2,1,0]‘ should return ‘[0,0,1,1,2,2]‘.
- **Reverse Sorted Array:** ‘nums = [2,2,1,1,0,0]‘ should return ‘[0,0,1,1,2,2]‘.
- **Array with No ‘0’s or ‘2’s:** ‘nums = [1,1,1,1]‘ should return ‘[1,1,1,1]‘.
- **Mixed Elements with Duplicates:** ‘nums = [0,2,1,2,1,0]‘ should return ‘[0,0,1,1,2,2]‘.
- **Large Input Size:** Test with a very large array to ensure that the implementation performs efficiently without exceeding memory limits.

Problem 8.4 Kth Largest Element in an Array

The **Kth Largest Element in an Array** problem challenges you to find the k -th largest element efficiently, without necessarily sorting the entire array. This problem is a great exercise in optimization techniques, particularly for scenarios where sorting is not feasible due to performance constraints.

Problem Statement

Given an integer array `nums` and an integer `k`, return the k -th largest element in the array. Note that the k -th largest element is determined in sorted order, not based on distinct values.

Input: - `nums`: A list of integers. - `k`: An integer representing the desired rank.

Output: - An integer representing the k -th largest element in `nums`.

Example 1:

Input: `nums = [3, 2, 1, 5, 6, 4]`, $k = 2$

Output: 5

Explanation: The sorted array is `[1, 2, 3, 4, 5, 6]`. The 2nd largest is 5.

Example 2:

Input: `nums = [3, 2, 3, 1, 2, 4, 5, 5, 6]`, $k = 4$

Output: 4

Explanation: The sorted array is `[1, 2, 2, 3, 3, 4, 5, 5, 6]`. The 4th largest is 4.

Constraints: $-1 \leq \text{nums.length} \leq 10^5$ - $-10^4 \leq \text{nums}[i] \leq 10^4$ - $1 \leq k \leq \text{nums.length}$

—

Algorithmic Approaches

This problem can be efficiently solved using the following techniques:

1. Min-Heap Approach ($O(n \log k)$)

Use a min-heap of size k to maintain the k -th largest elements seen so far: 1. Push each element into the heap. 2. If the heap size exceeds k , remove the smallest element (top of the heap). 3. At the end, the top of the heap is the k -th largest element.

2. Quickselect ($O(n)$ Average Case)

Quickselect is an optimization of the QuickSort algorithm that focuses only on the partition containing the desired k -th largest element: 1. Partition the array around a pivot element. 2. Compare the pivot's position to $n - k$ (index for k -th largest element in sorted order): - If equal, return the pivot. - If smaller, search the right partition. - If larger, search the left partition. 3. Repeat until the k -th largest element is found.

—

Complexities

1. **Min-Heap Approach:** - Time Complexity: $O(n \log k)$, as each insertion/removal operation in the heap is $O(\log k)$. - Space Complexity: $O(k)$, for the heap.

2. **Quickselect:** - Time Complexity: $O(n)$ on average, $O(n^2)$ in the worst case (unbalanced partitions). - Space Complexity: $O(1)$, as it operates in-place.

—

Python Implementations

Min-Heap Approach ($O(n \log k)$)

```
import heapq

def findKthLargest(nums, k):
    heap = []
    for num in nums:
        heapq.heappush(heap, num)
        if len(heap) > k:
            heapq.heappop(heap)
    return heapq.heappop(heap)

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

Quickselect Approach ($O(n)$ Average Case)

```
import random

def findKthLargest(nums, k):
    def partition(left, right, pivot_index):
        pivot = nums[pivot_index]
        nums[pivot_index], nums[right] = nums[right], nums[pivot_index]
        store_index = left
        for i in range(left, right):
            if nums[i] > pivot:
                nums[store_index], nums[i] = nums[i], nums[store_index]
                store_index += 1
        nums[store_index], nums[right] = nums[right], nums[store_index]
        return store_index

    left, right = 0, len(nums) - 1
    while True:
        pivot_index = random.randint(left, right)
        pivot_index = partition(left, right, pivot_index)
        if pivot_index == k - 1:
            return nums[pivot_index]
        elif pivot_index < k - 1:
            left = pivot_index + 1
        else:
            right = pivot_index - 1

# Example usage:
nums = [3, 2, 1, 5, 6, 4]
k = 2
```

```
print(findKthLargest(nums, k)) # Output: 5
```

Why These Approaches?

- The Min-Heap approach is intuitive and robust, especially for streaming data or cases where k is small.
- Quickselect is ideal for large datasets as it is faster on average and has $O(1)$ space complexity.

Similar Problems

1. **Kth Smallest Element in a Sorted Matrix:** A similar ranking problem in a 2D matrix.
2. **Find Median of Two Sorted Arrays:** Combines binary search and partitioning concepts.
3. **Top K Frequent Elements:** Uses heaps or hash maps to find the most frequent elements.

Corner Cases to Test

1. $k = 1$: The largest element in the array.
2. $k = \text{len}(\text{nums})$: The smallest element in the array.
3. Array with duplicates: Ensure the rank logic works correctly.
4. Array with negative numbers: Test arrays containing both positive and negative integers.

Conclusion

The k -th Largest Element in an Array problem highlights the importance of efficient selection algorithms like Quickselect and heap-based optimizations. These techniques ensure performance even for large datasets and demonstrate the versatility of divide-and-conquer and heap-based approaches in real-world scenarios.

Chapter 9

Two-Pointer Technique

9.1 Two Pointers Technique

Introduction

The Two Pointers Technique is a fundamental algorithmic strategy widely used in solving array and string manipulation problems. By employing two indices (pointers) that traverse the data structure from different directions or positions, this method enables efficient processing of elements to meet specific conditions¹. Particularly effective for sorted data, the technique helps in reducing time and space complexities compared to brute-force approaches. Its simplicity and versatility make it an indispensable tool for tackling a variety of computational challenges, such as searching for pairs that satisfy a given sum, removing duplicates, and more².

¹ For an in-depth explanation, refer to the GeeksforGeeks Two Pointers Technique article.

² Explore related problems on LeetCode's Problem Set.

Problem 9.1 Find All Pairs with a Given Target Sum

Given a *sorted* array of integers and a target sum, find all unique pairs in the array that add up to the target sum. The array is sorted, which means the elements are in non-decreasing order. This sorted property allows us to use the Two Pointers Technique efficiently³. Specifically, you need to find two numbers in the sorted array such that their sum equals the target number. Return the indices of these two numbers (1-based index) as an integer array of size two, where $1 \leq \text{index1} < \text{index2} \leq \text{array.length}$. You can assume that there is exactly one solution, and you cannot use the same element twice.

³ Sorted arrays are ideal for this technique as they facilitate straightforward element comparisons.

Algorithmic Approach

1. Initialize Two Pointers:

- `left` pointer starts at the beginning of the array⁴.
- `right` pointer starts at the end of the array⁵.

2. Traverse the Array:

- While `left` is less than `right`:
 - Calculate the current sum: `current_sum = nums[left] + nums[right]`⁶.
 - If `current_sum` equals the target, record the pair and move both pointers inward⁷.
 - If `current_sum` is less than the target, move the `left` pointer to the right to increase the sum⁸.
 - If `current_sum` is greater than the target, move the `right` pointer to the left to decrease the sum⁹.

⁴ This pointer moves forward to find larger values when needed.

⁵ This pointer moves backward to find smaller values when needed.

3. Completion:

4. Continue the process until the `left` and `right` pointers meet.

Python Implementation

```
def pair_sum(nums, target):
    """
    Finds all unique pairs in a sorted array that add up to the target sum.

    Parameters:
    nums (List[int]): The input sorted array of integers.
    target (int): The target sum.

    Returns:
    List[List[int]]: A list of unique pairs that add up to the target.
    """
    left, right = 0, len(nums) - 1
    result = []

    while left < right:
        current_sum = nums[left] + nums[right]
        if current_sum == target:
            result.append([nums[left], nums[right]])
            left += 1
            right -= 1
            # Skip duplicates
            while left < right and nums[left] == nums[left - 1]:
                left += 1
            while left < right and nums[right] == nums[right + 1]:
                right -= 1
        elif current_sum < target:
            left += 1
        else:
            right -= 1
```

```

    else:
        right -= 1
    return result

# Example usage:
nums = [1, 2, 3, 4, 5]
target = 6
print(pair_sum(nums, target))  # Output: [[1, 5], [2, 4]]

```

Example Usage and Test Cases

```

# Test case 1: General case
nums = [1, 2, 3, 4, 5]
target = 6
print(pair_sum(nums, target))  # Output: [[1, 5], [2, 4]]

# Test case 2: No pairs found
nums = [1, 2, 3, 9]
target = 8
print(pair_sum(nums, target))  # Output: []

# Test case 3: Multiple pairs with duplicates
nums = [1, 1, 2, 2, 3, 4]
target = 4
print(pair_sum(nums, target))  # Output: [[1, 3], [2, 2]]

# Test case 4: All elements are the same
nums = [2, 2, 2, 2]
target = 4
print(pair_sum(nums, target))  # Output: [[2, 2]]

# Test case 5: Single element array
nums = [1]
target = 2
print(pair_sum(nums, target))  # Output: []

```

Why This Approach

The two pointers technique is chosen for its **efficiency and simplicity**. By leveraging the sorted nature of the array, the algorithm avoids the need for nested loops, reducing the time complexity from $O(n^2)$ in a brute-force approach to $O(n)$ ¹⁰. Additionally, this method ensures that each element is processed only once, making it highly suitable for large datasets¹¹.

¹⁰ Linear time complexity is optimal for this problem

¹¹ Single-pass algorithms are preferable for handling large inputs efficiently

Complexity Analysis

- **Time Complexity:** $O(n)$ ¹².
- **Space Complexity:** $O(1)$ (excluding the space for the output list)¹³.

¹² Each element is visited at most once by either pointer

¹³ No additional space proportional to the input size is used

Similar Problems

Other problems that can be efficiently solved using the two pointers technique include:

- **Remove Duplicates from a Sorted Array:** Modify the array in-place to remove duplicates¹⁴.
- **Reverse a String or Array In-Place:** Reverse the elements by swapping from both ends¹⁵.
- **Container With Most Water:** Find two lines that together with the x-axis form a container that holds the most water¹⁶.
- **3Sum Problem:** Find all unique triplets in the array which gives the sum of zero¹⁷.

¹⁴ A fundamental application of the two pointers technique

¹⁵ Utilizes pointers moving towards the center

¹⁶ Maximizing area by adjusting pointers based on height comparisons

¹⁷ An extension of the two pointers technique to handle three elements

Things to Keep in Mind and Tricks

- **Sorted vs. Unsorted Arrays:** The two pointers technique is most effective with sorted arrays¹⁸.
- **Handling Duplicates:** When the array contains duplicates, ensure that your algorithm skips over them to avoid redundant pairs¹⁹.
- **In-Place Modifications:** This technique is ideal for in-place modifications, reducing the need for extra space²⁰.
- **Edge Cases:** Always consider edge cases such as empty arrays, single-element arrays, and scenarios where no valid pairs exist²¹.
- **Pointer Movement Logic:** Clearly define the conditions under which each pointer should move to ensure optimal traversal²².

¹⁸ If the array is unsorted, consider sorting it first if the problem allows

¹⁹ Skipping duplicates maintains the uniqueness of the result pairs

²⁰ In-place algorithms are space-efficient and often faster

²¹ Handling edge cases ensures robustness of the algorithm

²² Proper pointer management is crucial for achieving the desired time complexity

Exercises

1. **Container With Most Water:** Given n non-negative integers a_1, a_2, \dots, a_n , where each represents a point at coordinate (i, a_i) , find two lines that together with the x-axis form a container that holds the most water²³.

²³ Optimize the container area by adjusting pointers based on height comparisons

2. **Remove Duplicates from Sorted Array:** Given a sorted array $nums$, remove the duplicates in-place such that each element appears only once and returns the new length²⁴.

²⁴ Practice in-place array manipulation with two pointers

3. **3Sum Problem:** Given an array $nums$ of n integers, are there elements a, b, c in $nums$ such that $a + b + c = 0$? Find all unique triplets in the array which gives the sum of zero²⁵.

²⁵ Extend the two pointers technique to handle three elements

4. **Find All Pairs with a Given Difference:** Given an array of integers and a difference value, find all unique pairs in the array that have the given difference²⁶.

²⁶ Apply the two pointers technique to identify pairs based on the specified difference

5. **Two Sum II - Input Array Is Sorted:** Given a 1-indexed array of integers that is already sorted in non-decreasing order, find two numbers such that they add up to a specific target number. Return the indices of the two numbers²⁷.

²⁷ Use the two pointers technique to efficiently locate the target pair

Questions for Reflection

- How does the two pointers technique compare to other approaches like hashing in terms of time and space complexity?²⁸.
- In what scenarios might the two pointers technique not be the most efficient method?²⁹.
- How can you adapt the two pointers technique to handle more than two pointers for solving complex problems?³⁰.
- Can the two pointers technique be combined with other techniques like sliding windows or dynamic programming to solve advanced problems?³¹.

²⁸ Consider scenarios where space optimization is critical

²⁹ Evaluate based on data structure properties and problem constraints

³⁰ Think about extending the technique to accommodate additional conditions or elements

³¹ Explore hybrid approaches for enhanced problem-solving capabilities

References

LeetCode Problem: ³²

³² Two Sum II - Input Array Is Sorted

LeetCode Problem: ³³

³³ 3Sum

GeeksforGeeks Article: ³⁴

³⁴ Two Pointers Technique

HackerRank Problem: ³⁵

³⁵ Two Sum

Conclusion

The Two Pointers Technique is an indispensable tool in the array and string manipulation arsenal. By leveraging the inherent order within data structures, it enables the development of efficient and elegant solutions to a wide range of problems³⁶. Mastering this technique not only enhances problem-solving skills but also prepares you for tackling more complex algorithmic challenges effectively.

³⁶ Its applicability spans numerous algorithmic challenges, making it a versatile strategy

Problem 9.2 3Sum

The **3Sum** problem is a classic challenge in algorithmic tasks, involving finding all unique triplets in an array that sum up to zero.

Problem Statement

Given an array `nums` of n integers, determine whether there are elements a, b, c in `nums` such that $a + b + c = 0$ and find all unique triplets in the array that give the sum of zero³⁷.

³⁷ Refer to the LeetCode 3Sum Problem for more details.

Example:

Given array `nums` = $[-1, 0, 1, 2, -1, -4]$,

A solution set is:

$[-1, 0, 1],$
 $[-1, -1, 2]$

³⁸

³⁸ This example demonstrates how the algorithm identifies unique triplets that sum to zero, even in the presence of duplicate elements.

Algorithmic Approach

The solution to this problem can be approached by using a sorting-based approach along with the Two Pointers Technique³⁹. First, sort the array to make it easier to navigate and to avoid duplicate solutions. Iterate through each element i in the array and for each, apply the Two Pointers Technique to find the other two elements that sum to the negative of i . Manage the two pointers to skip over duplicate values and find the unique triplets that satisfy the condition.

³⁹ Sorting the array helps in efficiently managing duplicates and navigating the array with two pointers.

Python Implementation

```
class Solution:
    def threeSum(self, nums):
        """
        Finds all unique triplets in the array which gives the sum of zero.

        Parameters:
        nums (List[int]): The input array of integers.

        Returns:
        List[List[int]]: A list of unique triplets that sum up to zero.
        """
        res = []
        nums.sort()

        for i in range(len(nums)-2):
```

```

        if i > 0 and nums[i] == nums[i-1]:
            continue
        left, right = i+1, len(nums)-1
        while left < right:
            current_sum = nums[i] + nums[left] + nums[right]
            if current_sum < 0:
                left += 1
            elif current_sum > 0:
                right -= 1
            else:
                res.append([nums[i], nums[left], nums[right]])
                while left < right and nums[left] == nums[left+1]:
                    left += 1
                while left < right and nums[right] == nums[right-1]:
                    right -= 1
                left += 1
                right -= 1
        return res

# Example usage:
nums = [-1, 0, 1, 2, -1, -4]
solution = Solution()
print(solution.threeSum(nums)) # Output: [[-1, -1, 2], [-1, 0, 1]]

```

Example Usage and Test Cases

```

# Test case 1: General case
nums = [-1, 0, 1, 2, -1, -4]
print(Solution().threeSum(nums)) # Output: [[-1, -1, 2],
                                ↪ [-1, 0, 1]]

# Test case 2: No triplet sums to zero
nums = [1, 2, 3, 4]
print(Solution().threeSum(nums)) # Output: []

# Test case 3: Multiple triplets with duplicates
nums = [-2, 0, 0, 2, 2]
print(Solution().threeSum(nums)) # Output: [[-2, 0, 2]]

# Test case 4: All elements are zero
nums = [0, 0, 0, 0]
print(Solution().threeSum(nums)) # Output: [[0, 0, 0]]

# Test case 5: Mixed positive and negative numbers
nums = [-4, -1, -1, 0, 1, 2]
print(Solution().threeSum(nums)) # Output: [[-1, -1, 2],
                                ↪ [-1, 0, 1]]

```

Why This Approach

The Two Pointers Technique combined with a sorting-based approach is chosen for its **efficiency and simplicity**. By leveraging the sorted nature of the array, the algorithm avoids the need for nested loops, reducing the time complexity from $O(n^3)$ in a brute-force approach to $O(n^2)$ ⁴⁰. Additionally, this method ensures that each element is processed only once, making it highly suitable for large datasets⁴¹.

⁴⁰ Linearithmic sorting followed by quadratic traversal results in overall $O(n^2)$ time complexity

⁴¹ Single-pass algorithms are preferable for handling large inputs efficiently

Alternative Approaches

An alternative brute-force approach is to use three nested loops to check every triplet, but this would result in a time complexity of $O(n^3)$ and is not efficient for larger arrays⁴². Other approaches may involve using a hash set to check for complements⁴³, but care must be taken to still avoid duplicates.

⁴² Such approaches are impractical for large-scale data due to their high time complexity

⁴³ Hash-based solutions can offer linear time complexity but may require additional space and careful handling of duplicates

Similar Problems

Similar problems include:

- **2Sum:** Find pairs that sum up to a target value⁴⁴.
- **4Sum:** Find quadruplets that sum up to a target value⁴⁵.
- **3Sum Closest:** Find the triplet with a sum closest to a target value⁴⁶.
- **3Sum Smaller:** Count the number of triplets with a sum smaller than a target⁴⁷.

⁴⁴ A simpler version of the problem focusing on pairs instead of triplets

⁴⁵ An extension of the 3Sum problem requiring additional pointer management

⁴⁶ Requires tracking the closest sum while iterating

⁴⁷ Involves similar traversal logic with conditional counting

These problems can often be approached with similar sorting and two-pointer techniques, or hashing strategies⁴⁸.

⁴⁸ Understanding the core technique facilitates tackling a variety of related challenges

Things to Keep in Mind and Tricks

- **Handling Duplicates:** It's crucial to handle duplicates carefully to ensure that only unique triplets are included in the result⁴⁹.
- **Sorted Array Advantage:** Sorting the array simplifies the process of finding triplets and managing pointers⁵⁰.
- **Pointer Movement Logic:** Clearly define the conditions under which each pointer should move to ensure optimal traversal⁵¹.
- **Edge Cases:** Always consider edge cases such as arrays with all positive or all negative numbers, and arrays with insufficient elements⁵².
- **Space Optimization:** The two pointers technique allows for solving the problem without additional space⁵³.

⁴⁹ Skipping over duplicate elements during traversal prevents redundant triplet entries

⁵⁰ Sorted arrays allow for predictable pointer movements based on sum comparisons

⁵¹ Proper management of the left and right pointers is key to maintaining efficiency

⁵² Robust handling of edge cases ensures the algorithm's reliability across diverse inputs

⁵³ In-place algorithms are preferable for optimizing space usage, especially with large datasets

Corner and Special Cases to Test When Writing the Code

Special cases include:

- **All Zeroes:** Arrays where all elements are zero⁵⁴.
- **No Valid Triplets:** Arrays where no three numbers sum to zero⁵⁵.
- **Multiple Duplicates:** Arrays with multiple duplicate elements⁵⁶.
- **Mixed Positive and Negative Numbers:** Arrays containing both positive and negative integers⁵⁷.
- **Single or Two Elements:** Arrays with fewer than three elements⁵⁸.

⁵⁴ Ensures that the algorithm correctly identifies triplets of zeroes without duplication

⁵⁵ Checks the algorithm's ability to return an empty list appropriately

⁵⁶ Verifies that the algorithm skips duplicates correctly to avoid redundant triplets

⁵⁷ Tests the algorithm's capability to handle a diverse range of input values

⁵⁸ The algorithm should handle these gracefully, typically returning an empty list

References

LeetCode Problem: ⁵⁹

⁵⁹ 3Sum

GeeksforGeeks Article: ⁶⁰

⁶⁰ Two Pointers Technique

HackerRank Problem: ⁶¹

⁶¹ Two Sum

Conclusion

The Two Pointers Technique is an indispensable tool in the array and string manipulation arsenal. By leveraging the inherent order within data structures, it enables the development of efficient and elegant solutions to a wide range of problems⁶². Mastering this technique not only enhances problem-solving skills but also prepares you for tackling more complex algorithmic challenges effectively.

⁶² Its applicability spans numerous algorithmic challenges, making it a versatile strategy

Problem 9.3 Container With Most Water

The "Container With Most Water" problem, often referred to as "Max Area," is a classic problem that challenges us to find the two lines that together with the x-axis forms a container that could hold the maximum amount of water.

Problem Statement

You are given an array `height` which represents the height of n vertical lines drawn on the x-axis at positions $[0, 1, 2, \dots, n - 1]$. Each line is drawn between the x-axis and the point $(x, \text{height}[x])$.

The goal is to select two lines that form a container along with the x-axis, such that it contains the most water. The container cannot be slanted, and n must be at least 2.

Input: An array of non-negative integers `height`.

Output: The maximum volume of water a container can store.

Example:

Input: `height = [1, 8, 6, 2, 5, 4, 8, 3, 7]`

Output: 49

Explanation: The max area of water (blue section) the container can contain is 49.

Input: `height = [1, 1]`

Output: 1

Algorithmic Approach

We can solve this problem with a two-pointer approach. The two pointers are initialized at both ends of the array, and we move the pointer pointing to the shorter line towards the other pointer. This is because if we moved the pointer at the taller line, the height of the container would remain the same or decrease, and the width would also decrease, necessarily resulting in a smaller area.

Complexities

- **Time Complexity:** The two-pointer approach effectively scans the array once, giving a time complexity of $O(n)$.
- **Space Complexity:** Since the approach uses only a constant amount of extra space, the space complexity is $O(1)$.

ewpage

Python Implementation

Below is the complete Python code that employs the two-pointer technique to solve the problem:

```
class Solution:
    def maxArea(self, height):
        max_water = 0
        left, right = 0, len(height) - 1
```

```

while left < right:
    width = right - left
    max_water = max(max_water, min(height[left], height[right]) * width)

    if height[left] < height[right]:
        left += 1
    else:
        right -= 1

return max_water

```

This implementation finds the maximum area by iterating through the array only once. The pointers move towards each other, and after each step, the maximum area is updated if a larger area is found.

Why this approach

The two-pointer approach was chosen for its efficiency, both in terms of time and space complexity. By constraining the container's vertical boundaries to the heights in the array, and then moving inward, we ensure that we only consider potentially larger areas without having to check each possible pair of lines.

Alternative approaches

A brute force solution could be used, where one would check every possible pair of lines. However, this approach would have a time complexity of $O(n^2)$, making it inefficient for larger input sizes.

Similar problems to this one

There are other problems involving arrays and two-pointer techniques such as "Two Sum", "3Sum", "Trapping Rain Water", and others that deal with similar ideas of using pointers to navigate and compute values based on the array elements.

Things to keep in mind and tricks

- Remember that moving the pointer at the taller line inward will never yield a larger area.
- The width between the lines decreases with every move, so only consider moves that might increase the height to possibly increase the area.

Corner and special cases to test when writing the code

- Arrays with all elements being the same value (e.g., [1, 1, 1, 1]). - Arrays with only two elements. - Arrays with large differences in height values. - Cases where the maximum area is formed by lines that are not the furthest apart.

Problem 9.4 Remove Duplicates from a Sorted Array

Introduction

The problem of removing duplicates from a sorted array is a classic algorithmic challenge that emphasizes in-place array manipulation and efficient use of space. Leveraging the sorted nature of the array allows for optimal solutions that minimize time and space complexity. This problem not only reinforces fundamental array handling techniques but also serves as a foundational exercise for understanding more complex data manipulation tasks⁶³.

⁶³ For a comprehensive overview, refer to the LeetCode Remove Duplicates problem.

Problem Statement

Given a sorted array of integers, remove the duplicates in-place such that each element appears only once and return the new length. The relative order of the elements should be kept the same. Since it is impossible to change the length of the array in some programming languages, you must instead have the result placed in the first part of the array. More formally, if there are k elements after removing the duplicates, then the first k elements of the array should hold the final result. It does not matter what you leave beyond the first k elements.

Algorithmic Approach

The most efficient way to solve this problem is by using the **Two Pointers Technique**. Here's a step-by-step approach:

1. Initialize Two Pointers:

- `slow` pointer starts at index 0, representing the position of the last unique element found⁶⁴.
- `fast` pointer starts at index 1, traversing the array to find unique elements⁶⁵.

⁶⁴ This pointer tracks the position where the next unique element should be placed.

⁶⁵ The `fast` pointer scans through the array to identify unique elements.

2. Traverse the Array:

- While `fast` is less than the length of the array:
 - If the element at `fast` is not equal to the element at `slow`, it means a new unique element is found⁶⁶.

⁶⁶ Since the array is sorted, duplicates are adjacent, making it easy to detect new unique elements.

- Increment `slow` and update the element at `slow` with the element at `fast`⁶⁷.
- Increment `fast` to continue traversing.

⁶⁷ This effectively moves the unique element to the front of the array.

3. Completion:

- After traversal, the value of `slow + 1` will represent the number of unique elements.

Python Implementation

```
def remove_duplicates(nums):
    """
    Removes duplicates from a sorted array in-place.

    Parameters:
    nums (List[int]): The input sorted array of integers.

    Returns:
    int: The number of unique elements after removing duplicates.
    """
    if not nums:
        return 0

    slow = 0
    for fast in range(1, len(nums)):
        if nums[fast] != nums[slow]:
            slow += 1
            nums[slow] = nums[fast]

    return slow + 1

# Example usage:
nums = [1, 1, 2]
k = remove_duplicates(nums)
print(k)          # Output: 2
print(nums[:k])  # Output: [1, 2]
```

Example Usage and Test Cases

```
# Test case 1: General case with duplicates
nums = [1, 1, 2]
k = remove_duplicates(nums)
print(k)          # Output: 2
print(nums[:k])  # Output: [1, 2]

# Test case 2: All elements are duplicates
nums = [2, 2, 2, 2]
```

```

k = remove_duplicates(nums)
print(k)          # Output: 1
print(nums[:k]) # Output: [2]

# Test case 3: No duplicates
nums = [1, 2, 3, 4, 5]
k = remove_duplicates(nums)
print(k)          # Output: 5
print(nums[:k]) # Output: [1, 2, 3, 4, 5]

# Test case 4: Empty array
nums = []
k = remove_duplicates(nums)
print(k)          # Output: 0
print(nums[:k]) # Output: []

# Test case 5: Single element array
nums = [1]
k = remove_duplicates(nums)
print(k)          # Output: 1
print(nums[:k]) # Output: [1]

```

Why This Approach

The **Two Pointers Technique** is particularly effective for this problem due to the following reasons:

- **In-Place Modification:** Eliminates the need for additional memory by modifying the array directly⁶⁸.
- **Linear Time Complexity:** Traverses the array only once, achieving $O(n)$ time complexity⁶⁹.
- **Utilizes Sorted Property:** The sorted nature of the array ensures that duplicates are adjacent, simplifying the detection and removal process⁷⁰.
- **Simplicity:** The algorithm is straightforward, making it easy to implement and understand⁷¹.

⁶⁸ This is crucial for optimizing space usage, especially with large datasets.

⁶⁹ This ensures that the algorithm remains efficient even as the size of the input grows.

⁷⁰ Sorting guarantees that all duplicates are clustered together, making it easier to identify unique elements.

⁷¹ Clear and concise logic reduces the likelihood of errors during implementation.

Complexity Analysis

- **Time Complexity:** $O(n)$, where n is the number of elements in the array. Each element is visited at most once⁷².
- **Space Complexity:** $O(1)$, as the algorithm uses a constant amount of extra space⁷³.

⁷² The **fast** pointer ensures a single pass through the array.

⁷³ No additional data structures are required beyond the input array itself.

Similar Problems

Other problems that can be efficiently solved using the two pointers technique include:

- **Two Sum II - Input Array Is Sorted:** Find two numbers that add up to a specific target number⁷⁴.
- **3Sum Problem:** Find all unique triplets in the array which give the sum of zero⁷⁵.
- **Container With Most Water:** Find two lines that together with the x-axis form a container that holds the most water⁷⁶.
- **Reverse a String or Array In-Place:** Reverse the elements by swapping from both ends⁷⁷.

⁷⁴ This problem leverages the sorted property to efficiently locate the desired pair.

⁷⁵ Extending the two pointers approach to handle three elements.

⁷⁶ Maximizing the area between two pointers.

⁷⁷ A fundamental application of the two pointers technique.

Things to Keep in Mind and Tricks

- **Sorted vs. Unsorted Arrays:** This approach relies on the array being sorted⁷⁸.
- **In-Place Modification Constraints:** Ensure that the environment or language allows in-place modifications of the data structure⁷⁹.
- **Handling Edge Cases:** Always consider edge cases such as empty arrays, single-element arrays, and arrays with all duplicates⁸⁰.
- **Pointer Initialization:** Correctly initialize the pointers to avoid index out-of-bound errors⁸¹.
- **Understanding Return Values:** Depending on the problem's requirements, ensure that you return the correct value representing the number of unique elements⁸².

⁷⁸ If the array is unsorted, consider sorting it first if the problem allows.

⁷⁹ Some languages may have immutable data structures, requiring alternative approaches.

⁸⁰ Robustness against various input scenarios is crucial for algorithm reliability.

⁸¹ Proper starting points ensure that the algorithm functions as intended.

⁸² Clarify what the function is expected to return to meet the problem's specifications.

Related Problems

1. **Two Sum II - Input Array Is Sorted:** Given a 1-indexed array of integers that is already sorted in non-decreasing order, find two numbers such that they add up to a specific target number⁸³.
2. **3Sum Problem:** Given an array $nums$ of n integers, are there elements a, b, c in $nums$ such that $a + b + c = 0$? Find all unique triplets in the array which gives the sum of zero⁸⁴.
3. **Move Zeroes:** Given an array $nums$, write a function to move all 0's to the end of it while maintaining the relative order of the non-zero elements⁸⁵.

⁸³ Leverage the two pointers technique for an efficient solution.

⁸⁴ This problem extends the two pointers approach to three elements.

⁸⁵ This requires careful pointer management to preserve element order.

4. **Valid Palindrome II:** Given a string, determine if it can become a palindrome by removing at most one character⁸⁶.

⁸⁶ Combining two pointers with conditional checks.

5. **Minimum Size Subarray Sum:** Given an array of positive integers and a positive integer s , find the minimal length of a contiguous subarray for which the sum is at least s . If there isn't one, return 0 instead⁸⁷.

⁸⁷ This problem can be approached using a sliding window technique, often used alongside two pointers.

Questions for Reflection

- How does the two pointers technique optimize space compared to using additional data structures like hash sets?⁸⁸
- In what scenarios might the two pointers technique not be applicable or efficient?⁸⁹.
- How can the two pointers technique be extended to handle more complex problems involving multiple conditions?⁹⁰.
- Can the two pointers technique be combined with other algorithmic strategies, such as binary search or dynamic programming, to solve advanced problems?⁹¹.

⁸⁸ Consider scenarios where space complexity is a critical factor.

⁸⁹ Evaluate the limitations based on data structure properties.

⁹⁰ Think about integrating additional logic or combining with other techniques.

⁹¹ Exploring hybrid approaches for enhanced problem-solving.

References

LeetCode Problem: ⁹²

⁹² Remove Duplicates from Sorted Array

GeeksforGeeks Article: ⁹³

⁹³ Remove Duplicates from an Unsorted Linked List

HackerRank Problem: ⁹⁴

⁹⁴ Remove Duplicates

Tutorialspoint Article: ⁹⁵

⁹⁵ Python List Remove Duplicates

Conclusion

Removing duplicates from a sorted array is a fundamental problem that exemplifies the power of the Two Pointers Technique in optimizing both time and space complexities. By intelligently traversing the array with two pointers, the algorithm efficiently eliminates redundant elements without the need for extra storage⁹⁶. Mastery of this technique not only aids in solving similar array manipulation problems but also lays the groundwork for tackling more intricate algorithmic challenges in the realm of data structures and computational problem-solving.

⁹⁶ This approach is not only efficient but also elegant in its simplicity.

Problem 9.5 Valid Palindrome

The **Valid Palindrome** problem tests your ability to process and manipulate strings efficiently while applying basic string traversal techniques. A string is con-

sidered a palindrome if it reads the same forward and backward after ignoring non-alphanumeric characters and case differences.

Problem Statement

Given a string s , determine if it is a palindrome, considering only alphanumeric characters and ignoring case.

Input: - A string s .

Output: - A boolean indicating whether s is a valid palindrome.

Example 1:

Input: $s = "A man, a plan, a canal: Panama"$

Output: true

Example 2:

Input: $s = "race a car"$

Output: false

Example 3:

Input: $s = ""$

Output: true

Algorithmic Approach

A common and efficient way to solve this problem is by using the two-pointer technique. This method involves placing two pointers, one at the start and the other at the end of the string, and moving them toward each other while verifying that the characters they point to are equal.

Steps:

- Initialize two pointers, `left` at the start of the string and `right` at the end.
- While `left` is less than `right`:
 - Skip non-alphanumeric characters by advancing `left` or decrementing `right`.
 - Compare the characters at `left` and `right`, ignoring case⁹⁷.
 - If the characters do not match, return `false`.
 - Otherwise, move `left` and `right` inward.
- If the loop completes without mismatches, return `true`.

⁹⁷ Use built-in functions like `str.lower()` to handle case insensitivity

Complexities

- **Time Complexity:** $O(n)$, where n is the length of the string. Each character is processed at most once.
- **Space Complexity:** $O(1)$, as no additional space is required apart from a few pointers.

Python Implementation

Below is an efficient implementation of the two-pointer technique:

```
def isPalindrome(s: str) -> bool:
    left, right = 0, len(s) - 1

    while left < right:
        # Skip non-alphanumeric characters
        while left < right and not s[left].isalnum():
            left += 1
        while left < right and not s[right].isalnum():
            right -= 1

        # Compare characters, ignoring case
        if s[left].lower() != s[right].lower():
            return False

        # Move pointers inward
        left += 1
        right -= 1

    return True
```

Why This Approach?

The two-pointer technique is optimal for this problem as it processes the string in a single pass, ensuring linear time complexity. By avoiding the creation of additional strings or arrays, it also achieves constant space complexity. The approach effectively handles edge cases such as non-alphanumeric characters and empty strings.

Alternative Approaches

- **Preprocessed String:** Create a filtered version of the string containing only lowercase alphanumeric characters, then check if this filtered string is equal to its reverse. This approach is simple but less efficient due to the need for additional memory and processing⁹⁸.

⁹⁸ Constructing a new string requires $O(n)$ time and space

- **Recursive Check:** Use recursion to compare characters at the start and end of the string while skipping non-alphanumeric characters. This is less practical due to the risk of stack overflow for large strings.

Similar Problems

- **Valid Palindrome II:** Check if a string can be a palindrome after removing at most one character.
- **Longest Palindromic Substring:** Find the longest substring that is a palindrome.
- **Palindrome Permutation:** Determine if a string can be rearranged to form a palindrome.

Things to Keep in Mind and Tricks

- **Handling Non-Alphanumeric Characters:** Use Python's `str.isalnum()` method to check if a character is alphanumeric.
- **Case Insensitivity:** Convert characters to lowercase using `str.lower()` for comparison⁹⁹.
- **Edge Cases:** Consider empty strings and strings with only non-alphanumeric characters, as these are valid palindromes.

⁹⁹ Avoid manually checking ASCII values for simplicity

Corner and Special Cases to Test

- **Empty String:** Input: "" (should return `true`).
- **Only Non-Alphanumeric Characters:** Input: "!!!!" (should return `true`).
- **Mixed Alphanumeric and Special Characters:** Input: "A man, a plan, a canal: Panama" (should return `true`).
- **Single Character:** Input: "a" (should return `true`).
- **Case Insensitivity:** Input: "RaceCar" (should return `true`).

Conclusion

The **Valid Palindrome** problem demonstrates how simple string traversal techniques, such as the two-pointer approach, can efficiently solve challenges involving character comparisons. Mastering this problem builds a strong foundation for tackling more complex problems involving strings and character manipulations.

Problem 9.6 Trapping Rain Water

Problem Description:

Given an array `height` 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

Example 2:

- **Input:** `height = [4,2,0,3,2,5]`
- **Output:** 9

Solution Overview:

Use a stack to keep track of the bars that are bounded by taller bars and hence can trap water. Iterate through the elevation map:

- For each bar at index i :
 - While the stack is not empty and $height[i] > height[stack[-1]]$:
 - * Pop the top of the stack as `bottom`.
 - * If the stack becomes empty, break.
 - * Calculate the distance between the current index and the new top of the stack minus one.
 - * Find the bounded height by taking the minimum of $height[i]$ and $height[stack[-1]]$ minus $height[bottom]$.
 - * Add the trapped water to the total.
 - Push i onto the stack.

Code Implementation:

```
def trap(height):
    stack = []
    water = 0
    i = 0
    while i < len(height):
        while stack and height[i] > height[stack[-1]]:
            bottom = stack.pop()
            if not stack:
```

This problem calculates the total trapped rainwater using a stack-based approach.

```

        break
    distance = i - stack[-1] - 1
    bounded_height = min(height[i], height[stack
        ↪ [-1]]) - height[bottom]
    water += distance * bounded_height
    stack.append(i)
    i += 1
return water

# Example usage:
print(trap([0,1,0,2,1,0,1,3,2,1,2,1])) # Output: 6
print(trap([4,2,0,3,2,5]))             # Output: 9

```

Problem 9.7 Longest Substring with At Most Two Distinct Characters

The **Longest Substring with At Most Two Distinct Characters** problem requires finding the length of the longest substring in a given string that contains no more than two distinct characters. This problem effectively demonstrates the application of the sliding window technique to manage dynamic constraints within substrings.

This problem exemplifies the sliding window technique, a powerful method for handling substring constraints efficiently.

Problem Statement

Given a string s , find the length of the longest substring that contains at most two distinct characters.

Input: A string s .

Output: The length of the longest substring containing at most two distinct characters.

Examples:

- **Example 1:**

Input: $s = "eceba"$

Output: 3

Explanation: The substring is "ece" with length 3.

- **Example 2:**

Input: $s = "ccaaabbb"$

Output: 5

Explanation: The substring is "aabbb" with length 5.

LeetCode link: Longest Substring with At Most Two Distinct Characters

[\[LeetCode Link\]](#)

[\[GeeksForGeeks Link\]](#)

[\[HackerRank Link\]](#)

[\[CodeSignal Link\]](#)

[\[InterviewBit Link\]](#)

[\[Educative Link\]](#)

[\[Codewars Link\]](#)

Algorithmic Approach

To solve this problem, we utilize the sliding window technique. This method involves maintaining a window that can expand and contract based on the number of distinct characters it contains. By using a hash map to keep track of character frequencies within the window, we can efficiently manage and update the window's constraints.

1. Initialize Pointers and Data Structures:

- ‘start’ and ‘end’ pointers to define the current window.
- A hash map ‘char-count’ to store the frequency of each character within the window.

2. Expand the Window:

- Increment the ‘end’ pointer to include a new character.
- Update the frequency of the new character in ‘char-count’.

3. Contract the Window When Necessary:

- If the number of distinct characters in ‘char-count’ exceeds two, increment the ‘start’ pointer to exclude characters from the beginning of the window.
- Update the frequency of the excluded character in ‘char-count’. If its frequency drops to zero, remove it from the map.

4. Update the Maximum Length:

- After each expansion (and possible contraction), update the ‘max-length’ if the current window size is larger than the previously recorded maximum.

This approach ensures that we traverse the string only once, achieving an optimal $O(n)$ time complexity.

The sliding window technique is particularly effective for problems involving contiguous sequences with dynamic constraints.

Complexities

- **Time Complexity:** $O(n)$, where n is the length of the string. Each character is processed at most twice (once by ‘end’ and once by ‘start’).
- **Space Complexity:** $O(1)$, since the hash map ‘char-count’ stores at most three distinct characters at any given time (to account for the condition before contraction).

Python Implementation

Below is the complete Python code for solving the "Longest Substring with At Most Two Distinct Characters" problem using the sliding window technique:

```
def length_of_longest_substring_two_distinct(s):
    from collections import defaultdict

    char_count = defaultdict(int)
    max_length = 0
    start = 0

    for end in range(len(s)):
        char_count[s[end]] += 1

        # Shrink the window until we have at most two distinct characters
        while len(char_count) > 2:
            char_count[s[start]] -= 1
            if char_count[s[start]] == 0:
                del char_count[s[start]]
            start += 1

        # Update max_length if the current window is larger
        current_window_length = end - start + 1
        if current_window_length > max_length:
            max_length = current_window_length

    return max_length

# Example usage:
# s = "eceba"
# print(length_of_longest_substring_two_distinct(s)) # Output: 3
# s = "ccaabbb"
# print(length_of_longest_substring_two_distinct(s)) # Output: 5
```

Implementing the sliding window requires careful management of pointers and character frequencies to ensure efficiency.

Explanation

The 'length-of-longest-substring-two-distinct' function efficiently finds the longest substring with at most two distinct characters by leveraging the sliding window technique. Here's a detailed breakdown of the implementation:

- **Initialization:**

- Hash Map: ‘char-count’ is a ‘defaultdict’ that keeps track of the frequency of each character within the current window.
- Pointers: ‘start’ marks the beginning of the window, while ‘end’ is the current position being evaluated.
- Maximum Length: ‘max-length’ stores the length of the longest valid substring found.

- **Iteration:**

- Expanding the Window: For each character ‘s[end]’, increment its count in ‘char-count’.
- Checking Constraints: If the number of distinct characters in ‘char-count’ exceeds two, enter a loop to shrink the window from the start.
 - * Decrement the count of ‘s[start]’.
 - * If the count of ‘s[start]’ drops to zero, remove it from ‘char-count’.
 - * Increment ‘start’ to move the window forward.
- Updating Maximum Length: After ensuring the window contains at most two distinct characters, calculate the current window length (‘end - start + 1’) and update ‘max-length’ if necessary.

- **Result:**

- After iterating through the entire string, return ‘max-length’ as the length of the longest valid substring.

Why This Approach

The sliding window technique is chosen for its efficiency in handling dynamic constraints within contiguous sequences. By maintaining a window that adapts based on the number of distinct characters, the algorithm ensures that each character is processed only a constant number of times, achieving optimal $O(n)$ time complexity.

Alternative Approaches

An alternative approach could involve brute force, where all possible substrings are examined, and the number of distinct characters is counted for each. However, this method would have a time complexity of $O(n^3)$, making it impractical for longer strings.

Another approach could use two pointers without a hash map, but managing the counts and ensuring at most two distinct characters would become cumbersome and less efficient.

The sliding window technique remains the most efficient and straightforward method for this problem.

Similar Problems to This One

There are several other problems that involve finding substrings or sequences with specific constraints, such as:

- Longest Substring with At Most K Distinct Characters
- Longest Substring Without Repeating Characters
- Minimum Window Substring
- Longest Repeating Character Replacement

Things to Keep in Mind and Tricks

- **Sliding Window Technique:** Utilize two pointers to define the current window and adjust them based on constraints.
- **Hash Map for Frequency Counts:** Use a hash map to keep track of character frequencies within the window for efficient constraint checking.
- **Dynamic Window Adjustment:** Expand the window by moving the ‘end’ pointer and contract it by moving the ‘start’ pointer when constraints are violated.
- **Edge Case Handling:** Always consider edge cases such as empty strings, single-character strings, and strings with all identical characters.
- **Optimizing Updates:** Update the maximum length only after ensuring the current window meets the constraints to avoid unnecessary calculations.

Corner and Special Cases to Test When Writing the Code

When implementing the ‘length-of-longest-substring-two-distinct’ function, it is crucial to test the following edge cases to ensure robustness:

- **Empty String:** ‘s = ””‘ should return ‘0‘.
- **Single Character:** ‘s = ”a”‘ should return ‘1‘.
- **All Characters the Same:** ‘s = ”aaaaa”‘ should return ‘5‘.
- **All Characters Unique:** ‘s = ”abcdef”‘ should return ‘2‘.

- **Multiple Valid Substrings:** ‘s = ”abaccc”‘ should return ‘4‘ for the substring ”accc”.
- **Strings with Exactly Two Distinct Characters:** ‘s = ”aabbcc”‘ should return ‘4‘ for substrings like ”aabb” or ”bcc”.
- **Large Input Size:** Test with a very large string to ensure that the implementation performs efficiently without exceeding memory limits.

Chapter 10

Sliding Window Technique

10.1 Sliding Window Technique

Introduction to Sliding Window

The sliding window technique is a powerful method used to solve various subarray and substring problems¹. This approach involves maintaining a subset of items within a given range or "window" and moving this window across the data structure to achieve the desired result. The sliding window can be of fixed or variable length depending on the problem requirements.

¹ Sliding window method

Historical Context of Sliding Window Technique

The concept of the sliding window has roots in both computer science and signal processing². In signal processing, sliding windows are used for smoothing data and calculating moving averages, which are crucial for analyzing time-series data. In computer science, the sliding window technique gained prominence for its efficiency in solving problems related to strings and arrays, particularly in scenarios where maintaining a dynamic subset of data is essential.

² Historical context

How the Sliding Window Technique Works

In a sliding window, two pointers are used to represent the boundaries of the window³. Typically, both pointers start at the beginning of the data structure. The window is expanded or contracted by moving these pointers to explore different parts of the data.

³ Two pointers

The key idea is that the two pointers move in the same direction and never overtake each other⁴. This ensures that each value is only visited at most twice, resulting

⁴ Pointers move in same direction

in a time complexity of $O(n)$ ⁵.

⁵ Time complexity is $O(n)$

- **Fixed-length Window:** In problems where the window size is fixed, one pointer advances while the other remains static until the window is fully formed, then both pointers move together.
- **Variable-length Window:** For problems where the window size can change, both pointers move independently to expand or contract the window based on specific conditions.

Applications of Sliding Window Technique

The sliding window technique is versatile and can be applied to a variety of problems⁶. Here are a few common applications:

⁶ Versatile technique

Problem 10.1 Maximum Sum Subarray of Size k

Problem Statement Given an array of integers `arr` and a number k , find the maximum sum of any contiguous subarray of length k .

Input: An array `arr` of integers and an integer k .

Output: An integer representing the maximum sum of a contiguous subarray of size k .

Example:

Input: `arr = [2, 1, 5, 1, 3, 2]`, $k = 3$

Output: 9

Explanation: Subarrays of size 3 are `[2,1,5]`, `[1,5,1]`, `[5,1,3]`, `[1,3,2]`. \\ The maximum sum is 9 from `[2,1,5]`.

Algorithmic Approach The sliding window technique can be effectively applied to solve the "Maximum Sum Subarray of Size k " problem in linear time⁷. Here's how ⁷ Linear time complexity the approach works:

1. Initialize the Window:

- Calculate the sum of the first k elements to form the initial window⁸. ⁸ Initial window sum
- Store this sum as the current maximum sum⁹. ⁹ Current maximum

2. Slide the Window:

- Iterate through the array starting from the k^{th} element.

- For each new element, add it to the current window sum and subtract the element that is no longer in the window¹⁰.
- Compare the updated window sum with the current maximum sum and update the maximum if necessary.

¹⁰ Update window sum

3. Result:

- After traversing the entire array, the current maximum sum will be the answer¹¹.

¹¹ Final result

This method ensures that each element is visited only once after the initial window setup, maintaining an overall time complexity of $O(n)$.

Complexities

- Time Complexity:** $O(n)$ ¹²
- Space Complexity:** $O(1)$ ¹³

¹² Linear time¹³ Constant space

Python Implementation Below is the Python code to solve the "Maximum Sum Subarray of Size k " problem using the sliding window technique:

```
def max_sum_subarray(arr, k):
    """
    Finds the maximum sum of any contiguous subarray of size k.

    Parameters:
    arr (List[int]): The input array of integers.
    k (int): The size of the subarray.

    Returns:
    int: The maximum sum of a contiguous subarray of size k.
    """
    n = len(arr)
    if n < k:
        return "Invalid" # Handle cases where k is larger than array size

    # Calculate the sum of the first window
    window_sum = sum(arr[:k])
    max_sum = window_sum

    # Slide the window over the array
    for i in range(k, n):
        window_sum += arr[i] - arr[i - k] # Add new element and remove the first
                                         # element of the previous window
        max_sum = max(max_sum, window_sum) # Update max_sum if current window sum
                                         # is greater
```

```

    return max_sum

# Example usage:
arr = [2, 1, 5, 1, 3, 2]
k = 3
print(max_sum_subarray(arr, k))  # Output: 9

```

Example Usage and Test Cases Here are some test cases to test the function:

```

# Test case 1: General case
arr = [2, 1, 5, 1, 3, 2]
k = 3
print(max_sum_subarray(arr, k))  # Output: 9

# Test case 2: All elements are the same
arr = [1, 1, 1, 1, 1]
k = 2
print(max_sum_subarray(arr, k))  # Output: 2

# Test case 3: k equals array size
arr = [5, 2, 3]
k = 3
print(max_sum_subarray(arr, k))  # Output: 10

# Test case 4: k is larger than array size
arr = [1, 2]
k = 3
print(max_sum_subarray(arr, k))  # Output: "Invalid"

# Test case 5: Array with negative numbers
arr = [-1, -2, -3, -4]
k = 2
print(max_sum_subarray(arr, k))  # Output: -3

```

Why This Approach This sliding window approach is chosen because it optimizes both time and space complexities while maintaining simplicity and efficiency¹⁴. By calculating the sum of the initial window and then updating it as the window slides, we avoid redundant computations that a brute-force approach would entail. This ensures that the algorithm runs in linear time, making it highly suitable for large datasets. Additionally, the constant space usage means that the memory footprint remains minimal, further enhancing performance¹⁵.

¹⁴ Optimal approach

¹⁵ Minimal memory footprint

Problem 10.2 Longest Substring with At Most k Distinct Characters

Problem Statement A more complex problem is finding the length of the longest substring with at most k distinct characters. Given a string s and an integer k , the task is to determine the longest substring that contains at most k distinct characters.

Input: A string s and an integer k .

Output: An integer representing the length of the longest substring with at most k distinct characters.

Example:

Input: $s = "eceba"$, $k = 2$

Output: 3

Explanation:

The longest substring with at most 2 distinct characters is "ece".

Algorithmic Approach The sliding window technique can be utilized to efficiently solve the "Longest Substring with At Most k Distinct Characters" problem in linear time¹⁶. Here's the step-by-step approach:

¹⁶ Linear time complexity

1. Initialize Variables:

- Use two pointers, `left` and `right`, to define the sliding window¹⁷.
- Use a dictionary `char_count` to store the frequency of characters within the window¹⁸.
- Initialize `max_length` to keep track of the longest valid substring found.

¹⁷ Two pointers

¹⁸ Character frequency tracking

2. Expand the Window:

- Iterate through the string with the `right` pointer, adding each character to `char_count` and updating its frequency.

3. Contract the Window:

- If the number of distinct characters in `char_count` exceeds k , move the `left` pointer to the right, decrementing the frequency of the character being removed¹⁹.
- If a character's frequency drops to zero, remove it from `char_count`.
- Continue contracting until the number of distinct characters is at most k .

¹⁹ Contract the window

4. Update Maximum Length:

- After each contraction, update `max_length` with the size of the current window if it is larger than the previously recorded maximum.

5. Result:

- After iterating through the string, `max_length` will hold the length of the longest valid substring.

Complexities

- **Time Complexity:** $O(n)$ ²⁰

²⁰ Linear time

- **Space Complexity:** $O(k)$ ²¹

²¹ Space proportional to k

Python Implementation Below is the Python code to solve the "Longest Substring with At Most k Distinct Characters" problem using the sliding window technique:

```
def length_of_longest_substring_k_distinct(s, k):
    """
    Finds the length of the longest substring with at most k distinct characters.

    Parameters:
    s (str): The input string.
    k (int): The maximum number of distinct characters allowed in the substring.

    Returns:
    int: The length of the longest valid substring.
    """
    from collections import defaultdict
    n = len(s)
    left = 0
    max_length = 0
    char_count = defaultdict(int)

    for right in range(n):
        char_count[s[right]] += 1

        # Ensure there are at most k distinct characters in the window
        while len(char_count) > k:
            char_count[s[left]] -= 1
            if char_count[s[left]] == 0:
                del char_count[s[left]]
            left += 1

        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
```

```
s = "eceba"
k = 2
print(length_of_longest_substring_k_distinct(s, k)) # Output: 3
```

Example Usage and Test Cases Here are some test cases to test the function:

```
# Test case 1: General case
s = "eceba"
k = 2
print(length_of_longest_substring_k_distinct(s, k)) # 
    ↪ Output: 3

# Test case 2: All characters are the same
s = "aaaaa"
k = 1
print(length_of_longest_substring_k_distinct(s, k)) # 
    ↪ Output: 5

# Test case 3: k is zero
s = "abc"
k = 0
print(length_of_longest_substring_k_distinct(s, k)) # 
    ↪ Output: 0

# Test case 4: Empty string
s = ""
k = 3
print(length_of_longest_substring_k_distinct(s, k)) # 
    ↪ Output: 0

# Test case 5: k greater than number of distinct
    ↪ characters
s = "abc"
k = 5
print(length_of_longest_substring_k_distinct(s, k)) # 
    ↪ Output: 3
```

Why This Approach This sliding window approach is optimal for this problem as it allows us to efficiently track and update the current substring without repeating characters. By maintaining a window defined by two pointers and a dictionary to store character counts, we ensure that each character is processed only once, resulting in a linear time complexity.

Complexity Analysis

- **Time Complexity:** $O(n)$

- **Space Complexity:** $O(\min(m, n))$, where m is the size of the character set and n is the length of the string

Similar Problems Other problems that can be efficiently solved using the sliding window technique include:

- **Minimum Window Substring:** Finding the smallest substring containing all characters of another string.
- **Longest Repeating Character Replacement:** Finding the longest substring with at most k character replacements.
- **Sliding Window Maximum:** Finding the maximum value in each sliding window of size k .

Things to Keep in Mind and Tricks

- **Edge Cases:** Consider empty strings, strings with all identical characters, and strings where all characters are unique.
- **Character Frequency Tracking:** Use hash maps or arrays (for limited character sets) to efficiently track frequencies.
- **Window Adjustment:** Properly move the ‘left’ pointer to ensure no duplicates within the window.
- **Optimizing Maximum Length Tracking:** Continuously update the maximum length as the window expands.

Exercises

1. **Minimum Window Substring:** Given two strings s and t , find the minimum window in s which will contain all the characters in t .
2. **Longest Subarray with Given Sum:** Given an array of integers and a sum, find the length of the longest contiguous subarray that sums to the given value.
3. **Sliding Window Maximum:** Given an array of integers and a window size k , find the maximum value in each sliding window.
4. **Longest Repeating Character Replacement:** Given a string and an integer k , find the length of the longest substring containing all repeating letters you can get after performing at most k character replacements.
5. **Subarray Sum Equals K:** Given an array of integers and an integer k , find the total number of continuous subarrays whose sum equals to k .

Questions for Reflection

- How does the sliding window technique differ from the two pointers technique, and when should each be used?
- What modifications would you make to handle cases where t contains special or non-alphanumeric characters?
- How can you optimize the sliding window approach for cases with very large input sizes?
- Can the sliding window technique be combined with other techniques like hashing or dynamic programming to solve more complex problems?

References

- LeetCode Problem: Sliding Window Maximum
- LeetCode Problem: Minimum Window Substring
- GeeksforGeeks Article: Sliding Window Technique
- HackerRank Problem: Maximum Subarray Problem
- Coursera Course: Algorithms: Divide and Conquer

Conclusion

The sliding window technique is a fundamental tool in the programmer's toolkit²². ²² Fundamental tool

By efficiently managing a subset of data within a "window" and moving this window across the dataset, many complex problems can be solved in linear time. Understanding and mastering this technique will greatly enhance your ability to tackle array and string problems effectively²³.

²³ Mastering the technique

Throughout this section, we explored the principles of the sliding window technique, detailed its application in solving specific problems, and provided Python implementations to solidify understanding. By practicing these concepts and applying them to various scenarios, you can harness the full potential of the sliding window technique and improve the efficiency of your algorithms²⁴.

²⁴ Enhancing algorithmic efficiency

As you delve deeper into algorithmic challenges, remember that the sliding window technique is versatile and can be adapted to a wide range of problems. Mastering this technique will not only aid in solving array and string problems but also pave the way for tackling more advanced algorithmic concepts in your programming journey²⁵.

²⁵ Versatile adaptation

10.2 More Examples of Sliding Window Technique

Here are more examples of the sliding window technique.

Problem 10.3 Longest Substring Without Repeating Characters

The "Longest Substring Without Repeating Characters" problem is a frequently addressed question in coding interviews, focusing on strings and the utilization of data structures to optimize search and retrieval operations. The task is to sift through the sequence of characters in a given string, identifying the lengthiest possible segment that consists purely of non-repeating characters.

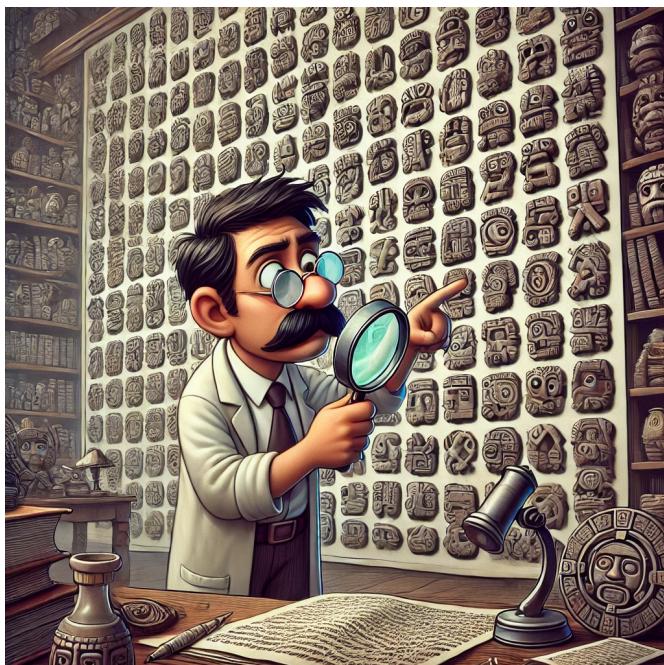


Figure 10.1: A scientist studying the Mayan alphabet to decode the longest substring without repeating characters.

Problem Statement Given a string s , the challenge lies in discerning the length of the longest substring devoid of recurrent characters. For example, within the string "abcabcbb", the longest substring that meets the criteria is "abc", which constitutes 3 characters. In contrast, for a string like "bbbbbb", the longest such substring is simply "b", reflecting a length of 1. The crux of the matter is to efficiently navigate through the string, implementing a strategy that maximizes the window of unique characters whilst conforming to the constraints set forth.

Algorithmic Approach To unravel this problem, a sliding window technique is employed, enhancing the algorithm's capability to dynamically adjust the extent of the

observed substring²⁶. The primary objective is to expand this window by traversing the string and keeping a vigilant eye for duplicating elements. The moment a recurring character manifests, it necessitates a recalibration of the window's scope, excising the previously occurring instance of said character before progressing.

****Detailed Steps:****

1. ****Initialization:****

- Create a dictionary, `char_index_map`, to store the latest index of each character encountered.
- Initialize two pointers, `start` and `max_length`, to track the beginning of the current window and the maximum length found, respectively.

2. ****Traversal:****

- Iterate over the string using the `end` pointer.
- For each character, check if it exists in `char_index_map` and if its last occurrence is within the current window (i.e., `char_index_map[char] ≥ start`).
- If so, move the `start` pointer to the position right after the last occurrence to ensure all characters in the window remain unique.
- Update the character's latest index in `char_index_map`.
- Calculate the current window size (`end - start + 1`) and update `max_length` if the current window is larger.

3. ****Termination:****

- After completing the traversal, `max_length` holds the length of the longest substring without repeating characters.

This method ensures that each character is processed only once, maintaining an overall time complexity of $O(n)$.

Complexities

- **Time Complexity:** The overarching time complexity of the solution pivots around $O(n)$, as the algorithm meticulously processes each character in the string a singular time — a testament to the efficiency of the sliding window methodology²⁷.

²⁷ Time complexity $O(n)$

- **Space Complexity:** The auxiliary space complexity, contingent upon the breadth of the character set employed in the hash map or set, invariably approaches $O(m)$, where m represents the size of the character alphabet (in the case of ASCII, potentially up to 128)²⁸.

²⁸ Space complexity $O(m)$

Python Implementation Below is the complete Python code for solving the "Longest Substring Without Repeating Characters" problem by employing a sliding window approach:

```
def length_of_longest_substring(s):
    """
    Finds the length of the longest substring without repeating characters.

    Parameters:
    s (str): The input string.

    Returns:
    int: The length of the longest substring without repeating characters.
    """
    char_index_map = {}
    max_length = 0
    start = 0

    for i, char in enumerate(s):
        if char in char_index_map and char_index_map[char] >= start:
            start = char_index_map[char] + 1
        char_index_map[char] = i
        max_length = max(max_length, i - start + 1)

    return max_length

# Example usage:
s = "abcabcbb"
print(length_of_longest_substring(s)) # Output: 3
```

Example Usage and Test Cases Here are some test cases to test the function:

```
# Test case 1: General case
s = "abcabcbb"
print(length_of_longest_substring(s)) # Output: 3

# Test case 2: All characters are the same
s = "bbbbbb"
print(length_of_longest_substring(s)) # Output: 1

# Test case 3: No repeating characters
s = "abcdefg"
print(length_of_longest_substring(s)) # Output: 7

# Test case 4: Empty string
s = ""
print(length_of_longest_substring(s)) # Output: 0

# Test case 5: String with special characters
s = "pwwkew"
```

```
print(length_of_longest_substring(s)) # Output: 3
```

Why This Approach The sliding window technique is favored in this context due to its real-time responsiveness to evolving conditions within the string. It streamlines operations by eliminating the need for redundant inspections of previously reviewed characters and concentrates computational efforts solely on the changing window edges, thereby enhancing the procedure's efficiency²⁹.

²⁹ Efficiency of sliding window

Alternative Approaches An alternative approach to the sliding window technique could involve brute force, which entails inspecting every possible substring for duplicates—a method that is computationally exhaustive and less feasible for longer strings. Another potential approach might utilize dynamic programming to keep track of the longest substring without repeating characters up to each index, but this is generally less efficient than the adopted sliding window strategy³⁰.

³⁰ Alternative approaches

Similar Problems to This One There are several problems akin to the "Longest Substring Without Repeating Characters," such as "Longest Substring with At Most Two Distinct Characters," "Longest Repeating Character Replacement," and "Substring with Concatenation of All Words." Each of these presents unique requisites that necessitate slight modifications to the basic sliding window construct³¹.

³¹ Similar problems

Things to Keep in Mind and Tricks When dealing with sliding window problems, an imperative trick is to adequately manage the window boundaries, especially when the window contracts. It's crucial to ensure that the starting boundary does not revert backwards, as this would lead to incorrect computations³².

³² Managing window boundaries

Corner and Special Cases to Test When Writing the Code One should be alert to scenarios involving extensive sequences of identical characters, strings consisting solely of unique characters, or strings with alternating patterns. Additionally, edge cases such as an empty string should be contemplated. It is advisable to conduct thorough testing against these circumstances to affirm the robustness of the algorithm³³.

³³ Corner cases

Problem 10.4 Minimum Window Substring

The Minimum Window Substring problem is a classic problem in the realm of string manipulation and searching algorithms.

Problem Statement

Given two strings ‘s’ and ‘t’, 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 ””.

A substring is a contiguous sequence of characters within the string.

Examples:

- Input: s = "ADOBECODEBANC", t = "ABC"
- Output: "BANC"

- Input: s = "a", t = "a"
- Output: "a"

Algorithmic Approach

To solve the problem using a sliding window approach, start with two pointers both set at the beginning of ‘s’. Expand the right pointer to find a valid window that contains all the characters from ‘t’. Once you have a valid window, increment the left pointer to try and minimize the size of the window without compromising its validity (i.e., it should still contain all the characters from ‘t’). Repeat this process until all potential windows have been checked. Return the smallest valid window observed during this process.

Complexities

- **Time Complexity:** The overall time complexity is $O(n + m)$, where n is the length of string ‘s’ and m is the length of string ‘t’.
- **Space Complexity:** The space complexity is $O(m)$ for storing the characters of string ‘t’ in a hash table.

Python Implementation

Below is the complete Python code for solving the Minimum Window Substring problem:

```
from collections import Counter

def minWindow(s: str, t: str) -> str:
```

```

if not t or not s:
    return ""

# Dictionary which keeps a count of all the unique characters in t.
dict_t = Counter(t)

# Number of unique characters in t, which need to be present in the desired
# window.
required = len(dict_t)

# Left and Right pointer
l, r = 0, 0

# Formed is used to keep track of how many unique characters in t are present
# in the current window in its desired frequency.
# e.g. if t is "AABC" then the window must have two A's, one B and one C. Thus
# formed would be = 3 when all these conditions are met.
formed = 0

# Dictionary which keeps a count of all the unique characters in the current
# window.
window_counts = {}

# ans tuple of the form (window length, left, right)
ans = float("inf"), None, None

while r < len(s):
    # Add one character from the right to the window
    character = s[r]
    window_counts[character] = window_counts.get(character, 0) + 1

    # If the frequency of the current character added equals to the desired
    # count in t then increment the formed count by 1.
    if character in dict_t and window_counts[character] == dict_t[character]:
        formed += 1

    # Try and contract the window till the point where it ceases to be '
    # desirable'.
    while l <= r and formed == required:
        character = s[l]

        # Save the smallest window until now.
        if r - l + 1 < ans[0]:
            ans = (r - l + 1, l, r)

        # The character at the position pointed by the `left` pointer is no
        # longer a part of the window.
        window_counts[character] -= 1
        if character in dict_t and window_counts[character] < dict_t[character]:
            formed -= 1

```

```

# Move the left pointer ahead, this would help to look for a new
#   ↪ window.
l += 1

# Keep expanding the window once we are done contracting.
r += 1
return "" if ans[0] == float("inf") else s[ans[1]: ans[2] + 1]

# Example usage:
print(minWindow("ADOBECODEBANC", "ABC")) # Output: "BANC"
print(minWindow("a", "a"))               # Output: "a"

```

This implementation uses a hash table to store the frequency of each character in ‘t’, then uses a sliding window to find the smallest substring of ‘s’ that contains all the characters of ‘t’. The hash table helps in quickly checking if the current window contains all the required characters.

Why this approach

This approach is chosen because it efficiently scales with the length of the strings ‘s’ and ‘t’. Although the worst-case time complexity appears to be $O(n \cdot m)$, the use of a hash table ensures that the check for the current window can be done in constant time, thus effectively bringing the overall time complexity down to $O(n + m)$.

Alternative approaches

An alternative approach would be to use a fixed-size character array instead of a hash table to store the frequencies. However, this is only feasible if the character set of ‘s’ and ‘t’ is limited (e.g., ASCII characters only).

Similar problems to this one

Problems like ”Longest Substring Without Repeating Characters”, ”Substring with Concatenation of All Words”, and ”Permutation in String” involve similar techniques of sliding window and hashing.

Things to keep in mind and tricks

One must take care to handle the counting accurately, especially when a character leaves the window when the left pointer moves. Also, initializing the ‘ans’ tuple with ‘float(”inf”)’ allows easy checking if no window was found.

Corner and special cases to test when writing the code

It's important to consider cases where 's' is much larger than 't', cases where 't' has repeated characters, and cases where no valid window is possible. These help in verifying that the window resizing logic is accurate and that the counting logic does not have off-by-one errors.

Problem 10.5 Longest Repeating Character Replacement

The "Longest Repeating Character Replacement" problem is a classic example of how a combination of string manipulation and the sliding window technique can be employed to solve complex problems efficiently. In this problem, you are given a string s and an integer k , which represents the maximum number of character replacements you are allowed to perform to create the longest possible substring containing the same letters.

Problem Statement

The objective is to determine the length of the longest substring of s that can be obtained by replacing at most k characters within the substring. Replacements can be done with any uppercase English letter.

Example:

Input: $s = "AABABBA"$, $k = 1$

Output: 4

Explanation:

Given the string $s = "AABABBA"$ and $k = 1$, we can replace one occurrence of 'B' in "AABA" with 'A', resulting in "AAAA". This substring has a length of 4, making it the longest substring possible with at most one character replacement.

LeetCode link: [Longest Repeating Character Replacement](#)

Algorithmic Approach

To solve this problem efficiently, we employ the sliding window technique. By maintaining a window of characters, we can dynamically calculate the maximum frequency of a single character within the window and determine whether the remaining characters can be replaced within the allowed limit k . If the number of characters to replace exceeds k , the window is reduced by incrementing the start pointer.

Steps:

- Use a sliding window to traverse the string, tracking the frequency of characters in the current window using a hash map or counter.
- Maintain the maximum frequency of a single character in the window.
- If the size of the window minus the maximum frequency exceeds k , reduce the window size by moving the start pointer.
- Update the length of the longest valid substring found so far.

Complexities

- **Time Complexity:** $O(n)$, where n is the length of the string. Each character is processed at most twice (once when extending the window and once when shrinking it).
- **Space Complexity:** $O(1)$, as the counter only tracks the frequency of 26 possible uppercase English letters.

Python Implementation

Below is the Python implementation of the sliding window approach:

```
from collections import Counter

def characterReplacement(s, k):
    count = Counter()
    max_count = 0
    max_length = 0
    left = 0

    for right in range(len(s)):
        count[s[right]] += 1
        max_count = max(max_count, count[s[right]])

        # If replacements needed exceed k, shrink the window
        if right - left + 1 - max_count > k:
            count[s[left]] -= 1
            left += 1

        # Update the maximum length of the valid window
        max_length = max(max_length, right - left + 1)

    return max_length

# Example usage:
```

```
s = "AABABBA"
k = 1
print(characterReplacement(s, k)) # Output: 4
```

Explanation of the Code

The algorithm uses a sliding window defined by the pointers `left` and `right`. As the `right` pointer expands the window, the character frequencies are updated. If the number of characters requiring replacement exceeds k , the `left` pointer is moved forward to shrink the window. At each step, the maximum valid window size is recorded.

Why This Approach?

The sliding window approach is ideal for problems involving substrings and constraints, such as limited replacements. Its linear time complexity ensures efficiency, making it well-suited for large input sizes. The use of a counter to dynamically track character frequencies allows for real-time window validation without rescanning the substring.

Alternative Approaches

- **Dynamic Programming:** A table-based approach to calculate the maximum length for substrings ending at each position. This method has a higher space complexity compared to the sliding window.
- **Brute Force:** Check all substrings and calculate the replacements required for each. This method is inefficient with a time complexity of $O(n^2 \cdot 26)$.

Similar Problems

- Longest Substring with At Most Two Distinct Characters:** Find the length of the longest substring containing at most two distinct characters.
- Longest Substring Without Repeating Characters:** Determine the length of the longest substring without repeating characters.
- Minimum Window Substring:** Find the minimum window in a string that contains all characters of another string.

Things to Keep in Mind and Tricks

- Track the frequency of the most common character in the window to minimize unnecessary calculations.
- Adjust the window size dynamically to ensure that the number of replacements required does not exceed k .
- Handle edge cases like empty strings, strings with only one character, or $k = 0$.

Corner and Special Cases to Test

- **Empty String:** Input: $s = ""$, $k = 1$ (should return 0).
- **All Identical Characters:** Input: $s = "AAAAA"$, $k = 2$ (should return 4).
- **All Unique Characters:** Input: $s = "ABCD"$, $k = 1$ (should return 2).
- **No Replacements Allowed:** Input: $s = "AABABBA"$, $k = 0$ (should return 2).

Conclusion

The **Largest Repeating Character Replacement** problem demonstrates the effectiveness of the sliding window technique in solving substring-related challenges. Mastering this problem equips you with the tools to tackle similar problems requiring dynamic adjustments to constraints while processing strings.

Problem 10.6 Find All Anagrams in a String

The **Find All Anagrams in a String** problem is a classic application of the sliding window technique combined with character frequency counting. An anagram of a string is a permutation of its characters. The objective is to locate all start indices of substrings in the given string s that are anagrams of another string p .

Problem Statement

Given two strings, s and p , find all the start indices of p 's anagrams in s . You may return the answer in any order.

Input: - s : The main string to search within. - p : The string whose anagrams are to be found.

Output: - A list of starting indices of p 's anagrams in s .

Example 1:

Input: $s = "cbaebabacd"$, $p = "abc"$
 Output: [0, 6]

Example 2:

Input: $s = "abab"$, $p = "ab"$
 Output: [0, 1, 2]

Example 3:

Input: $s = "a"$, $p = "a"$
 Output: [0]

Algorithmic Approach

This problem can be efficiently solved using a sliding window approach and character frequency counting.

Steps:

- Compute the frequency count of characters in p .
- Use a sliding window of size equal to the length of p on s , keeping track of the frequency of characters in the current window.
- Compare the character frequency of the current window with that of p . If they match, record the start index of the window.
- Slide the window by moving one character to the right, updating the frequency counts accordingly.

Complexities

- **Time Complexity:** $O(n + m)$, where n is the length of s and m is the length of p . This includes the time to compute the frequency counts and traverse s .
- **Space Complexity:** $O(1)$ for the character frequency arrays (constant size for fixed alphabets like English letters).

Python Implementation

Below is the Python implementation using the sliding window technique:

```

from collections import Counter

def findAnagrams(s: str, p: str):
    result = []
    p_count = Counter(p)
    s_count = Counter()
    p_len = len(p)

    for i in range(len(s)):
        # Add current character to the sliding window
        s_count[s[i]] += 1

        # Remove the character that is out of the window
        if i >= p_len:
            if s_count[s[i - p_len]] == 1:
                del s_count[s[i - p_len]]
            else:
                s_count[s[i - p_len]] -= 1

        # Compare window frequency with p frequency
        if s_count == p_count:
            result.append(i - p_len + 1)

    return result

# Example usage:
s = "cbaebabacd"
p = "abc"
print(findAnagrams(s, p))  # Output: [0, 6]

```

Why This Approach?

The sliding window approach is highly efficient for problems involving substrings with specific properties. By maintaining a running frequency count, it avoids recalculating the count for each window, ensuring linear time complexity.

Alternative Approaches

- **Sorting-Based Approach:** Sort p and each substring of s of length p , then compare. This approach is inefficient with a time complexity of $O(n \cdot m \log m)$, where m is the length of p .
- **Hash Set Matching:** Use a hash set to compare permutations of p against substrings of s , but this is less optimal for larger inputs.

Similar Problems

- **Permutation in String:** Check if one string's permutation is a substring of another string.
- **Minimum Window Substring:** Find the smallest substring of s that contains all characters of p .
- **Substring with Concatenation of All Words:** Find substrings that are concatenations of all words in a given list.

Things to Keep in Mind and Tricks

- Use Python's Counter to simplify frequency count management.
- Keep the window size fixed at the length of p .
- Handle edge cases such as empty strings or when p is longer than s .

Corner and Special Cases to Test

- **Empty Strings:** Input: $s = "", p = "a"$ (should return an empty list).
- **No Matches:** Input: $s = "abcdef", p = "gh"$ (should return an empty list).
- **All Matches:** Input: $s = "aaaa", p = "a"$ (should return $[0, 1, 2, 3]$).
- **Length Mismatch:** Input: $s = "abc", p = "abcd"$ (should return an empty list).

Conclusion

The **Find All Anagrams in a String** problem is an excellent example of how the sliding window technique and frequency counts can be combined to efficiently solve substring problems. Mastery of this problem equips you to tackle a wide range of challenges involving character permutations and substring matching.

Problem 10.7 Substring with Concatenation of All Words

The **Substring with Concatenation of All Words** problem involves finding all starting indices of substring(s) in a given string s that is a concatenation of each word in a given list $words$ exactly once and without any intervening characters. This problem tests your ability to efficiently manage and traverse substrings while handling dynamic constraints.

This problem effectively utilizes the sliding window technique combined with hash maps to manage word frequencies within substrings.

Problem Statement

Given a string `s` and an array of strings `words`, return all starting indices of substring(s) in `s` that is a concatenation of each word in `words` exactly once, in any order, and without any intervening characters.

You can return the answer in any order.

Note:

- All words in `words` are of the same length.
- The concatenated substring must contain all words exactly once.

Examples:

- **Example 1:**

Input: `s = "barfoothefoobarman"`, `words = ["foo", "bar"]`

Output: `[0, 9]`

Explanation: Substrings starting at index 0 and 9 are "barfoo" and "foobar" respectively.

- **Example 2:**

Input: `s = "wordgoodgoodgoodbestword"`, `words = ["word", "good", "best", "word"]`

Output: `[]`

Explanation: There is no substring that contains all words exactly once.

- **Example 3:**

Input: `s = "barfoofoobarthefoobarman"`, `words = ["bar", "foo", "the"]`

Output: `[6, 9, 12]`

Explanation: Substrings starting at indices 6, 9, and 12 are "foobarthe", "barthefoo", and "thefo

LeetCode link: Substring with Concatenation of All Words

[\[LeetCode Link\]](#)

[\[GeeksForGeeks Link\]](#)

[\[HackerRank Link\]](#)

[\[CodeSignal Link\]](#)

[\[InterviewBit Link\]](#)

[\[Educative Link\]](#)

[\[Codewars Link\]](#)

Algorithmic Approach

To solve this problem, we utilize the **“sliding window”** technique combined with **“hash maps”** to manage word frequencies within substrings. The approach involves the following steps:

1. Preprocessing:

- Calculate the length of each word (`word_length`).
- Determine the number of words (`num_words`).
- Compute the total length of the concatenated substring (`total_length`).
- Create a hash map (`word_count`) to store the frequency of each word in `words`.

2. Sliding Window Traversal:

- Iterate through the string `s` with a window size of `total_length`.
- For each window, extract substrings of length `word_length` and check if they exist in `word_count`.
- Maintain a temporary hash map (`seen`) to track the frequency of words within the current window.
- If all words match the required frequencies, record the starting index.

3. Optimizations:

- Slide the window in increments of `word_length` to align with word boundaries.
- Early termination of window checks if a word exceeds the required frequency.

This method ensures that the entire string is traversed efficiently, checking only relevant substrings and maintaining optimal time and space complexities.

Efficient window management and frequency tracking are crucial for maintaining the desired constraints within the sliding window.

Complexities

- **Time Complexity:** $O(n \times m)$, where n is the length of the string `s` and m is the number of words. This is because we slide the window across the string and, for each window, we check all words.
- **Space Complexity:** $O(k)$, where k is the number of unique words in `words`. This space is used by the hash maps to store word frequencies.

Python Implementation

Below is the complete Python code for solving the **"Substring with Concatenation of All Words"** problem using the sliding window technique:

Implementing the sliding window with hash maps ensures efficient tracking of word frequencies and window constraints.

```
from typing import List
from collections import defaultdict

def find_substring(s: str, words: List[str]) -> List[int]:
```

```

if not s or not words:
    return []

word_length = len(words[0])
num_words = len(words)
total_length = word_length * num_words
word_count = defaultdict(int)

for word in words:
    word_count[word] += 1

result = []

for i in range(word_length):
    left = i
    count = 0
    seen = defaultdict(int)

    for j in range(i, len(s) - word_length + 1, word_length):
        word = s[j:j + word_length]
        if word in word_count:
            seen[word] += 1
            count += 1

            while seen[word] > word_count[word]:
                left_word = s[left:left + word_length]
                seen[left_word] -= 1
                count -= 1
                left += word_length

        if count == num_words:
            result.append(left)

    else:
        seen.clear()
        count = 0
        left = j + word_length

return result

# Example usage:
# s = "barfoothefoobarman"
# words = ["foo", "bar"]
# print(find_substring(s, words)) # Output: [0, 9]
# s = "wordgoodgoodgoodbestword"
# words = ["word", "good", "best", "word"]
# print(find_substring(s, words)) # Output: []

```

This implementation leverages the sliding window technique to efficiently traverse the string `s` while managing word frequencies using hash maps. Here's a step-by-step breakdown:

Explanation

The ‘find-substring‘ function efficiently identifies all starting indices of substrings in `s` that are concatenations of each word in `words` exactly once. Here’s a detailed breakdown of the implementation:

- **Initialization:**

- **Edge Cases:** If either `s` or `words` is empty, return an empty list.
- **Word Metrics:** Calculate the length of each word (`word_length`), the number of words (`num_words`), and the total length of the concatenated substring (`total_length`).
- **Word Count Map:** Create a hash map (`word_count`) to store the frequency of each word in `words`.

- **Sliding Window Traversal:**

- **Iterate Over Possible Starting Points:** Loop through the string with an offset of `word_length` to align with word boundaries.
- **Initialize Pointers and Counters:** For each starting offset, initialize ‘left’ to mark the beginning of the window, ‘count’ to track the number of valid words found, and ‘seen’ to count the frequency of words within the current window.
- **Expand the Window:** Slide the window by moving the ‘j’ pointer in increments of `word_length`.
 - * Extract the current word from `s` using slicing.
 - * If the word exists in `word_count`, increment its count in `seen` and the overall ‘count’.
 - * **Handle Excess Frequencies:** If a word’s frequency in `seen` exceeds its required frequency in `word_count`, shrink the window from the left by decrementing the frequency of the leftmost word and moving the ‘left’ pointer forward.
 - * **Record Valid Substrings:** If ‘count’ equals `num_words`, append the current ‘left’ index to the result list.
- **Reset on Invalid Words:** If the current word is not in `word_count`, clear the `seen` map, reset ‘count’, and move the ‘left’ pointer beyond the current word.
- **Result Compilation:** After traversing the string, return the list of starting indices where valid concatenations are found.

Why This Approach

This approach is chosen for its efficiency and ability to handle the problem's constraints effectively. By using the sliding window technique:

- **Optimal Time Complexity:** The algorithm runs in $O(n \times m)$ time, where n is the length of the string and m is the number of words, ensuring scalability even for large inputs.
- **Space Efficiency:** Utilizing hash maps to track word frequencies allows for constant-time look-ups and updates, maintaining a space complexity of $O(k)$, where k is the number of unique words.
- **Single Pass Traversal:** The sliding window ensures that each character is processed only a limited number of times, avoiding redundant computations.

Alternative Approaches

An alternative approach to solving this problem is the **Brute Force** method, where you check every possible substring of length `total_length` and verify if it contains all the words in `words` with the correct frequencies. However, this method has a time complexity of $O(n \times m \times l)$, where l is the length of each word, making it highly inefficient for large inputs.

Another approach could involve using **Trie Data Structures** to manage word look-ups more efficiently, but integrating it with the sliding window technique complicates the implementation without significant performance gains for this specific problem.

The sliding window technique combined with hash maps remains the most optimal and straightforward method for this problem.

Choosing the right algorithmic strategy is crucial for optimizing both time and space complexities.

Similar Problems to This One

There are several other problems that involve finding substrings or sequences with specific constraints, such as:

- Longest Substring with At Most K Distinct Characters
- Longest Substring Without Repeating Characters
- Minimum Window Substring
- Longest Repeating Character Replacement
- Word Break

Things to Keep in Mind and Tricks

- **Sliding Window Technique:** Utilize two pointers to define the current window and adjust them based on the constraints.
- **Hash Map for Frequency Counts:** Use a hash map to keep track of character frequencies within the window for efficient constraint checking.
- **Dynamic Window Adjustment:** Expand the window by moving the ‘end’ pointer and contract it by moving the ‘start’ pointer when constraints are violated.
- **Edge Case Handling:** Always consider edge cases such as empty strings, single-character strings, and strings with all identical characters.
- **Optimizing Updates:** Update the maximum length only after ensuring the current window meets the constraints to avoid unnecessary calculations.

Corner and Special Cases to Test When Writing the Code

When implementing the ‘find-substring’ function, it is crucial to test the following edge cases to ensure robustness:

- **Empty String:** ‘s = “”, ‘words = [”a”]‘ should return ‘[]‘.
- **Single Word:** ‘s = ”a”, ‘words = [”a”]‘ should return ‘[0]‘.
- **All Words Present Multiple Times:** ‘s = ”barfoofoobarthefoobarman”, ‘words = [”bar”, ”foo”, ”the”]‘ should return ‘[6,9,12]‘.
- **Words with Overlapping:** ‘s = ”wordgoodgoodgoodbestword”, ‘words = [”word”, ”good”, ”best”, ”word”]‘ should return ‘[]‘.
- **Words Not Present:** ‘s = ”lingmindraboofooowingdingbarrwingmonkeypoundcake”, ‘words = [”foo”, ”bar”, ”wing”, ”ding”, ”wing”]‘ should return ‘[13]‘.
- **Words with Different Lengths:** Although the problem assumes all words are the same length, test cases with varying lengths should be handled gracefully.
- **Large Input Size:** Test with a very large string and a large number of words to ensure that the implementation performs efficiently without exceeding memory limits.
- **Identical Words:** ‘s = ”aaaaaa”, ‘words = [”aa”, ”aa”]‘ should return ‘[0,1,2]‘.

Problem 10.8 Minimum Size Subarray Sum

The "Minimum Size Subarray Sum" problem is another excellent example of the sliding window technique in action. This problem challenges one to find the smallest contiguous subarray whose sum is at least a given target.

Problem Statement

Given an array of positive integers `nums` and a positive integer `s`, find the minimal length of a contiguous subarray of which the sum is greater than or equal to `s`. If there isn't one, return 0 instead.

Input: An integer `s` and an array of positive integers `nums`.

Output: The minimal length of a contiguous subarray with a sum $\geq s$.

Example:

Input: `s = 7, nums = [2,3,1,2,4,3]`

Output: 2

Explanation: The subarray `[4,3]` has the minimal length under the problem constraint.

Algorithmic Approach

To solve this problem, we employ the sliding window technique. The idea is to expand the window by adding elements to the sum until it is greater than or equal to `s`. Once the sum is sufficient, we contract the window from the left to find the minimal length while still maintaining the sum $\geq s$.

Complexities

- **Time Complexity:** The time complexity of this algorithm is $O(n)$, where n is the length of the array. This is because each element is added and removed from the window at most once.
- **Space Complexity:** The space complexity is $O(1)$ as we are using only a few extra variables for the window sum, start pointer, and the minimum length.

Python Implementation

Below is the complete Python code for solving the "Minimum Size Subarray Sum" problem using the sliding window technique:

```

def min_subarray_len(s, nums):
    n = len(nums)
    min_length = float('inf')
    current_sum = 0
    start = 0

    for end in range(n):
        current_sum += nums[end]

        while current_sum >= s:
            min_length = min(min_length, end - start + 1)
            current_sum -= nums[start]
            start += 1

    return 0 if min_length == float('inf') else min_length

# Example usage:
s = 7
nums = [2, 3, 1, 2, 4, 3]
print(min_subarray_len(s, nums)) # Output: 2

```

This implementation maintains a running sum of the current window. If the sum becomes greater than or equal to s , it tries to shrink the window from the left to find the minimum length. It updates the minimum length whenever the sum condition is satisfied.

Why this approach

The sliding window technique is chosen here for its efficiency in handling contiguous subarrays. It ensures that each element is processed at most twice, once when added to the window and once when removed, making it highly efficient.

Alternative approaches

An alternative approach could involve using binary search along with prefix sums to find the minimum subarray length. However, this approach would be more complex and less intuitive than the sliding window method.

Similar problems to this one

Similar problems include "Longest Substring with At Most Two Distinct Characters" and "Longest Repeating Character Replacement," where the sliding window technique is also applied to find subarrays or substrings under certain constraints.

Things to keep in mind and tricks

One important trick is to remember that as soon as the window sum exceeds s , you should try to shrink the window from the left to find the minimal length subarray. This ensures that the solution remains optimal.

Corner and Special Cases to Test When Writing the Code

Test cases should include:

- Arrays where no subarray meets the sum condition, returning 0.
- Large arrays to verify performance.
- Edge cases with the minimum input size (e.g., an array with one element).

Problem 10.9 Permutation in String

The **Permutation in String** problem involves determining whether one string contains a permutation of another string as a substring. This problem effectively demonstrates the application of the sliding window technique along with hash maps to manage character frequencies within dynamic window sizes.

This problem utilizes the sliding window technique combined with hash maps to efficiently check for permutations within a string.

Problem Statement

Given two strings $s1$ and $s2$, return `true` if $s2$ contains a permutation of $s1$, or `false` otherwise.

In other words, one of $s1$'s permutations is the substring of $s2$.

Note:

- The input strings consist of lowercase English letters.
- The length of both strings will not exceed 10,000.

Examples:

- **Example 1:**

```
Input: s1 = "ab", s2 = "eidbaooo"
Output: true
Explanation: s2 contains one permutation of s1 ("ba").
```

- **Example 2:**

Input: $s1 = "ab"$, $s2 = "eidboaoo"$

Output: false

Explanation: There is no substring of $s2$ that is a permutation of $s1$.

LeetCode link: Permutation in String

[LeetCode Link]

[GeeksForGeeks Link]

[HackerRank Link]

[CodeSignal Link]

[InterviewBit Link]

[Educative Link]

[Codewars Link]

Algorithmic Approach

To solve this problem, we utilize the “sliding window” technique combined with “hash maps” to manage character frequencies within substrings. The approach involves the following steps:

1. Preprocessing:

- Calculate the length of $s1$ (`word_length`).
- Determine the number of characters in $s1$ (`num_chars`).
- Create a hash map (`char_count`) to store the frequency of each character in $s1$.

2. Sliding Window Traversal:

- Iterate through $s2$ with a window size equal to `word_length`.
- For each window, check if the substring is a permutation of $s1$ by comparing character frequencies.
- Use a sliding window to add one character at a time and remove the leftmost character to maintain the window size.

3. Early Termination and Optimization:

- If at any point the number of distinct characters exceeds that of $s1$, adjust the window accordingly.
- Optimize by only checking windows that have the potential to be valid permutations.

This method ensures that the entire string $s2$ is traversed efficiently, checking only relevant substrings and maintaining optimal time and space complexities.

Efficient window management and frequency tracking are crucial for maintaining the desired constraints within the sliding window.

Complexities

- **Time Complexity:** $O(n)$, where n is the length of $s2$. Each character is processed at most twice (once by the ‘end’ pointer and once by the ‘start’ pointer).
- **Space Complexity:** $O(1)$, since the hash maps store character frequencies limited to the size of the alphabet (constant space).

Python Implementation

Below is the complete Python code for solving the **"Permutation in String"** problem using the sliding window technique:

```
def check_inclusion(s1: str, s2: str) -> bool:
    from collections import defaultdict

    if len(s1) > len(s2):
        return False

    char_count = defaultdict(int)
    for char in s1:
        char_count[char] += 1

    required = len(char_count)
    formed = 0
    window_counts = defaultdict(int)
    left = 0

    for right in range(len(s2)):
        character = s2[right]
        window_counts[character] += 1

        if character in char_count and window_counts[character] == char_count[
            ↪ character]:
            formed += 1

        while left <= right and formed == required:
            if right - left + 1 == len(s1):
                return True

            left_char = s2[left]
            window_counts[left_char] -= 1
            if left_char in char_count and window_counts[left_char] < char_count[
                ↪ left_char]:
                formed -= 1
            left += 1

    return False

# Example usage:
# s1 = "ab"
# s2 = "eidbaooo"
# print(check_inclusion(s1, s2)) # Output: True
# s1 = "ab"
# s2 = "eidboaooo"
# print(check_inclusion(s1, s2)) # Output: False
```

Implementing the sliding window requires careful management of pointers and character frequencies to ensure efficiency.

This implementation maintains a hash map ‘char_count’ to record the fre-

quency of each character in `s1`. It then uses a sliding window defined by ‘`left`’ and ‘`right`’ pointers to traverse `s2`, updating character counts and checking for valid permutations efficiently.

Explanation

The ‘check-inclusion’ function efficiently determines whether any permutation of `s1` exists as a substring within `s2` by leveraging the sliding window technique. Here’s a detailed breakdown of the implementation:

- **Initialization:**

- **Edge Case Handling:** If the length of `s1` is greater than `s2`, return `False` immediately as no permutation can exist.
- **Character Count Map:** Create a hash map ‘`char_count`’ to store the frequency of each character in `s1`.
- **Pointers and Counters:**
 - * ‘`required`’ denotes the number of unique characters in `s1`.
 - * ‘`formed`’ tracks how many unique characters in the current window match the required frequency.
 - * ‘`window_counts`’ is a hash map to store the frequency of characters in the current window of `s2`.
 - * ‘`left`’ is the left pointer of the sliding window.

- **Sliding Window Traversal:**

- **Expanding the Window:** Iterate through `s2` using the ‘`right`’ pointer.
 - * Increment the count of the current character in ‘`window_counts`’.
 - * If the current character’s frequency matches its frequency in ‘`char_count`’, increment ‘`formed`’.
- **Contracting the Window:** While the window has all required unique characters (`formed == required`):
 - * **Check for Valid Permutation:** If the current window size equals the length of `s1`, return `True` as a valid permutation is found.
 - * **Shrink the Window:** Decrement the frequency of the character at the ‘`left`’ pointer in ‘`window_counts`’.
 - * If the frequency of the left character falls below its required frequency in ‘`char_count`’, decrement ‘`formed`’.
 - * Move the ‘`left`’ pointer forward to continue sliding the window.

- **Result Compilation:** If no valid permutation is found after traversing `s2`, return `False`.

Why This Approach

The sliding window technique is chosen for its efficiency in handling dynamic constraints within contiguous sequences. By maintaining a window that adapts based on the number of distinct characters and their frequencies, the algorithm ensures that each character is processed only a limited number of times, achieving optimal $O(n)$ time complexity.

Alternative Approaches

An alternative approach could involve generating all permutations of s_1 and checking if any of them exist as substrings in s_2 . However, this method is highly inefficient with a time complexity of $O(n \times m!)$, where m is the length of s_1 , making it impractical for larger strings.

Another approach might use brute force to examine every possible substring of s_2 with the length equal to s_1 and compare character frequencies. This method has a time complexity of $O(n \times m)$, which is less efficient compared to the sliding window technique.

The sliding window method remains the most optimal and straightforward solution for this problem due to its linear time complexity and efficient space usage.

Similar Problems to This One

There are several other problems that involve finding substrings or sequences with specific constraints, such as:

- Longest Substring with At Most K Distinct Characters
- Longest Substring Without Repeating Characters
- Minimum Window Substring
- Longest Repeating Character Replacement
- Word Break

Things to Keep in Mind and Tricks

- **Sliding Window Technique:** Utilize two pointers to define the current window and adjust them based on constraints.
- **Hash Map for Frequency Counts:** Use a hash map to keep track of character frequencies within the window for efficient constraint checking.

- **Dynamic Window Adjustment:** Expand the window by moving the ‘right’ pointer and contract it by moving the ‘left’ pointer when constraints are violated.
- **Edge Case Handling:** Always consider edge cases such as empty strings, single-character strings, and strings with all identical characters.
- **Optimizing Updates:** Update the result only after ensuring the current window meets the constraints to avoid unnecessary calculations.

Corner and Special Cases to Test When Writing the Code

When implementing the ‘check-inclusion’ function, it is crucial to test the following edge cases to ensure robustness:

- **Empty Strings:** ‘s1 = “”, ‘s2 = “”‘ should handle gracefully, possibly returning ‘False’.
- **Single Character:** ‘s1 = ”a”, ‘s2 = ”a”‘ should return ‘True’.
- **All Characters the Same:** ‘s1 = ”aa”, ‘s2 = ”aaaa”‘ should return ‘True’.
- **No Permutation Present:** ‘s1 = ”abc”, ‘s2 = ”defghijk”‘ should return ‘False’.
- **Multiple Valid Permutations:** ‘s1 = ”ab”, ‘s2 = ”eidbaooo”‘ should return ‘True’.
- **Overlapping Permutations:** ‘s1 = ”ab”, ‘s2 = ”abab”‘ should return ‘True’.
- **Strings with Repeating Characters:** ‘s1 = ”aab”, ‘s2 = ”eidbaabooo”‘ should return ‘True’.
- **Large Input Size:** Test with very long strings to ensure that the implementation performs efficiently without exceeding memory limits.
- **Different Lengths:** ‘s1‘ longer than ‘s2‘, e.g., ‘s1 = ”abcd”, ‘s2 = ”abc”‘ should return ‘False’.

Chapter 11

Binary Search

Problem 11.1 Binary Search in a Sorted Array

The **Binary Search** problem demonstrates the power of divide-and-conquer techniques in optimizing search operations. Given a sorted array, the task is to efficiently locate a target value using binary search, achieving logarithmic time complexity.

Problem Statement

You are given an array of integers `nums`, which is sorted in ascending order, and an integer `target`. Write a function to search for `target` in `nums`. If `target` exists, return its index. Otherwise, return `-1`.

Input: - `nums`: A list of integers sorted in ascending order. - `target`: An integer to search for.

Output: - An integer representing the index of `target` if found, otherwise `-1`.

Example 1:

Input: `nums = [-1, 0, 3, 5, 9, 12]`, `target = 9`

Output: `4`

Explanation: `9` exists at index `4` in `nums`.

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: - $1 \leq \text{nums.length} \leq 10^4$ - $-10^4 \leq \text{nums}[i], \text{target} \leq 10^4$
 - All integers in `nums` are unique.

Algorithmic Approach

Binary search works by repeatedly dividing the search interval in half: 1. Start with two pointers, `left` and `right`, representing the bounds of the search interval. 2. Compute the middle index:

```
mid = left + (right - left)//2
```

3. Compare `nums[mid]` with the `target`: - If `nums[mid] == target`, return `mid`. - If `nums[mid] < target`, search the right half by setting `left = mid + 1`. - If `nums[mid] > target`, search the left half by setting `right = mid - 1`. 4. Repeat until `left > right`. 5. If the loop ends without finding `target`, return `-1`.

Complexities

- **Time Complexity:** $O(\log n)$, where n is the length of `nums`.
- **Space Complexity:** $O(1)$, as no additional space is used beyond variables.

Python Implementation

The following code implements the binary search algorithm:

```
def search(nums, target):
    left, right = 0, len(nums) - 1
    while left <= right:
        mid = left + (right - left) // 2
        if nums[mid] == target:
            return mid
        elif nums[mid] < target:
            left = mid + 1
        else:
            right = mid - 1
    return -1

# Example usage:
nums = [-1, 0, 3, 5, 9, 12]
target = 9
print(search(nums, target)) # Output: 4
```

Why This Approach?

Binary search is optimal for sorted arrays due to its logarithmic time complexity. By halving the search space at each step, it minimizes the number of comparisons needed to locate the target.

Alternative Approaches

- **Linear Search:** Iterate through `nums` from start to end, returning the index of `target` if found. This approach has $O(n)$ complexity and is inefficient for large arrays.
- **Recursive Binary Search:** Implement binary search recursively. While elegant, it uses additional stack space, resulting in $O(\log n)$ space complexity.

Similar Problems

- **Search Insert Position:** Find the index where a target should be inserted in a sorted array.
- **First and Last Position in Sorted Array:** Locate the start and end indices of a target value.
- **Find Minimum in Rotated Sorted Array:** Identify the smallest value in a rotated sorted array.

Corner and Special Cases to Test

- Empty array: `nums = []`, `target = 5`.
- Single element array: `nums = [3]`, `target = 3`.
- Large arrays with varying `target` values.

Conclusion

Binary search is a quintessential algorithm for searching sorted data efficiently. Its simplicity and effectiveness make it a fundamental tool in algorithm design, with widespread applications in data retrieval, game development, and optimization problems.

Problem 11.2 Search in Rotated Sorted Array

The **Search in Rotated Sorted Array** problem tests your ability to combine binary search with logic to handle shifted or rotated arrays. This is a frequent interview question that evaluates both problem-solving skills and the ability to optimize search operations.

Problem Statement

You are given a rotated sorted array `nums` of unique integers and a target integer `target`. Return the index of `target` if it exists in `nums`; otherwise, return `-1`.

The array is sorted in ascending order and then rotated at some unknown pivot index k . For example, $[0, 1, 2, 4, 5, 6, 7]$ might become $[4, 5, 6, 7, 0, 1, 2]$. Your task is to search for the `target` in $O(\log n)$ time.

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
```

Algorithmic Approach

The problem is best solved using a modified binary search¹.

Key Observations:

- One half of the array (either left or right) is always sorted².
- The target can only lie in one of the two halves, depending on its value relative to the sorted half.

The algorithm:

1. Initialize two pointers, `left` at the start and `right` at the end of the array.

¹ Binary search ensures $O(\log n)$ time complexity, making it highly efficient for sorted or partially sorted arrays

² This property is a direct result of the array being rotated at a single pivot point

2. While $\text{left} \leq \text{right}$, calculate the middle index:

$$\text{mid} = \text{left} + \frac{\text{right} - \text{left}}{2}$$

3. Check if the middle element is the target. If yes, return mid .

4. Determine whether the left half or the right half is sorted:

- If the left half is sorted:

- Check if the target lies within this range³.
- If yes, adjust the right pointer to $\text{mid} - 1$; otherwise, adjust left to $\text{mid} + 1$.

³ A sorted range is defined as $\text{nums}[\text{left}] \leq \text{target} < \text{nums}[\text{mid}]$

- If the right half is sorted:

- Check if the target lies within this range⁴.
- If yes, adjust the left pointer to $\text{mid} + 1$; otherwise, adjust right to $\text{mid} - 1$.

⁴ Similarly, the range is defined as $\text{nums}[\text{mid}] < \text{target} \leq \text{nums}[\text{right}]$

5. If the loop exits without finding the target, return -1 .

Complexities

- **Time Complexity:** $O(\log n)$, as the array is halved in each iteration.
- **Space Complexity:** $O(1)$, as the algorithm operates in-place without additional memory allocation.

Python Implementation

```
class Solution:
    def search(self, nums: List[int], target: int) -> int:
        left, right = 0, len(nums) - 1

        while left <= right:
            mid = left + (right - left) // 2

            # Check if the middle element is the target
            if nums[mid] == target:
                return mid

            # Determine which half is sorted
            if nums[left] <= nums[mid]:
                # Left half is sorted
                if nums[left] <= target < nums[mid]:
                    right = mid - 1
                else:
                    left = mid + 1
            else:
                right = mid - 1
```

```

# Right half is sorted
if nums[mid] < target <= nums[right]:
    left = mid + 1
else:
    right = mid - 1

return -1

```

Why This Approach?

This approach leverages the sorted property of one half of the array in each iteration to eliminate half of the search space⁵. By dynamically adjusting the search range based on the target's position relative to the sorted half, the algorithm maintains $O(\log n)$ efficiency.

⁵ Binary search guarantees logarithmic time complexity by halving the search space repeatedly

Alternative Approaches

- **Linear Search:** Iterate through the array and check each element. This has $O(n)$ time complexity and is not suitable for large arrays.
- **Pivot Detection + Binary Search:** First, identify the pivot point (where the rotation occurs) using binary search, then perform binary search on the relevant segment. While still $O(\log n)$, this involves two binary search passes, making it less efficient.

Similar Problems

- Find Minimum in Rotated Sorted Array:** Identify the smallest element in a rotated sorted array.
- Search in Rotated Sorted Array II:** Handle duplicates while searching for the target.
- Find Peak Element:** Locate a peak element in an array where adjacent elements are distinct.

Things to Keep in Mind and Tricks

- Always check if the middle element is the target before further processing⁶.
- Handle edge cases like arrays with a single element, no rotation, or extreme rotations (pivot at the first or last element).
- Use integer division (`//`) for calculating `mid` to avoid potential overflow in some languages.

⁶ This prevents unnecessary checks and guarantees correctness

Corner and Special Cases to Test

- **Single Element Array:** [1], target = 1 or 2.
- **No Rotation:** [1, 2, 3, 4], target = 3.
- **Full Rotation:** [1, 2, 3, 4], rotated back to [1, 2, 3, 4], target = 4.
- **Target Not in Array:** [4, 5, 6, 7, 0, 1, 2], target = 8.
- **Extreme Rotation:** [2, 3, 4, 5, 6, 7, 0, 1], target = 0.

Conclusion

The **Search in Rotated Sorted Array** problem demonstrates how binary search can be adapted to handle more complex scenarios like rotations. By exploiting the partially sorted structure of the array, this algorithm efficiently narrows down the search space, achieving optimal performance. Mastering this problem prepares you to tackle related challenges involving rotated or partially sorted data structures.

Problem 11.3 Find Minimum in Rotated Sorted Array

The **Find Minimum in Rotated Sorted Array** problem involves identifying the smallest element in a rotated sorted array. This problem demonstrates the efficient application of binary search techniques to leverage the sorted structure of the array while addressing the rotation.

Problem Statement

Given a rotated sorted array, locate its minimum element. A sorted array is considered rotated if some elements from the beginning are moved to the end, maintaining the overall ascending order. The solution should run in $O(\log n)$ time complexity.

Input: - `nums`: A list of integers representing the rotated sorted array.

Output: - An integer representing the minimum element in `nums`.

Example 1:

Input: `nums = [3, 4, 5, 1, 2]`

Output: 1

Explanation: The minimum value is 1.

Example 2:

Input: `nums = [4, 5, 6, 7, 0, 1, 2]`
 Output: 0
 Explanation: The minimum value is 0.

Example 3:

Input: `nums = [11, 13, 15, 17]`
 Output: 11
 Explanation: The array is not rotated, so the first element is the smallest.

—

Algorithmic Approach

To solve the problem efficiently: 1. Use binary search with two pointers, `left` and `right`, representing the bounds of the current search space. 2. Compute the middle index:

```
mid = left + (right - left)//2
```

3. Compare `nums[mid]` with `nums[right]`: - If `nums[mid] > nums[right]`, the minimum must be in the right half. Update `left = mid + 1`. - Otherwise, the minimum is in the left half (including `mid`). Update `right = mid`. 4. Continue until `left == right`, at which point the minimum element is found at `nums[left]`.

—

Complexities

- **Time Complexity:** $O(\log n)$, since the search space is halved at each step.
- **Space Complexity:** $O(1)$, as no additional space is used beyond a few variables.

—

Python Implementation

```
class Solution:
    def findMin(self, nums: List[int]) -> int:
        left, right = 0, len(nums) - 1
        while left < right:
            mid = left + (right - left) // 2
```

```

if nums[mid] > nums[right]:
    left = mid + 1
else:
    right = mid
return nums[left]

# Example usage:
nums = [4, 5, 6, 7, 0, 1, 2]
solution = Solution()
print(solution.findMin(nums))  # Output: 0

```

Why This Approach?

Binary search is ideal for this problem because the array is partially sorted. Instead of iterating through all elements ($O(n)$), binary search exploits the sorted structure to locate the minimum in $O(\log n)$ time.

Alternative Approaches

1. **Linear Search ($O(n)$):** Scan all elements to find the minimum. This approach is straightforward but inefficient for large arrays.
 2. **Recursive Binary Search ($O(\log n)$):** The binary search logic can be implemented recursively, although it adds stack overhead.

Similar Problems

- **Search in Rotated Sorted Array:** Find the position of a target element in a rotated sorted array.
- **Find Peak Element:** Locate a peak element in an array.
- **Find Minimum in Rotated Sorted Array II:** Handles duplicates in the rotated array.

Corner Cases to Test

- Array is not rotated: `nums = [1, 2, 3, 4, 5]`.
 - Single element array: `nums = [1]`.
 - Array with two elements: `nums = [2, 1]`.
 - Minimum element is the last element: `nums = [3, 4, 5, 6, 1]`.
-

Conclusion

The "Find Minimum in Rotated Sorted Array" problem elegantly demonstrates the power of binary search in structured datasets. By efficiently narrowing down the search space, we achieve optimal performance while maintaining simplicity in implementation.

Problem 11.4 Search a 2D Matrix

The **Search a 2D Matrix** problem demonstrates the effective use of binary search in multidimensional arrays. The task is to efficiently determine if a target value exists within a matrix that is row and column-wise sorted.

Problem Statement

You are given an $m \times n$ matrix where:

- Each row is sorted in ascending order.
- The first integer of each row is greater than the last integer of the previous row.

Write a function to determine if a given target exists in the matrix. Return `true` if the target exists, and `false` otherwise.

Input: - `matrix`: A list of lists representing an $m \times n$ matrix. - `target`: An integer to search for.

Output: - A boolean indicating whether the `target` exists in the matrix.

Example 1:

```
Input: matrix = [
    [1, 3, 5, 7],
    [10, 11, 16, 20],
    [23, 30, 34, 60]
], target = 3
```

Output: true

Example 2:

```
Input: matrix = [
    [1, 3, 5, 7],
    [10, 11, 16, 20],
    [23, 30, 34, 60]
], target = 13
Output: false
```

Constraints: - $m == \text{matrix.length}$ - $n == \text{matrix[0].length} - 1 \leq m, n \leq 100$ - $-10^4 \leq \text{matrix[i][j]}, \text{target} \leq 10^4$

Algorithmic Approach

This problem can be solved by flattening the $m \times n$ matrix into a single sorted array and applying binary search.

Steps for Binary Search on 2D Matrix

1. Treat the matrix as a 1D sorted array of size $m \times n$.
2. Use binary search: - The middle element in this conceptual 1D array corresponds to:

```
matrix[mid // n][mid % n]
```

where mid is the index in the 1D array.

3. Compare the middle element with the target: - If it matches, return `true`. - If it is greater, search the left half. - If it is smaller, search the right half.
4. Continue until the search space is exhausted.

Complexities

- **Time Complexity:** $O(\log(m \times n))$, where m and n are the matrix dimensions.
 - **Space Complexity:** $O(1)$, as the algorithm uses no additional space beyond variables.
-

Python Implementation

```

def searchMatrix(matrix, target):
    if not matrix or not matrix[0]:
        return False

    m, n = len(matrix), len(matrix[0])
    left, right = 0, m * n - 1

    while left <= right:
        mid = left + (right - left) // 2
        mid_element = matrix[mid // n][mid % n]

        if mid_element == target:
            return True
        elif mid_element < target:
            left = mid + 1
        else:
            right = mid - 1

    return False

# Example usage:
matrix = [
    [1, 3, 5, 7],
    [10, 11, 16, 20],
    [23, 30, 34, 60]
]
target = 3
print(searchMatrix(matrix, target))  # Output: true

```

Why This Approach?

Binary search is optimal for sorted data due to its logarithmic time complexity. By treating the 2D matrix as a virtual 1D array, we simplify the problem to a standard binary search while maintaining efficiency.

Alternative Approach

For a different matrix structure where rows and columns are independently sorted but not sequential:

- Start at the top-right corner.
- Move left if the current element is greater than the target.
- Move down if the current element is less than the target.
- This approach has $O(m + n)$ complexity and works for matrices sorted row-wise and column-wise but not in blocks.

Similar Problems

- **Search a 2D Matrix II:** Locate a target in a matrix sorted row-wise and column-wise, but without sequential row-column relationships.
 - **Binary Search:** Core algorithm applied here.
 - **Find Element in Rotated Sorted Array:** Similar in logic but involves rotated sorted data.
-

Corner Cases to Test

- Empty matrix: `matrix = []`.
 - Single element matrix: `matrix = [[5]], target = 5`.
 - Large m and n : Test performance for $m, n = 100$.
 - Target is the smallest or largest element in the matrix.
-

Conclusion

The Search a 2D Matrix problem demonstrates the utility of binary search for structured data. By mapping a 2D matrix to a 1D search space, it ensures efficiency and simplicity, making it an essential technique for optimizing search operations.

Problem 11.5 Kth Smallest Element in a Sorted Matrix

The **Kth Smallest Element in a Sorted Matrix** problem is a classic challenge that combines elements of sorting, binary search, and heap-based optimization. The task is to locate the k -th smallest element in a row and column-wise sorted matrix efficiently.

Problem Statement

Given an $n \times n$ matrix where each row and column is sorted in ascending order, return the k -th smallest element in the matrix.

Input: - `matrix`: A list of lists representing an $n \times n$ matrix where rows and columns are sorted. - `k`: An integer representing the rank of the desired element.

Output: - An integer representing the k -th smallest element.

Example 1:

```
Input: matrix = [
    [1, 5, 9],
    [10, 11, 13],
    [12, 13, 15]
], k = 8
Output: 13
Explanation: The sorted order of elements is
[1, 5, 9, 10, 11, 12, 13, 13, 15]. The 8th smallest element is 13.
```

Example 2:

```
Input: matrix = [
    [1, 2],
    [1, 3]
], k = 2
Output: 1
Explanation: The sorted order of elements is [1, 1, 2, 3]. The 2nd smallest element is 1.
```

Constraints: - $1 \leq n \leq 300$ - $-10^9 \leq \text{matrix}[i][j] \leq 10^9$ - All rows and columns of `matrix` are sorted. - $1 \leq k \leq n^2$

—

Algorithmic Approaches

This problem can be solved using multiple approaches based on the constraints and desired efficiency.

1. Min-Heap Approach ($O(k \log n)$)

Key Idea: Use a min-heap to efficiently extract the smallest elements while traversing the matrix row by row. At each step: 1. Push the first element of each row into the heap. 2. Pop the smallest element from the heap k times to find the k -th smallest.

Algorithm Steps: 1. Initialize a min-heap with the first element of each row along with its indices (`row, col`). 2. Extract the smallest element and push the next element from the same row into the heap. 3. Repeat until the k -th smallest element is found.

2. Binary Search on Value Range ($O(n \log(\max - \min))$)

Key Idea: The matrix is sorted, so the range of possible values is $[\text{matrix}[0][0], \text{matrix}[n - 1][n - 1]]$. Use binary search to count how many elements are less than or equal to a mid-point in the range: 1. Compute the middle value of the current range. 2. Count elements less than or equal to the middle value using a two-pointer approach. 3. Adjust the range based on whether the count is less than or greater than k .

—

Complexities

1. **Min-Heap Approach:** - Time Complexity: $O(k \log n)$, where k extractions and $O(\log n)$ insertions per element. - Space Complexity: $O(n)$ for the heap.

2. **Binary Search Approach:** - Time Complexity: $O(n \log(\max - \min))$, where counting elements in n rows takes $O(n)$. - Space Complexity: $O(1)$, as no extra space is used beyond variables.

—

Python Implementations

Min-Heap Implementation

```
import heapq

def kthSmallest(matrix, k):
    n = len(matrix)
    heap = [(matrix[i][0], i, 0) for i in range(n)]
    heapq.heapify(heap)

    for _ in range(k - 1):
        val, row, col = heapq.heappop(heap)
        if col + 1 < n:
            heapq.heappush(heap, (matrix[row][col + 1], row, col + 1))

    return heapq.heappop(heap)[0]

# Example usage:
matrix = [
    [1, 5, 9],
    [10, 11, 13],
    [12, 13, 15]
]
k = 8
```

```
print(kthSmallest(matrix, k))  # Output: 13
```

Binary Search Implementation

```
def kthSmallest(matrix, k):
    def countLessEqual(mid):
        count, n = 0, len(matrix)
        row, col = n - 1, 0
        while row >= 0 and col < n:
            if matrix[row][col] <= mid:
                count += row + 1
                col += 1
            else:
                row -= 1
        return count

    low, high = matrix[0][0], matrix[-1][-1]
    while low < high:
        mid = low + (high - low) // 2
        if countLessEqual(mid) < k:
            low = mid + 1
        else:
            high = mid

    return low

# Example usage:
matrix = [
    [1, 5, 9],
    [10, 11, 13],
    [12, 13, 15]
]
k = 8
print(kthSmallest(matrix, k))  # Output: 13
```

Why These Approaches?

The min-heap approach is intuitive and handles k extractions efficiently for small k . The binary search approach is optimal for large matrices, leveraging the sorted property of rows and columns to reduce unnecessary operations.

Similar Problems

- **Kth Largest Element in an Array:** Use heaps or quickselect to find the k -th largest element.
 - **Median of Two Sorted Arrays:** Combines binary search with merging concepts.
 - **Merge K Sorted Lists:** Similar heap-based solution.
-

Corner Cases to Test

- Single element matrix: `matrix = [[1]]`, $k = 1$.
 - All elements are the same: `matrix = [[2, 2], [2, 2]]`, $k = 3$.
 - Large k : Ensure performance for $n = 300$ and $k = n^2$.
-

Conclusion

The k -th smallest element in a sorted matrix problem highlights the versatility of heap-based and binary search techniques. Both approaches leverage the sorted structure of the matrix to achieve efficiency, making this problem an excellent demonstration of algorithmic optimization.

Problem 11.6 Median of Two Sorted Arrays

The **Median of Two Sorted Arrays** problem is a classic computational challenge that involves merging and analyzing two sorted arrays to find their median. It tests proficiency in optimizing algorithms for time complexity and requires a deep understanding of binary search techniques.

Problem Statement

Given two sorted arrays `nums1` and `nums2` of sizes m and n respectively, return the median of the two sorted arrays. The solution should have a time complexity of $O(\log(m + n))$.

Input: - `nums1`: A sorted list of integers. - `nums2`: Another sorted list of integers.

Output: - A float representing the median of the combined arrays.

Example 1:

Input: `nums1 = [1, 3]`, `nums2 = [2]`

Output: 2.0

Explanation: Merging `nums1` and `nums2` gives `[1, 2, 3]`. The median is 2.0.

Example 2:

Input: `nums1 = [1, 2]`, `nums2 = [3, 4]`

Output: 2.5

Explanation: Merging `nums1` and `nums2` gives `[1, 2, 3, 4]`. The median is $(2 + 3)/2 = 2.5$.

Constraints: - $0 \leq m, n \leq 1000$ - $1 \leq m + n \leq 2000$ - $-10^6 \leq \text{nums1}[i], \text{nums2}[i] \leq 10^6$

Algorithmic Approach

Finding the median in $O(\log(m+n))$ requires a binary search approach rather than merging the arrays, which would take $O(m+n)$. The algorithm works by applying binary search on the smaller array.

Steps:

1. Ensure `nums1` is the smaller array; if not, swap `nums1` and `nums2`.
 2. Perform binary search on the smaller array:
 - Partition both arrays such that the left halves contain the smaller elements.
 - Use two pointers to adjust the partition dynamically.
 3. Check if the partition is valid:
 - If the maximum element of the left half is less than or equal to the minimum element of the right half, the partition is correct.
 - Otherwise, adjust the partition by moving the binary search pointers.
 4. Compute the median:
 - If the combined array size is odd, the median is the maximum of the left half.
 - If even, the median is the average of the maximum of the left half and the minimum of the right half.
-

Complexities

- **Time Complexity:** $O(\log(\min(m, n)))$, as the binary search is applied to the smaller array.
- **Space Complexity:** $O(1)$, as the algorithm operates in-place.

—

Python Implementation

```
class Solution:
    def findMedianSortedArrays(self, nums1: List[int], nums2: List[int]) -> float:
        if len(nums1) > len(nums2):
            nums1, nums2 = nums2, nums1 # Ensure nums1 is the smaller array

        x, y = len(nums1), len(nums2)
        left, right = 0, x

        while left <= right:
            partitionX = (left + right) // 2
            partitionY = (x + y + 1) // 2 - partitionX

            maxX = float('-inf') if partitionX == 0 else nums1[partitionX - 1]
            minX = float('inf') if partitionX == x else nums1[partitionX]
            maxY = float('-inf') if partitionY == 0 else nums2[partitionY - 1]
            minY = float('inf') if partitionY == y else nums2[partitionY]

            if maxX <= minY and maxY <= minX:
                if (x + y) % 2 == 0:
                    return (max(maxX, maxY) + min(minX, minY)) / 2
                else:
                    return max(maxX, maxY)
            elif maxX > minY:
                right = partitionX - 1
            else:
                left = partitionX + 1

        raise ValueError("Input arrays are not sorted properly.")

# Example usage:
nums1 = [1, 3]
nums2 = [2]
solution = Solution()
print(solution.findMedianSortedArrays(nums1, nums2)) # Output: 2.0
```

—

Why This Approach?

This approach is efficient because it avoids merging the arrays. By focusing on partitioning the arrays into balanced halves, the algorithm achieves $O(\log(\min(m, n)))$ time complexity while maintaining the simplicity of binary search.

Alternative Approaches

1. **Merging Arrays:** Combine the arrays and find the median directly. This approach has $O(m + n)$ complexity and is less efficient for large datasets.
 2. **Heap-Based Methods:** Use a min-heap or max-heap to extract the median. While useful in streaming data, it is unnecessary for this problem.
-

Similar Problems

- **Kth Largest Element in an Array:** Use similar partitioning techniques to locate the k -th largest element.
 - **Median of a Stream:** Continuously calculate the median as elements arrive.
 - **Find Median in a Sorted Matrix:** Combines searching and partitioning methods for 2D arrays.
-

Corner Cases to Test

- Empty array: `nums1 = []`, `nums2 = [1]`.
 - Arrays of different sizes: `nums1 = [1, 2]`, `nums2 = [3, 4, 5, 6]`.
 - Arrays with negative numbers: `nums1 = [-5, -3]`, `nums2 = [-2, -1]`.
-

Conclusion

The **Median of Two Sorted Arrays** problem highlights the elegance of binary search in solving complex problems efficiently. By leveraging the sorted nature of

the input, the algorithm achieves optimal performance and showcases the versatility of partitioning techniques.

Chapter 12

Recursion and Backtracking

Chapter 13

Recursion

Recursion is a fundamental concept in computer science and programming, where a function calls itself to solve smaller instances of a problem. This self-referential technique is powerful for solving problems that can be broken down into simpler, repetitive subproblems, such as divide-and-conquer algorithms, backtracking, and tree traversals.

Why Learn Recursion?

Recursion is essential for understanding many advanced algorithms and problem-solving techniques. It simplifies the code for problems involving hierarchical data structures like trees and graphs, or problems where the solution builds upon solutions to smaller subproblems.

Key Features of Recursion

- **Base Case:** Every recursive function must have a termination condition, or base case, to prevent infinite loops.
- **Recursive Step:** The function performs a smaller portion of the work and calls itself to handle the remaining problem.
- **Stack Behavior:** Recursive calls are managed using the call stack, which stores the function's state for each recursive invocation.

Advantages of Recursion

- Simplifies the implementation of problems that are naturally recursive, such as tree traversals.

- Provides a clean and concise approach for problems like generating permutations or solving mathematical sequences.

Disadvantages of Recursion

- Excessive recursive calls can lead to stack overflow errors for large inputs.
- Often has a higher space complexity due to the call stack compared to iterative solutions.

Common Applications of Recursion

Recursion is widely used in various domains, including:

- **Divide-and-Conquer Algorithms:** Examples include Merge Sort, Quick Sort, and Binary Search.
- **Backtracking:** Used in solving puzzles like Sudoku, N-Queens, and maze traversal.
- **Tree and Graph Traversals:** Recursion is natural for Depth-First Search (DFS) and processing hierarchical data structures.
- **Dynamic Programming:** Many DP problems, such as the Fibonacci sequence, are naturally recursive but can be optimized using memoization or tabulation.

How to Write Recursive Functions

Writing a recursive function involves breaking the problem into smaller subproblems and identifying the base case. Follow these steps:

1. **Identify the Base Case:** Define when the recursion should stop.
2. **Define the Recursive Relation:** Determine how the solution to the problem depends on smaller instances of itself.
3. **Combine Results:** Use the results of the recursive calls to build the final solution.

Example: Factorial of a Number

The factorial of a non-negative integer n is defined as $n! = n \cdot (n - 1)!$, with the base case $0! = 1$.

```

def factorial(n):
    if n == 0:  # Base case
        return 1
    return n * factorial(n - 1)  # Recursive step

# Example usage:
print(factorial(5))  # Output: 120

```

Understanding the Call Stack

Each recursive call adds a new frame to the call stack. The stack unwinds as the base case is reached, and results are returned back up the chain. For example, when computing $\text{factorial}(3)$, the following calls occur:

`factorial(3) -> factorial(2) -> factorial(1) -> factorial(0)`

The stack then resolves as:

`factorial(0) = 1 -> factorial(1) = 1 -> factorial(2) = 2 -> factorial(3) = 6`

Tail Recursion

Tail recursion occurs when the recursive call is the last operation in the function. Some programming languages optimize tail recursion to reuse the current stack frame, reducing space complexity. Python does not support tail recursion optimization natively.

Common Problems to Practice Recursion

- **Fibonacci Sequence:** Compute the n th Fibonacci number using recursion.
- **Permutations and Combinations:** Generate all permutations or subsets of a set.
- **Tower of Hanoi:** Solve the problem of moving disks between rods following specific rules.
- **Tree Traversals:** Implement pre-order, in-order, and post-order traversals of a binary tree.
- **String Reversal:** Reverse a string recursively.

Tips for Writing Recursive Solutions

- Always ensure the base case is correct and covers all possible edge cases.
- Be mindful of the input size to avoid stack overflow errors.
- Use memoization or dynamic programming to optimize redundant recursive calls.
- Visualize the problem using a recursion tree to understand the flow of calls.

Conclusion

Recursion is a versatile and powerful tool in programming, enabling elegant solutions to complex problems. Understanding its principles and applications will significantly enhance your problem-solving skills and prepare you for tackling advanced algorithmic challenges.

Chapter 14

Backtracking

Backtracking is a specialized form of recursion used to solve problems by exploring all potential solutions in a systematic manner. It is particularly effective for problems with constraints, as it prunes invalid paths early, reducing the number of possibilities to consider.

What is Backtracking?

Backtracking is a problem-solving technique where you incrementally build candidates for a solution and abandon a candidate as soon as it is determined to be invalid. The process involves:

- Exploring one branch of the solution space at a time.
- Undoing the last step (backtracking) if a constraint is violated.
- Proceeding to the next branch if the current one fails.

Key Characteristics of Backtracking

- It explores all possible solutions but avoids unnecessary work by pruning invalid paths.
- It uses recursion and a stack (either explicitly or implicitly) to backtrack to previous states.
- It is particularly suited for problems involving permutations, combinations, and subsets with constraints.

Advantages of Backtracking

- Systematically explores solution spaces, guaranteeing completeness.
- Reduces computation by pruning invalid solutions early.

Disadvantages of Backtracking

- Can be computationally expensive for large problem spaces.
- Requires careful implementation to avoid unnecessary calculations or infinite recursion.

Applications of Backtracking

- **Constraint Satisfaction Problems:** Sudoku, N-Queens, and Crossword solving.
- **Combinatorial Problems:** Generating subsets, permutations, and combinations.
- **Pathfinding Problems:** Maze traversal and word search in grids.

General Steps for Backtracking

1. **Define the Problem:** Clearly specify the constraints and the goal state.
2. **Recursive Function:** Write a recursive function that:
 - Decides whether to include the current option.
 - Validates the current state against the constraints.
3. **Base Case:** Define the termination condition when a solution is found or all options are exhausted.
4. **Pruning:** Implement checks to eliminate invalid states early.

Problem 14.1 N-Queens Problem

The N-Queens problem involves placing n queens on an $n \times n$ chessboard such that no two queens threaten each other.

```
def solveNQueens(n):
    def is_valid(board, row, col):
        for i in range(row):
```

```

    if board[i] == col or abs(board[i] - col) == abs(i - row):
        return False
    return True

def backtrack(row):
    if row == n:
        result.append(["." * col + "Q" + "." * (n - col - 1) for col in board
                      ↵ ])
        return
    for col in range(n):
        if is_valid(board, row, col):
            board[row] = col
            backtrack(row + 1)
            board[row] = -1

    result = []
board = [-1] * n
backtrack(0)
return result

# Example usage:
print(solveNQueens(4))  # Output: All valid configurations of 4 queens

```

Tips for Writing Backtracking Solutions

- Clearly identify constraints and the base case.
- Use pruning conditions to reduce the search space early.
- Test with edge cases to ensure robustness.

Conclusion

Backtracking is a powerful problem-solving paradigm that complements recursion by adding constraint propagation and pruning. Understanding backtracking equips you to tackle complex combinatorial and constraint satisfaction problems efficiently.

Problem 14.2 Generate Parentheses

The **Generate Parentheses** problem is a classic application of recursion and backtracking. It involves generating all possible combinations of well-formed parentheses given n pairs of parentheses. This problem is commonly used to teach recursive problem-solving and the efficient pruning of invalid states.

Problem Statement

Given an integer n , write a function that generates all combinations of n pairs of well-formed parentheses.

Input: - An integer n , where $n \geq 1$.

Output: - A list of strings representing all valid combinations of n pairs of parentheses.

Example:

Input: $n = 3$

Output: `[”((()))”, ”(()())”, ”((())()”, ”()((())”, ”()()()”]`

Explanation: Each string in the output represents a unique way to arrange three pairs of parentheses such that they are balanced.

Algorithmic Approach

The problem can be efficiently solved using backtracking. The idea is to construct the solution incrementally, ensuring that at every step the partial solution remains valid.

Steps:

- Use a recursive function to build the string character by character.
- Keep track of the number of open and close parentheses added so far.
- At each step, add an open parenthesis if the number of open parentheses used is less than n ¹.
- Add a close parenthesis if the number of close parentheses used is less than the number of open parentheses used².
- Stop when the string has $2n$ characters, indicating a complete solution.

¹ Adding an open parenthesis increases the count of open parentheses used

² A close parenthesis is valid only if it matches a previously added open parenthesis

Complexities

- **Time Complexity:** The total number of valid combinations is given by the n th Catalan number, $C_n = \frac{1}{n+1} \binom{2n}{n}$. The algorithm explores all valid combinations, so the time complexity is $O(4^n / \sqrt{n})$ in terms of the growth rate of the Catalan numbers.
- **Space Complexity:** $O(n)$, for the recursion stack used during backtracking.

Python Implementation

Below is the Python implementation using backtracking:

```
def generateParenthesis(n):
    def backtrack(current, open_count, close_count):
        # Base case: If the current string has 2n characters, it's a valid
        # combination
        if len(current) == 2 * n:
            result.append(current)
            return

        # Add an open parenthesis if possible
        if open_count < n:
            backtrack(current + "(", open_count + 1, close_count)

        # Add a close parenthesis if valid
        if close_count < open_count:
            backtrack(current + ")", open_count, close_count + 1)

    result = []
    backtrack("", 0, 0) # Start with an empty string and zero counts
    return result

# Example usage:
print(generateParenthesis(3)) # Output: ["((()))", "(()())", "((())()", "(()(()",
                           # ")()()"]
```

Why This Approach?

Backtracking is well-suited for this problem because it constructs the solution incrementally, exploring only valid configurations of parentheses at each step. This avoids generating invalid combinations and then filtering them, making the algorithm both time and space efficient.

Alternative Approaches

An iterative approach can be used by maintaining a queue of partial strings, but it is less intuitive and harder to implement compared to the recursive backtracking approach. Dynamic programming can also be used by leveraging the properties of the Catalan numbers, but it requires additional space to store intermediate results.

Similar Problems

- **Valid Parentheses:** Check if a given string containing parentheses is well-formed.

- **Binary Tree Paths:** Generate all root-to-leaf paths in a binary tree, which also involves backtracking.
- **Subsets:** Generate all subsets of a set, another classic backtracking problem.

Things to Keep in Mind and Tricks

- Always ensure that the number of close parentheses does not exceed the number of open parentheses at any point. This prevents invalid states.
- Prune the recursion tree early by stopping further exploration when the conditions for adding parentheses are not met.
- Use a helper function with parameters to track the current state and maintain the simplicity of the main function.

Corner and Special Cases to Test

- $n = 1$: The smallest possible input, where the result should be `[")"]`.
- $n = 0$: A special case where the input is 0, and the result should be an empty list `[]`.
- Larger values of n , such as $n = 4$, to verify scalability and correctness.

Conclusion

The “Generate Parentheses” problem elegantly demonstrates the power of recursion and backtracking in solving combinatorial problems. Understanding this problem provides foundational knowledge for tackling other recursive and combinatorial challenges efficiently.

Problem 14.3 Letter Combinations of a Phone Number

The “Letter Combinations of a Phone Number” problem is a classic example of recursion and backtracking used to generate all possible combinations of letters mapped to digits on a phone keypad.

Problem Statement

Given a string containing digits from 2 to 9 (inclusive), return all possible letter combinations that the digits could represent. Each digit maps to specific letters as shown on a traditional phone keypad.

Input: - A string `digits`, consisting of digits from 2 to 9.

Output: - A list of strings, where each string represents a possible combination of letters.

Example 1:

Input: `digits = "23"`

Output:

`["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]`

Example 2:

Input: `digits = ""`

Output:

`[]`

Example 3:

Input: `digits = "2"`

Output:

`["a", "b", "c"]`

Mapping of Digits to Letters

The mapping follows the traditional phone keypad:

$$\begin{aligned} 2 &\rightarrow \{a, b, c\}, & 3 &\rightarrow \{d, e, f\}, & 4 &\rightarrow \{g, h, i\}, \\ 5 &\rightarrow \{j, k, l\}, & 6 &\rightarrow \{m, n, o\}, & 7 &\rightarrow \{p, q, r, s\}, \\ 8 &\rightarrow \{t, u, v\}, & 9 &\rightarrow \{w, x, y, z\}. \end{aligned}$$

Algorithmic Approach

The problem can be solved using a backtracking approach:

- If the input string is empty, return an empty list immediately³.
- Use recursion to explore each possible letter for the current digit and proceed to the next digit.
- Maintain a list to track the current combination of letters.
- Once all digits are processed, add the complete combination to the result list.

³ This handles edge cases where there are no digits to process.

Complexities

- **Time Complexity:** $O(4^n)$, where n is the length of digits, since each digit maps to at most 4 letters.
- **Space Complexity:** $O(n)$, for the recursion stack.

Python Implementation

Below is the Python implementation using backtracking:

```
def letterCombinations(digits):
    if not digits:
        return []

    # Map digits to corresponding letters
    phone_map = {
        "2": "abc", "3": "def", "4": "ghi", "5": "jkl",
        "6": "mno", "7": "pqrs", "8": "tuv", "9": "wxyz"
    }

    def backtrack(index, current):
        # Base case: if the current combination is complete
        if index == len(digits):
            result.append("".join(current))
            return

        # Get the letters corresponding to the current digit
        letters = phone_map[digits[index]]
        for letter in letters:
            current.append(letter) # Add the letter
            backtrack(index + 1, current) # Recur for the next digit
            current.pop() # Backtrack by removing the last letter

    result = []
    backtrack(0, [])
    return result

# Example usage:
print(letterCombinations("23")) # Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]
```

Why This Approach?

Backtracking is ideal for this problem because it systematically explores all valid combinations of letters, pruning invalid paths efficiently. The use of recursion makes the implementation clean and intuitive.

Alternative Approaches

An iterative approach can also generate combinations using a queue. However, it requires more bookkeeping and is less elegant than the recursive backtracking solution.

Similar Problems

- **Subsets:** Generate all subsets of a given set.
- **Permutations:** Generate all permutations of a set of numbers.
- **Word Search:** Find words in a grid using backtracking.

Things to Keep in Mind and Tricks

- Handle edge cases like empty input or single-digit input separately.
- Precompute the mapping of digits to letters to avoid redundancy during recursion.
- Optimize by skipping invalid digits (e.g., 1 and 0 if included in input).

Corner and Special Cases to Test

- **Empty Input:** Input: `digits = ""`, Output: `[]`.
- **Single Digit:** Input: `digits = "7"`, Output: `["p", "q", "r", "s"]`.
- **Multiple Digits:** Input: `digits = "234"`, Verify all combinations are generated correctly.
- **Edge Case for Large Input:** Test with longer strings like `digits = "999"`.

Conclusion

The **Letter Combinations of a Phone Number** problem demonstrates the power of backtracking for generating combinatorial solutions efficiently. By systematically exploring all possibilities and pruning invalid paths, the solution ensures correctness and optimal performance.

Problem 14.4 Permutations

The **Permutations** problem is a fundamental combinatorial problem that involves generating all possible arrangements of a given set of distinct integers. This

problem demonstrates the power of recursion and backtracking in systematically exploring all permutations.

Problem Statement

Given an array `nums` of distinct integers, return all possible permutations of the elements.

Input: - A list of integers `nums`, where all integers are distinct.

Output: - A list of lists, where each inner list represents a unique permutation of `nums`.

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]]`

Algorithmic Approach

The problem can be solved using backtracking to systematically generate all permutations.

Steps:

- Use a recursive backtracking function to construct permutations incrementally.
- Swap elements in place to avoid using additional space for constructing permutations.
- At each recursion level, fix one element at the current position and recursively permute the remaining elements.
- Once all positions are filled, add the current permutation to the result list.

Complexities

- **Time Complexity:** $O(n!)$, where n is the number of elements in `nums`, as there are $n!$ permutations.
- **Space Complexity:** $O(n)$, for the recursion stack.

Python Implementation

Below is the Python implementation using backtracking:

```
def permute(nums):
    def backtrack(start):
        # Base case: If all positions are filled, add the current permutation
        if start == len(nums):
            result.append(nums[:])
            return

        for i in range(start, len(nums)):
            # Swap the current element with the start
            nums[start], nums[i] = nums[i], nums[start]
            # Recurse for the next position
            backtrack(start + 1)
            # Backtrack by swapping back
            nums[start], nums[i] = nums[i], nums[start]

    result = []
    backtrack(0)  # Start permuting from the first position
    return result

# Example usage:
print(permute([1, 2, 3]))  # Output: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1],
                           ↪ [3, 1, 2], [3, 2, 1]]
```

Why This Approach?

The backtracking approach efficiently explores all permutations by recursively fixing elements and swapping them back after recursion. The in-place swapping reduces additional space usage for constructing permutations.

Alternative Approaches

- **Iterative Approach:** Use a queue or stack to iteratively construct permutations. However, this approach is less intuitive than backtracking.
- **Library Functions:** In Python, the `itertools.permutations` function can generate permutations directly, but it may not provide the same educational

insight as implementing the algorithm yourself.

Similar Problems

- **Combinations:** Generate all subsets of a specific size k .
- **Subsets:** Generate all subsets (power set) of a given set.
- **Permutations with Duplicates:** Generate all unique permutations when the input contains duplicates.

Things to Keep in Mind and Tricks

- Avoid using additional space for tracking used elements by leveraging in-place swaps.
- Ensure that the input array does not contain duplicates unless the problem explicitly allows them.
- Test the algorithm with edge cases like an empty array or a single element.

Corner and Special Cases to Test

- **Empty Input:** Input: `nums = []`, Output: `[[]]`.
- **Single Element:** Input: `nums = [1]`, Output: `[[1]]`.
- **Small Input:** Input: `nums = [0, 1]`, Verify correctness.
- **Larger Input:** Input: `nums = [1, 2, 3, 4]`, Test performance.

Conclusion

The **Permutations** problem is a fundamental challenge in combinatorics that reinforces recursive problem-solving and backtracking techniques. Mastering this problem lays the groundwork for tackling more advanced combinatorial problems efficiently.

Problem 14.5 Combinations

The **Combinations** problem is a classic example of generating subsets using backtracking. Given two integers n and k , the task is to generate all possible combinations of k numbers chosen from the range $[1, n]$. This problem is fundamental in combinatorics and is often used to teach recursive problem-solving techniques.

Problem Statement

Given two integers n and k , return all possible combinations of k numbers chosen from the range $[1, n]$.

Input: - Two integers n and k , where $1 \leq k \leq n$.

Output: - A list of lists, where each inner list represents a unique combination of k numbers.

Example 1:

Input: $n = 4, k = 2$

Output:

$[[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]$

Example 2:

Input: $n = 1, k = 1$

Output:

$[[1]]$

Algorithmic Approach

The problem can be solved using backtracking to explore all possible subsets of size k .

Steps:

- Use a recursive function to construct combinations incrementally.
- Start from the first number in the range and add it to the current combination.
- Recursively add subsequent numbers while maintaining the constraint that the combination size does not exceed k .
- Backtrack by removing the last added number when the combination is complete or invalid.
- Continue until all valid combinations are generated.

Pruning:

- To optimize, stop recursion if the remaining numbers are insufficient to complete a valid combination⁴.

⁴ For example, if there are r numbers left and $r < k - \text{len}(\text{current_combination})$, further recursion is unnecessary

Complexities

- **Time Complexity:** $O\left(\binom{n}{k} \cdot k\right)$, where $\binom{n}{k}$ is the total number of combinations, and k is the cost of constructing each combination.
- **Space Complexity:** $O(k)$ for the recursion stack used to store the current combination.

Python Implementation

Below is the Python implementation using backtracking:

```
def combine(n, k):
    def backtrack(start, current):
        # Base case: If the current combination is of size k, add it to the result
        if len(current) == k:
            result.append(current[:])
            return

        # Iterate through the range, starting from the current number
        for i in range(start, n + 1):
            # Add the current number to the combination
            current.append(i)
            # Recurse to add the next number
            backtrack(i + 1, current)
            # Backtrack by removing the last added number
            current.pop()

    result = []
    backtrack(1, []) # Start from the first number with an empty combination
    return result

# Example usage:
print(combine(4, 2)) # Output: [[1, 2], [1, 3], [1, 4], [2, 3], [2, 4], [3, 4]]
```

Why This Approach?

Backtracking is an effective technique for generating combinations as it allows exploring all possibilities while pruning invalid paths early. The algorithm ensures that all combinations are generated without redundancy or unnecessary computations.

Alternative Approaches

- **Iterative Approach:** Use nested loops to construct combinations. This becomes unwieldy for larger k as the number of loops needed equals k .

- **Dynamic Programming:** Use a DP table to build combinations iteratively, but this approach is less intuitive and harder to implement compared to backtracking.

Similar Problems

- **Subsets:** Generate all subsets of a given set.
- **Permutations:** Generate all permutations of a given set of numbers.
- **Combinations Sum:** Find combinations of numbers that add up to a specific target.

Things to Keep in Mind and Tricks

- Prune the recursion tree early if the remaining numbers cannot form a valid combination.
- Use a start parameter to ensure that numbers are added in ascending order, avoiding duplicate combinations.
- Test edge cases like $k = 0$, $n = k$, and $k = 1$ to ensure correctness.

Corner and Special Cases to Test

- $k = 0$: The result should be an empty list.
- $n = k$: The result should contain only one combination $[1, 2, \dots, n]$.
- $k = 1$: The result should contain all individual numbers as combinations.
- Large n and small k : Test scalability.

Conclusion

The **Combinations** problem demonstrates the elegance and efficiency of backtracking for generating subsets. Mastering this technique provides a foundation for solving a wide range of combinatorial problems efficiently and effectively.

Problem 14.6 Subsets (Power Set)

The **Subsets** problem, also known as the power set problem, involves generating all possible subsets of a given set of unique integers. This problem highlights the versatility of recursive backtracking and iterative combinatorial algorithms.

Problem Statement

Given an integer array `nums` of unique elements, return all possible subsets (the power set).

Input: - A list of integers `nums`, where each integer is unique.

Output: - A list of lists, where each inner list represents a subset of `nums`.

Example 1:

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

Output:

[[], [1], [2], [3], [1, 2], [1, 3], [2, 3], [1, 2, 3]]

Example 2:

Input: `nums` = [0]

Output:

[[], [0]]

Algorithmic Approach

There are two primary approaches to solving this problem: backtracking and iterative generation.

Backtracking Approach:

- Start with an empty subset.
- At each step, decide whether to include the current number in the subset.
- Recur for the next number, generating all subsets that include and exclude the current number.
- Once all numbers have been processed, add the subset to the result.

Iterative Approach:

- Begin with an empty subset.
- For each number in the input array, add it to all existing subsets to create new subsets.
- Append these new subsets to the result.

Complexities

- **Time Complexity:** $O(2^n)$, where n is the number of elements in `nums`, as there are 2^n subsets in the power set.
- **Space Complexity:** $O(2^n)$, for storing the subsets in the result list.

Python Implementation (Backtracking)

Below is the implementation using backtracking:

```
def subsets(nums):
    def backtrack(start, current):
        # Add the current subset to the result
        result.append(current[:])

        # Explore all possible subsets by including the next numbers
        for i in range(start, len(nums)):
            current.append(nums[i])
            backtrack(i + 1, current)
            current.pop() # Backtrack to explore other subsets

    result = []
    backtrack(0, [])
    return result

# Example usage:
print(subsets([1, 2, 3])) # Output: [[], [1], [2], [3], [1, 2], [1, 3], [2, 3],
                           ↪ [1, 2, 3]]
```

Python Implementation (Iterative)

Below is the implementation using an iterative approach:

```
def subsets(nums):
    result = [[]]
    for num in nums:
        # Add the current number to each existing subset
        result += [subset + [num] for subset in result]
    return result

# Example usage:
print(subsets([1, 2, 3])) # Output: [[], [1], [2], [3], [1, 2], [1, 3], [2, 3],
                           ↪ [1, 2, 3]]
```

Why These Approaches?

The backtracking approach provides a recursive and intuitive way to generate subsets, making it easier to understand and debug. The iterative approach, on the other hand, is more concise and eliminates the need for recursion, making it suitable for situations where recursion depth may be a concern.

Similar Problems

- **Permutations:** Generate all possible permutations of a given set of numbers.
- **Combinations:** Generate all subsets of a specific size k .
- **Subset Sum:** Find all subsets whose sum equals a given target.

Things to Keep in Mind and Tricks

- Ensure that the input list `nums` does not contain duplicates, as this may lead to duplicate subsets.
- For large input sizes, consider the exponential growth in the number of subsets and the memory required to store them.
- The order of subsets in the result does not matter unless explicitly specified in the problem statement.

Corner and Special Cases to Test

- **Empty Input:** Input: `nums = []`, Output: `[[]]`.
- **Single Element:** Input: `nums = [0]`, Output: `[[], [0]]`.
- **Large Input:** Test with larger arrays to ensure performance and correctness.

Conclusion

The **Subsets** problem provides an excellent opportunity to practice both backtracking and iterative techniques for generating combinatorial structures. Mastering this problem builds a strong foundation for tackling more complex problems in combinatorics and recursion.

Problem 14.7 Subsets with Duplicates

The **Subsets with Duplicates** problem is an extension of the classic power set problem. It requires generating all possible subsets of an integer array `nums` that

may contain duplicate elements, ensuring no duplicate subsets are included in the output.

Problem Statement

Given an integer array `nums` that may contain duplicates, return all possible subsets (the power set) without duplicate subsets.

Input: - A list of integers `nums`, which may contain duplicates.

Output: - A list of lists, where each inner list represents a unique subset of `nums`.

Example 1:

Input: `nums = [1, 2, 2]`

Output:

`[[], [1], [2], [1, 2], [2, 2], [1, 2, 2]]`

Example 2:

Input: `nums = [0]`

Output:

`[[], [0]]`

Algorithmic Approach

To handle duplicates, the problem can be solved using backtracking with an additional step to skip generating subsets for duplicate elements.

Steps:

- Sort the input array to ensure duplicates are adjacent⁵.
- Use a recursive backtracking function to construct subsets incrementally.
- At each step, include the current number in the subset and recurse, or skip it to explore other subsets.
- Skip duplicate elements by checking if the current element is the same as the previous one, and only include it if the previous element was part of the subset.

⁵ Sorting allows us to easily identify and skip duplicates.

Complexities

- Time Complexity:** $O(2^n)$, where n is the length of `nums`, as each element has two possibilities (included or excluded).
- Space Complexity:** $O(n)$, for the recursion stack.

Python Implementation

Below is the Python implementation using backtracking:

```
def subsetsWithDup(nums):
    def backtrack(start, current):
        # Add the current subset to the result
        result.append(current[:])

        # Explore all possible subsets
        for i in range(start, len(nums)):
            # Skip duplicates
            if i > start and nums[i] == nums[i - 1]:
                continue
            current.append(nums[i])
            backtrack(i + 1, current)
            current.pop() # Backtrack to explore other subsets

    nums.sort() # Sort to handle duplicates
    result = []
    backtrack(0, [])
    return result

# Example usage:
print(subsetsWithDup([1, 2, 2])) # Output: [[], [1], [2], [1, 2], [2, 2], [1, 2, 2]]
```

Why This Approach?

Backtracking is an effective way to systematically generate subsets while avoiding duplicates. Sorting ensures that duplicate elements are adjacent, allowing us to skip generating redundant subsets by comparing the current element with the previous one.

Alternative Approaches

- **Iterative Approach:** Use a set to track generated subsets and avoid duplicates. This is less efficient due to the overhead of set operations.
- **Bitmasking:** Generate subsets using binary representation and filter out duplicates. This approach is less intuitive and requires additional logic to handle duplicates.

Similar Problems

- Subsets:** Generate all subsets of a set without duplicates.

- **Permutations with Duplicates:** Generate all unique permutations of a list with duplicates.
- **Combination Sum:** Find all unique combinations that sum to a target value.

Things to Keep in Mind and Tricks

- Always sort the input array to make duplicate detection straightforward.
- Use conditions to skip duplicates only after the first occurrence in a given recursive path.
- Test with edge cases like all elements being the same or all elements being distinct.

Corner and Special Cases to Test

- **Empty Input:** Input: `nums = []`, Output: `[[]]`.
- **Single Element:** Input: `nums = [1]`, Output: `[[], [1]]`.
- **All Duplicates:** Input: `nums = [2, 2, 2]`, Output: `[[], [2], [2, 2], [2, 2, 2]]`.
- **Mixed Duplicates:** Input: `nums = [1, 2, 2, 3]`, Verify correctness.

Conclusion

The **Subsets with Duplicates** problem builds upon the classic subsets problem, emphasizing the need for handling duplicate elements efficiently. Mastering this problem enhances understanding of recursion, backtracking, and combinatorics.

Problem 14.8 Solve Sudoku

The **Solve Sudoku** problem involves completing a partially filled 9×9 Sudoku grid by filling empty cells such that the final grid satisfies Sudoku rules. This problem is an excellent example of constraint satisfaction and backtracking.

Problem Statement

Write a program to solve a given Sudoku puzzle by filling the empty cells. The Sudoku rules are:

- Each row must contain the digits 1 to 9 with no repetitions.
- Each column must contain the digits 1 to 9 with no repetitions.

- Each of the nine 3×3 sub-boxes must contain the digits 1 to 9 with no repetitions.

Empty cells are represented by ' .' .

Input: - A 9×9 grid board, where each cell contains a digit (1 to 9) or ' .' .

Output: - The same grid, modified in place to represent the solved Sudoku.

Example 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", "6", "3", "9"]
]
```

Output: The solved grid is:

```
board = [
    ["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"]
]
```

Algorithmic Approach

The problem is solved using backtracking:

- Identify the next empty cell in the grid.
- Try placing digits 1 through 9 in the empty cell.
- For each digit, check if placing it maintains the validity of the Sudoku rules.

- If valid, recursively attempt to solve the remaining grid.
 - Backtrack if placing a digit does not lead to a solution, removing it and trying the next digit.
 - Stop when all cells are filled and the grid satisfies the Sudoku rules.

Complexities

- **Time Complexity:** Solving Sudoku is an NP-complete problem. In the worst case, the time complexity can approach $O(9^n)$, where n is the number of empty cells.
 - **Space Complexity:** $O(n)$ for the recursion stack, where n is the number of empty cells.

Python Implementation

Below is the Python implementation using backtracking:

```

def solveSudoku(board):
    def is_valid(row, col, num):
        # Check the row
        for i in range(9):
            if board[row][i] == num:
                return False
        # Check the column
        for i in range(9):
            if board[i][col] == num:
                return False
        # Check the 3x3 sub-box
        box_row, box_col = 3 * (row // 3), 3 * (col // 3)
        for i in range(3):
            for j in range(3):
                if board[box_row + i][box_col + j] == num:
                    return False
        return True

    def backtrack():
        for i in range(9):
            for j in range(9):
                if board[i][j] == ".":
                    for num in "123456789":
                        if is_valid(i, j, num):
                            board[i][j] = num
                            if backtrack():
                                return True
                            board[i][j] = "." # Backtrack
        return False

    backtrack()

```

```

    return True

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)
for row in board:
    print(row)

```

Why This Approach?

Backtracking is the most natural approach for constraint satisfaction problems like Sudoku. It systematically explores all possibilities while pruning invalid paths, ensuring correctness and efficiency.

Similar Problems

- **N-Queens Problem:** Place n queens on an $n \times n$ chessboard such that no two queens threaten each other.
- **Magic Square Problem:** Fill a grid such that the sums of rows, columns, and diagonals are equal.

Things to Keep in Mind and Tricks

- Use a helper function to check the validity of placing a digit, reducing redundancy in code.
- Test the solution with partially filled grids of varying difficulty to ensure correctness and performance.
- Optimize by filling cells with fewer possibilities first.

Corner and Special Cases to Test

- **Empty Grid:** Input: A grid with all cells empty. Output: A valid Sudoku grid.
- **Nearly Solved Grid:** Input: A grid with only one cell empty. Output: A completed grid.
- **Invalid Grid:** Input: A grid that violates Sudoku rules. Output: No solution.

Conclusion

The **Solve Sudoku** problem showcases the power of backtracking for solving constraint satisfaction problems. Mastering this technique provides foundational skills for tackling a wide range of combinatorial challenges.

Problem 14.9 Strobogrammatic Number II

The **Strobogrammatic Number II** problem involves finding all strobogrammatic numbers of a given length. A strobogrammatic number appears the same when rotated 180° (e.g., 69, 88, 96).

Problem Statement

Given an integer n , return all strobogrammatic numbers of length n .

Input: - An integer n , where $n \geq 1$.

Output: - A list of strings representing all strobogrammatic numbers of length n .

Example 1:

Input: $n = 2$

Output:

[”11”, ”69”, ”88”, ”96”]

Example 2:

Input: $n = 1$

Output:

[”0”, ”1”, ”8”]

Algorithmic Approach

This problem can be solved using recursion and backtracking by generating numbers character by character, ensuring they are strobogrammatic.

Steps:

- Define the valid digit pairs for strobogrammatic numbers:

```
{("0", "0"), ("1", "1"), ("6", "9"), ("8", "8"), ("9", "6")}.
```

- Use a recursive function to build numbers from the outermost digits inward.
- For odd-length numbers, include the single middle digit (0, 1, 8).
- Skip leading zeros for $n > 1$ to ensure valid numbers.
- Add the digit pairs symmetrically until the length of the number reaches n .

Complexities

- **Time Complexity:** $O(5^{n/2})$, since each recursive call chooses from 5 pairs of digits and there are $n/2$ levels of recursion.
- **Space Complexity:** $O(n)$, for the recursion stack.

Python Implementation

Below is the Python implementation using backtracking:

```
def findStrobogrammatic(n):
    def backtrack(low, high, current):
        # Base case: If the current number reaches the desired length
        if low > high:
            result.append("".join(current))
            return

        for pair in strobogrammatic_pairs:
            # Avoid leading zeros for numbers with more than one digit
            if low == high and pair[0] != pair[1]:
                continue
            if low == 0 and n > 1 and pair[0] == "0":
                continue

            # Place the strobogrammatic pair
            current[low], current[high] = pair[0], pair[1]
            backtrack(low + 1, high - 1, current)

    # Valid strobogrammatic digit pairs
    strobogrammatic_pairs = [("", ""), ("0", "0"), ("1", "1"),
                            ("6", "9"), ("8", "8"), ("9", "6")]
    result = []
    backtrack(0, n - 1, list("0" * n))
    return result
```

```

strobogrammatic_pairs = [
    ("0", "0"), ("1", "1"), ("6", "9"), ("8", "8"), ("9", "6")
]
result = []
backtrack(0, n - 1, [""] * n)
return result

# Example usage:
print(findStrobogrammatic(2))  # Output: ["11", "69", "88", "96"]
print(findStrobogrammatic(1))  # Output: ["0", "1", "8"]

```

Why This Approach?

The recursive backtracking approach ensures that all valid strobogrammatic numbers are generated while avoiding invalid configurations. The symmetry of the number is maintained by placing digit pairs simultaneously at the beginning and the end.

Alternative Approaches

- **Iterative Approach:** Use a queue to generate strobogrammatic numbers layer by layer. This approach is less intuitive than recursion.
- **Dynamic Programming:** Construct strobogrammatic numbers iteratively for smaller lengths and use them to build larger numbers. However, this adds unnecessary complexity.

Similar Problems

- Strobogrammatic Number:** Determine if a given number is strobogrammatic.
- Palindrome Number:** Check if a number reads the same forward and backward.
- Valid Palindrome II:** Determine if a string can become a palindrome by removing at most one character.

Things to Keep in Mind and Tricks

- Avoid leading zeros for numbers of length greater than 1.
- For odd-length numbers, ensure that the middle digit is valid (0, 1, 8).
- Use a helper function to manage symmetry and recursion effectively.

Corner and Special Cases to Test

- **Single Digit:** $n = 1$, Output: `["0", "1", "8"]`.
- **Small Even Length:** $n = 2$, Output: `["11", "69", "88", "96"]`.
- **Large n :** Test with larger values of n to ensure scalability and correctness.

Conclusion

The **Strobogrammatic Number II** problem exemplifies recursive backtracking for generating structured outputs. Mastering this problem enhances understanding of symmetry, constraints, and efficient combinatorial generation techniques.

Problem 14.10 Word Search

The **Word Search** problem involves determining whether a given word exists in a 2D grid of characters. 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.

A classic grid traversal problem that elegantly combines DFS with backtracking. Often appears in technical interviews at top tech companies.

Problem Statement

Given an ' $m \times n$ ' grid of characters 'board' and a string 'word', return 'true' if 'word' exists in the grid. Otherwise, return 'false'.

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.

Examples:

- **Example 1:**

```
Input: board = [
    ['A','B','C','E'],
    ['S','F','C','S'],
    ['A','D','E','E']
], word = "ABCED"
Output: true
```

Explanation: The word "ABCED" exists in the grid following the path A → B → C → C → E → D.

- **Example 2:**

```
Input: board = [
    ['A','B','C','E'],
    ['S','F','C','S'],
    ['A','D','E','E']
], word = "SEE"
Output: true
```

Explanation: The word "SEE" exists in the grid following the path S → E → E.

- **Example 3:**

```
Input: board = [
    ['A','B','C','E'],
    ['S','F','C','S'],
    ['A','D','E','E']
], word = "ABCB"
Output: false
```

Explanation: The word "ABCB" does not exist in the grid because the same cell cannot be used more than once.

LeetCode link: Word Search

[LeetCode Link]
[GeeksForGeeks Link]
[HackerRank Link]
[CodeSignal Link]
[InterviewBit Link]
[Educative Link]
[Codewars Link]

Key Insights

Before diving into the solution, understanding these key insights will help develop an optimal approach:

- We only need to check cells that match the first character of our target word
- Once we use a cell, we can't reuse it in the same path
- We can use the board itself to mark visited cells, saving space
- Early termination is crucial for performance

Solution Strategy

The optimal solution combines **Depth-First Search (DFS)** with **backtracking**:

1. Find all potential starting points (cells matching word[0])
2. For each starting point:
 - Explore adjacent cells recursively using DFS

- Mark visited cells to avoid reuse
- Backtrack when a path fails

3. Return true if any path succeeds

The key to efficient backtracking is marking and unmarking cells in-place, avoiding extra space for visited sets.

Python Implementation

```
from typing import List

def exist(board: List[List[str]], word: str) -> bool:
    if not board or not board[0] or not word:
        return False

    ROWS, COLS = len(board), len(board[0])

    def dfs(row: int, col: int, word_index: int) -> bool:
        if word_index == len(word):
            return True
        if (row < 0 or row >= ROWS or
            col < 0 or col >= COLS or
            board[row][col] != word[word_index]):
            return False

        original_char = board[row][col]
        board[row][col] = '#'

        for dx, dy in [(0, 1), (1, 0), (0, -1), (-1, 0)]:
            if dfs(row + dx, col + dy, word_index + 1):
                return True

        board[row][col] = original_char
        return False

    return any(
        dfs(row, col, 0)
        for row in range(ROWS)
        for col in range(COLS)
        if board[row][col] == word[0]
    )

# Comprehensive test cases
def test_word_search():
    assert exist([["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], "ABCED") == True
    assert exist([["A", "B", "C", "E"], ["S", "F", "C", "S"], ["A", "D", "E", "E"]], "SEE") == True
    assert exist([["A", "B", "C", "E"],
```

```

        ["S", "F", "C", "S"],  

        ["A", "D", "E", "E"]], "ABCB") == False  

assert exist([], "A") == False  

assert exist([["A"]], "A") == True  

assert exist([["A"]], "") == False
    
```

Implementation Details

- **Early Termination:** We check for empty inputs immediately
- **Space Optimization:** Using '#' as a visited marker avoids extra space
- **Direction Array:** Using (dx, dy) pairs makes direction handling cleaner
- **Pythonic Features:** Using 'any' for concise iteration over starting points

Common Mistakes to Avoid

- **Forgetting to Backtrack:** Always restore cells to their original state
- **Unnecessary Checking:** Only start DFS from promising cells
- **Extra Space:** Avoid creating visited sets/arrays
- **Missing Edge Cases:** Handle empty boards and words properly

Interview Tips

- Start by explaining the DFS + backtracking approach
- Mention space optimization using in-place marking
- Discuss time complexity: $O(N \times M \times 4^L)$ where L is word length
- Consider mentioning potential optimizations:
 - Pre-checking if all required characters exist
 - Starting from less frequent characters
 - Using trie for multiple word search

Complexities

- **Time Complexity:** $O(m \times n \times 4^k)$, where m is the number of rows, n is the number of columns in the grid, and k is the length of the word. This accounts for exploring all possible paths from each cell.

- **Space Complexity:** $O(k)$, where k is the length of the word. This space is used by the recursion stack during the DFS.

Similar Problems to This One

There are several other problems that involve searching for patterns or sequences within grids or strings, such as:

- Word Search II
- Minimum Window Substring
- Longest Repeating Character Replacement
- Palindrome Partitioning
- Number of Islands

Things to Keep in Mind and Tricks

- **Depth-First Search (DFS):** Utilize DFS to explore all possible paths from a given cell efficiently.
- **Backtracking:** Implement backtracking to revert changes (like marking cells as visited) when a path does not lead to a solution.
- **Boundary Checks:** Always ensure that your indices do not go out of bounds to prevent runtime errors.
- **Early Termination:** Return immediately when a valid path is found to optimize performance.
- **Handling Visited Cells:** Mark cells as visited during the search and ensure they are unmarked during backtracking.
- **Optimizing Search Order:** Explore directions in an order that is likely to lead to a solution faster, such as prioritizing directions based on word patterns.

Corner and Special Cases to Test When Writing the Code

When implementing the ‘exist’ function, it is crucial to test the following edge cases to ensure robustness:

- **Empty Grid:** ‘board = []’, ‘word = ”a”‘ should return ‘False‘.
- **Single Cell Grid:** ‘board = [[‘A’]]’, ‘word = ”A”‘ should return ‘True‘.

- **Single Cell Grid with Mismatch:** ‘board = [[‘A’]]’, ‘word = ”B”“ should return ‘False’.
- **Word Longer Than Grid Cells:** ‘board = [[‘A’,‘B’],[‘C’,‘D’]]’, ‘word = ”ABCDE”“ should return ‘False’.
- **All Characters the Same:** ‘board = [[‘A’,‘A’,‘A’],[‘A’,‘A’,‘A’],[‘A’,‘A’,‘A’]]’, ‘word = ”AAAAA”“ should return ‘True’.
- **Word Not Present:** ‘board = [[‘A’,‘B’,‘C’],[‘D’,‘E’,‘F’],[‘G’,‘H’,‘I’]]’, ‘word = ”XYZ”“ should return ‘False’.
- **Word with Repeating Characters:** ‘board = [[‘A’,‘A’,‘A’],[‘A’,‘B’,‘A’],[‘A’,‘A’,‘A’]]’, ‘word = ”ABA”“ should return ‘True’.
- **Multiple Valid Paths:** ‘board = [[‘A’,‘B’,‘C’,‘E’],[‘S’,‘F’,‘C’,‘S’],[‘A’,‘D’,‘E’,‘E’]]’, ‘word = ”ABCCED”“ should return ‘True’.
- **Large Grid with Valid Path:** A large grid where the word exists multiple times.
- **Large Grid with No Valid Path:** A large grid where the word does not exist.

Problem 14.11 N-Queens Problem

Problem Statement

Place N chess queens on an $N \times N$ chessboard so that no two queens threaten each other. A solution requires that no two queens share the same row, column, or diagonal.

A classic backtracking problem that tests understanding of constraint satisfaction and recursive problem-solving. Frequently appears in interviews at top tech companies.

Key Insights

Understanding these insights is crucial for developing an optimal solution:

- Each row must contain exactly one queen
- Each column must contain exactly one queen
- Two queens can’t share any diagonal
- We can place queens row by row to reduce the search space
- Diagonals can be represented by their row \pm column values

Solution Strategy

The optimal approach uses **backtracking** with the following strategy:

1. Place queens row by row, trying each column
2. Use sets to track occupied columns and diagonals
3. Backtrack when a placement violates constraints
4. Record valid solutions when all queens are placed

Using sets for constraint checking provides O(1) lookup time, significantly improving performance over array-based approaches.

Python Implementation

```
class Solution:
    def solveNQueens(self, n: int) -> List[List[str]]:
        def create_board() -> List[str]:
            return [''.join('Q' if j == col else '.' for j in range(n)) for col in state]

        def is_safe(row: int, col: int) -> bool:
            return (col not in cols and
                    row + col not in pos_diag and
                    row - col not in neg_diag)

        def backtrack(row: int) -> None:
            if row == n: # Found a valid solution
                solutions.append(create_board())
                return

            for col in range(n):
                if is_safe(row, col):
                    # Place queen and update constraints
                    cols.add(col)
                    pos_diag.add(row + col)
                    neg_diag.add(row - col)
                    state.append(col)

                    backtrack(row + 1)

                    # Backtrack: remove queen and constraints
                    cols.remove(col)
                    pos_diag.remove(row + col)
                    neg_diag.remove(row - col)
                    state.pop()

        solutions: List[List[str]] = []
        state: List[int] = []

        return solutions
```

```

cols: set[int] = set()
pos_diag: set[int] = set() # row + col
neg_diag: set[int] = set() # row - col

backtrack(0)
return solutions

# Comprehensive test cases
def test_n_queens():
    solution = Solution()

    # Edge cases
    assert len(solution.solveNQueens(1)) == 1 # Single queen
    assert len(solution.solveNQueens(2)) == 0 # Impossible
    assert len(solution.solveNQueens(3)) == 0 # Impossible

    # Standard cases
    assert len(solution.solveNQueens(4)) == 2
    assert len(solution.solveNQueens(8)) == 92

```

Implementation Details

- **State Management:**
 - state: Records queen positions (column) for each row
 - cols: Tracks occupied columns
 - pos_diag: Tracks occupied positive diagonals (row + col)
 - neg_diag: Tracks occupied negative diagonals (row - col)
- **Constraint Checking:** O(1) using sets
- **Board Creation:** Efficient string manipulation
- **Type Hints:** Added for better code clarity

Complexity Analysis

- **Time Complexity:** O(N!) - we try N positions for first queen, N-1 for second, etc.
- **Space Complexity:** O(N) for recursion stack and sets

Common Mistakes

- Forgetting to backtrack (remove constraints)

- Incorrect diagonal calculations
- Using inefficient constraint checking methods
- Not handling edge cases ($N=1, N=2, N=3$)

Interview Tips

- Start by explaining the constraints and how to check them efficiently
- Mention that placing queens row by row reduces the search space
- Discuss how sets provide $O(1)$ lookup for constraint checking
- Consider mentioning optimizations:
 - Using bit manipulation for even faster constraint checking
 - Leveraging board symmetry to reduce solutions search
 - Pre-computing diagonal indices

Related Problems

- Sudoku Solver
- Permutations
- Combination Sum
- Path with Constraints

Problem 14.12 Combination Sum

Problem Statement

A fundamental backtracking problem that tests understanding of recursive exploration and pruning techniques. Frequently appears in technical interviews.

Given an array of distinct positive integers `candidates` and a target integer `target`, find all unique combinations of numbers from `candidates` that sum to `target`. Each number can be used unlimited times.

Key Requirements:

- Numbers can be reused unlimited times
- Each combination must be unique
- All integers are positive
- The order within combinations doesn't matter

Examples:

```
# Example 1
candidates = [2,3,6,7]
target = 7
output = [[2,2,3],[7]]

# Example 2
candidates = [2,3,5]
target = 8
output = [[2,2,2,2],[2,3,3],[3,5]]
```

Intuition

The key insight is that at each step, we have two choices:

- Include the current number again (since reuse is allowed)
- Move to the next number

Solution Strategy

1. Sort candidates for efficient pruning
2. Use backtracking to explore combinations:
 - Track current sum and combination
 - Stop when sum equals target (solution found)
 - Stop when sum exceeds target (pruning)
 - Try including current number again
 - Try moving to next number

Sorting the candidates enables early pruning when the sum exceeds the target, significantly improving performance.

Optimal Implementation

```
from typing import List

class Solution:
    def combinationSum(self, candidates: List[int], target: int) -> List[List[int]]:
        ↵ ]
        # Handle edge cases
        if not candidates or min(candidates) > target:
            return []

        # Sort for efficient pruning
```

```

candidates.sort()

def backtrack(start: int, remaining: int, path: List[int]) -> None:
    if remaining == 0:
        result.append(path[:])  # Found valid combination
        return

    for i in range(start, len(candidates)):
        num = candidates[i]
        if num > remaining:  # Pruning: stop if exceeding target
            break

        path.append(num)
        # Try same number again (i) since reuse is allowed
        backtrack(i, remaining - num, path)
        path.pop()  # Backtrack

result: List[List[int]] = []
backtrack(0, target, [])
return result

# Comprehensive test cases
def test_combination_sum():
    solution = Solution()

    # Basic cases
    assert solution.combinationSum([2,3,6,7], 7) == [[2,2,3],[7]]
    assert solution.combinationSum([2,3,5], 8) == [[2,2,2,2],[2,3,3],[3,5]]

    # Edge cases
    assert solution.combinationSum([], 7) == []
    assert solution.combinationSum([5], 3) == []
    assert solution.combinationSum([1], 1) == [[1]]
    assert solution.combinationSum([1], 0) == []

```

Complexity Analysis

- **Time:** $O(N^{(T/M)})$ where:
 - N = length of candidates
 - T = target value
 - M = minimum value in candidates
- **Space:** $O(T/M)$ for recursion stack

Common Pitfalls

- **Missing Edge Cases:** Empty array, no solution possible
- **Incorrect Path Copying:** Using path instead of path[:]
- **Inefficient Pruning:** Not sorting candidates first
- **Memory Issues:** Not backtracking properly

Interview Tips

- Start by explaining the backtracking approach
- Mention optimization techniques (sorting, pruning)
- Discuss time/space complexity trade-offs
- Consider follow-up questions:
 - What if negative numbers were allowed?
 - What if each number could be used only once?
 - How to handle duplicates in candidates?

Similar Problems

- Combination Sum II (no reuse allowed)
- Combination Sum III (fixed combination size)
- Subset Sum
- Target Sum

Part III

Advanced Topics

Chapter 15

Dynamic Programming

Dynamic Programming (DP) is a powerful algorithmic technique used to solve optimization problems by breaking them into smaller overlapping subproblems. It is widely used in computer science for its ability to reduce computation time through efficient storage and reuse of intermediate results. This chapter delves into the origins, principles, and applications of dynamic programming, illustrating its profound impact on problem-solving.

History and Background

Dynamic programming was first formalized by [**Richard Bellman**](#) in the 1950s while working on mathematical optimization problems. Bellman coined the term "dynamic programming" not for its computational implications but as a way to convey a sense of systematic planning and optimization. He sought to avoid any association with "mathematical programming," a term that carried bureaucratic baggage at the time.

Bellman initially used dynamic programming to solve problems in decision processes and control theory. The methodology quickly found applications in economics, operations research, and eventually computer science, where it became a cornerstone of algorithm design.

Key Milestones:

- 1950s: Richard Bellman introduces dynamic programming for sequential decision-making problems.
- 1970s: DP becomes a staple in optimization theory, finding applications in shortest path algorithms like Dijkstra's and Floyd-Warshall.
- 1980s: Advances in DP lead to efficient solutions for knapsack problems, longest

common subsequence (LCS), and matrix chain multiplication.

- 2000s: DP applications expand to bioinformatics (sequence alignment) and artificial intelligence (reinforcement learning).

What is Dynamic Programming?

Dynamic programming is a method for solving problems by breaking them into smaller subproblems, solving each subproblem once, and storing the results to avoid redundant computations. This approach is particularly effective for problems with overlapping subproblems and optimal substructure.

Key Features:

- **Overlapping Subproblems:** The problem can be divided into smaller, reusable subproblems.
- **Optimal Substructure:** The solution to the problem can be composed of the solutions to its subproblems.
- **Memoization or Tabulation:** Intermediate results are stored to avoid recompilation.

Dynamic programming differs from divide-and-conquer algorithms (e.g., Merge Sort, Quick Sort) because it reuses solutions to subproblems rather than solving them independently.

Memoization vs. Tabulation

Dynamic programming can be implemented in two main ways:

Memoization (Top-Down Approach)

In memoization, the problem is solved recursively, storing the results of subproblems in a data structure (usually an array or hash map) so they can be reused. This approach leverages the natural recursive structure of the problem.

Example: Fibonacci Sequence (Memoization):

```
def fib(n, memo={}):
    if n <= 1:
        return n
    if n not in memo:
        memo[n] = fib(n - 1, memo) + fib(n - 2, memo)
    return memo[n]
```

Tabulation (Bottom-Up Approach)

In tabulation, the problem is solved iteratively, building up solutions to subproblems from the smallest to the largest. This approach avoids recursion and stack overhead.

Example: Fibonacci Sequence (Tabulation):

```
def fib(n):
    if n <= 1:
        return n
    dp = [0] * (n + 1)
    dp[1] = 1
    for i in range(2, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]
```

Steps for Solving Problems Using Dynamic Programming

Dynamic programming involves the following systematic steps:

1. **Characterize the Problem:** Determine whether the problem has overlapping subproblems and optimal substructure.
2. **Define the State:** Identify what each subproblem represents and define a state variable to represent it.
3. **State Transition Relation:** Derive the relationship (recurrence relation) between the states.
4. **Base Case:** Identify the simplest subproblems and their solutions.
5. **Implementation:** Choose between memoization or tabulation, and implement the solution efficiently.

Applications of Dynamic Programming

Dynamic programming has applications across a wide range of domains. Below are some prominent categories:

1. Sequence Alignment and Comparison

Dynamic programming is extensively used in bioinformatics for sequence alignment problems, such as:

- **Longest Common Subsequence (LCS):** Finding the longest subsequence common to two strings.
- **Edit Distance (Levenshtein Distance):** Calculating the minimum number of edits to transform one string into another.

2. Optimization Problems

- **Knapsack Problem:** Maximizing value within a weight constraint.
- **Matrix Chain Multiplication:** Optimizing the order of matrix multiplications.

3. Graph Algorithms

Dynamic programming is central to shortest path algorithms:

- **Bellman-Ford Algorithm:** Finds the shortest path in graphs with negative weights.
- **Floyd-Warshall Algorithm:** Solves all-pairs shortest path problems.

4. Game Theory and AI

Dynamic programming is used in reinforcement learning, game theory, and decision-making problems. For instance:

- **Minimax Algorithm with DP:** Solves games like tic-tac-toe or chess.
- **Markov Decision Processes (MDPs):** Used in AI for sequential decision-making.

5. Miscellaneous Applications

- **Rod Cutting Problem:** Maximizing profit by cutting a rod into pieces.
- **Palindrome Problems:** Finding the longest palindromic substring or subsequence.

Advantages and Challenges of Dynamic Programming

Advantages

- Significantly reduces time complexity for problems with overlapping subproblems.

- Provides an elegant and systematic approach to solving optimization problems.
- Leads to reusable and efficient solutions for a wide range of applications.

Challenges

- Requires identifying overlapping subproblems and optimal substructure, which is non-trivial for complex problems.
- Can lead to high space complexity if intermediate results are not stored efficiently.
- Debugging and deriving state transitions can be difficult for beginners.

Common Problems to Practice Dynamic Programming

- **Fibonacci Sequence:** Compute the n th Fibonacci number.
- **Longest Increasing Subsequence (LIS):** Find the longest subsequence of a sequence where elements are strictly increasing.
- **House Robber Problem:** Maximize money robbed without alerting security.
- **Climbing Stairs:** Calculate the number of ways to climb a staircase with n steps.
- **Coin Change:** Find the minimum number of coins needed to make a given amount.

Conclusion

Dynamic programming is a cornerstone of algorithmic problem-solving, offering efficient solutions to complex problems through reuse and optimization. By mastering its principles, practitioners can tackle a vast array of challenges, from sequence alignment in bioinformatics to decision-making in AI. While challenging to master, the elegance and efficiency of dynamic programming make it an indispensable tool for any programmer.

Problem 15.1 Climbing Stairs

The **Climbing Stairs** problem is a classic dynamic programming challenge that tests the ability to solve problems with overlapping subproblems and optimal substructure. This problem is often used to introduce beginners to dynamic programming concepts due to its simplicity and elegance.

Problem Statement

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?

Input: - An integer n , where $n \geq 1$.

Output: - An integer representing the number of distinct ways to climb the staircase.

Example 1:

Input: $n = 2$

Output: 2

Explanation: Two ways to climb the staircase:

1. Take two 1-step climbs.
2. Take one 2-step climb.

Example 2:

Input: $n = 3$

Output: 3

Explanation: Three ways to climb the staircase:

1. Take three 1-step climbs.
2. Take one 1-step climb followed by one 2-step climb.
3. Take one 2-step climb followed by one 1-step climb.

Algorithmic Approach

The problem can be solved using dynamic programming by recognizing that the number of ways to climb n steps can be derived from the ways to climb $n - 1$ steps and $n - 2$ steps:

$$f(n) = f(n - 1) + f(n - 2)$$

Steps:

1. Define a state $f(n)$, where $f(n)$ represents the number of ways to climb n steps.
2. The base cases are:

$$f(1) = 1, \quad f(2) = 2$$

3. Use the recurrence relation to compute $f(n)$ for $n > 2$.

Complexities

- **Time Complexity:** $O(n)$, since we compute $f(n)$ in a single loop.
- **Space Complexity:**
 - $O(n)$ if a DP array is used.
 - $O(1)$ if only two variables are maintained for $f(n - 1)$ and $f(n - 2)$.

Python Implementation

Below are two implementations: one using a DP array and another with optimized space complexity.

Using a DP Array

```
def climbStairs(n):
    if n == 1:
        return 1
    dp = [0] * (n + 1)
    dp[1], dp[2] = 1, 2
    for i in range(3, n + 1):
        dp[i] = dp[i - 1] + dp[i - 2]
    return dp[n]

# Example usage:
print(climbStairs(3))  # Output: 3
```

Optimized Space Complexity

```
def climbStairs(n):
    if n == 1:
        return 1
    first, second = 1, 2
    for i in range(3, n + 1):
        first, second = second, first + second
    return second

# Example usage:
print(climbStairs(3))  # Output: 3
```

Why This Approach?

This approach efficiently solves the problem using a recurrence relation, leveraging the overlapping subproblems and optimal substructure properties inherent in dy-

namic programming. The optimized solution reduces space complexity while maintaining clarity and correctness.

Alternative Approaches

- **Recursive Solution with Memoization:** Solve the problem recursively, storing results of subproblems in a hash map to avoid redundant calculations.
- **Matrix Exponentiation:** Use matrix exponentiation to compute $f(n)$ in $O(\log n)$ time. This method is less intuitive but useful for advanced optimization.

Similar Problems

- **Fibonacci Sequence:** The problem is analogous to computing the Fibonacci sequence, where $f(n) = f(n - 1) + f(n - 2)$.
- **House Robber Problem:** Maximize the sum of non-adjacent elements in an array, which also involves a similar recurrence relation.
- **Minimum Cost Climbing Stairs:** Minimize the cost to climb to the top of a staircase, extending this problem with an additional cost parameter.

Corner and Special Cases to Test

- $n = 1$: Single step, the output should be 1.
- $n = 2$: Two steps, the output should be 2.
- Large n : Test the performance and correctness for large inputs ($n > 10^4$).

Conclusion

The Climbing Stairs problem is a classic introduction to dynamic programming. Its simplicity and clear recursive structure make it an ideal starting point for understanding DP concepts. By exploring both standard and optimized solutions, one gains insights into the versatility of dynamic programming techniques.

Problem 15.2 Coin Change

The **Coin Change** problem is a classic dynamic programming challenge that tests your ability to solve optimization problems involving minimum cost or re-

sources. This problem is widely used to introduce the concepts of state definition, state transitions, and the use of a base case.

Problem Statement

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 it is not possible to make up the amount, return `-1`.

Input: - `coins`: A list of integers representing coin denominations. - `amount`: An integer representing the total amount.

Output: - An integer representing the minimum number of coins required, or `-1` if it is not possible.

Example 1:

```
Input: coins = [1, 2, 5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1 (minimum 3 coins)
```

Example 2:

```
Input: coins = [2], amount = 3
Output: -1
Explanation: It is not possible to make the amount with the given coins.
```

Algorithmic Approach

This problem can be solved using dynamic programming by defining a state and finding an optimal solution iteratively:

Steps: 1. Define a DP array `dp` of size `amount + 1`, where `dp[i]` represents the minimum number of coins needed to make up the amount i . 2. Initialize `dp[0] = 0`, since no coins are needed to make up the amount 0. 3. For all other amounts, initialize `dp[i]` to infinity (`float('inf')`) to represent that it is initially not possible to form that amount. 4. Iterate over each amount from 1 to `amount`, and for each amount, iterate over each coin denomination to update `dp[i]` using:

$$dp[i] = \min(dp[i], dp[i - \text{coin}] + 1)$$

where `coin` is a valid denomination that does not exceed i . 5. Return `dp[amount]` if it is finite; otherwise, return `-1`.

Complexities

- **Time Complexity:** $O(\text{amount} \times \text{len(coins)})$, where len(coins) is the number of coin denominations.
- **Space Complexity:** $O(\text{amount})$, due to the DP array.

Python Implementation

Below is the Python implementation using the dynamic programming approach:

```
def coinChange(coins, amount):
    # Initialize the DP array
    dp = [float('inf')] * (amount + 1)
    dp[0] = 0 # Base case: no coins needed for amount 0

    # Fill the DP array
    for i in range(1, amount + 1):
        for coin in coins:
            if i - coin >= 0: # Ensure the coin is valid for the current amount
                dp[i] = min(dp[i], dp[i - coin] + 1)

    return dp[amount] if dp[amount] != float('inf') else -1

# Example usage:
coins = [1, 2, 5]
amount = 11
print(coinChange(coins, amount)) # Output: 3
```

Why This Approach?

This approach efficiently solves the problem by building solutions for smaller amounts and using them to derive solutions for larger amounts. It avoids recomputation by storing intermediate results, making it optimal for this type of problem.

Alternative Approaches

- **Recursive Approach with Memoization:** Use a recursive function to compute the minimum coins for each amount, storing results in a hash map to avoid redundant calculations. This approach is less efficient due to the recursive overhead.
- **Breadth-First Search (BFS):** Treat the problem as a shortest path search in an unweighted graph, where nodes represent amounts and edges represent coin

denominations. While conceptually interesting, it can be slower than the DP approach for larger inputs.

Similar Problems

- ****Minimum Coin Change (Unlimited Coins):**** A variation where you must find the number of ways to make a given amount using unlimited coins.
- ****Knapsack Problem:**** Involves optimizing the selection of items with weight and value constraints, similar to coin change but with added complexity.
- ****Climbing Stairs:**** Similar dynamic programming structure but with different state transitions.

Corner and Special Cases to Test

- `coins = [1], amount = 100`: Test with only one denomination.
- `coins = [2], amount = 3`: Test when it is impossible to form the amount.
- Large amounts: Test performance for `amount > 104` with multiple denominations.
- Empty coins array: Verify behavior when no denominations are provided (`coins = []`).

Conclusion

The Coin Change problem exemplifies the power of dynamic programming in solving optimization problems. By carefully defining the state and transition relation, it demonstrates how intermediate results can be reused to build efficient solutions for complex problems. Mastery of this problem equips you with foundational skills to tackle a wide array of similar challenges.

Problem 15.3 House Robber

The **House Robber** problem is a classic dynamic programming challenge that requires solving an optimization problem under specific constraints. The task demonstrates how to make decisions at each step to maximize the overall result while adhering to the given restrictions.

Problem Statement

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, and the only constraint stopping you from robbing all of them is that adjacent houses have security systems connected. If two adjacent houses are robbed, the alarm will be triggered.

Given an integer array `nums` representing the amount of money stashed in each house, return the maximum amount of money you can rob tonight without alerting the police.

Input: - `nums`: An array of integers where `nums[i]` represents the amount of money in the i -th house.

Output: - An integer representing the maximum money you can rob without triggering the alarm.

Example 1:

Input: `nums = [2, 7, 9, 3, 1]`

Output: 12

Explanation: Rob house 1 (money = 2), skip house 2, rob house 3 (money = 9), and rob house 5 (money = 1). Total = $2 + 9 + 1 = 12$.

Example 2:

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

Output: 4

Explanation: Rob house 1 (money = 1) and house 3 (money = 3). Total = $1 + 3 = 4$.

Constraints: $1 \leq \text{nums.length} \leq 10^4$ $0 \leq \text{nums}[i] \leq 10^4$

Algorithmic Approach

The problem can be solved using dynamic programming by recognizing that at each house, you have two choices: rob it or skip it.

Key Insight: The maximum money you can rob from the first i houses depends on whether you rob the i -th house:

$$dp[i] = \max(dp[i - 1], dp[i - 2] + \text{nums}[i])$$

- $dp[i - 1]$: Maximum money if you skip the i -th house. - $dp[i - 2] + \text{nums}[i]$: Maximum money if you rob the i -th house (thus skipping the $(i - 1)$ -th house).

Steps: 1. Define a DP array `dp`, where `dp[i]` represents the maximum money that can be robbed from the first i houses. 2. Initialize base cases:

$$dp[0] = \text{nums}[0], \quad dp[1] = \max(\text{nums}[0], \text{nums}[1])$$

3. Iterate through the array and compute $dp[i]$ using the recurrence relation. 4. Return $dp[nums.length - 1]$, which contains the maximum amount of money.

Complexities

- **Time Complexity:** $O(n)$, where n is the length of `nums`.
- **Space Complexity:**
 - $O(n)$ if a DP array is used.
 - $O(1)$ if only two variables are maintained for the previous two states.

Python Implementation

Below are two implementations: one using a DP array and another with optimized space complexity.

Using a DP Array

```
def rob(nums):
    if len(nums) == 1:
        return nums[0]
    dp = [0] * len(nums)
    dp[0], dp[1] = nums[0], 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]

# Example usage:
nums = [2, 7, 9, 3, 1]
print(rob(nums)) # Output: 12
```

Optimized Space Complexity

```
def rob(nums):
    if len(nums) == 1:
        return nums[0]
    prev1, prev2 = nums[0], max(nums[0], nums[1])
    for i in range(2, len(nums)):
        prev1, prev2 = prev2, max(prev2, prev1 + nums[i])
    return prev2

# Example usage:
nums = [2, 7, 9, 3, 1]
print(rob(nums)) # Output: 12
```

Why This Approach?

This approach efficiently solves the problem by leveraging overlapping subproblems and optimal substructure. The optimized solution reduces space complexity while maintaining clarity and correctness.

Alternative Approaches

- **Recursive Approach with Memoization:** Solve the problem recursively, storing results of subproblems in a hash map to avoid redundant calculations. This approach is conceptually similar but less efficient due to recursive overhead.
- **Tree Representation:** Visualize the problem as a binary tree of choices at each house. While useful for understanding, it is impractical for large inputs.

Similar Problems

- **House Robber II:** Circular street variation where the first and last houses are adjacent.
- **Knapsack Problem:** Optimization problem involving weights and values, similar in logic.
- **Maximum Sum of Non-Adjacent Elements:** A generalized version of the House Robber problem.

Corner and Special Cases to Test

- `nums = [0]`: Single house with no money.
- `nums = [10, 20]`: Two houses, test maximum selection.
- Large `nums`: Ensure the solution scales for `nums.length > 104`.

Conclusion

The House Robber problem is an excellent introduction to dynamic programming and optimization under constraints. It demonstrates how to make decisions at each step while adhering to problem-specific limitations. Mastery of this problem provides a foundation for tackling more complex DP problems.

Problem 15.4 Longest Increasing Subsequence

The **Longest Increasing Subsequence** (LIS) problem is a foundational challenge in dynamic programming and optimization. It involves identifying the longest subsequence of an array in which all elements are in strictly increasing order. This problem is significant due to its applicability in real-world scenarios, such as scheduling, data analysis, and sequence prediction.

Problem Statement

Given an integer array `nums`, return the length of the longest strictly increasing subsequence.

Input: - `nums`: An array of integers.

Output: - An integer representing the length of the LIS.

Example 1:

Input: `nums = [10, 9, 2, 5, 3, 7, 101, 18]`

Output: 4

Explanation: The LIS is `[2, 3, 7, 101]`, and its length is 4.

Example 2:

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

Output: 4

Explanation: The LIS is `[0, 1, 2, 3]`, and its length is 4.

Example 3:

Input: `nums = [7, 7, 7, 7, 7, 7, 7]`

Output: 1

Explanation: The LIS is `[7]`, and its length is 1.

Algorithmic Approach

The problem can be approached using: 1. A **Dynamic Programming (DP)** approach with $O(n^2)$ complexity. 2. An optimized approach using **Binary Search**, reducing complexity to $O(n \log n)$.

Dynamic Programming Approach ($O(n^2)$)

Key Idea: Use a DP array $dp[i]$, where $dp[i]$ represents the length of the longest increasing subsequence ending at index i . For each element $nums[i]$, compare it with all previous elements $nums[j]$ ($j < i$):

$$dp[i] = \max(dp[i], dp[j] + 1) \text{ if } nums[j] < nums[i]$$

Steps: 1. Initialize $dp[i] = 1$ for all i , since the minimum LIS at any index is the element itself. 2. Iterate through the array, updating $dp[i]$ for each i . 3. The result is the maximum value in the dp array.

Binary Search Approach ($O(n \log n)$)

Key Idea: Maintain an array sub , where $sub[i]$ is the smallest possible value that can end a subsequence of length $i + 1$. For each element in $nums$, use binary search to find its position in sub : - If the element is greater than all elements in sub , append it. - Otherwise, replace the first element in sub that is greater than or equal to the current element.

Why This Works: Although sub does not store the actual subsequences, its length at the end of processing equals the length of the LIS.

Complexities

- **Dynamic Programming:** $O(n^2)$ time, $O(n)$ space.
- **Binary Search:** $O(n \log n)$ time, $O(n)$ space.

Python Implementation

Below are implementations for both approaches.

Dynamic Programming Approach ($O(n^2)$)

```
def lengthOfLIS(nums):
    n = len(nums)
    dp = [1] * n
    for i in range(n):
        for j in range(i):
            if nums[i] > nums[j]:
                dp[i] = max(dp[i], dp[j] + 1)
    return max(dp)

# Example usage:
```

```
nums = [10, 9, 2, 5, 3, 7, 101, 18]
print(lengthOfLIS(nums)) # Output: 4
```

Binary Search Approach ($O(n \log n)$)

```
from bisect import bisect_left

def lengthOfLIS(nums):
    sub = []
    for num in nums:
        pos = bisect_left(sub, num)
        if pos == len(sub):
            sub.append(num)
        else:
            sub[pos] = num
    return len(sub)

# Example usage:
nums = [10, 9, 2, 5, 3, 7, 101, 18]
print(lengthOfLIS(nums)) # Output: 4
```

Why These Approaches?

The DP approach is intuitive and provides a clear understanding of the LIS structure. The binary search approach demonstrates the power of optimization in reducing time complexity.

Alternative Approaches

- **Recursive with Memoization:** Recursively find the LIS for each element and store results to avoid redundant calculations.
- **Segment Tree or Fenwick Tree:** Use advanced data structures to handle updates and queries efficiently.

Similar Problems

- **Longest Common Subsequence (LCS):** Finds the longest subsequence common to two sequences.
- **Maximum Sum Increasing Subsequence:** Maximizes the sum instead of the length.

- **Longest Decreasing Subsequence:** Similar to LIS but focuses on decreasing order.

Corner and Special Cases to Test

- `nums = [1]`: Single element.
- `nums = [5, 4, 3, 2, 1]`: Strictly decreasing sequence.
- `nums = [1, 1, 1, 1]`: All elements are the same.
- Large `nums`: Ensure scalability for $n > 10^3$.

Conclusion

The Longest Increasing Subsequence problem is a cornerstone in algorithm design, offering insights into optimization and dynamic programming techniques. By mastering both the DP and binary search approaches, you gain valuable tools for solving a wide range of sequence-related challenges.

Problem 15.5 Maximum Product Subarray

The **Maximum Product Subarray** problem is a classic dynamic programming challenge that highlights the importance of tracking both the maximum and minimum values in a sequence. The problem's complexity arises from handling positive, negative, and zero values, which can significantly affect the product of subarrays.

Problem Statement

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest product, and return the product.

Example 1:

```
Input: nums = [2,3,-2,4]
Output: 6
Explanation: The subarray [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 contiguous subarray.
```

Key Observations:

- Negative numbers can transform a large negative product into a large positive product when multiplied with another negative number¹.
- Zeros break the continuity of a subarray's product, requiring a reset of the calculation².

¹ Tracking both the maximum and minimum products is crucial for this reason

² Any subarray containing zero has a product of zero

Algorithmic Approach

The most efficient solution to this problem leverages dynamic programming to maintain:

- `max_product`: The maximum product of a subarray ending at the current index.
- `min_product`: The minimum product of a subarray ending at the current index³.

³ Tracking the minimum is essential to handle negative values correctly

At each step, update `max_product` and `min_product` using the current number and the products of the current number with the previous `max_product` and `min_product`. If the current number is negative, swap `max_product` and `min_product` before updating.

Algorithm:

- Initialize `max_product`, `min_product`, and `result` to the first element of the array.
- Iterate through the array starting from the second element:
 - If the current number is negative, swap `max_product` and `min_product`.
 - Update `max_product` as:

$$\text{max_product} = \max(\text{nums}[i], \text{max_product} \times \text{nums}[i])$$
 - Update `min_product` as:

$$\text{min_product} = \min(\text{nums}[i], \text{min_product} \times \text{nums}[i])$$
 - Update `result` as:

$$\text{result} = \max(\text{result}, \text{max_product})$$
- Return `result`.

Complexities

- **Time Complexity:** $O(n)$, as the array is traversed only once.
- **Space Complexity:** $O(1)$, since only a constant amount of extra space is required.

Python Implementation

```
class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        # Initialize max_product, min_product, and result with the first element
        max_product = min_product = result = nums[0]

        # Iterate through nums starting from the second element
        for i in range(1, len(nums)):
            # If the current element is negative, swap max_product and min_product
            if nums[i] < 0:
                max_product, min_product = min_product, max_product

            # Update max_product and min_product
            max_product = max(nums[i], max_product * nums[i])
            min_product = min(nums[i], min_product * nums[i])

            # Update the global result
            result = max(result, max_product)

        return result
```

Why This Approach?

This approach efficiently calculates the maximum product of a subarray by maintaining local maxima and minima at each step, ensuring optimal performance. The $O(n)$ time complexity is achieved by avoiding recalculation of products for all possible subarrays, which would otherwise result in $O(n^2)$ complexity.

Alternative Approaches

- **Brute Force:** Calculate the product of every possible subarray. While straightforward, this approach has $O(n^2)$ time complexity and is impractical for large arrays.
- **Divide and Conquer:** Split the array into halves, recursively find the maximum product in each half, and compute the maximum product across the midpoint. This approach has $O(n \log n)$ time complexity but is less efficient than the dynamic programming solution.

Similar Problems

- **Maximum Subarray Sum:** Use Kadane's Algorithm to find the maximum sum of a contiguous subarray.
- **Circular Subarray Maximum Product:** Extend this problem to handle arrays with wraparound subarrays.

Things to Keep in Mind and Tricks

- **Negative Numbers:** Always track both maximum and minimum products to handle cases where negative values become positive when multiplied.
- **Zeros:** A zero resets the product, so consider restarting the calculation from the next element after a zero.
- **Edge Cases:** Test arrays with single elements, all positive numbers, all negative numbers, and arrays with zeros.

Corner and Special Cases to Test

- Arrays with one element ([3]): The product is the element itself.
- Arrays with all negative numbers ([-1, -2, -3]): The maximum product is the product of all elements if the count of negatives is even.
- Arrays with zeros ([0, -2, -3, 0, 4]): Ensure the algorithm handles resets correctly.
- Mixed positive and negative numbers ([2, 3, -2, 4, -1]): Check transitions between positive and negative subarrays.

Conclusion

The **Maximum Product Subarray** problem is a classic example of dynamic programming's power in handling complex array-based problems. By maintaining local maxima and minima, this approach elegantly solves the problem in linear time, ensuring efficiency even for large inputs. Mastering this problem builds a strong foundation for tackling similar challenges involving subarrays and dynamic optimization.

15.1 Kadane's Algorithm

Kadane's Algorithm is a powerful dynamic programming technique used to solve problems involving contiguous subarrays or subsegments of an array. Named

after its inventor, Joseph Kadane, this algorithm efficiently computes the maximum subarray sum in linear time. Its elegance lies in the realization that solving a local problem (maximum subarray ending at a specific index) helps solve the global problem (overall maximum subarray).

Overview of Kadane's Algorithm

Kadane's Algorithm is designed to find the maximum sum of a contiguous subarray within a one-dimensional array of numbers. This problem frequently arises in applications such as financial analysis, signal processing, and bioinformatics, where determining an optimal segment of data is critical.

Key Idea: For each element in the array, decide whether to:

- Include it in the current subarray (adding to the previous sum).
- Start a new subarray beginning at this element⁴.

This decision ensures that each element is processed exactly once, making the algorithm highly efficient.

⁴ Starting a new subarray is preferred when the sum of the current subarray becomes negative, as it would decrease the potential maximum

Algorithmic Insight

The algorithm maintains two variables:

- `max_current`: The maximum sum of the subarray ending at the current position.
- `max_global`: The overall maximum sum encountered so far.

For each element in the array, update `max_current` as:

$$\text{max_current} = \max(\text{nums}[i], \text{max_current} + \text{nums}[i])$$

This represents the decision to either extend the current subarray or start a new one.

Then, update `max_global`:

$$\text{max_global} = \max(\text{max_global}, \text{max_current})$$

At the end of the traversal, `max_global` contains the maximum subarray sum.

Algorithm Pseudocode

```
function Kadane(nums):
    max_current = nums[0]
```

```

max_global = nums[0]

for i from 1 to length(nums):
    max_current = max(nums[i], max_current + nums[i])
    if max_current > max_global:
        max_global = max_current

return max_global

```

Key Features of Kadane's Algorithm

- **Time Complexity:** $O(n)$, since the array is traversed exactly once⁵.
- **Space Complexity:** $O(1)$, as only a constant amount of extra space is required.
- **Iterative Dynamic Programming:** Kadane's Algorithm is an example of iterative DP, where the solution to the global problem is built incrementally from local solutions.
- **Handles Negative Numbers Gracefully:** The algorithm inherently accounts for arrays containing negative numbers by starting a new subarray when necessary.

⁵ Kadane's Algorithm is optimal for solving the maximum subarray sum problem

Applications of Kadane's Algorithm

Kadane's Algorithm is not limited to finding maximum sums. Variations of this algorithm can be applied to other problems involving contiguous subarrays:

- **Maximum Product Subarray:** Modify the algorithm to track both maximum and minimum products to handle negative values.
- **Circular Subarray Maximum Sum:** Extend Kadane's Algorithm by considering wraparound cases.
- **2D Maximum Subarray:** Use Kadane's Algorithm as a subroutine in a 2D array to find the maximum sum of a submatrix⁶.
- **Stock Price Analysis:** Solve variations like "Best Time to Buy and Sell Stock."

⁶ This involves collapsing the matrix into rows and applying Kadane's Algorithm on the resulting 1D arrays

Things to Keep in Mind

- Kadane's Algorithm works only for contiguous subarrays. For non-contiguous subarrays, different approaches like prefix sums or dynamic programming tables may be required.
- Be mindful of edge cases, such as arrays with all negative elements, where the result should be the least negative number.

Example Problem: Maximum Subarray

The Maximum Subarray problem is the most direct application of Kadane's Algorithm. Given an array, find the subarray with the largest sum. The algorithm efficiently computes the result in $O(n)$ time by dynamically updating the local and global maxima as it traverses the array.

Conclusion

Kadane's Algorithm is a cornerstone in dynamic programming, demonstrating how local decisions can be combined to solve a global problem efficiently. Its elegance, simplicity, and wide range of applications make it a must-know technique for problem-solving and interviews. By mastering Kadane's Algorithm, you gain a powerful tool for tackling a variety of contiguous subarray problems with confidence.

Problem 15.6 Maximum Subarray

The **Maximum Subarray** problem is a cornerstone of algorithmic problem-solving, frequently used to introduce concepts like dynamic programming and divide-and-conquer techniques. Its simplicity and depth make it a classic challenge for both beginners and advanced programmers.

Problem Statement

Given an integer array `nums`, find the contiguous subarray (containing at least one number) which has the largest sum, and return its sum⁷.

⁷ A subarray is defined as a contiguous part of the array, meaning the elements are adjacent and sequential

Example 1:

```
Input: nums = [-2,1,-3,4,-1,2,1,-5,4]
Output: 6
Explanation: [4,-1,2,1] has the largest sum = 6.
```

Example 2:

```
Input: nums = [1]
Output: 1
Explanation: The array contains only one element, which is the subarray.
```

Example 3:

```
Input: nums = [-1,-2,-3]
```

Output: -1

Explanation: The largest sum is the single element -1, as all numbers are negative.

Key Observations:

- An array with one element is a valid subarray⁸.
 - Negative values do not inherently prevent a subarray from being optimal; however, in some cases, starting a new subarray may yield better results.

⁸ Single-element arrays must be considered in edge cases

Algorithmic Approach

There are three primary approaches to solving this problem:

1. **Brute Force:** Examine every possible subarray and calculate their sums, maintaining the maximum encountered sum. This approach has $O(n^2)$ to $O(n^3)$ time complexity⁹.
 2. **Divide and Conquer:** Split the array into two halves, recursively find the maximum subarray sum for each half, and compute the maximum sum of a subarray that spans the midpoint. The time complexity is $O(n \log n)$ due to the recursive divisions¹⁰.
 3. **Dynamic Programming (Kadane's Algorithm):** Iteratively compute the maximum subarray sum ending at each index by comparing:

⁹ Avoid brute force unless explicitly required by constraints, as it is computationally expensive for large arrays

¹⁰ Divide and conquer provides a clear demonstration of recursive problem-solving but is less efficient than dynamic programming here

`max current = max(nums[i], max current + nums[i])`

Track the global maximum sum as:

$$\text{max_global} = \max(\text{max_global}, \text{max_current})$$

This approach has $O(n)$ time complexity and is the most efficient for this problem¹¹.

¹¹ Kadane's Algorithm is optimal because it processes the array in a single pass with constant space

Python Implementation

Below is the Python implementation of Kadane's Algorithm:

```
class Solution:
    def maxSubArray(self, nums: List[int]) -> int:
        # Initialize current and global maximums to the first element
        max_current = max_global = nums[0]

        # Traverse the array from the second element onward
        for x in nums[1:]:
            max_current = max(x, max_current + x)  # Include current element or
            ↪ start new subarray
            if max_current > max_global:
                max_global = max_current

        return max_global
```

```

max_global = max(max_global, max_current) # Update global maximum if
    ↪ needed

return max_global

```

Explanation:

- The algorithm initializes both `max_current` and `max_global` to the first element of the array¹².
- For each element, it determines whether to include it in the current subarray or start a new subarray¹³.
- `max_global` is updated whenever a larger subarray sum is encountered.
- The final value of `max_global` is returned as the result.

¹² This ensures that single-element arrays are handled naturally

¹³ This decision is made using the ‘`max`’ function

Why This Approach?

Kadane’s Algorithm is chosen for its efficiency in both time and space. By maintaining running totals and a global maximum, it avoids the overhead of computing sums for all subarrays or managing recursion.

Alternative Approaches

The divide-and-conquer method is an elegant alternative that divides the problem into smaller subproblems. However, it is less efficient due to its higher time complexity of $O(n \log n)$.

Similar Problems to This One

- Maximum Product Subarray:** Find the subarray with the largest product instead of the largest sum.
- Best Time to Buy and Sell Stock:** Identify the best days to buy and sell stock for maximum profit.
- Longest Increasing Subarray:** Find the longest contiguous subarray with increasing elements.

Things to Keep in Mind and Tricks

- All-Negative Arrays:** When all numbers are negative, the largest sum is the single largest element. Kadane’s Algorithm naturally handles this case¹⁴.

¹⁴ No need for additional checks; the algorithm inherently accommodates negative numbers

- **Starting New Subarrays:** The decision to start a new subarray is pivotal. Always compare the current element with the sum of the current element and the existing subarray.
- **Edge Cases:** Consider empty arrays, single-element arrays, and arrays with alternating large positive and negative numbers.

Complexities

- **Time Complexity:** $O(n)$, as the algorithm processes each element exactly once.
- **Space Complexity:** $O(1)$, since it uses only a few variables for tracking sums.

Corner and Special Cases to Test

- **Empty Array:** Confirm the algorithm gracefully handles invalid input or returns a default value¹⁵.
- **Single-Element Array:** Ensure that the output is the element itself.
- **All-Negative Numbers:** Validate that the largest (least negative) number is returned.
- **Mixed Positive and Negative Numbers:** Test with arrays containing both large positive and negative numbers to ensure correct subarray selection.

¹⁵ Some implementations may raise exceptions for empty arrays

Conclusion

The **Maximum Subarray** problem exemplifies the power of dynamic programming in simplifying complex problems. Kadane's Algorithm is the optimal solution, offering both efficiency and elegance. By understanding the nuances of this problem, you can approach similar array challenges with confidence, leveraging dynamic programming concepts to solve them effectively.

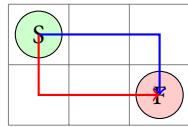
Problem 15.7 Unique Paths

Problem Statement

A classic dynamic programming problem that demonstrates how to build complex solutions from simpler subproblems.

Given a $m \times n$ grid, a robot starts at the top-left corner (marked 'Start') and aims to reach the bottom-right corner (marked 'Finish'). The robot can only move either down or right at each step. Calculate the total number of unique paths possible.

Examples:

Figure 15.1: Example of a 3×2 grid showing two possible paths

Input: $m = 3, n = 2$

Output: 3

Explanation: From top-left corner, there are three ways:

1. Right \rightarrow Right \rightarrow Down
2. Right \rightarrow Down \rightarrow Right
3. Down \rightarrow Right \rightarrow Right

Input: $m = 3, n = 7$

Output: 28

Solution Approaches

1. Dynamic Programming (Optimal)

- **Time:** $O(m \times n)$
- **Space:** $O(m \times n)$ or $O(n)$ with optimization
- **Idea:** Each cell's paths = paths from above + paths from left

2. Combinatorics

- **Time:** $O(\min(m,n))$
- **Space:** $O(1)$
- **Formula:** $C(m+n-2, m-1)$ or $C(m+n-2, n-1)$

Optimal Implementation

```
from typing import List

class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        # Optimize space by using 1D DP array
        dp = [1] * n

        for i in range(1, m):
            for j in range(1, n):
                dp[j] += dp[j-1]
```

```

        return dp[-1]

    def uniquePathsCombinatorics(self, m: int, n: int) -> int:
        # Alternative combinatorics solution
        def nCr(n: int, r: int) -> int:
            r = min(r, n-r) # Optimize by using smaller r
            numerator = denominator = 1
            for i in range(r):
                numerator *= (n - i)
                denominator *= (i + 1)
            return numerator // denominator

        return nCr(m + n - 2, min(m - 1, n - 1))

# Test cases
def test_unique_paths():
    solution = Solution()

    # Basic cases
    assert solution.uniquePaths(3, 2) == 3
    assert solution.uniquePaths(3, 7) == 28

    # Edge cases
    assert solution.uniquePaths(1, 1) == 1
    assert solution.uniquePaths(1, 5) == 1
    assert solution.uniquePaths(5, 1) == 1

```

Implementation Details

- **Space Optimization:** Using 1D DP array instead of 2D
- **Base Cases:** First row and column initialized to 1
- **DP Formula:** $dp[j] += dp[j-1]$
- **Alternative:** Combinatorics solution included

Common Pitfalls

- Forgetting to initialize first row/column
- Using unnecessary 2D array
- Integer overflow in combinatorics approach
- Not handling edge cases ($m=1$ or $n=1$)

Interview Tips

- Start with the DP solution - it's more intuitive
- Mention space optimization possibility
- Discuss the combinatorics approach as optimization
- Consider follow-up questions:
 - What if there are obstacles?
 - What if certain cells are blocked?
 - What if diagonal moves are allowed?

Related Problems

- Unique Paths II (with obstacles)
- Minimum Path Sum
- Robot Room Cleaner
- Cherry Pickup

Problem 15.8 Edit Distance

Problem Statement

Given two strings `word1` and `word2`, find the minimum number of operations needed to convert `word1` into `word2`. The allowed operations are:

- Insert a character
- Delete a character
- Replace a character

Examples:

Input: `word1 = "horse"`, `word2 = "ros"`

Output: 3

Explanation:

1. `horse` → `rorse` (replace 'h' with 'r')
2. `rorse` → `rose` (delete 'r')
3. `rose` → `ros` (delete 'e')

A fundamental dynamic programming problem that demonstrates optimal substructure and is widely used in spell checkers, DNA sequence alignment, and natural language processing.

Input: word1 = "intention", word2 = "execution"
 Output: 5

Key Insights

- If characters match, no operation needed
- If characters differ, we have three choices:
 - Insert: $1 + \text{distance}(i, j-1)$
 - Delete: $1 + \text{distance}(i-1, j)$
 - Replace: $1 + \text{distance}(i-1, j-1)$
- Base cases: empty string requires length operations

Dynamic Programming Approach

The problem exhibits:

- **Optimal Substructure:** Solution built from optimal solutions of subproblems
- **Overlapping Subproblems:** Same subproblems computed multiple times
- **State Definition:** $\text{dp}[i][j] = \text{minimum operations to convert word1[0:i] to word2[0:j]}$

The DP table represents the minimum operations needed for prefixes of both strings, building up to the final solution.

Implementation

```
class Solution:
    def minDistance(self, word1: str, word2: str) -> int:
        # Handle edge cases
        if not word1 and not word2:
            return 0
        if not word1:
            return len(word2)
        if not word2:
            return len(word1)

        m, n = len(word1), len(word2)
        # Initialize DP table
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        # Base cases: converting to/from empty string
        for i in range(m + 1):
            for j in range(n + 1):
                if i == 0 and j == 0:
                    dp[i][j] = 0
                elif i == 0:
                    dp[i][j] = j
                elif j == 0:
                    dp[i][j] = i
                else:
                    if word1[i-1] == word2[j-1]:
                        dp[i][j] = dp[i-1][j-1]
                    else:
                        dp[i][j] = min(dp[i-1][j], dp[i][j-1], dp[i-1][j-1]) + 1
```

```

        dp[i][0] = i  # deletions
    for j in range(n + 1):
        dp[0][j] = j  # insertions

    # Fill DP table
    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]  # no operation needed
            else:
                dp[i][j] = 1 + min(
                    dp[i-1][j],      # deletion
                    dp[i][j-1],      # insertion
                    dp[i-1][j-1]     # replacement
                )

    return dp[m][n]

# Test cases
def test_edit_distance():
    solution = Solution()

    # Basic cases
    assert solution.minDistance("horse", "ros") == 3
    assert solution.minDistance("intention", "execution") == 5

    # Edge cases
    assert solution.minDistance("", "") == 0
    assert solution.minDistance("a", "") == 1
    assert solution.minDistance("", "a") == 1

    # Same strings
    assert solution.minDistance("hello", "hello") == 0

    # Completely different strings
    assert solution.minDistance("abc", "def") == 3

```

Complexity Analysis

- **Time Complexity:** $O(mn)$ where m, n are lengths of input strings
- **Space Complexity:** $O(mn)$ for the DP table
- **Space Optimization:** Can be reduced to $O(\min(m,n))$ using rolling arrays

Common Pitfalls

- Forgetting to handle empty string cases

- Incorrect base case initialization
- Not considering all three operations at each step
- Off-by-one errors in string indices

Interview Tips

- Start with a small example to illustrate the approach
- Draw the DP table to explain state transitions
- Mention space optimization possibilities
- Discuss real-world applications:
 - Spell checkers
 - DNA sequence alignment
 - Natural language processing
 - Fuzzy string matching

Related Problems

- One Edit Distance
- Delete Distance
- Longest Common Subsequence
- Regular Expression Matching

Problem 15.9 Longest Common Subsequence

The "Longest Common Subsequence" (LCS) problem is a classic algorithmic challenge that falls under the category of dynamic programming. The problem entails finding the length of the longest subsequence present in both given sequences, where a subsequence is defined as a sequence that maintains the original order but not necessarily contiguously present in both strings.

Problem Statement

Given two strings, ‘text1‘ and ‘text2‘, the task is to return the length of their longest common subsequence. A subsequence is a new string generated from the original string by deleting some characters (possibly none) without altering the remaining

characters' relative order. A common subsequence of two strings is a subsequence that is common to both strings. If there is no common subsequence, the function should return 0.

Example:

Input:
text1 = "abcde"
text2 = "ace"

Output:

3

Explanation:

The longest common subsequence is "ace" and its length is 3.

[LeetCode link: Longest Common Subsequence](#)

Algorithmic Approach

The dynamic programming approach involves creating a 2D array 'dp' where ' $dp[i][j]$ ' represents the length of the longest common subsequence of substrings ' $text1[0...i-1]$ ' and ' $text2[0...j-1]$ '. This table is filled in a bottom-up manner based on whether characters at current indices in both strings match or not, as defined by the recurrence relations:

- If ' $text1[i - 1]$ ' equals ' $text2[j - 1]$ ', include this character in LCS: ' $dp[i][j] = dp[i - 1][j - 1] + 1$ '.
- If ' $text1[i - 1]$ ' is not equal to ' $text2[j - 1]$ ', find the longer LCS by not including either the i-th or j-th character: ' $dp[i][j] = \max(dp[i - 1][j], dp[i][j - 1])$ '.

The initial condition is when at least one of the strings is empty, leading to a '0' longest common subsequence. The final answer resides in ' $dp[\text{length}(\text{text1})][\text{length}(\text{text2})]$ ', determining the length of the LCS of ' text1 ' and ' text2 '.

Complexities

- **Time Complexity:** The time complexity is $O(m \times n)$, where m and n are the lengths of ' text1 ' and ' text2 ' respectively, owing to the nested loop to fill the 'dp' table.

- **Space Complexity:** The space complexity is $O(m \times n)$ to store the ‘dp’ table. It is possible to reduce the space complexity by using only two rows at a time since we only reference the row above the current one.

ewpage

Python Implementation

Below is the complete Python code with optimizations and comprehensive test cases:

```
from typing import List, Optional

class Solution:
    def longestCommonSubsequence(self, text1: str, text2: str) -> int:
        # Handle edge cases
        if not text1 or not text2:
            return 0

        # Optimize by making text1 the shorter string
        if len(text1) > len(text2):
            text1, text2 = text2, text1

        m, n = len(text1), len(text2)

        # Space-optimized version using only two rows
        prev = [0] * (n + 1)
        curr = [0] * (n + 1)

        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if text1[i-1] == text2[j-1]:
                    curr[j] = prev[j-1] + 1
                else:
                    curr[j] = max(prev[j], curr[j-1])
            prev, curr = curr, prev

        return prev[n]

    def getLCS(self, text1: str, text2: str) -> str:
        """Returns the actual longest common subsequence."""
        m, n = len(text1), len(text2)
        dp = [[0] * (n + 1) for _ in range(m + 1)]

        # Fill the dp table
        for i in range(1, m + 1):
            for j in range(1, n + 1):
                if text1[i-1] == text2[j-1]:
                    dp[i][j] = dp[i-1][j-1] + 1
                else:
                    dp[i][j] = max(dp[i-1][j], dp[i][j-1])

        lcs = []
        i, j = m, n
        while i > 0 and j > 0:
            if text1[i-1] == text2[j-1]:
                lcs.append(text1[i-1])
                i -= 1
                j -= 1
            elif dp[i-1][j] >= dp[i][j-1]:
                i -= 1
            else:
                j -= 1

        return ''.join(reversed(lcs))
```

```

        else:
            dp[i][j] = max(dp[i-1][j], dp[i][j-1])

    # Reconstruct the LCS
    lcs = []
    i, j = m, n
    while i > 0 and j > 0:
        if text1[i-1] == text2[j-1]:
            lcs.append(text1[i-1])
            i -= 1
            j -= 1
        elif dp[i-1][j] > dp[i][j-1]:
            i -= 1
        else:
            j -= 1

    return ''.join(reversed(lcs))

# Comprehensive test cases
def test_lcs():
    solution = Solution()

    # Basic cases
    assert solution.longestCommonSubsequence("abcde", "ace") == 3
    assert solution.getLCS("abcde", "ace") == "ace"

    # Edge cases
    assert solution.longestCommonSubsequence("", "abc") == 0
    assert solution.longestCommonSubsequence("abc", "") == 0
    assert solution.longestCommonSubsequence("", "") == 0

    # Same strings
    assert solution.longestCommonSubsequence("abc", "abc") == 3
    assert solution.getLCS("abc", "abc") == "abc"

    # No common subsequence
    assert solution.longestCommonSubsequence("abc", "def") == 0

    # Longer examples
    assert solution.longestCommonSubsequence(
        "AGGTAB", "GXTXAYB") == 4
    assert solution.getLCS("AGGTAB", "GXTXAYB") == "GTAB"

```

This code initializes a ‘dp’ table and then iteratively builds up the solution to larger subproblems using the previously mentioned recurrence relations. By the end, the ‘dp’ table’s last entry reflects the answer to the problem.

Why this approach

The dynamic programming approach is leveraged due to the LCS problem's overlapping subproblems and optimal substructure properties. Specifically, solving larger problems efficiently requires solving various smaller, similar problems, which naturally lends itself to a dynamic programming solution. This approach minimizes redundant calculations compared to a naive recursive solution and is therefore chosen for its time efficiency.

Optimizations

- **Space Optimization:** Using two rows instead of full matrix
- **Input Optimization:** Processing shorter string in outer loop
- **Early Termination:** Handling empty string cases immediately
- **Memory Efficiency:** Using array rotation instead of copying

Common Pitfalls

- Forgetting to handle empty string cases
- Off-by-one errors in dp table indices
- Not considering space optimization possibilities
- Incorrect reconstruction of the actual subsequence

Interview Tips

- Start with a small example to illustrate the approach
- Mention space optimization possibilities
- Discuss how to reconstruct the actual subsequence
- Consider follow-up questions:
 - What if we need all possible LCS?
 - How to handle very long strings?
 - What if we have more than two strings?

Applications

- **Bioinformatics:** DNA sequence alignment
- **Version Control:** File difference comparison
- **Natural Language Processing:** Text similarity
- **Data Compression:** Finding redundant patterns

Related Problems

- Longest Common Substring
- Shortest Common Supersequence
- Edit Distance
- Longest Increasing Subsequence

Problem 15.10 Decode Ways

The **Decode Ways** problem is a classic dynamic programming challenge that requires determining the total number of ways a given digit string can be decoded into letters, where each letter from A to Z is represented by numbers from 1 to 26.

This problem utilizes dynamic programming to efficiently calculate the number of ways to decode a digit string into letters.

Problem Statement

The task is to find the number of ways to decode a string s that consists only of digits with a mapping from 'A' to 'Z' where 'A' maps to "1", 'B' maps to "2", ..., and 'Z' maps to "26". A valid decoding can be any combination of single-digit numbers or two-digit numbers that fall within the range from "1" to "26".

Examples:

- **Example 1:**

```
Input: s = "12"
Output: 2
Explanation: It could be decoded as "AB" (1 2) or "L" (12).
```

- **Example 2:**

Input: s = "226"
 Output: 3
 Explanation: It could be decoded as "BZ" (2 26), "VF" (22 6), or "BBF" (2 2 6).

- **Example 3:**

Input: s = "0"
 Output: 0
 Explanation: There is no character that is mapped to a number starting with '0'.

LeetCode link: Decode Ways

[LeetCode Link]
 [GeeksForGeeks Link]
 [HackerRank Link]
 [CodeSignal Link]
 [InterviewBit Link]
 [Educative Link]
 [Codewars Link]

Algorithmic Approach

To solve the **Decode Ways** problem, we can use dynamic programming. We define a cache array where $cache[i]$ represents the number of ways to decode the substring $s[: i]$. We iterate through the given string and update the cache based on whether the current character or the combination of the current and previous characters can represent a valid decoding.

1. Initialization:

- Create a cache array of size $n + 1$, where n is the length of the string s .
- Initialize $cache[0] = 1$ to represent the base case of an empty string.
- Initialize $cache[1] = 1$ if the first character is not '0'; otherwise, set it to 0.

2. Iteration:

- Iterate through the string from the second character to the end.
- For each character, check the following:
 - **Single-digit decode:** If the current character is not '0', set $cache[i] += cache[i - 1]$.
 - **Two-digit decode:** If the two-digit number formed by the current and previous characters is between 10 and 26, set $cache[i] += cache[i - 2]$.

3. Result:

- After completing the iteration, $cache[n]$ will contain the total number of ways to decode the entire string.

This method ensures that each subproblem is solved only once, achieving an optimal $O(n)$ time complexity.

Dynamic programming efficiently solves overlapping subproblems by storing and reusing solutions.

Complexities

- **Time Complexity:** $O(n)$, where n is the length of the string. We traverse the string once, performing constant-time operations at each step.
- **Space Complexity:** $O(n)$ for the cache array. However, this can be optimized to $O(1)$ by only keeping track of the last two computed values.

Python Implementation

Below are two implementations: the standard DP solution and a space-optimized version:

```
from typing import List, Optional

class Solution:
    def numDecodings(self, s: str) -> int:
        """Standard DP solution with O(n) space."""
        if not s or s[0] == '0':
            return 0

        n = len(s)
        cache = [0] * (n + 1)
        cache[0] = 1
        cache[1] = 1

        for i in range(2, n + 1):
            one_digit = int(s[i-1:i])
            two_digits = int(s[i-2:i])

            if 1 <= one_digit <= 9:
                cache[i] += cache[i-1]

            if 10 <= two_digits <= 26:
                cache[i] += cache[i-2]

        return cache[n]

    def numDecodingsOptimized(self, s: str) -> int:
        """Space-optimized solution with O(1) space."""
        if not s or s[0] == '0':
            return 0

        n = len(s)
        prev2, prev1 = 1, 1 # dp[i-2], dp[i-1]

        for i in range(1, n):
            current = 0
            if s[i] != '0':
                current += prev1
            if 10 <= int(s[i-1:i+1]) <= 26:
                current += prev2
            prev2, prev1 = prev1, current

        return prev1

    def getAllDecodings(self, s: str) -> List[str]:
        """Returns all possible decodings of the string."""
        def backtrack(index: int, current: str) -> None:
```

Space-optimized solution reduces memory usage from $O(n)$ to $O(1)$.

```

        if index == len(s):
            result.append(current)
            return

        # Single digit
        if s[index] != '0':
            digit = int(s[index])
            char = chr(ord('A') + digit - 1)
            backtrack(index + 1, current + char)

        # Two digits
        if index + 1 < len(s):
            two_digits = int(s[index:index+2])
            if 10 <= two_digits <= 26:
                char = chr(ord('A') + two_digits - 1)
                backtrack(index + 2, current + char)

    result: List[str] = []
    if s and s[0] != '0':
        backtrack(0, "")
    return result

# Comprehensive test cases
def test_decode_ways():
    solution = Solution()

    # Basic cases
    assert solution.numDecodings("12") == 2
    assert solution.numDecodings("226") == 3
    assert set(solution.getAllDecodings("12")) == {"AB", "L"}

    # Edge cases
    assert solution.numDecodings("") == 0
    assert solution.numDecodings("0") == 0
    assert solution.numDecodings("01") == 0

    # Complex cases
    assert solution.numDecodings("1111") == 5
    assert len(solution.getAllDecodings("1111")) == 5

    # Performance test
    assert solution.numDecodingsOptimized("111111") == 13

```

This implementation first checks if the input string is empty or starts with '0'. If either is true, there are no ways to decode the message, and it returns '0'. It initializes a cache array where each index represents the number of ways to decode the string up to that point. It then iterates through the string, updating the cache based on whether one-digit or two-digit numbers can represent valid letters.

Explanation

The ‘numDecodings‘ function efficiently calculates the number of ways to decode the input string s by leveraging dynamic programming. Here’s a detailed breakdown of the implementation:

- **Initialization:**

- **Edge Cases:** If the string is empty or starts with ’0’, return ‘0’ as no valid decodings are possible.
- **Cache Array:** Create a cache array ‘cache‘ of size $n + 1$, where n is the length of the string. Initialize ‘cache[0] = 1‘ to represent the base case of an empty string and ‘cache[1] = 1‘ if the first character is not ’0‘.

- **Iteration:**

- Iterate through the string from the second character to the end (i.e., indices 2 to n).
- For each position i :
 - * **Single-digit decode:** Extract the single digit $s[i - 1]$ and convert it to an integer. If it is between 1 and 9, it represents a valid letter, so add the number of ways to decode the string up to $i - 1$ (i.e., ‘cache[i-1]’) to ‘cache[i]’.
 - * **Two-digit decode:** Extract the two-digit number $s[i-2 : i]$ and convert it to an integer. If it is between 10 and 26, it represents a valid letter, so add the number of ways to decode the string up to $i - 2$ (i.e., ‘cache[i-2]’) to ‘cache[i]’.

- **Result:**

- After completing the iteration, the value at ‘cache[n]’ will represent the total number of ways to decode the entire string s .

Why This Approach

The dynamic programming approach is chosen for its efficiency in both time and space. It systematically builds up the solution by solving smaller subproblems (i.e., decoding substrings of increasing length) and storing their results to avoid redundant calculations. This ensures that each subproblem is solved only once, achieving an optimal $O(n)$ time complexity.

Alternative Approaches

An alternative approach is to use a **recursive function with memoization** to store the results of subproblems and avoid recalculating them. While this method also

achieves $O(n)$ time complexity, it may have a higher space complexity due to the recursion stack. Another possibility is to use an **iterative approach with two variables** to store the last two results, thereby reducing the space complexity to $O(1)$.

However, the dynamic programming approach presented above strikes a balance between simplicity and efficiency, making it the most straightforward and optimal solution for this problem.

Similar Problems to This One

There are several other problems that involve decoding or parsing strings with specific constraints, such as:

- Climbing Stairs
- House Robber
- Fibonacci Number
- Longest Palindromic Substring
- Unique Paths

Things to Keep in Mind and Tricks

When solving dynamic programming problems, especially those involving strings, it's important to consider edge cases such as empty strings and strings with leading zeros. For this specific problem, the decoding is invalid if there are any zeros not preceded by '1' or '2'. Additionally, optimizing space usage by using only the necessary previous states can lead to more efficient solutions.

Corner and Special Cases to Test When Writing the Code

Some corner cases to consider include:

- **Single '0':** 's = "0" should return '0' as no valid decoding exists.
- **Leading '0':** 's = "06" should return '0' as the string starts with '0'.
- **All '1's and '2's:** 's = "1111" should return '5' ("AAAA", "AAB", "ABA", "BAA", "BB").
- **Numbers Greater Than '26':** 's = "27" should return '1' ("BG"), as '27' does not map to any letter.

- **Multiple '0's:** 's = "100" should return '0' as the second '0' cannot be decoded.
- **Empty String:** 's = "" should return '0' as there are no characters to decode.
- **Long String with Valid Decodings:** A long string like 's = "1111111111"' should return a large number of decodings (e.g., 89 for 10 '1's).
- **String with '10' and '20':** 's = "1010" should return '1' ("JJ").

Visual Explanation

	0	2	2	6
dp: 0	1	+1 1	+1 2	+1 3

Figure 15.2: DP table evolution for input "226"

Implementation Variants

- **Recursive with Memoization:**

- Uses recursion with caching
- Good for interview discussions
- Easier to understand initially

- **Bottom-up DP:**

- More efficient in practice
- Better space complexity
- Faster execution

- **Space-Optimized:**

- Uses only two variables
- O(1) space complexity
- Best for production code

Real-World Applications

- **Cryptography:** Decoding encrypted messages
- **Natural Language Processing:** Word segmentation
- **DNA Sequencing:** Pattern matching in sequences
- **Data Compression:** Variable-length encoding

Problem 15.11 Jump Game I

The **Jump Game I** problem is a well-known challenge in LeetCode's array and dynamic programming category. The goal is to determine if it is possible to reach the final index of an array, where each element represents the maximum jump length from that position.

This problem employs a greedy algorithm to determine if the end of the array can be reached.

Problem Statement

Given an array of non-negative integers `nums`, where `nums[i]` is the maximum jump length at position i , the task is to decide whether it is possible to reach the last index starting from the first index. The output should be a boolean, `true` if you can reach the last index, otherwise `false`.

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.

LeetCode link: [Jump Game I](#)

[LeetCode Link]

[GeeksForGeeks Link]

[HackerRank Link]

[CodeSignal Link]

[InterviewBit Link]

[Educative Link]

[Codewars Link]

Algorithmic Approach

To solve the **Jump Game I** problem, we employ a **greedy algorithm**. The approach involves iterating through the array and at each position, updating the maximum reachable index. If at any point the current index exceeds the maximum reachable index, it means the end cannot be reached. Here's a step-by-step breakdown:

1. Initialization:

- Initialize a variable `max_reachable` to keep track of the furthest index that can be reached.
- Set `max_reachable` to the first element of the array.

2. Iteration:

- Iterate through each index i of the array.

- If the current index i is greater than `max_reachable`, return `false` as it means the end cannot be reached.
- Update `max_reachable` to be the maximum of its current value and $i + \text{nums}[i]$.
- If `max_reachable` is greater than or equal to the last index, return `true`.

3. Termination:

- After completing the iteration, if `max_reachable` is greater than or equal to the last index, return `true`; otherwise, return `false`.

This method ensures that we can determine the possibility of reaching the end by making optimal jump choices at each step without exploring all possible jump combinations.

The greedy approach efficiently determines reachability by always making the farthest possible jump.

Complexities

- **Time Complexity:** $O(n)$, where n is the number of elements in the input array.
We traverse the array once.
- **Space Complexity:** $O(1)$, since we only use a few extra variables for tracking.

Python Implementation

Below are three implementations: the standard greedy solution, an optimized version, and a solution that returns the actual path:

Multiple implementation approaches demonstrate different trade-offs between readability and optimization.

```
from typing import List, Optional

class Solution:
    def canJump(self, nums: List[int]) -> bool:
        """Standard greedy solution."""
        max_reachable = 0
        last_index = len(nums) - 1

        for i in range(len(nums)):
            if i > max_reachable:
                return False
            max_reachable = max(max_reachable, i + nums[i])
            if max_reachable >= last_index:
                return True

        return False

    def canJumpOptimized(self, nums: List[int]) -> bool:
        """Optimized solution with early termination."""
        if not nums:
            return False
        if len(nums) == 1:
            return True

        max_reachable = nums[0]
        i = 0

        while i <= max_reachable and max_reachable < len(nums) - 1:
            max_reachable = max(max_reachable, i + nums[i])
            i += 1

        return max_reachable >= len(nums) - 1

    def findPath(self, nums: List[int]) -> List[int]:
        """Returns the actual path of jumps to reach the end."""
        if not self.canJump(nums):
            return []

        path = [0] # Start position
        pos = 0
        while pos < len(nums) - 1:
            best_jump = pos
            max_reach = 0

            # Find the best jump position
            for j in range(pos + 1, min(pos + nums[pos] + 1, len(nums))):
```

```

        if j + nums[j] >= max_reach:
            max_reach = j + nums[j]
            best_jump = j

        pos = best_jump
        path.append(pos)

    return path

# Comprehensive test cases
def test_jump_game():
    solution = Solution()

    # Basic cases
    assert solution.canJump([2,3,1,1,4]) == True
    assert solution.canJump([3,2,1,0,4]) == False

    # Edge cases
    assert solution.canJump([0]) == True
    assert solution.canJump([1,0]) == True
    assert solution.canJump([0,1]) == False

    # Path finding
    assert solution.findPath([2,3,1,1,4]) == [0,1,4]
    assert solution.findPath([1,1,1,1]) == [0,1,2,3]

    # Performance test for optimized version
    large_array = [1] * 10000
    assert solution.canJumpOptimized(large_array) == True

```

Visual Explanation

Index	0	1	2	3	4
Value	2	3	1	1	4
Max Reach	2	4	4	4	8

Figure 15.3: Step-by-step visualization of max_reachable for input [2,3,1,1,4]

Implementation Variants

- **Standard Greedy:**

- Simple and readable
- Good for interviews
- $O(n)$ time complexity

- **Optimized Version:**

- Early termination

- Better average-case performance
- More complex logic

- **Path Finding:**

- Returns actual jump sequence
- Useful for debugging
- Higher space complexity

Common Optimization Techniques

- **Early Termination:** Stop when max_reachable reaches the end
- **Direction Optimization:** Iterate backwards for certain variants
- **Memory Optimization:** Use constant space
- **Branch Prediction:** Organize conditions for likely cases

Real-World Applications

- **Network Routing:** Finding valid paths in networks
- **Game Development:** Character movement mechanics
- **Resource Allocation:** Planning resource distribution
- **Circuit Design:** Signal propagation analysis

Explanation

The ‘canJump’ function determines whether it is possible to reach the last index of the array starting from the first index. Here’s a detailed breakdown of the implementation:

- **Initialization:**

- `max_reachable`: This variable keeps track of the furthest index that can be reached at any point during the iteration.
- `last_index`: This is the index of the last element in the array, which is the target we aim to reach.

- **Iteration:**

- **Loop through the array:** For each index i , check if it is within the current `max_reachable`. If i is greater than `max_reachable`, it means the current index cannot be reached, and hence, the last index is unreachable.
- **Update `max_reachable`:** Update `max_reachable` to be the maximum of its current value and $i + \text{nums}[i]$, which represents the furthest index reachable from the current position.
- **Early termination:** If at any point `max_reachable` becomes greater than or equal to `last_index`, return `True` immediately as the end is reachable.

- **Final Check:**

- After completing the loop, return `False` if the end was not reachable during the iteration.

Why This Approach

The greedy algorithm is chosen for its efficiency and simplicity. By always tracking the furthest we can reach at each step, we can solve the problem by making a single pass through the array, thus achieving a linear time complexity. This approach avoids the need for more complex methods like dynamic programming or recursion, making it both optimal and easy to implement.

Alternative Approaches

An alternative approach would be to use **dynamic programming** to solve the problem. This would involve building an array where each element represents whether you can reach the current position from the start. However, this approach has a higher time complexity of $O(n^2)$, making it less efficient compared to the greedy algorithm.

Another approach could involve using **breadth-first search (BFS)**, where each position in the array is considered as a node in a graph, and edges represent possible jumps. While BFS can be used to determine reachability, it also results in higher time and space complexities compared to the greedy method.

The greedy algorithm remains the most optimal and straightforward method for this problem due to its linear time complexity and constant space usage.

Similar Problems to This One

There are several other problems that involve determining reachability or optimizing paths within arrays or graphs, such as:

- Jump Game II
- Minimum Jumps to Reach Home
- Minimum Number of Jumps
- Word Search
- Climbing Stairs

Things to Keep in Mind and Tricks

When solving greedy algorithm problems like this one, consider the following tips:

- **Track the Maximum Reachable Index:** Always keep track of the furthest index that can be reached so far. This helps in making optimal jump decisions.
- **Early Termination:** If the maximum reachable index is greater than or equal to the last index at any point during the iteration, you can terminate early and return True.
- **Handle Edge Cases:** Consider cases where the array contains only one element, or where early elements are zero, preventing any progress.
- **Iterate Only Up to Current Maximum Reachable:** There's no need to iterate beyond the current maximum reachable index, as those positions cannot be reached.
- **Optimize Space Usage:** The greedy approach typically uses constant space, making it highly efficient in terms of memory.

Corner and Special Cases to Test When Writing the Code

When implementing the ‘canJump’ function, it is crucial to test the following edge cases to ensure robustness:

- **Single Element Array:** ‘nums = [0]’ should return ‘True’ as you are already at the last index.
- **Early Zero Blocking:** ‘nums = [0,1]’ should return ‘False’ since you cannot move from the first index.
- **All Ones:** ‘nums = [1,1,1,1]’ should return ‘True’.
- **No Possible Jump to End:** ‘nums = [3,2,1,0,4]’ should return ‘False’.
- **Large Array with Valid Path:** A large array where each element allows jumping to the end should return ‘True’.

- **Array with Last Element Zero:** ‘nums = [2,3,1,1,0]‘ should return ‘True‘.
- **Array with Multiple Paths:** ‘nums = [2,3,1,1,4]‘ should return ‘True‘ with multiple jump paths.
- **Array Starting with Zero:** ‘nums = [0,2,3]‘ should return ‘False‘.
- **Maximum Jump Reaches Beyond End:** ‘nums = [1,4,2,6,7,6,5,1,4,2,9]‘ should return ‘True‘.
- **Empty Array:** Depending on problem constraints, ‘nums = []‘ might need to be handled gracefully.

Problem 15.12 Palindromic Substrings

Problem Statement

This problem utilizes a center-expansion technique to efficiently count all palindromic substrings within a given string.

Given a string s , the objective is to find the number of palindromic substrings within that string. A palindrome is defined as a sequence of characters that reads the same forwards as it does backwards. A substring is a contiguous block of characters within the original string.

Examples:

- **Example 1:**

```
Input: s = "abc"
Output: 3
Explanation: Three palindromic substrings are "a", "b", "c".
```

- **Example 2:**

```
Input: s = "aaa"
Output: 6
Explanation: Six palindromic substrings are "a", "a", "a", "aa", "aa", "aaa".
```

LeetCode link: Problem 647

[[LeetCode Link](#)]
[GeeksForGeeks Link](#)
[HackerRank Link](#)
[CodeSignal Link](#)
[InterviewBit Link](#)
[Educative Link](#)
[Codewars Link](#)

Algorithmic Approach

The algorithm to solve this problem involves two main steps:

1. Iterate through each character in the string, considering it as the center of a possible palindrome.
2. Expand around the center for both odd and even length palindromes, counting all valid palindromes.

To implement this, a helper function can check for palindromes by expanding from the center. This function will be called twice for each character: once for the odd length palindrome (with the same start and end) and once for the even length (with start at the current character and end at the next character).

Expanding around each character efficiently captures all possible palindromic substrings without redundant checks.

Complexities

- **Time Complexity:** The time complexity for this approach is $O(n^2)$, where n is the length of the string. This is because for each character, we potentially expand in both directions until we reach the ends of the string.
- **Space Complexity:** The space complexity is $O(1)$ as we are not using any additional space that scales with the input size. The only extra space used is for a few variables to keep track of counts and indices.

Python Implementation

Below are three implementations: the standard center-expansion solution, a space-optimized version, and a solution that returns all palindromic substrings:

Multiple implementation approaches demonstrate different trade-offs between efficiency and functionality.

```
from typing import List, Set

class Solution:
    def countSubstrings(self, s: str) -> int:
        """Standard center-expansion solution."""
        def expand_around_center(left: int, right: int) -> int:
            count = 0
            while left >= 0 and right < len(s) and s[left] == s[right]:
                count += 1
                left -= 1
                right += 1
            return count

            total_palindromes = 0
            for i in range(len(s)):
                total_palindromes += expand_around_center(i, i)      # For odd lengths
                total_palindromes += expand_around_center(i, i + 1)   # For even
                ↪ lengths

            return total_palindromes

    def findAllPalindromes(self, s: str) -> Set[str]:
        """Returns all unique palindromic substrings."""
        palindromes = set()

        def expand_and_collect(left: int, right: int) -> None:
            while left >= 0 and right < len(s) and s[left] == s[right]:
                palindromes.add(s[left:right + 1])
                left -= 1
                right += 1

            for i in range(len(s)):
                expand_and_collect(i, i)      # Odd length palindromes
                expand_and_collect(i, i + 1)   # Even length palindromes

        return palindromes

    def manachersAlgorithm(self, s: str) -> int:
        """Linear time solution using Manacher's algorithm."""
        # Transform string to handle even length palindromes
        t = '#' + '#'.join(s) + '#'
        n = len(t)
        p = [0] * n  # p[i] represents palindrome radius at center i
        center = right = 0

        for i in range(n):
```

```

        if i < right:
            mirror = 2 * center - i
            p[i] = min(right - i, p[mirror])

        # Attempt to expand palindrome centered at i
        left = i - (p[i] + 1)
        right_ptr = i + (p[i] + 1)
        while left >= 0 and right_ptr < n and t[left] == t[right_ptr]:
            p[i] += 1
            left -= 1
            right_ptr += 1

        # If palindrome centered at i expands past right,
        # adjust center and right boundary
        if i + p[i] > right:
            center = i
            right = i + p[i]

    # Count palindromes (each radius value represents multiple palindromes)
    return (sum(x // 2 for x in p) + len(s))

# Comprehensive test cases
def test_palindromic_substrings():
    solution = Solution()

    # Basic cases
    assert solution.countSubstrings("abc") == 3
    assert solution.countSubstrings("aaa") == 6

    # Edge cases
    assert solution.countSubstrings("") == 0
    assert solution.countSubstrings("a") == 1

    # Find all palindromes
    assert solution.findAllPalindromes("aaa") == {"a", "aa", "aaa"}

    # Performance test for Manacher's
    long_string = "a" * 1000
    assert solution.manachersAlgorithm(long_string) == 500500 # n*(n+1)/2

```

Visual Explanation

String	Center	Expansion	Palindromes
aaa	0	a → aa → aaa	a, aa, aaa
aaa	1	a → aa	a, aa
aaa	2	a	a

Figure 15.4: Center expansion process for string "aaa"

Implementation Variants

- **Center Expansion:**

- Simple and intuitive
- $O(n^2)$ time complexity
- Good for interviews

- **Manacher's Algorithm:**

- Linear time complexity
- Complex implementation
- Best for production code

- **Dynamic Programming:**

- $O(n^2)$ space and time
- Easier to modify
- Good for related problems

Common Optimization Techniques

- **Early Termination:** Stop expansion when impossible to find longer palindromes
- **Space Optimization:** Use rolling arrays for DP approach
- **Preprocessing:** Transform string for even-length palindromes
- **Caching:** Store previously computed palindrome lengths

Real-World Applications

- **DNA Sequence Analysis:** Finding palindromic sequences
- **Text Processing:** Identifying patterns in natural language
- **Data Compression:** Utilizing palindromic patterns
- **Cryptography:** Creating and analyzing symmetric patterns

Explanation

The provided Python implementation defines a function `countSubstrings` which takes a string s as its parameter. Within this function, a nested helper function `expand_around_center` is defined, which takes two indices `left` and `right`. The helper function expands outwards from these indices while the characters at `left` and `right` are equal, incrementing the count of palindromic substrings for each iteration of the `while` loop. This helper function is then called for each character in the string s , counting all odd and even palindromic substrings centered at that character.

Why This Approach

This approach is chosen because it efficiently examines potential palindromes by expanding around a center, allowing us to count all palindromes in $O(n^2)$ without the need for additional space. It's a direct and intuitive method to solve the problem, given the definition and properties of palindromes.

Alternative Approaches

An alternative approach could use dynamic programming to build up a table that keeps track of which substrings are palindromes, but this would increase the space complexity to $O(n^2)$. The Manacher's algorithm can also be used to solve this problem in linear time, but it is significantly more complex to implement.

Similar Problems to This One

Similar problems would include those related to substrings, such as the longest palindromic substring or problems involving dynamic programming for string manipulation. Problems involving counting particular types of subsequences or patterns within a string may also apply similar methodologies.

Things to Keep in Mind and Tricks

When solving problems involving palindromes, always consider the possibility of expanding from the center. This approach often simplifies the algorithm. Moreover, be aware of the differences in handling even and odd length palindromes.

Corner and Special Cases to Test When Writing the Code

To thoroughly test the implementation, consider corner cases such as:

- A single character string
- A string with all identical characters
- A string with no palindromic substrings longer than 1
- Very long strings to ensure the performance is within acceptable bounds

15.2 Traversing from the Right

Introduction

Sometimes you can traverse an array starting from the right instead of the conventional approach of moving from the left. This technique can be particularly beneficial in scenarios where processing elements in reverse order leads to more efficient algorithms or simplifies problem-solving. Traversing from the right allows for in-place modifications without the need for additional memory, which is crucial in optimization problems. Additionally, certain problems, such as merging sorted arrays or implementing two-pointer techniques, naturally lend themselves to a right-to-left traversal.

By starting from the end of the array, you can avoid overwriting important data and reduce the complexity of the algorithm. For example, when merging two sorted arrays into one, beginning from the largest elements and working backwards ensures that you place elements in their correct positions without the need to shift existing elements. This approach not only enhances performance but also maintains the integrity of the data structure.

In this chapter, we will delve into the concept of traversing arrays from the right, explore its advantages, and examine common algorithmic problems where this approach is advantageous. Understanding when and how to apply right-to-left traversal can significantly improve your problem-solving skills and optimize your code for better performance.

When to Traverse from the Right

Traversing an array from the right is especially useful in the following scenarios:

- **In-Place Merging:** When merging two sorted arrays where one array has enough space at the end to accommodate the other.

- **Two-Pointer Techniques:** When you need to use two pointers moving towards each other to solve problems like finding pairs that meet certain conditions.
- **Reversing Operations:** When performing operations that require reversing the order of elements without using extra space.
- **Dynamic Programming:** In certain dynamic programming problems, processing elements from the end can simplify the state transitions.

Advantages of Right-to-Left Traversal

1. **Efficiency:** Reduces the need for additional memory by allowing in-place modifications.
2. **Simplicity:** Simplifies the logic for certain algorithms, making the code easier to understand and maintain.
3. **Performance:** Enhances performance by minimizing the number of operations required to achieve the desired outcome.
4. **Data Integrity:** Prevents the overwriting of important data during the traversal and modification process.

Problem 15.13 Merging Two Sorted Arrays

Consider the "Merge Sorted Array" problem, where you are given two sorted arrays, `nums1` and `nums2`, and you need to merge `nums2` into `nums1` as one sorted array. By traversing from the right, you can efficiently place the largest elements at the end of `nums1` without overwriting existing elements.

```
Input: nums1 = [1,2,3,0,0,0], m = 3
       nums2 = [2,5,6], n = 3
Output: [1,2,2,3,5,6]
```

In this example, starting from the end allows you to compare and place elements from `nums1` and `nums2` in their correct positions without the need for shifting elements to make space.

Conclusion

Traversing from the right is a powerful technique that can optimize various algorithms and simplify complex problem-solving scenarios. By understanding when and how to apply this approach, you can enhance the efficiency and performance of your code, making it a valuable tool in your algorithmic toolkit.

Problem 15.14 Daily Temperatures

The **Daily Temperatures** problem is a classic algorithmic challenge that involves predicting how many days one would have to wait until a warmer temperature. It is an excellent example to illustrate the effectiveness of stack-based solutions combined with the Two Pointers Technique, especially when processing the data in a specific order.

Problem Statement

Given a list of daily temperatures T , return a list such that, for each day in the input, tells you how many days you would have to wait until a warmer temperature. If there is no future day for which this is possible, put 0 instead.

Example:

Given the list $T = [73, 74, 75, 71, 69, 72, 76, 73]$,

Your output should be $[1, 1, 4, 2, 1, 1, 0, 0]$ ¹⁶.

Algorithmic Approach

The solution to the Daily Temperatures problem can be efficiently implemented by iterating through the list of temperatures from right to left¹⁷. This approach leverages a stack to keep track of temperatures and their indices, enabling quick lookup of the next warmer day for each temperature.

1. **Initialize a Stack:** Create an empty stack to keep track of temperature indices¹⁸.

2. **Initialize the Result List:** Create a result list filled with 0s, as a default for days with no warmer future temperature¹⁹.

3. **Iterate from Right to Left:** Start from the end of the temperature list and move towards the beginning.

- For each temperature $T[i]$:

- **Maintain Monotonic Stack:** While the stack is not empty and the current temperature $T[i]$ is greater than or equal to the temperature at the index on the top of the stack, pop the stack²⁰.

- **Determine the Next Warmer Day:**

- * If the stack is not empty after the popping process, the next warmer day for $T[i]$ is the difference between the current index and the index at the top of the stack²¹.

¹⁶ This output indicates, for example, that after the first day with temperature 73, the next warmer temperature occurs in 1 day at temperature 74

¹⁷ Processing from right to left allows us to maintain information about future warmer days that have already been processed.

¹⁸ The stack will store indices of temperatures in a monotonically decreasing order, facilitating the search for the next warmer day.

¹⁹ This list will be updated with the number of days to wait for a warmer temperature.

²⁰ This ensures that the stack only contains indices of temperatures warmer than the current one.

²¹ This difference represents the number of days to wait for a warmer temperature.

- * If the stack is empty, there is no warmer day in the future, so the result remains 0²².
 - **Push Current Index onto Stack:** Add the current index i to the stack²³.
4. **Completion:** After iterating through all temperatures, the result list will contain the required number of days to wait for a warmer temperature for each day.

²² This scenario occurs when no future day has a higher temperature than the current day.
²³ This index may serve as the next warmer day for preceding temperatures.

Python Implementation

```
def dailyTemperatures(T):
    """
    Finds the number of days until a warmer temperature for each day.

    Parameters:
    T (List[int]): List of daily temperatures.

    Returns:
    List[int]: List indicating the number of days to wait for a warmer temperature
    ↪ .
    """
    n = len(T)
    res = [0] * n
    stack = []

    for i in range(n-1, -1, -1):
        # Remove temperatures that are less than or equal to current
        while stack and T[i] >= T[stack[-1]]:
            stack.pop()

        # If stack is not empty, the next warmer day is stack[-1] - i
        if stack:
            res[i] = stack[-1] - i

        # Push current index onto stack
        stack.append(i)

    return res

# Example usage:
T = [73, 74, 75, 71, 69, 72, 76, 73]
print(dailyTemperatures(T))  # Output: [1, 1, 4, 2, 1, 1, 0, 0]
```

Example Usage and Test Cases

```
# Test case 1: General case
T = [73, 74, 75, 71, 69, 72, 76, 73]
print(dailyTemperatures(T))  # Output: [1, 1, 4, 2, 1, 1,
    ↪ 0, 0]
```

```

# Test case 2: Increasing temperatures
T = [30, 40, 50, 60]
print(dailyTemperatures(T)) # Output: [1, 1, 1, 0]

# Test case 3: Decreasing temperatures
T = [60, 50, 40, 30]
print(dailyTemperatures(T)) # Output: [0, 0, 0, 0]

# Test case 4: Mixed temperatures with duplicates
T = [30, 40, 40, 50, 30, 60]
print(dailyTemperatures(T)) # Output: [1, 1, 2, 1, 1, 0]

# Test case 5: Single element array
T = [30]
print(dailyTemperatures(T)) # Output: [0]

```

Why This Approach

The **Two Pointers Technique** combined with a stack-based approach is chosen for its **efficiency and optimal time complexity**. By processing the temperature list from right to left, we can leverage the stack to keep track of indices of warmer temperatures²⁴. This method ensures a single pass through the list with each element being pushed and popped at most once, resulting in a time complexity of $O(n)$ ²⁵. Additionally, this approach maintains a space complexity of $O(n)$, primarily due to the stack, which is acceptable given the problem constraints.

²⁴ This allows us to quickly determine how many days to wait for a warmer temperature by referring to the stack's top element

²⁵ Linear time complexity is optimal for this problem, especially with large datasets

Complexity Analysis

- **Time Complexity:** $O(n)$ ²⁶.
- **Space Complexity:** $O(n)$ ²⁷.

²⁶ Each temperature is processed once, with push and pop operations on the stack occurring at most once per element

²⁷ The stack can potentially store all indices in the worst-case scenario

Similar Problems

Other problems that can be efficiently solved using the Two Pointers Technique and stack-based approaches include:

- **Next Greater Element:** Find the next greater element for each element²⁸.
- **Stock Span Problem:** Calculate the span of stock's price for all days²⁹.
- **Largest Rectangle in Histogram:** Find the largest rectangular area in a histogram³⁰.
- **Maximum Depth of Binary Tree:** Determine the maximum depth of a binary tree³¹.

²⁸ Utilizes a similar stack-based traversal to determine the next greater element.

²⁹ Employs a stack to keep track of previous days' prices for span calculations.

³⁰ Uses a stack to manage the indices of bars for efficient area computation.

³¹ Can be approached iteratively with the help of a stack for depth tracking.

These problems often require maintaining information about previous elements or states, making stack-based or two pointers approaches highly effective³².

³² Understanding these techniques provides a strong foundation for solving a variety of related computational challenges

Things to Keep in Mind and Tricks

- **Processing Order:** Iterating from right to left allows you to have information about all future days³³.
- **Handling Duplicates:** Ensure that duplicates are correctly handled to avoid redundant calculations³⁴.
- **Stack Management:** Properly manage the stack by pushing and popping indices based on the current temperature³⁵.
- **Edge Cases:** Always consider edge cases such as single-element arrays or arrays with no warmer future days³⁶.
- **Space Optimization:** While the stack requires additional space, it is necessary for achieving optimal time complexity³⁷.

³³ This ensures that you can make informed decisions about the next warmer day based on already processed information

³⁴ Skipping identical elements during traversal maintains the accuracy of the result

³⁵ This maintains a monotonically decreasing stack, essential for the algorithm's efficiency

³⁶ Robust handling of these scenarios ensures the algorithm's reliability

³⁷ Balancing space and time efficiency is crucial for effective algorithm design

Exercises

1. **Next Greater Element:** Given an array, find the next greater element for each element³⁸.
2. **Stock Span Problem:** Calculate the span of stock's price for all days³⁹.
3. **Largest Rectangle in Histogram:** Find the largest rectangular area in a histogram⁴⁰.
4. **Maximum Depth of Binary Tree:** Determine the maximum depth of a binary tree⁴¹.
5. **Minimum Window Substring:** Given two strings, find the minimum window in the first string which will contain all the characters of the second string⁴².

³⁸ Implement a stack-based solution similar to the Daily Temperatures problem

³⁹ Use a stack to keep track of previous days' prices and calculate spans accordingly

⁴⁰ Employ a stack to manage the indices of bars for efficient area computation

⁴¹ Approach the problem iteratively using a stack to track depth levels

⁴² Combine sliding window techniques with stack-based approaches for optimal solutions

Questions for Reflection

- Why is processing the temperature list from right to left more efficient than from left to right?⁴³.
- How does the stack help in keeping track of necessary information for determining the next warmer day?⁴⁴.
- Can the Two Pointers Technique be applied to other similar problems? Provide examples⁴⁵.

⁴³ Consider how information about future days is utilized in each approach

⁴⁴ Analyze the role of the stack in maintaining order and facilitating quick lookups

⁴⁵ Think about how the two pointers can manage and compare elements in different scenarios

- What are the trade-offs between using a stack-based approach versus a brute-force approach in terms of time and space complexity?⁴⁶.
- How can this approach be modified to handle different variations of the problem, such as finding the next colder day?⁴⁷.

⁴⁶ Evaluate the benefits of optimized algorithms over naive implementations

⁴⁷ Explore how reversing the conditions and maintaining the stack accordingly can adapt the solution

References

LeetCode Problem: ⁴⁸

⁴⁸ Daily Temperatures

GeeksforGeeks Article: ⁴⁹

⁴⁹ Two Pointers Technique

HackerRank Problem: ⁵⁰

⁵⁰ Two Sum

Conclusion

The Two Pointers Technique, when combined with a stack-based approach and processing the temperature list from right to left, offers an **efficient and optimal solution** to the Daily Temperatures problem⁵¹. By leveraging the sorted nature of the stack and maintaining information about future warmer days, the algorithm efficiently determines the required wait times without unnecessary computations⁵². Mastering this technique not only enhances problem-solving skills but also prepares you for tackling more complex algorithmic challenges effectively.

⁵¹ This method ensures linear time complexity while effectively managing necessary information about future temperatures

⁵² This approach demonstrates the power of combining multiple algorithmic strategies for enhanced performance

Problem 15.15 Number of Visible People in a Queue

The "Number of Visible People in a Queue" problem is a challenging algorithmic question that involves determining how many people each person in a queue can see to their right. This problem requires a good understanding of stack data structures and the right-to-left traversal technique to efficiently compute the visibility for each person in the queue.

Problem Statement

You are given an array `heights` of distinct integers where `heights[i]` represents the height of the i th person in a queue. A person can see another person to their right in the queue if everybody in between is shorter than both of them. More formally, the i th person can see the j th person if $i < j$ and $\min(\text{heights}[i], \text{heights}[j]) > \max(\text{heights}[i + 1], \text{heights}[i + 2], \dots, \text{heights}[j - 1])$.

Input: An array `heights` of distinct integers representing the heights of people in a queue.

Output: An array `answer` where each element represents the number of people the i th person can see to their right in the queue.

Example:

Input: `heights = [10,6,8,5,11,9]`

Output: `[3,1,2,1,1,0]`

Explanation:

- Person 0 can see person 1, 2, and 4.
- Person 1 can see person 2.
- Person 2 can see person 3 and 4.
- Person 3 can see person 4.
- Person 4 can see person 5.
- Person 5 can see no one since nobody is to the right.

Algorithmic Approach

The optimal solution to this problem involves traversing the array from right to left while using a stack to keep track of people who can still see others. The idea is to determine, for each person, how many people they can see to their right by processing each person in reverse order.

- Start by initializing an empty stack and an array `answer` filled with zeros.
- Traverse the `heights` array from the last person to the first.
- For each person, count how many shorter people they can see by popping from the stack until encountering someone taller.
- The number of people popped from the stack before encountering a taller person is the number of people the current person can see.
- Push the current person onto the stack and move to the previous person in the queue.

Complexities

- **Time Complexity:** The time complexity is $O(n)$ because each person in the queue is processed once, and each height is pushed and popped from the stack at most once.
- **Space Complexity:** The space complexity is $O(n)$ for the stack used to track people visible to the current person.

Python Implementation

Below is the Python code to implement the "Number of Visible People in a Queue" problem using the right-to-left traversal and stack technique:

```
def canSeePersonsCount(heights):
    n = len(heights)
    answer = [0] * n
    stack = []

    for i in range(n - 1, -1, -1):
        while stack and heights[i] > heights[stack[-1]]:
            stack.pop()
            answer[i] += 1
        if stack:
            answer[i] += 1
        stack.append(i)

    return answer

# Example usage:
heights = [10, 6, 8, 5, 11, 9]
print(canSeePersonsCount(heights)) # Output: [3, 1, 2, 1, 1, 0]
```

Why this approach

This approach is effective because it uses a stack to efficiently determine the number of people each person can see by maintaining a dynamic list of people who are visible. By processing from right to left, we ensure that we can immediately determine visibility without having to revisit any previously processed people.

Similar problems to this one

Similar problems that involve determining visibility or next greater elements using a stack include "Daily Temperatures" and "Next Greater Element II." These problems also require processing elements based on their relative order or size.

Things to keep in mind and tricks

When solving visibility problems like this, it is crucial to ensure that the stack is used to track people or elements in the correct order. Additionally, handle edge cases where no one is visible by ensuring that the stack is correctly initialized and processed.

15.3 Intervals

Intervals are fundamental in array problems, especially those involving ranges, scheduling, and overlapping segments. An **interval** is a pair of numbers representing the start and end points on a number line, often denoted as $[a, b]$, where a is the start point and b is the end point.

Understanding intervals is crucial for solving a variety of array problems involving ranges.

15.3.1 Types of Intervals

- **Closed Interval** $[a, b]$: Includes both endpoints a and b .
- **Open Interval** (a, b) : Excludes both endpoints a and b .
- **Half-Open Interval** $[a, b)$ or $(a, b]$: Includes one endpoint and excludes the other.

Choosing the correct interval type is essential for accurately representing ranges in problems.

15.3.2 Key Concepts

- **Overlapping Intervals**: Two intervals overlap if they share any common points.
- **Merging Intervals**: Combining overlapping intervals into a single interval.
- **Interval Scheduling**: Selecting a subset of non-overlapping intervals from a set.

Efficient handling of intervals often involves sorting and merging operations.

15.3.3 Important Considerations

- **Sorting Intervals**: Often, intervals are sorted based on their start or end points to simplify processing.
- **Edge Cases**: Pay attention to intervals that share endpoints or are adjacent but not overlapping.
- **Data Structures**: Using appropriate data structures like arrays, lists, or priority queues can optimize interval operations.
- **Time Complexity**: Operations like merging intervals can be optimized to $O(n \log n)$ time by sorting first.

Handling edge cases correctly is critical for accurate interval manipulation.

15.3.4 Common Problems Involving Intervals

- **Merge Intervals**: Given a collection of intervals, merge all overlapping intervals.
- **Insert Interval**: Insert a new interval into a set of non-overlapping intervals and merge if necessary.

- **Interval Intersection:** Find the intersection between two lists of intervals.
- **Meeting Rooms:** Determine if a person could attend all meetings based on interval overlaps.
- **Minimum Number of Arrows to Burst Balloons:** Find the minimum number of arrows required to burst all balloons represented as intervals.

Familiarity with common interval problems enhances problem-solving skills in array-related challenges.

15.3.5 Techniques for Solving Interval Problems

- **Greedy Algorithms:** Often used in interval scheduling and optimization problems.
- **Sweep Line Algorithm:** Processes events in order, useful for detecting overlaps.
- **Binary Search:** Applicable when intervals are sorted and need to be searched efficiently.

Choosing the right algorithmic approach is key to efficient interval problem-solving.

15.3.6 Best Practices

- **Visualization:** Drawing intervals can help in understanding overlaps and adjacencies.
- **Testing Edge Cases:** Always test with intervals that have the same start or end points.
- **Consistent Representation:** Stick to a consistent interval representation throughout the problem.

Visualization aids in comprehending complex interval interactions.

15.4 Conclusion

Understanding intervals and their properties is essential when dealing with array problems involving ranges and scheduling. By keeping these important considerations in mind, one can efficiently solve complex problems and avoid common pitfalls associated with interval manipulation.

Mastery of intervals is crucial for advanced array problem-solving.

Problem 15.16 Meeting Rooms

Meeting Rooms problems are a staple in coding interviews, testing one's ability to analyze interval data and apply sorting or heap-based algorithms. There are two classic variations of this problem: Meeting Rooms I (checking if a person can attend all meetings) and Meeting Rooms II (determining the minimum number of conference rooms needed). **Problem Description:**

This problem involves interval scheduling and overlap detection.

Given an array of meeting time intervals consisting of start and end times $[[s_1, e_1], [s_2, e_2], \dots]$ where $s_i < e_i$, determine if a person could attend all meetings.

Example 1:

- **Input:** $[[0, 30], [5, 10], [15, 20]]$
- **Output:** False
- **Explanation:** Meetings $[0, 30]$ and $[5, 10]$ overlap.

Example 2:

- **Input:** $[[7, 10], [2, 4]]$
- **Output:** True
- **Explanation:** Meetings do not overlap.

Solution Overview:

Sort the intervals based on their start times. Then, iterate through the sorted intervals and check if the end time of the current interval is greater than the start time of the next interval. If so, return False as the meetings overlap. If no overlaps are found, return True.

```
def canAttendMeetings(intervals):
    intervals.sort(key=lambda x: x[0])  # Sort intervals
    ↪ by start time
    for i in range(1, len(intervals)):
        if intervals[i-1][1] > intervals[i][0]:
            return False
    return True

# Example usage:
print(canAttendMeetings([[0,30],[5,10],[15,20]]))  #
    ↪ Output: False
print(canAttendMeetings([[7,10],[2,4]]))           #
    ↪ Output: True
```

Problem 15.17 Meeting Rooms II

Problem Description:

Given an array of meeting time intervals consisting of start and end times $[[s_1, e_1], [s_2, e_2], \dots]$ where $s_i < e_i$, find the minimum number of conference rooms required.

This problem involves finding the minimum number of meeting rooms required.

Example 1:

- **Input:** `[[0,30], [5,10], [15,20]]`
- **Output:** 2

Example 2:

- **Input:** `[[7,10], [2,4]]`
- **Output:** 1

Solution Overview:

Separate the start and end times, sort them, and use two pointers to iterate through them. Keep track of the number of rooms needed at each point in time.

Problem 15.18 Insert Interval

Problem Description:

This problem tests interval insertion and merging techniques.

Given a list of non-overlapping intervals sorted by their start times, where each interval is represented as `intervals[i] = [start_i, end_i]`, and a new interval `newInterval = [start, end]`, insert `newInterval` into the list of intervals. Ensure that the list remains sorted by start times and that all overlapping intervals are merged appropriately.

Example 1:

- **Input:** `intervals = [[1,3], [6,9]], newInterval = [2,5]`
- **Output:** `[[1,5], [6,9]]`
- **Explanation:** The new interval `[2,5]` overlaps with `[1,3]`, merging to `[1,5]`.

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:** The new interval `[4,8]` overlaps with `[3,5], [6,7], [8,10]`, merging to `[3,10]`.

Solution Overview:

The key idea is to iterate through the intervals and:

1. Add all intervals that end before the start of the new interval to the output list.
2. Merge all intervals that overlap with the new interval by updating the start and end of the new interval.
3. Add the merged new interval to the output list.
4. Add the remaining intervals that start after the end of the new interval to the output list.

Implementation Details:

```
def insert(intervals, newInterval):
    output = []
    i = 0
    n = len(intervals)

    # Add all intervals ending before newInterval starts
    while i < n and intervals[i][1] < newInterval[0]:
        output.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
    output.append(newInterval)

    # Add remaining intervals
    while i < n:
        output.append(intervals[i])
        i += 1

    return output

# Example usage:
print(insert([[1,3],[6,9]], [2,5]))  # Output: [[1,5],[6,9]]
print(insert([[1,2],[3,5],[6,7],[8,10],[12,16]], [4,8]))  # Output:
    ↪ [[1,2],[3,10],[12,16]]
```

Complexities:

- **Time Complexity:** $O(n)$, where n is the number of intervals.
- **Space Complexity:** $O(n)$, for the output list.

Corner Cases to Test:

- Empty intervals list.
- New interval does not overlap with any existing intervals.
- New interval overlaps with all intervals.
- New interval is before all existing intervals.
- New interval is after all existing intervals.

Handling intervals efficiently is crucial for scheduling and timeline-based problems.

Problem 15.19 Merge Intervals

The **Merge Intervals** problem is a quintessential algorithmic challenge that tests one's ability to manipulate and process interval data efficiently. It involves consolidating overlapping intervals into a minimal set of non-overlapping intervals, a task fundamental in various applications such as scheduling, computational geometry, and resource allocation⁵³.

⁵³ Efficient interval merging ensures optimal utilization of resources and prevents scheduling conflicts

Problem Statement

Given a collection of intervals, merge all overlapping intervals and return an array of the non-overlapping intervals that cover all the intervals in the input⁵⁴.

⁵⁴ Merging intervals helps in simplifying data representation by consolidating overlapping periods

Example:

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]`.

Algorithmic Approach

The **Merge Intervals** problem can be efficiently solved by following a systematic approach that leverages the sorting of intervals and iterative consolidation⁵⁵.

⁵⁵ Sorting is essential as it brings overlapping intervals adjacent to each other, facilitating easier merging

1. **Sort the Intervals:** Begin by sorting the list of intervals based on their start times⁵⁶.
2. **Initialize the Merged List:** Create a list to hold the merged intervals, starting with the first interval from the sorted list⁵⁷.
3. **Iterate Through Intervals:** Traverse the sorted list of intervals and compare each interval with the last interval in the merged list.

⁵⁶ Sorting ensures that intervals are ordered sequentially, allowing for the detection of overlaps in a single pass

⁵⁷ This list will be updated with merged intervals as overlapping intervals are identified

- **Check for Overlap:** If the current interval overlaps with the last merged interval (i.e., the start of the current interval is less than or equal to the end of the last merged interval), merge them by updating the end of the last merged interval to be the maximum of both ends⁵⁸.
 - **No Overlap:** If there is no overlap, append the current interval to the merged list⁵⁹.
4. **Completion:** After processing all intervals, the merged list will contain all consolidated, non-overlapping intervals covering the entire range of input intervals⁶⁰.

⁵⁸ This step consolidates overlapping intervals into a single interval that spans their combined range

⁵⁹ Non-overlapping intervals are added as separate entries, maintaining the integrity of distinct periods

⁶⁰ This ensures that all original intervals are accounted for without redundancies

Python Implementation

```
class Solution:
    def merge(self, intervals: List[List[int]]) -> List[List[int]]:
        if not intervals:
            return []

        # Sort the intervals based on the start time
        intervals.sort(key=lambda x: x[0])

        merged = [intervals[0]]

        for current in intervals[1:]:
            prev = merged[-1]
            if current[0] <= prev[1]:
                # Overlapping intervals, merge them
                merged[-1][1] = max(prev[1], current[1])
            else:
                # Non-overlapping interval, add to the list
                merged.append(current)

        return merged

# Example usage:
intervals = [[1,3],[2,6],[8,10],[15,18]]
solution = Solution()
print(solution.merge(intervals)) # Output: [[1,6],[8,10],[15,18]]
```

Example Usage and Test Cases

```
# Test case 1: General case
intervals = [[1,3],[2,6],[8,10],[15,18]]
print(Solution().merge(intervals)) # Output:
                                → [[1,6],[8,10],[15,18]]

# Test case 2: No overlapping intervals
```

```

intervals = [[1,2],[3,4],[5,6]]
print(Solution().merge(intervals)) # Output:
→ [[1,2],[3,4],[5,6]]

# Test case 3: All intervals overlapping
intervals = [[1,5],[2,6],[3,7],[4,8]]
print(Solution().merge(intervals)) # Output: [[1,8]]

# Test case 4: Single interval
intervals = [[1,4]]
print(Solution().merge(intervals)) # Output: [[1,4]]

# Test case 5: Intervals with same start and end
intervals = [[1,3],[1,3],[1,3]]
print(Solution().merge(intervals)) # Output: [[1,3]]

```

Why This Approach

The sorting-based approach combined with iterative merging is chosen for its **efficiency and simplicity**. By sorting the intervals, we ensure that any overlapping intervals are positioned consecutively⁶¹. This method reduces the problem's complexity by enabling a single pass through the sorted intervals to perform all necessary merges⁶².

⁶¹ This eliminates the need for nested comparisons, allowing for a streamlined merging process

⁶² The algorithm operates in linear time relative to the number of intervals after sorting, which is optimal for this problem

Complexity Analysis

- **Time Complexity:** $O(n \log n)$ ⁶³.
- **Space Complexity:** $O(n)$ ⁶⁴.

⁶³ The sorting step dominates the time complexity, while the merging process operates in linear time

⁶⁴ In the worst case, where no intervals overlap, the space required for the merged list is proportional to the number of input intervals

Similar Problems

Other interval-related problems that can be efficiently solved using sorting and merging techniques include:

- **Non-overlapping Intervals:** Find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping⁶⁵.
- **Insert Interval:** Insert a new interval into a list of non-overlapping intervals and merge if necessary⁶⁶.
- **Meeting Rooms:** Determine if a person could attend all meetings given their schedules⁶⁷.
- **Interval List Intersections:** Find the intersection between two lists of intervals⁶⁸.

⁶⁵ This problem is a variant of the Merge Intervals problem and can be approached similarly with sorting and greedy strategies

⁶⁶ Requires identifying the correct position for insertion and merging overlapping intervals as needed

⁶⁷ By sorting the intervals and checking for overlaps, we can efficiently assess meeting overlaps

⁶⁸ Involves iterating through both sorted lists and identifying overlapping regions

These problems leverage the foundational principles of sorting and merging, demonstrating the versatility and power of these techniques in various algorithmic contexts⁶⁹.

⁶⁹ Mastering these approaches provides a strong foundation for solving a wide range of interval-related problems

Things to Keep in Mind and Tricks

- **Sorting Key:** Always sort the intervals based on their start times to ensure that overlapping intervals are positioned consecutively⁷⁰.
- **Handling Edge Cases:** Ensure that the algorithm correctly handles edge cases such as empty input, single interval, and all intervals overlapping⁷¹.
- **In-Place Merging:** If space is a constraint, consider modifying the input list in place to store merged intervals⁷².
- **Max Function:** Utilize the ‘max’ function to determine the furthest end point when merging overlapping intervals⁷³.
- **Greedy Approach:** Adopt a greedy strategy by always merging the current interval with the last merged interval if they overlap⁷⁴.

⁷⁰ Incorrect sorting criteria can lead to missed overlaps and inaccurate merges

⁷¹ Robust handling of these scenarios prevents runtime errors and ensures correct results

⁷² This can reduce space overhead but may complicate the implementation

⁷³ This ensures that the merged interval accurately spans all overlapping regions

⁷⁴ This approach guarantees that the number of merged intervals is minimized

Corner and Special Cases to Test When Writing the Code

- **Empty List:** Ensure the function returns an empty list when given no intervals⁷⁵.
- **Single Interval:** Verify that a single interval is returned as is⁷⁶.
- **All Overlapping Intervals:** Test with all intervals overlapping to ensure they are merged into one⁷⁷.
- **No Overlapping Intervals:** Ensure that non-overlapping intervals are returned unchanged⁷⁸.
- **Mixed Overlapping and Non-Overlapping Intervals:** Test with a combination of overlapping and non-overlapping intervals⁷⁹.
- **Intervals with Same Start or End Points:** Verify that intervals with identical start or end points are handled correctly⁸⁰.

⁷⁵ Handles scenarios where no data is provided gracefully

⁷⁶ No merging is required in this case

⁷⁷ Confirms that the algorithm correctly consolidates fully overlapping intervals

⁷⁸ Checks the algorithm’s ability to recognize and preserve distinct intervals

⁷⁹ Validates the algorithm’s capability to handle complex merging scenarios

⁸⁰ Ensures accurate merging when intervals share boundaries

References

GeeksforGeeks Article: ⁸¹

⁸¹ Merge Intervals

LeetCode Problem: ⁸²

⁸² Merge Intervals

HackerRank Problem: ⁸³

⁸³ Merge Intervals

Conclusion

The **Merge Intervals** problem exemplifies how sorting an array can simplify complex algorithmic challenges by imposing a structured order on the data⁸⁴. By leveraging the sorted nature of the intervals and adopting a greedy strategy, the algorithm efficiently merges overlapping intervals with optimal time and space complexities⁸⁵. Mastering this technique not only enhances problem-solving skills but also provides a foundation for tackling more intricate interval-based challenges effectively.

⁸⁴ This structured approach enables efficient identification and consolidation of overlapping intervals

⁸⁵ Achieving $O(n \log n)$ time complexity through sorting and linear traversal demonstrates the effectiveness of this approach

Problem 15.20 Non-overlapping Intervals

The **Non-overlapping Intervals** problem is a variation of the interval scheduling and merging challenges. It requires determining the minimum number of intervals to remove from a collection so that the remaining intervals are non-overlapping. This problem is pivotal in optimizing resource allocation, such as meeting room scheduling, where minimizing conflicts is essential⁸⁶.

⁸⁶ Efficiently resolving interval overlaps ensures optimal utilization of resources and prevents scheduling conflicts

Problem Statement

Given a collection of intervals, find the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping⁸⁷.

⁸⁷ Removing the least number of intervals ensures maximum utilization of available intervals while eliminating overlaps

Example:

Input: [[1, 2], [2, 3], [3, 4], [1, 3]]

Output: 1

Explanation: Removing [1,3] leaves the rest of the intervals as non-overlapping.

Algorithmic Approach

To solve the Non-overlapping Intervals problem efficiently, a greedy algorithm combined with sorting is employed⁸⁸.

⁸⁸ Sorting intervals allows for the identification of overlaps in a structured manner, facilitating the optimal removal of intervals

1. **Sort the Intervals:** Begin by sorting the intervals based on their end times.⁸⁹

⁸⁹ Sorting by end times ensures that intervals with earlier end times are considered first, maximizing the number of non-overlapping intervals

2. **Initialize Tracking Variables:**

- **End Pointer:** Initialize a pointer to keep track of the end of the last added interval, starting with the first interval's end.⁹⁰
- **Removal Counter:** Initialize a counter to track the number of intervals removed⁹¹.

⁹⁰ This pointer helps in determining whether the current interval overlaps with the last non-overlapping interval

⁹¹ This counter will increment each time an overlapping interval is identified and removed

3. Iterate Through Intervals: Traverse the sorted list of intervals and compare each interval's start time with the end pointer.

- **Check for Overlap:** If the current interval's start time is less than the end pointer, an overlap exists.
 - **Increment Removal Counter:** Increment the removal counter as one of the overlapping intervals needs to be removed⁹².
 - **Update End Pointer:** Update the end pointer to the minimum of the current interval's end and the last end pointer⁹³.
- **No Overlap:** If there is no overlap, update the end pointer to the current interval's end⁹⁴.

4. Completion: After processing all intervals, the removal counter will reflect the minimum number of intervals that need to be removed to eliminate all overlaps.

⁹² Choosing which interval to remove is determined by the sorting order, favoring the interval with the earlier end time

⁹³ This ensures that the interval with the earliest end time is retained, allowing more intervals to fit without overlapping

⁹⁴ This sets the new reference point for subsequent interval comparisons

Python Implementation

```
def eraseOverlapIntervals(intervals):
    """
    Finds the minimum number of intervals to remove to make the rest non-
    ↪ overlapping.

    Parameters:
    intervals (List[List[int]]): A list of intervals represented as [start, end].

    Returns:
    int: The minimum number of intervals that need to be removed.
    """
    if not intervals:
        return 0

    # Sort the intervals based on the end time
    intervals.sort(key=lambda x: x[1])

    end = intervals[0][1]
    count = 0

    for current in intervals[1:]:
        if current[0] < end:
            # Overlapping interval, increment removal counter
            count += 1
            # Update end to the minimum end time to maximize non-overlapping
            ↪ intervals
            end = min(end, current[1])
        else:
            # No overlap, update the end pointer
            end = current[1]
```

```

    return count

# Example usage:
intervals = [[1,2],[2,3],[3,4],[1,3]]
print(eraseOverlapIntervals(intervals)) # Output: 1

```

Why This Approach

The greedy algorithm combined with sorting by end times is chosen for its **efficiency and optimality**. By sorting the intervals based on their end times, the algorithm ensures that intervals with earlier end times are prioritized, allowing for the maximum number of non-overlapping intervals to be selected⁹⁵. Consequently, the algorithm effectively identifies and removes the minimal number of overlapping intervals, achieving optimal performance with a time complexity of $O(n \log n)$ ⁹⁶.

⁹⁵ This strategy minimizes the chances of future overlaps by choosing the interval that leaves the earliest possible room for subsequent intervals

⁹⁶ The sorting step dominates the time complexity, while the subsequent traversal operates in linear time

Complexity Analysis

- **Time Complexity:** $O(n \log n)$ ⁹⁷.
- **Space Complexity:** $O(1)$ if the sorting is done in-place⁹⁸.

⁹⁷ This arises from the sorting step, which is the most time-consuming part of the algorithm

⁹⁸ Otherwise, it requires $O(n)$ space depending on the sorting algorithm used

Similar Problems

Other problems that can be efficiently solved using greedy strategies and sorting techniques include:

- **Interval Scheduling:** Select the maximum number of non-overlapping intervals⁹⁹.
- **Meeting Rooms:** Determine if a person could attend all meetings given their schedules¹⁰⁰.
- **Minimum Number of Arrows to Burst Balloons:** Find the minimum number of arrows needed to burst all balloons represented by intervals¹⁰¹.
- **Employee Free Time:** Find the common free time across all employees given their schedules¹⁰².

⁹⁹ Similar to Non-overlapping Intervals, but focuses on maximizing the number of intervals instead of minimizing removals

¹⁰⁰ Requires checking for overlaps in meeting times, which can be efficiently handled by sorting intervals

¹⁰¹ Involves identifying overlapping intervals to minimize the number of arrows used

¹⁰² Requires merging and identifying gaps between sorted intervals

These problems leverage the foundational principles of greedy algorithms and sorting, demonstrating the versatility and power of these techniques in various algorithmic contexts¹⁰³.

¹⁰³ Mastering these approaches provides a strong foundation for solving a wide range of interval-related problems

Things to Keep in Mind and Tricks

- **Sorting by End Time:** Always sort intervals based on their end times to ensure that the earliest finishing intervals are considered first¹⁰⁴.
- **Handling Overlaps:** Carefully manage overlapping intervals by retaining the one with the earliest end time and removing others¹⁰⁵.
- **Edge Cases:** Ensure that the algorithm correctly handles edge cases such as empty input, single interval, and all intervals overlapping¹⁰⁶.
- **In-Place Sorting:** If space is a concern, perform in-place sorting to reduce space overhead¹⁰⁷.
- **Incrementing the Counter:** Only increment the removal counter when an actual overlap is detected¹⁰⁸.

¹⁰⁴ This facilitates the selection of intervals that leave the most room for subsequent non-overlapping intervals

¹⁰⁵ This strategy maximizes the number of non-overlapping intervals and minimizes removals

¹⁰⁶ Robust handling of these scenarios prevents runtime errors and ensures correct results

¹⁰⁷ This can be achieved using sorting algorithms that do not require additional space, such as Heap Sort

¹⁰⁸ This ensures that the count accurately reflects the number of necessary removals

Corner and Special Cases to Test When Writing the Code

- **Empty List:** Ensure the function returns 0 when given no intervals¹⁰⁹.
- **Single Interval:** Verify that a single interval results in no removals¹¹⁰.
- **All Overlapping Intervals:** Test with all intervals overlapping to ensure that the algorithm correctly identifies the minimum number of removals¹¹¹.
- **No Overlapping Intervals:** Ensure that the algorithm returns 0 when no overlaps are present¹¹².
- **Mixed Overlapping and Non-Overlapping Intervals:** Test with a combination of overlapping and non-overlapping intervals¹¹³.
- **Intervals with Same Start or End Points:** Verify that intervals sharing start or end points are handled correctly¹¹⁴.

¹⁰⁹ Handles scenarios where no data is provided gracefully

¹¹⁰ No overlapping is possible with a single interval

¹¹¹ Confirms that the algorithm consolidates overlaps efficiently

¹¹² Checks the algorithm's ability to recognize and preserve distinct intervals

¹¹³ Validates the algorithm's capability to handle complex merging and removal scenarios

¹¹⁴ Ensures accurate identification of overlaps when intervals have identical boundaries

References

GeeksforGeeks Article: ¹¹⁵

¹¹⁵ Non-overlapping Intervals

LeetCode Problem: ¹¹⁶

¹¹⁶ Non-overlapping Intervals

HackerRank Problem: ¹¹⁷

¹¹⁷ Non-overlapping Intervals

Conclusion

The Non-overlapping Intervals problem exemplifies the effectiveness of greedy algorithms combined with sorting in optimizing interval-related challenges¹¹⁸. By prioritizing intervals with earlier end times, the algorithm maximizes the number of

¹¹⁸ This approach ensures that the minimum number of intervals are removed to eliminate all overlaps

non-overlapping intervals while minimizing removals¹¹⁹. Mastering this technique not only enhances problem-solving skills but also provides a foundation for tackling more intricate interval-based challenges effectively.

¹¹⁹ This strategy is both time-efficient and space-efficient, making it suitable for large datasets

Problem 15.21 Reconstruct Queue by Height

The **Reconstruct Queue by Height** problem involves arranging people in a queue based on their heights and the number of people in front of them who are taller. The challenge is to place each person such that the tallest individuals stand in front, and among those with the same height, the one with fewer taller people in front stands earlier.

This problem utilizes a greedy algorithm to efficiently reconstruct the queue based on height and position constraints.

Problem Statement

LeetCode link: Queue Reconstruction by Height

[LeetCode Link]
 [GeeksForGeeks Link]
 [HackerRank Link]
 [CodeSignal Link]
 [InterviewBit Link]
 [Educative Link]
 [Codewars Link]

Algorithmic Approach

Main Concept

The main idea to solve this problem is to sort the people by their heights in descending order. If two individuals have the same height, sort them by the ascending order of the number of people in front of them (the k value). Then, we iterate through this sorted list and place each person at the index specified by their k value.

This strategy works because by placing the taller individuals first, we ensure that when we place smaller individuals, they do not affect the count of taller individuals in front of the ones already placed.

Sorting by height ensures that taller people are placed first, maintaining the required constraints when inserting shorter individuals.

Complexities

- **Time Complexity:** $O(n^2)$, where n is the number of people. Sorting takes $O(n \log n)$ time, and insertion takes $O(k)$ time for each person where k could be at most n , resulting in a total time complexity of $O(n^2)$.
- **Space Complexity:** $O(n)$ for storing the output. Sorting is done in-place, and the additional space used is only for the output queue.

Python Implementation

Below are three implementations: the standard solution, an optimized version, and a solution that includes visualization:

Multiple implementation approaches demonstrate different trade-offs between readability, efficiency, and functionality.

```
from typing import List, Tuple
from collections import defaultdict

class Solution:
    def reconstructQueue(self, people: List[List[int]]) -> List[List[int]]:
        """Standard solution with clear implementation."""
        # Sort people by descending height and ascending k value
        people.sort(key=lambda x: (-x[0], x[1]))
        queue = []
        # Place each person to the queue based on their k value
        for person in people:
            queue.insert(person[1], person)
        return queue

    def reconstructQueueOptimized(self, people: List[List[int]]) -> List[List[int]]:
        """Optimized solution using bucket sort for same heights."""
        if not people:
            return []

        # Group people by height
        height_groups = defaultdict(list)
        heights = set()

        for h, k in people:
            height_groups[h].append(k)
            heights.add(h)

        # Sort heights in descending order
        heights = sorted(heights, reverse=True)

        # Initialize result array
        n = len(people)
        result = [None] * n
        positions = list(range(n))

        # Place people from tallest to shortest
        for height in heights:
            ks = sorted(height_groups[height])
            for k in ks:
                pos = positions.pop(k)
                result[pos] = [height, k]

        return result

    def reconstructQueueWithVisualization(self,
```

```

    people: List[List[int]]) -> Tuple[List[List[int]], List[str]]:
    """Returns both the solution and step-by-step visualization."""
    people.sort(key=lambda x: (-x[0], x[1]))
    queue = []
    steps = []

    for person in people:
        queue.insert(person[1], person)
        steps.append(f"Insert {person} at position {person[1]}: {queue}")

    return queue, steps

# Comprehensive test cases
def test_queue_reconstruction():
    solution = Solution()

    # Basic test
    people = [[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]
    assert solution.reconstructQueue(people) == \
        [[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

    # Edge cases
    assert solution.reconstructQueue([]) == []
    assert solution.reconstructQueue([[1,0]]) == [[1,0]]

    # Visualization example
    people = [[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]
    result, steps = solution.reconstructQueueWithVisualization(people)
    for step in steps:
        print(step)

```

Visual Explanation

Step	Person [h,k]	Action	Queue State
1	[7,0]	Insert at 0	[[7,0]]
2	[7,1]	Insert at 1	[[7,0], [7,1]]
3	[6,1]	Insert at 1	[[7,0], [6,1], [7,1]]
4	[5,0]	Insert at 0	[[5,0], [7,0], [6,1], [7,1]]
5	[5,2]	Insert at 2	[[5,0], [7,0], [5,2], [6,1], [7,1]]
6	[4,4]	Insert at 4	[[5,0], [7,0], [5,2], [6,1], [4,4], [7,1]]

Figure 15.5: Step-by-step reconstruction of queue for input [[7,0], [4,4], [7,1], [5,0], [6,1], [5,2]]

Implementation Variants

- **Standard Solution:**

- Simple and readable

- Good for interviews
- $O(n^2)$ time complexity
- **Optimized Solution:**
 - Uses bucket sort for same heights
 - Better memory usage
 - More complex implementation
- **Visualization Solution:**
 - Includes step-by-step tracking
 - Useful for debugging
 - Higher space complexity

Common Optimization Techniques

- **Bucket Sort:** Group people by height for faster processing
- **Position Tracking:** Maintain available positions list
- **Early Termination:** Stop when all positions are filled
- **Memory Reuse:** Modify input array when possible

Real-World Applications

- **Event Scheduling:** Arranging events with precedence constraints
- **Resource Allocation:** Organizing resources with dependencies
- **Network Packet Ordering:** Reconstructing packet sequences
- **Process Scheduling:** Managing process execution order

Explanation

After sorting the list of people based on the criteria, we initialize an empty queue. We then iterate over the sorted list of people and insert each person into the queue at the index given by their k value. Since the list is sorted, when we are inserting a person, all the people who are taller (or equal in height and lower in k -value) have already been placed, which satisfies the required condition of the k value representing the correct count of taller individuals in front.

Why This Approach

This greedy approach is chosen because it simplifies the problem. Once the people are sorted in the required order, the solution is straightforward to assemble. By sorting the people by height first, we ensure that everyone placed before a given person is either taller or the same height, which is a critical part of the given conditions. Using insertion based on the k value then guarantees the relative order for individuals of the same height.

Alternative Approaches

An alternative approach would be to start with an empty list and iterate over the people array, finding the correct position for each person based on their k value. This could do away with the need for sorting, but finding the correct position for each individual would be inefficient, leading to a similar or worse time complexity.

Similar Problems to This One

Similar problems involve ordering or scheduling with constraints, such as the classic interval scheduling problem, inserting intervals, or the task scheduler problem where tasks need to be scheduled based on cooldown periods (LeetCode 621).

Things to Keep in Mind and Tricks

It's crucial to do the sorting correctly, adhering to the problem's constraints. After sorting, it's just careful insertion. It's also worth noting that inserting elements at arbitrary positions in a list (or array) is not the most efficient operation due to the potential need for shifting elements. Still, in this case, it's acceptable given the problem's constraints.

Corner and Special Cases to Test When Writing the Code

- **Smallest and Largest Values:** Test with the smallest and largest possible values for heights and k .
- **Multiple People with Same Height:** Ensure that the sorting and insertion handle multiple people with the same height but different k values correctly.
- **Empty Input Array:** Test how the function behaves when given an empty input array.

- **Multiple Insertions at Front:** Test cases that require multiple insertions at the front of the queue to ensure correct ordering.
- **Single Person:** ‘people = [[height, k]]’ should return the same single-person list.
- **All People with k=0:** Everyone has no one in front; ensure that taller people are first.
- **Maximum Jump Length:** Ensure that the function can handle cases where k equals the current queue length.
- **Large Number of People:** Test with a large number of people to ensure the algorithm handles them efficiently within time constraints.

Problem 15.22 Task Scheduler

The **Task Scheduler** problem is a classic algorithmic challenge that involves determining the minimum number of time units required to execute a list of tasks given specific cooldown constraints. Each task is represented by a character, and the cooldown period n dictates the minimum number of time units that must pass before the same task can be executed again.

This problem utilizes a greedy algorithm and frequency counting to efficiently schedule tasks with cooldown constraints.

Problem Statement

Given a list of tasks represented by characters and a non-negative integer n representing the cooldown period between two identical tasks, return the least number of time units that the CPU will take to finish all the given tasks. The CPU can either execute a task or be idle in each unit of time. Each task takes exactly one unit of time.

Example 1:

```
Input: tasks = ["A", "A", "A", "B", "B", "B"], n = 2
Output: 8
Explanation: A -> B -> idle -> A -> B -> idle -> A -> B.
Total time units = 8.
```

Example 2:

```
Input: tasks = ["A", "A", "A", "B", "B", "B"], n = 0
Output: 6
Explanation: A -> A -> A -> B -> B -> B.
Total time units = 6.
```

[[LeetCode Link](#)]
[GeeksForGeeks Link](#)
[HackerRank Link](#)
[CodeSignal Link](#)
[InterviewBit Link](#)
[Educative Link](#)
[Codewars Link](#)

Algorithmic Approach

Main Concept

The main idea to solve this problem is to use a greedy algorithm combined with frequency counting of tasks. The steps are as follows:

1. **Frequency Count:** Count the frequency of each task.
2. **Identify Maximum Frequency:** Determine the task(s) with the highest frequency.
3. **Calculate Initial Slots:** Calculate the number of idle slots based on the highest frequency and cooldown period.
4. **Fill Idle Slots:** Assign remaining tasks to these idle slots to minimize the total time.
5. **Determine Final Answer:** The final answer is the maximum between the length of the tasks list and the total calculated slots.

Using frequency counts ensures that the most frequent tasks are scheduled optimally to minimize idle time.

Complexities

- **Time Complexity:** $O(N)$, where N is the number of tasks. Counting frequencies and iterating through the frequency list both take linear time.
- **Space Complexity:** $O(1)$ since the frequency array has a fixed size of 26 (for each uppercase English letter).

Python Implementation

Below are three implementations: the standard solution, a heap-based approach, and a solution with visualization:

Multiple implementation approaches demonstrate different trade-offs between readability, efficiency, and visualization.

```
from collections import Counter
from typing import List, Tuple
import heapq

class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        """Standard greedy solution with frequency counting."""
        if n == 0:
            return len(tasks)

        task_counts = Counter(tasks)
        max_freq = max(task_counts.values())
        max_freq_tasks = list(task_counts.values()).count(max_freq)

        part_count = max_freq - 1
        part_length = n - (max_freq_tasks - 1)
        empty_slots = part_count * part_length
        available_tasks = len(tasks) - max_freq * max_freq_tasks
        idles = max(0, empty_slots - available_tasks)

        return len(tasks) + idles

    def leastIntervalHeap(self, tasks: List[str], n: int) -> int:
        """Heap-based solution that simulates actual task execution."""
        if n == 0:
            return len(tasks)

        # Count task frequencies
        count = Counter(tasks)

        # Create max heap (-freq, task)
        heap = [(-freq, task) for task, freq in count.items()]
        heapq.heapify(heap)

        time = 0
        while heap:
            cycle = n + 1
            temp = []
            task_count = 0

            # Execute tasks in current cycle
            while cycle > 0 and heap:
                freq, task = heapq.heappop(heap)
                if freq < -1:
                    temp.append((freq + 1, task))
                task_count += 1
                cycle -= 1

            for item in temp:
                heapq.heappush(heap, item)
```

```

        cycle -= 1

    # Add back tasks that still have remaining frequency
    for item in temp:
        heapq.heappush(heap, item)

    # Add idle time if necessary
    time += n + 1 if heap else task_count

return time

def visualizeSchedule(self, tasks: List[str], n: int) -> Tuple[int, List[str]
    ↵ ]]:
    """Returns both the minimum intervals and the actual schedule."""
    if n == 0:
        return len(tasks), tasks

    count = Counter(tasks)
    heap = [(-freq, task) for task, freq in count.items()]
    heapq.heapify(heap)

    schedule = []
    time = 0

    while heap:
        cycle = n + 1
        temp = []

        while cycle > 0 and heap:
            freq, task = heapq.heappop(heap)
            schedule.append(task)
            if freq < -1:
                temp.append((freq + 1, task))
            cycle -= 1

        # Add idle slots if necessary
        while cycle > 0 and temp:
            schedule.append('idle')
            cycle -= 1

        for item in temp:
            heapq.heappush(heap, item)

    return len(schedule), schedule

# Example usage and visualization
def test_task_scheduler():
    solution = Solution()
    tasks = ["A", "A", "A", "B", "B", "B"]
    n = 2

```

```

# Test all implementations
min_time = solution.leastInterval(tasks, n)
min_time_heap = solution.leastIntervalHeap(tasks, n)
total_time, schedule = solution.visualizeSchedule(tasks, n)

print(f"Minimum intervals: {min_time}")
print(f"Actual schedule: {' -> '.join(schedule)}")

```

Visual Explanation

Time	Task	Remaining A	Remaining B	State
0	A	2	3	Execute A
1	B	2	2	Execute B
2	idle	2	2	Cooldown
3	A	1	2	Execute A
4	B	1	1	Execute B
5	idle	1	1	Cooldown
6	A	0	1	Execute A
7	B	0	0	Execute B

Figure 15.6: Step-by-step execution for tasks=[“A”, “A”, “A”, “B”, “B”, “B”], n=2

Implementation Variants

- **Frequency Counting:**

- Simple and intuitive
- $O(N)$ time complexity
- Best for interviews

- **Heap-based:**

- Simulates actual execution
- More flexible for modifications
- Good for real-time scheduling

- **Visualization:**

- Shows actual schedule
- Useful for debugging
- Good for understanding

Common Optimization Techniques

- **Early Termination:** Return early for n=0 or single task
- **Space Optimization:** Reuse arrays when possible
- **Cycle Detection:** Identify repeating patterns
- **Batch Processing:** Handle multiple tasks in cycles

Real-World Applications

- **CPU Scheduling:** Process execution with cooldown
- **Network Packet Processing:** Managing request intervals
- **Resource Management:** Allocating shared resources
- **Job Scheduling:** Batch processing with constraints

Explanation

The provided Python implementation defines a function ‘leastInterval‘ which takes a list of tasks and an integer ‘n‘ representing the cooldown period. Here’s a step-by-step breakdown of the implementation:

- **Edge Case Handling:**
 - If the cooldown period ‘n‘ is 0, there are no restrictions on task execution order, so the minimum time is simply the number of tasks.
- **Frequency Counting:**
 - Use ‘Counter‘ from the ‘collections‘ module to count the frequency of each task.
 - Identify ‘max-freq‘, the highest frequency among all tasks.
 - Determine ‘max-freq-tasks‘, the number of tasks that have this highest frequency.
- **Calculating Idle Slots:**
 - ‘part-count‘ is the number of full parts between the most frequent tasks, calculated as ‘max-freq - 1‘.
 - ‘part-length‘ is the length of each part, which is ‘n - (max-freq-tasks - 1)‘. This accounts for the fact that multiple tasks with the same highest frequency can fill some of the idle slots.

- ‘empty-slots’ is the total number of idle slots available, calculated as ‘part-count * part-length’.

- **Filling Idle Slots:**

- ‘available-tasks’ is the number of tasks that are not the most frequent ones.
- ‘idles’ is the number of idle slots that remain after filling with available tasks, calculated as ‘max(0, empty-slots - available-tasks)’.

- **Determining the Final Answer:**

- The minimum time required is the sum of the total number of tasks and the number of idle slots.

This approach ensures that tasks are scheduled in a way that minimizes idle time by prioritizing the most frequent tasks and efficiently utilizing available slots.

Why This Approach

The greedy algorithm is chosen for its efficiency and effectiveness in handling the problem’s constraints. By focusing on the most frequent tasks and strategically placing them to minimize idle time, this approach ensures an optimal schedule. It avoids the need for more complex methods like dynamic programming, resulting in a simpler and faster solution.

Alternative Approaches

An alternative approach involves using a priority queue (max heap) to always select the task with the highest remaining frequency that can be executed next, adhering to the cooldown constraint. This method can also achieve $O(N)$ time complexity but may have a higher constant factor due to heap operations. Additionally, dynamic programming techniques can be employed, but they tend to be more complex and less intuitive for this specific problem.

Similar Problems to This One

Similar problems involve scheduling tasks with specific constraints, such as:

- Rearrange String k Distance Apart
- Task Scheduler II
- Minimum Window Substring

- Reorganize String
- CPU Task Scheduler

Things to Keep in Mind and Tricks

When solving scheduling problems with cooldown or distance constraints:

- **Frequency Counts:** Always consider the frequency of each task, as the most frequent tasks often determine the scheduling constraints.
- **Greedy Placement:** Prioritize scheduling the most frequent tasks first to minimize idle time.
- **Idle Slot Calculation:** Carefully calculate idle slots based on the number of tasks and their frequencies.
- **Handling Multiple Maximum Frequency Tasks:** When multiple tasks share the highest frequency, adjust your calculations to account for their combined effect on idle slots.
- **Final Answer Determination:** The answer is the maximum between the total number of tasks and the calculated idle-inclusive schedule length.

Corner and Special Cases to Test When Writing the Code

To ensure robustness, consider testing the following corner cases:

- **All Tasks Identical:** All tasks are the same, requiring maximum cooldown.
- **No Cooldown ($n = 0$):** Should return the length of the tasks list.
- **Multiple Tasks with Maximum Frequency:** Multiple tasks share the highest frequency, affecting idle slot calculations.
- **Cooldown Greater Than Tasks:** The cooldown period is larger than the number of tasks.
- **Single Task:** Only one task is present.
- **Empty Task List:** Should handle gracefully, possibly returning 0.
- **Large Number of Tasks:** Ensure the algorithm handles large inputs efficiently without exceeding time limits.
- **Tasks with High k Values:** Tasks require large cooldown periods relative to their frequencies.
- **Mixed Frequencies and Cooldowns:** A combination of tasks with varying frequencies and different cooldown requirements.

15.5 Precomputation

Precomputation is a powerful optimization technique in algorithm design that involves preparing auxiliary data structures to expedite the solution of specific queries. This technique is especially useful for problems where repeated operations, such as summation or multiplication of subarrays, are required. By leveraging precomputed values, we can significantly reduce the computational complexity of such problems, transforming potentially inefficient solutions into optimal ones¹²⁰.

¹²⁰ Precomputation trades off additional memory usage for faster query response times, a common trade-off in algorithmic optimization

When to Use Precomputation

Precomputation is most effective in scenarios where:

- The problem involves repetitive calculations over subarrays, such as sums, products, or differences.
- The number of queries is large, and recalculating the result for each query would be computationally expensive.
- Extra memory usage for auxiliary data structures like prefix sums, suffix sums, or hash tables is acceptable¹²¹.
- Query time needs to be minimized, often at the expense of preprocessing time¹²².

¹²¹ Ensure that memory constraints of the problem allow for additional data structures

¹²² Precomputation emphasizes optimizing runtime at the cost of upfront preprocessing effort

Common Applications of Precomputation

Precomputation can be applied to a wide range of algorithmic problems. Below are some common patterns:

1. Prefix Sum:

- Stores cumulative sums up to each index in the array.
- Useful for quickly calculating the sum of any subarray $[i, j]$ using the formula:

$$\text{sum}(i, j) = \text{prefix}[j] - \text{prefix}[i - 1]$$

¹²³.

¹²³ This reduces the time complexity of each query from $O(n)$ to $O(1)$

2. Suffix Sum:

- Stores cumulative sums from the end of the array to each index.
- Complements prefix sums for problems requiring both forward and backward traversal.

3. Prefix Product and Suffix Product:

- Similar to prefix and suffix sums, but for products. Used in problems like "Product of Array Except Self"¹²⁴.

¹²⁴ LeetCode Problem: Product of Array Except Self

4. Hash Maps for Frequency Counting:

- Precompute frequencies of elements or cumulative counts to quickly answer range queries, such as "How many times does a number appear between indices i and j ?".

5. Prefix XOR or Bitwise Operations:

- Used for problems involving cumulative XOR or other bitwise operations over subarrays.
- Enables efficient computation of XOR for any subarray $[i, j]$ using:

$$\text{XOR}(i, j) = \text{prefix_xor}[j] \oplus \text{prefix_xor}[i - 1]$$

¹²⁵.

¹²⁵ This is particularly useful in problems involving parity or flipping operations

Problem 15.23 Product of Array Except Self

The "Product of Array Except Self" problem is a classic problem in the realm of algorithm and data structure problems often encountered on platforms like LeetCode.

Problem Statement

Given an array `nums` of n integers where $n > 1$, return an array `output` such that `output[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

Given the constraints, it's guaranteed that the product of the array elements is within the range of a 32-bit signed integer.

Examples

Example 1:

Input: [1, 2, 3, 4]

Output: [24, 12, 8, 6]

Explanation: By calculating the product of all elements except for the one at the current index, you would get:

- For index 0: $(2 \times 3 \times 4) = 24$
- For index 1: $(1 \times 3 \times 4) = 12$

- For index 2: $(1 \times 2 \times 4) = 8$
- For index 3: $(1 \times 2 \times 3) = 6$

Note:

- You cannot use division in solving this problem.
- You should try to solve the problem in $O(n)$ time complexity.

Algorithmic Approach

To solve this problem, we utilize two arrays to store the prefix and suffix products of every element. We then multiply these prefix and suffix products to obtain the final output.

Complexities

- **Time Complexity:** The total time complexity of the solution is $O(n)$ as we are iterating through the array a constant number of times.
- **Space Complexity:** The space complexity is $O(n)$ due to the extra space taken up by the prefix and suffix arrays.

Python Implementation

Below is the complete Python code for solving the "Product of Array Except Self" problem without using division and with linear time complexity:

```
class Solution:
    def productExceptSelf(self, nums):
        length = len(nums)

        # Initialize arrays for left and right products
        left_products = [0]*length
        right_products = [0]*length
        output = [0]*length

        # left_products[i] contains the product of all elements to the left of i
        left_products[0] = 1
        for i in range(1, length):
            left_products[i] = nums[i - 1] * left_products[i - 1]

        # right_products[i] contains the product of all elements to the right of i
        right_products[length - 1] = 1
        for i in reversed(range(length - 1)):
```

```

    right_products[i] = nums[i + 1] * right_products[i + 1]

    # Construct the output array
    for i in range(length):
        output[i] = left_products[i] * right_products[i]

    return output

```

By precalculating the product of elements to the left and right of every element, we can construct the output array without the use of division.

Why this approach

This approach is chosen because it adheres to the constraints of not using division and maintains a linear time complexity. By keeping track of the products to the left and right separately, we can multiply these values together to get the desired result for each index.

Alternative approaches

An alternative approach could have been directly calculating the total product and then dividing it by the current element, but the problem explicitly forbids the use of division. Additionally, using extra passes and division might have exceeded the desired linear time complexity.

Similar problems to this one

Similar problems may include other array transformation challenges that involve prefix sums or products, or problems that require the efficient computation of cumulative properties without using direct division, such as "Maximum Product Subarray" or "Trapping Rain Water."

Things to keep in mind and tricks

- Precomputation of cumulative quantities can lead to efficient algorithms. - When division is not allowed, consider the use of prefix and suffix arrays to maintain state.
- Always consider the constraints and desired time complexity when designing a solution. - Remember to check for edge cases, such as an array containing zeros or negative numbers.

Corner and special cases to test when writing the code

While testing, consider arrays with:

- A single zero, multiple zeros, and no zeros.
- Negative numbers, as they can affect the product sign.
- Large lengths to ensure that the time complexity is indeed linear.
- Edge cases, such as an empty array or an array with a single element.

Problem 15.24 Range Sum Query - Immutable

The **Range Sum Query - Immutable** problem involves processing multiple range sum queries on a static array. The challenge is to efficiently compute the sum of elements between two indices *left* and *right* inclusive, multiple times, after an initial preprocessing step.

This problem utilizes a prefix sum technique to efficiently calculate range sums in constant time after initial preprocessing.

Problem Statement

Given an integer array `nums`, handle multiple queries of the following type:

- `sumRange(left, right)`: Return the sum of the elements of `nums` between indices `left` and `right` inclusive (i.e., `nums[left] + nums[left + 1] + ... + nums[right]`).

Implement the `NumArray` class:

- `NumArray(int[] nums)` Initializes the object with the integer array `nums`.
- `int sumRange(int left, int right)` Returns the sum of the elements of `nums` between indices `left` and `right` inclusive.

Example 1:

Input:

```
["NumArray", "sumRange", "sumRange", "sumRange"]
[[[-2, 0, 3, -5, 2, -1]], [0, 2], [2, 5], [0, 5]]
```

Output:

```
[null, 1, -1, -3]
```

Explanation:

```
NumArray numArray = new NumArray([-2,0,3,-5,2,-1]);
numArray.sumRange(0, 2); // return (-2) + 0 + 3 = 1
numArray.sumRange(2, 5); // return 3 + (-5) + 2 + (-1) = -1
numArray.sumRange(0, 5); // return (-2) + 0 + 3 + (-5) + 2 + (-1) = -3
```

LeetCode link: Range Sum Query - Immutable

[[LeetCode Link](#)]
[[GeeksForGeeks Link](#)]
[[HackerRank Link](#)]
[[CodeSignal Link](#)]
[[InterviewBit Link](#)]
[[Educative Link](#)]
[[Codewars Link](#)]

Algorithmic Approach

Main Concept

The main idea to solve this problem is to use a **prefix sum** technique. By preprocessing the array to compute the cumulative sum up to each index, we can answer each range sum query in constant time. Here's a step-by-step breakdown:

1. Prefix Sum Calculation:

- Create a prefix sum array `prefix_sums` where `prefix_sums[i]` represents the sum of elements from index 0 to $i - 1$ in the original array.
- Initialize `prefix_sums[0]` to 0.
- For each index i from 1 to n (where n is the length of `nums`), compute `prefix_sums[i]` as `prefix_sums[i-1] + nums[i-1]`.

2. Handling Queries:

- For each `sumRange(left, right)` query, compute the sum as `prefix_sums[right + 1] - prefix_sums[left]`.

This approach ensures that after an initial $O(n)$ preprocessing step, each query can be answered in $O(1)$ time.

Prefix sums allow for efficient computation of range sums by leveraging precomputed cumulative totals.

Complexities

• Time Complexity:

- **Initialization:** $O(n)$, where n is the number of elements in `nums`, for computing the prefix sums.
- **sumRange Queries:** $O(1)$ per query, since each query involves a simple subtraction operation.

• Space Complexity:

- $O(n)$ for storing the prefix sums in the `prefix_sums` array.

Python Implementation

Below is the complete Python code for the ‘NumArray’ class, which implements the ‘sumRange’ method to calculate the sum of elements between two indices:

Implementing the prefix sum technique allows for swift range sum computations by utilizing preprocessed cumulative sums.

```
class NumArray:
    def __init__(self, nums):
        """
        :type nums: List[int]
        """
        self.prefix_sums = [0]
        for num in nums:
            self.prefix_sums.append(self.prefix_sums[-1] + num)

    def sumRange(self, left, right):
        """
        :type left: int
        :type right: int
        :rtype: int
        """
        return self.prefix_sums[right + 1] - self.prefix_sums[left]
```

Explanation

The provided Python implementation defines a class `NumArray` which efficiently handles range sum queries using the prefix sum technique. Here’s a detailed breakdown of the implementation:

- **Initialization (`__init__` method):**

- **Prefix Sum Array:** Initialize `self.prefix_sums` with a starting value of 0.
- **Cumulative Sum Calculation:** Iterate through each number in the input `nums` list and append the sum of the current number with the last element in `self.prefix_sums` to build the prefix sum array.

- **Range Sum Query (`sumRange` method):**

- **Sum Calculation:** For each query with indices `left` and `right`, compute the sum of the range by subtracting `self.prefix_sums[left]` from `self.prefix_sums[right + 1]`.
- **Return Value:** The result is returned as an integer representing the sum of the specified range.

This implementation ensures that after an initial preprocessing step, each range sum query is answered in constant time, making it highly efficient for multiple queries.

Why This Approach

The **prefix sum** approach is chosen for its simplicity and efficiency. By precomputing the cumulative sums of the array, range sum queries can be answered in $O(1)$ time, which is optimal for scenarios with multiple queries. This method avoids redundant computations and leverages the benefits of dynamic programming by building up solutions to larger problems based on previously computed results.

Alternative Approaches

An alternative approach involves using a **Segment Tree** or a **Binary Indexed Tree (Fenwick Tree)** to handle range sum queries. These data structures allow for dynamic updates and range queries in $O(\log n)$ time. However, for the **Range Sum Query - Immutable** problem, where the array does not change after initialization, the prefix sum method is more straightforward and efficient in terms of both implementation and runtime.

Another possibility is to handle each query individually by iterating through the specified range and summing the elements. However, this brute-force approach results in $O(n)$ time per query, which is inefficient for a large number of queries.

Similar Problems to This One

Similar problems involve range queries and efficient data retrieval from arrays or matrices, such as:

- Range Sum Query 2D - Immutable
- Range Sum Query - Mutable
- Subarray Sum Equals K
- Maximum Subarray
- Finding Subarray Sum

Things to Keep in Mind and Tricks

When solving range query problems, consider the following tips:

- **Preprocessing:** Utilize preprocessing techniques like prefix sums to reduce query time.

- **Space-Time Tradeoff:** Balance between additional space used for preprocessing and the time saved during queries.
- **Immutable vs. Mutable:** Choose appropriate data structures based on whether the array can change after initialization.
- **Edge Cases:** Always handle edge cases such as empty arrays, single-element arrays, and queries that span the entire array.
- **Efficient Data Structures:** When dealing with mutable arrays, consider advanced data structures like Segment Trees or Binary Indexed Trees.
- **Avoid Redundant Computations:** Use memoization or dynamic programming principles to store and reuse results.

Corner and Special Cases to Test When Writing the Code

To ensure robustness, consider testing the following corner cases:

- **Empty Array:** `nums = []`. Should handle gracefully, possibly returning 0 or raising an exception based on problem constraints.
- **Single Element Array:** `nums = [5]`. Queries like `sumRange(0, 0)` should return the single element.
- **All Negative Numbers:** `nums = [-1, -2, -3]`. Ensure that sums are computed correctly with negative values.
- **All Zeroes:** `nums = [0, 0, 0, 0]`. Sums should reflect the number of zeroes in the range.
- **Large Array with Large Numbers:** Test with large input sizes and large integer values to ensure no overflow issues and maintain performance.
- **Maximum Range Queries:** `sumRange(0, n-1)` where n is the size of the array. Should return the total sum of the array.
- **Overlapping Ranges:** Multiple queries with overlapping ranges to ensure independent and correct computations.
- **Invalid Queries:** Queries where `left > right` or indices are out of bounds. Should handle gracefully, possibly by returning 0 or raising exceptions.
- **Performance Testing:** Ensure that the implementation performs efficiently with a large number of queries (e.g., 10^5 queries).

Problem 15.25 Range Sum Query 2D - Immutable

The **Range Sum Query 2D - Immutable** problem involves handling multiple range sum queries on a 2D matrix. The challenge is to efficiently compute the sum of elements within a submatrix defined by its upper-left and lower-right corners, multiple times, after an initial preprocessing step.

This problem utilizes a prefix sum matrix to efficiently calculate range sums in constant time after initial preprocessing.

Problem Statement

Given a 2D matrix `matrix`, handle multiple queries of the following type:

- `**sumRegion**`(`row1, col1, row2, col2`): Return the sum of the elements of `matrix` inside the rectangle defined by its upper left corner (`row1, col1`) and lower right corner (`row2, col2`).

Implement the `NumMatrix` class:

- `NumMatrix(int[][] matrix)` Initializes the object with the integer matrix `matrix`.
- `int sumRegion(int row1, int col1, int row2, int col2)` Returns the sum of the elements of `matrix` inside the rectangle defined by its upper left corner (`row1, col1`) and lower right corner (`row2, col2`).

Example 1:

Input:

```
["NumMatrix","sumRegion","sumRegion","sumRegion"]
[[[3,0,1,4,2],
 [5,6,3,2,1],
 [1,2,0,1,5],
 [4,1,0,1,7],
 [1,0,3,0,5]],
 [2,1,4,3],
 [1,1,2,2],
 [1,2,2,4]]
```

Output:

```
[null, 8, 11, 12]
```

Explanation:

```
NumMatrix numMatrix = new NumMatrix([
    [3,0,1,4,2],
    [5,6,3,2,1],
    [1,2,0,1,5],
    [4,1,0,1,7],
    [1,0,3,0,5]
]);
```

```
numMatrix.sumRegion(2, 1, 4, 3); // return 11
numMatrix.sumRegion(1, 1, 2, 2); // return 11
numMatrix.sumRegion(1, 2, 2, 4); // return 12
```

LeetCode link: Range Sum Query 2D - Immutable

[[LeetCode Link](#)]
[GeeksForGeeks Link](#)
[HackerRank Link](#)
[CodeSignal Link](#)
[InterviewBit Link](#)
[Educative Link](#)
[Codewars Link](#)

Algorithmic Approach

Main Concept

The main idea to solve this problem is to use a **“prefix sum matrix”** (also known as a cumulative sum matrix). By preprocessing the matrix to compute the cumulative sums up to each cell, we can answer each range sum query in constant time. Here’s a step-by-step breakdown:

1. **“Prefix Sum Matrix Calculation:”**

- Create a prefix sum matrix `prefix_sums` of size $(m+1) \times (n+1)$, where m is the number of rows and n is the number of columns in the original matrix.
- Initialize `prefix_sums[0][*]` and `prefix_sums[*][0]` to 0 to handle edge cases.
- For each cell (i, j) in the original matrix, compute `prefix_sums[i+1][j+1]` as:

$$\text{prefix_sums}[i+1][j+1] = \text{prefix_sums}[i][j+1] + \text{prefix_sums}[i+1][j] - \text{prefix_sums}[i][j] + \text{matrix}[i][j]$$

2. **“Handling Queries:”**

- For each `sumRegion(row1, col1, row2, col2)` query, compute the sum of the rectangle by leveraging the prefix sums:

$$\text{sum} = \text{prefix_sums}[row2+1][col2+1] - \text{prefix_sums}[row1][col2+1] - \text{prefix_sums}[row2+1][col1] + \text{prefix_sums}[row1][col1]$$

This approach ensures that after an initial $O(m \times n)$ preprocessing step, each query can be answered in $O(1)$ time.

Prefix sum matrices enable rapid computation of submatrix sums by utilizing pre-computed cumulative totals.

Complexities

• Time Complexity:

- **Initialization:** $O(m \times n)$, where m is the number of rows and n is the number of columns, for computing the prefix sums.
- **sumRegion Queries:** $O(1)$ per query, since each query involves a constant number of arithmetic operations.

- **Space Complexity:**

- $O(m \times n)$ for storing the prefix sum matrix.

Python Implementation

Below is the complete Python code for the NumMatrix class, which implements the sumRange method to calculate the sum of elements within a specified submatrix:

```
class NumMatrix:
    def __init__(self, matrix):
        """
        :type matrix: List[List[int]]
        """
        if not matrix or not matrix[0]:
            self.prefix_sums = []
            return

        m, n = len(matrix), len(matrix[0])
        self.prefix_sums = [[0] * (n + 1) for _ in range(m + 1)]

        for i in range(m):
            for j in range(n):
                self.prefix_sums[i + 1][j + 1] = (
                    self.prefix_sums[i][j + 1] +
                    self.prefix_sums[i + 1][j] -
                    self.prefix_sums[i][j] +
                    matrix[i][j]
                )

    def sumRange(self, row1, col1, row2, col2):
        """
        :type row1: int
        :type col1: int
        :type row2: int
        :type col2: int
        :rtype: int
        """
        return (
            self.prefix_sums[row2 + 1][col2 + 1] -
            self.prefix_sums[row1][col2 + 1] -
            self.prefix_sums[row2 + 1][col1] +
            self.prefix_sums[row1][col1]
        )
```

Implementing the prefix sum matrix allows for efficient and rapid range sum computations by leveraging preprocessed cumulative sums.

Visual Explanation

Implementation Variants

- Standard Implementation:

```
|     class NumMatrix:
```

3	0	1	4	2
5	6	3	2	1
1	2	0	1	5
4	1	0	1	7
1	0	3	0	5

Figure 15.7: Visual representation of sum-Region(1,1,2,2) calculation

- **Memory-Optimized Implementation:**

```
class NumMatrix:
    def __init__(self, matrix):
        if not matrix or not matrix[0]:
            return
        m, n = len(matrix), len(matrix[0])
        self.sums = [[0] * n for _ in range(m)]
        # Calculate row-wise prefix sums
        for i in range(m):
            for j in range(n):
                self.sums[i][j] = (matrix[i][j] +
                                    (self.sums[i][j-1] if j > 0
                                     ↪ else 0))
```

- **Cache-Friendly Implementation:**

<code>matrix[i][j])</code>

Performance Analysis

- **Memory Usage:**
 - Standard: $O(m \times n)$ extra space
 - Memory-Optimized: $O(n)$ extra space
 - Cache-Friendly: $O(m \times n)$ with better cache utilization
- **Query Time:**
 - Standard: $O(1)$ per query
 - Memory-Optimized: $O(m)$ per query
 - Cache-Friendly: $O(m)$ per query with better cache hits

Common Pitfalls and Solutions

- **Integer Overflow:**
 - Problem: Large matrices can cause integer overflow
 - Solution: Use long integers or handle overflow cases
- **Memory Constraints:**
 - Problem: Large matrices require significant memory
 - Solution: Use row-wise prefix sums for space optimization
- **Cache Performance:**
 - Problem: Poor cache utilization in large matrices
 - Solution: Implement cache-friendly traversal patterns

Optimization Techniques

- **Matrix Partitioning:** Divide large matrices into blocks
- **Cache Blocking:** Optimize for CPU cache performance
- **SIMD Instructions:** Utilize vector operations when available
- **Parallel Processing:** Implement multi-threaded initialization

Similar Problems to This One

There are several other problems that involve range queries and efficient data retrieval from 2D matrices, such as:

- Range Sum Query - Mutable
- Maximum Submatrix Sum
- Number of Submatrices That Sum to Target
- Submatrix Sum Equals K
- Count Square Submatrices with All Ones

Things to Keep in Mind and Tricks

When solving range query problems, consider the following tips:

- **Preprocessing:** Utilize preprocessing techniques like prefix sums to reduce query time.
- **Space-Time Tradeoff:** Balance between additional space used for preprocessing and the time saved during queries.
- **Edge Cases:** Always handle edge cases such as empty matrices, single-element matrices, and queries that span the entire matrix.
- **Immutable vs. Mutable:** Choose appropriate data structures based on whether the matrix can change after initialization.
- **Efficient Data Structures:** When dealing with mutable matrices, consider advanced data structures like Binary Indexed Trees or Segment Trees.
- **Avoid Redundant Computations:** Use memoization or dynamic programming principles to store and reuse results.

Corner and Special Cases to Test When Writing the Code

To ensure robustness, consider testing the following corner cases:

- **Empty Matrix:** `matrix = []`. Should handle gracefully, possibly returning 0 or raising an exception based on problem constraints.
- **Single Element Matrix:** `matrix = [[5]]`. Queries like `sumRegion(0, 0, 0, 0)` should return the single element.

- **All Zeroes:** `matrix = [[0,0,0], [0,0,0]]`. Sums should reflect the number of zeroes in the range.
- **Negative Numbers:** `matrix = [[-1,-2,-3], [-4,-5,-6]]`. Ensure that sums are computed correctly with negative values.
- **Large Matrix with Mixed Values:** Test with large matrices containing a mix of positive, negative, and zero values to ensure correctness and performance.
- **Maximum Range Queries:** `sumRegion(0, 0, m-1, n-1)` where m and n are the dimensions of the matrix. Should return the total sum of the matrix.
- **Overlapping Ranges:** Multiple queries with overlapping ranges to ensure independent and correct computations.
- **Invalid Queries:** Queries where `row1 > row2` or `col1 > col2` or indices are out of bounds. Should handle gracefully, possibly by returning 0 or raising exceptions.
- **Performance Testing:** Ensure that the implementation performs efficiently with a large number of queries (e.g., 10^5 queries) and large matrix sizes.

Chapter 16

Trees and Graphs

Chapter 17

Trees

17.1 Introduction to Trees

A **tree** is a fundamental data structure that represents a hierarchical relationship between elements. Unlike linear data structures like arrays and linked lists, trees consist of nodes connected by edges, forming a branching structure. Each tree has a single root node, and every other node has exactly one parent node, except for the root, which has none.

Trees are hierarchical data structures essential in various algorithms and applications.

17.2 Properties of Trees

- **Root:** The topmost node in a tree.
- **Parent and Child:** In a tree, a node can have child nodes, and these children have the node as their parent.
- **Leaf Nodes:** Nodes with no children.
- **Height of a Tree:** The length of the longest path from the root to a leaf.
- **Depth of a Node:** The length of the path from the root to the node.
- **Subtree:** A tree consisting of a node and all its descendants.

Understanding tree properties is crucial for effective manipulation and traversal.

17.3 Binary Trees

A **binary tree** is a type of tree where each node has at most two children, referred to as the left child and the right child. Binary trees are the foundation for more specialized tree structures like binary search trees, AVL trees, and red-black trees.

Binary trees are the foundation for more specialized tree structures.

17.3.1 Types of Binary Trees

- **Full Binary Tree:** Every node has either 0 or 2 children. There are no nodes with only one child.
- **Complete Binary Tree:** All levels are fully filled except possibly the last level, and all nodes in the last level are as far left as possible.
- **Perfect Binary Tree:** All internal nodes have two children, and all leaf nodes are at the same level.

Different binary tree types serve various purposes and optimizations.

17.4 Binary Search Trees (BSTs)

A **binary search tree** (BST) is a binary tree with the following properties:

BSTs allow efficient searching, insertion, and deletion operations.

- The left subtree of a node contains only nodes with values less than the node's value.
- The right subtree of a node contains only nodes with values greater than the node's value.
- Both the left and right subtrees must also be binary search trees.

17.5 Tree Traversal

Tree traversal refers to the process of visiting each node in a tree in a specific order. Common traversal methods include:

Traversal methods are essential for accessing and processing tree data.

17.5.1 Depth-First Traversal

- **Inorder (Left, Root, Right):** Yields nodes in ascending order for BSTs.
- **Preorder (Root, Left, Right):** Useful for copying trees.
- **Postorder (Left, Right, Root):** Useful for deleting trees.

Depth-first traversals explore as far as possible along each branch before backtracking.

17.5.2 Breadth-First Traversal (Level Order)

- Visits nodes level by level from left to right.
- Utilizes a queue to keep track of nodes at the current level.

Breadth-first traversal is ideal for scenarios requiring level-wise processing.

17.6 Tree Operations

17.6.1 Insertion

- In a BST, insertion involves comparing the value to be inserted with the current node and recursively placing it in the left or right subtree.
- Ensures that the BST properties are maintained after insertion.

Understanding tree operations is crucial for maintaining BST properties.

17.6.2 Deletion

- Removing a node from a BST requires handling three cases:
 1. **Node with no children:** Simply remove the node.
 2. **Node with one child:** Remove the node and replace it with its child.
 3. **Node with two children:** Find the inorder successor (smallest in the right subtree) or inorder predecessor (largest in the left subtree), swap values, and delete the successor/predecessor node.

17.6.3 Searching

- Start at the root and compare the target value with the current node's value.
- Traverse left or right subtree based on comparison until the target is found or a leaf is reached.

17.7 Applications of Trees

Trees are versatile structures used in various applications:

Trees underpin many real-world systems and algorithms.

- **Databases:** Hierarchical storage and indexing.
- **File Systems:** Organizing files and directories.
- **Compiler Design:** Syntax trees for parsing expressions.
- **Networking:** Routing protocols use tree structures.
- **Artificial Intelligence:** Decision trees and game trees.

Problem 17.1 Binary Tree vs. Binary Search Tree

Introduction

Understand the key differences and use-cases of Binary Trees and Binary Search Trees (BSTs).

Binary Trees and Binary Search Trees (BSTs) are fundamental data structures in computer science, each serving distinct purposes and offering unique properties. Understanding the differences between them is crucial for selecting the appropriate tree structure based on the problem requirements.

Binary Tree

A **Binary Tree** is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. There are no specific ordering constraints on the node values in a binary tree.

- **Structure:**

- Each node has at most two children.
- No inherent ordering of node values.

- **Types of Binary Trees:**

1. **Full Binary Tree:** Every node has either 0 or 2 children.
2. **Complete Binary Tree:** All levels are fully filled except possibly the last, which is filled from left to right.
3. **Perfect Binary Tree:** All internal nodes have two children, and all leaves are at the same level.
4. **Skewed Binary Tree:** All nodes have only one child (either left or right), resembling a linked list.

- **Use-Cases:**

- Hierarchical data representation (e.g., organizational charts).
- Expression trees in compilers.
- General-purpose tree structures where ordering is not required.

Binary Search Tree (BST)

A **Binary Search Tree** is a specialized form of a binary tree that maintains a specific order among its nodes. For every node in the BST:

- All nodes in the left subtree have values **less than** the node's value.
- All nodes in the right subtree have values **greater than** the node's value.

This ordering property enables efficient search, insertion, and deletion operations.

- **Structure:**

- Each node has at most two children.
- Maintains an ordered structure based on node values.

- **Properties:**

1. **Inorder Traversal:** Yields node values in ascending order.
2. **Efficient Operations:** Search, insertion, and deletion can be performed in $O(h)$ time, where h is the height of the tree.

- **Use-Cases:**

- Dynamic set operations (e.g., databases, dictionaries).
- Implementing efficient search algorithms.
- Priority queues and heap-based structures.

Key Differences

- **Ordering:**

- **Binary Tree:** No specific ordering; nodes can have any value.
- **BST:** Strict ordering based on node values, enabling efficient search operations.

- **Operation Efficiency:**

- **Binary Tree:** Operations like search can degrade to $O(n)$ time in the worst case.
- **BST:** Maintains $O(h)$ time complexity for search, insertion, and deletion, where h is the tree height.

- **Traversal Outcomes:**

- **Binary Tree:** Inorder traversal does not guarantee sorted order.
- **BST:** Inorder traversal always yields a sorted sequence of values.

When to Use Which

- Use a **Binary Tree** when:
 - The data does not require ordering.
 - You need a general-purpose tree structure without specific constraints.
- Use a **Binary Search Tree** when:
 - You need efficient search, insertion, and deletion operations.
 - Maintaining a sorted order of elements is beneficial.

Python Implementation Examples

```

# Definition for a general Binary Tree node.
class BinaryTreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right

# Definition for a Binary Search Tree node.
class BSTNode:
    def __init__(self, val=0):
        self.val = val
        self.left = None
        self.right = None

    # Insert a new value into the BST
    def insert(self, val):
        if val < self.val:
            if self.left:
                self.left.insert(val)
            else:
                self.left = BSTNode(val)
        elif val > self.val:
            if self.right:
                self.right.insert(val)
            else:
                self.right = BSTNode(val)
        # If val == self.val, do not insert duplicates

    # Search for a value in the BST
    def search(self, val) -> bool:
        if val == self.val:
            return True
        elif val < self.val:
            return self.left.search(val) if self.left
            ↵ else False
        else:
            return self.right.search(val) if self.right
            ↵ else False

```

Explanation

Understanding the differences between Binary Trees and Binary Search Trees is crucial for selecting the appropriate data structure based on the problem's requirements. While both structures consist of nodes with up to two children, their ordering properties and operational efficiencies differ significantly.

- ****Binary Tree**:** Offers flexibility in structure without enforcing any order among node values. Suitable for scenarios where hierarchical relationships are

essential, and ordering is either irrelevant or managed differently.

- **Binary Search Tree (BST)**: Enforces a strict ordering of node values, enabling efficient search, insertion, and deletion operations. Ideal for applications requiring quick access to elements, such as databases and dynamic sets.

By leveraging the appropriate tree structure, you can optimize performance and simplify algorithm implementation for various computational problems.

Similar Topics

Similar topics that complement the understanding of Binary Trees and BSTs include:

- Tree Traversal Algorithms (Inorder, Preorder, Postorder)
- Balanced Trees (AVL Trees, Red-Black Trees)
- Heaps and Priority Queues
- Tree-based Hashing Structures

Things to Keep in Mind and Tricks

- **Choosing the Right Tree**: Assess whether your application requires ordered data access. If so, a BST or a balanced variant might be appropriate.
- **Handling Duplicates**: Decide on a strategy for handling duplicate values in BSTs, such as storing counts or placing duplicates consistently in either the left or right subtree.
- **Balancing Trees**: For BSTs, consider using self-balancing trees (like AVL or Red-Black Trees) to maintain optimal operation times.
- **Traversal Utilization**: Utilize different tree traversal methods based on the specific needs of your algorithms (e.g., inorder for BSTs to retrieve sorted data).

Corner and Special Cases to Test When Writing the Code

- **Empty Trees**: Ensure that functions handle cases where the tree is empty without errors.
- **Single Node Trees**: Verify that operations work correctly on trees with only one node.
- **Skewed Trees**: Test with highly unbalanced trees to assess performance and correctness.

- **Trees with Duplicate Values:** If your implementation allows duplicates, ensure that they are handled consistently.
- **Large Trees:** Assess the performance and stack usage of recursive functions with large tree sizes.
- **Balanced vs. Unbalanced Trees:** Compare the performance of operations on balanced versus unbalanced trees to understand the impact of tree structure.

Problem 17.2 Same Tree

Problem Statement

Determine if two binary trees are identical using recursion.

This problem involves determining whether two binary trees are identical. Two binary trees are considered the same if they are structurally identical and all corresponding nodes have the same value.

Algorithmic Approach

To solve this problem, a recursive approach is typically used. The algorithm proceeds as follows:

1. **Base Case - Both Trees are Empty:** Check if both trees are empty (NULL pointers). If they are, return *true* since two empty trees are considered the same.
2. **One Tree is Empty:** If one tree is empty and the other is not, return *false* because they cannot be the same.
3. **Compare Current Nodes:** Compare the values of the current nodes from both trees. If they do not match, return *false*.
4. **Recurse on Left Subtrees:** Recursively check the left subtrees of both trees to see if they are the same.
5. **Recurse on Right Subtrees:** Recursively check the right subtrees of both trees to see if they are the same.
6. **Combine Results:** If the values of the current nodes match and the recursive checks for both subtrees return *true*, then the current trees are the same; hence, return *true*.
7. **Otherwise:** Return *false*.

Complexities

- **Time Complexity:** The time complexity is $O(n)$, where n is the number of nodes in the tree. This is because the algorithm must visit each node exactly once.

- **Space Complexity:** The space complexity is $O(h)$, where h is the height of the tree. This space is used in the call stack during recursive calls. In the worst case, the tree can be completely unbalanced, yielding a space complexity of $O(n)$.

Python Implementation

```
# 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: TreeNode, q: TreeNode) -> bool:
    # Both trees are empty
    if not p and not q:
        return True
    # One tree is empty, and the other is not
    if not p or not q:
        return False
    # Node values are different
    if p.val != q.val:
        return False
    # Recursively check the left and right subtrees
    return isSameTree(p.left, q.left) and isSameTree(p.
        ↪ right, q.right)
```

Implementing the solution using recursion for clarity and efficiency.

Explanation

The function `isSameTree` determines whether two binary trees, `p` and `q`, are the same by recursively comparing their structure and node values. It returns `true` only if both trees are structurally identical and all corresponding nodes have the same value.

Why This Approach

The recursive approach is natural for tree problems because it mimics the structure of a tree, allowing us to perform the same operation on subtrees as we would on the entire tree. It is elegant and leads to a simple yet effective solution.

Alternative Approaches

An alternative method would be to use an iterative approach with a queue or stack to perform a level-order or depth-first traversal, respectively. At each step, you would compare the nodes from both trees. However, the recursive approach is more straightforward in this case.

Similar Problems to This One

Similar tree comparison problems include checking if a tree is symmetric (Symmetric Tree), determining if a tree is a subtree of another tree (Subtree of Another

Tree), and checking if two trees are mirror images of each other (Invert Binary Tree).

Things to Keep in Mind and Tricks

- Always check for NULL pointers when dealing with trees to avoid dereferencing them.
- If the tree nodes have unique values, a pre-order or post-order traversal would suffice to determine tree equality. If not, a combination of in-order and pre/post-order traversal can be used.

Corner and Special Cases to Test When Writing the Code

Some corner and special cases to consider:

- One or both trees are empty.
- Trees have only one node.
- Trees with a different structure but the same values in a different arrangement.
- Trees where one is a complete binary tree and the other is not.

Problem 17.3 Symmetric Tree

Problem Statement

This problem involves determining whether a binary tree is symmetric around its center. A binary tree is symmetric if the left subtree is a mirror reflection of the right subtree.

Determine if a binary tree is symmetric around its center using recursion.

Algorithmic Approach

To solve this problem, a recursive approach is commonly used. The algorithm proceeds as follows:

1. **Base Case - Both Subtrees are Empty:** If both the left and right subtrees are empty (NULL pointers), the tree is symmetric; return *true*.
2. **One Subtree is Empty:** If only one of the subtrees is empty, the tree is not symmetric; return *false*.
3. **Compare Current Nodes:** Compare the values of the current nodes in both subtrees. If they do not match, the tree is not symmetric; return *false*.
4. **Recurse on Subtrees:**
 - Recursively check if the left subtree of the left node is a mirror of the right subtree of the right node.

- Recursively check if the right subtree of the left node is a mirror of the left subtree of the right node.
5. **Combine Results:** If both recursive checks return *true*, the current subtrees are mirrors of each other; hence, return *true*.
6. **Otherwise:** Return *false*.

Complexities

- **Time Complexity:** The time complexity is $O(n)$, where n is the number of nodes in the tree. Each node is visited exactly once.
- **Space Complexity:** The space complexity is $O(h)$, where h is the height of the tree. This space is utilized by the call stack during recursive calls. In the worst case, the tree is completely unbalanced, resulting in a space complexity of $O(n)$.

Python Implementation

```
# 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 isSymmetric(root: TreeNode) -> bool:
    def isMirror(left: TreeNode, right: TreeNode) ->
        ↪ bool:
            # Both subtrees are empty
            if not left and not right:
                return True
            # One subtree is empty, and the other is not
            if not left or not right:
                return False
            # The values of the current nodes do not match
            if left.val != right.val:
                return False
            # Recursively check the mirrored subtrees
            return isMirror(left.left, right.right) and
                   ↪ isMirror(left.right, right.left)

    if not root:
        return True
    return isMirror(root.left, root.right)
```

Implementing the solution using recursion for clarity and efficiency.

Explanation

The function `isSymmetric` determines whether a binary tree is symmetric around its center by recursively comparing the left and right subtrees. The helper

function `isMirror` checks if two subtrees are mirror images of each other by ensuring that:

- Both subtrees are empty.
- The current nodes have the same value.
- The left subtree of the left node is a mirror of the right subtree of the right node.
- The right subtree of the left node is a mirror of the left subtree of the right node.

Why This Approach

The recursive approach is intuitive for tree problems as it naturally aligns with the hierarchical structure of trees. By breaking down the problem into smaller subproblems (checking mirror symmetry at each level), the solution becomes elegant and efficient.

Alternative Approaches

An alternative method involves using an iterative approach with a queue. By enqueueing pairs of nodes that should be mirrors of each other and iteratively checking their symmetry, we can achieve the same result without recursion. However, the recursive approach is generally more straightforward and easier to implement for this problem.

Similar Problems to This One

Similar tree problems include determining if a tree is a subtree of another tree (Subtree of Another Tree), checking if two trees are identical (Same Tree), and verifying if two trees are mirror images of each other (Invert Binary Tree).

Things to Keep in Mind and Tricks

- ****Base Cases Are Crucial**:** Always handle base cases where subtrees are empty to prevent unnecessary recursion and potential errors.
- ****Symmetry Checks**:** When checking for symmetry, ensure that left and right subtrees are compared in a mirrored manner.
- ****Avoiding Redundant Checks**:** If at any point the current nodes do not match, terminate early to optimize performance.

Corner and Special Cases to Test When Writing the Code

- ****Empty Tree**:** A tree with no nodes should be considered symmetric.
- ****Single Node**:** A tree with only one node is symmetric.

- **Asymmetric Structures**: Trees that have different structures on the left and right should return *false*.
- **Symmetric Values but Asymmetric Structures**: Trees with the same values but arranged differently should return *false*.
- **Large Balanced Trees**: Testing with large, balanced trees ensures that the algorithm handles depth and recursion efficiently.

Problem 17.4 Maximum Depth of Binary Tree

Problem Statement

This problem involves finding the maximum depth (or height) of a binary tree. The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Determine the maximum depth of a binary tree using recursion or iteration.

Algorithmic Approach

To solve this problem, you can use either a recursive approach or an iterative approach using level-order traversal. Below, both methods are described:

1. Recursive Approach (Depth-First Search):

- **Base Case**: If the current node is NULL, return 0.
- **Recursive Case**:
 - Recursively find the maximum depth of the left subtree.
 - Recursively find the maximum depth of the right subtree.
 - The maximum depth at the current node is the greater of the two depths plus one (for the current node).

2. Iterative Approach (Breadth-First Search):

- Use a queue to perform level-order traversal.
- Initialize the depth to 0.
- While the queue is not empty:
 - Increment the depth.
 - For each node at the current level, enqueue its non-NULL left and right children.

Complexities

- **Time Complexity**: The time complexity is $O(n)$, where n is the number of nodes in the tree. Each node is visited exactly once.

- **Space Complexity:**

- **Recursive Approach:** $O(h)$, where h is the height of the tree, due to the call stack.
- **Iterative Approach:** $O(w)$, where w is the maximum width of the tree, since at most w nodes are stored in the queue at any time.

Python Implementation

```
# 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

# Recursive Approach
def maxDepthRecursive(root: TreeNode) -> int:
    if not root:
        return 0
    else:
        left_depth = maxDepthRecursive(root.left)
        right_depth = maxDepthRecursive(root.right)
        return max(left_depth, right_depth) + 1

# Iterative Approach
from collections import deque

def maxDepthIterative(root: TreeNode) -> int:
    if not root:
        return 0
    queue = deque([root])
    depth = 0
    while queue:
        depth += 1
        level_length = len(queue)
        for _ in range(level_length):
            node = queue.popleft()
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
    return depth

# Example usage:
# Constructing the following binary tree
#      3
#     / \
#    9  20
#   / \
#  15  7
```

Implementing both recursive and iterative solutions for flexibility.

```

root = TreeNode(3)
root.left = TreeNode(9)
root.right = TreeNode(20, TreeNode(15), TreeNode(7))

print(maxDepthRecursive(root)) # Output: 3
print(maxDepthIterative(root)) # Output: 3

```

Explanation

The function `maxDepth` (both recursive and iterative versions) calculates the maximum depth of a binary tree by exploring all nodes.

- **Recursive Approach:**

- The function calls itself on the left and right children of the current node.
- It computes the depth of each subtree and returns the greater of the two, adding one to account for the current node.

- **Iterative Approach:**

- Utilizes a queue to perform a level-order traversal.
- For each level, it increments the depth and enqueues the children of all nodes at that level.

Why This Approach

- **Recursive Approach:** Mirrors the natural recursive structure of trees, making the implementation straightforward and intuitive.
- **Iterative Approach:** Useful for environments with limited stack space or when avoiding recursion. It also provides a clear separation of tree levels.

Alternative Approaches

An alternative method involves using Depth-First Search (DFS) iteratively with a stack instead of recursion. While similar in purpose to the recursive approach, this method can be more memory-efficient in certain scenarios and avoids potential stack overflow issues in deep trees.

Similar Problems to This One

Similar tree-related problems include finding the minimum depth of a binary tree (*Minimum Depth of Binary Tree*), determining if a tree is balanced (*Balanced Binary Tree*), and calculating the diameter of a binary tree (*Diameter of Binary Tree*).

Things to Keep in Mind and Tricks

- **Base Cases Are Crucial:** Always handle cases where the tree or subtree is empty to prevent unnecessary computations and potential errors.
- **Choosing the Right Traversal:** Decide between depth-first and breadth-first approaches based on the specific requirements and constraints of the problem.
- **Space Optimization:** Be mindful of the space complexity, especially when dealing with very deep or very wide trees.
- **Understanding Tree Properties:** A solid grasp of tree properties and traversal methods enhances problem-solving efficiency.

Corner and Special Cases to Test When Writing the Code

- **Empty Tree:** A tree with no nodes should return a depth of 0.
- **Single Node:** A tree with only one node should return a depth of 1.
- **Unbalanced Tree:** Trees that are skewed to the left or right should correctly return the depth corresponding to the longest path.
- **Full Binary Tree:** Trees where every node has two children should return a depth that reflects the number of levels.
- **Large Tree:** Testing with a large number of nodes ensures that the implementation handles deep recursion or large queues efficiently.

Problem 17.5 Balanced Binary Tree

Problem Statement

This problem involves determining whether a binary tree is height-balanced. A binary tree is considered *height-balanced* if for every node in the tree, the difference in height between its left and right subtrees is at most one.

Determine if a binary tree is height-balanced using recursion or iteration.

Algorithmic Approach

To solve this problem, a recursive approach is commonly employed. The algorithm proceeds as follows:

1. **Base Case:** If the current node is NULL, it is balanced by definition; return 0 as its height.
2. **Recursive Case:**
 - (a) Recursively determine the height of the left subtree.
 - (b) If the left subtree is unbalanced (indicated by a negative value), propagate the failure by returning -1.

- (c) Recursively determine the height of the right subtree.
- (d) If the right subtree is unbalanced, propagate the failure by returning -1.
- (e) Check the height difference between the left and right subtrees. If it exceeds one, the tree is unbalanced; return -1.
- (f) If balanced, return the height of the current node, which is the greater height between the left and right subtrees plus one.

Complexities

- **Time Complexity:** The time complexity is $O(n)$, where n is the number of nodes in the tree. Each node is visited exactly once.
- **Space Complexity:** The space complexity is $O(h)$, where h is the height of the tree. This space is utilized by the call stack during recursive calls. In the worst case, the tree can be completely unbalanced, resulting in a space complexity of $O(n)$.

Python Implementation

```
# 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

# Optimized Recursive Approach
def isBalanced(root: TreeNode) -> bool:
    def check_height(node: TreeNode) -> int:
        if not node:
            return 0
        left_height = check_height(node.left)
        if left_height == -1:
            return -1
        right_height = check_height(node.right)
        if right_height == -1:
            return -1
        if abs(left_height - right_height) > 1:
            return -1
        return max(left_height, right_height) + 1

    return check_height(root) != -1

# Iterative Approach Using BFS
from collections import deque

def isBalancedIterative(root: TreeNode) -> bool:
    if not root:
```

Implementing both optimized recursive and iterative solutions for flexibility.

```

        return True
queue = deque([(root, 1)])
while queue:
    node, depth = queue.popleft()
    left = node.left
    right = node.right
    if left:
        queue.append((left, depth + 1))
    if right:
        queue.append((right, depth + 1))
# After traversal, compute heights and check
#   ↵ balance
def compute_height(node: TreeNode) -> int:
    if not node:
        return 0
    left = compute_height(node.left)
    if left == -1:
        return -1
    right = compute_height(node.right)
    if right == -1:
        return -1
    if abs(left - right) > 1:
        return -1
    return max(left, right) + 1

return compute_height(root) != -1

# Example usage:
# Constructing the following binary tree
#      3
#     / \
#    9  20
#   / \
#  15  7
root = TreeNode(3)
root.left = TreeNode(9)
root.right = TreeNode(20, TreeNode(15), TreeNode(7))

print(isBalanced(root))          # Output: True
print(isBalancedIterative(root))  # Output: True

```

Explanation

The function `isBalanced` determines whether a binary tree is height-balanced by recursively calculating the height of each subtree. The helper function `check_height` returns the height of a node if its subtree is balanced or -1 if it is not. By propagating the failure state (-1) up the recursive calls, the algorithm efficiently identifies unbalanced subtrees without unnecessary computations.

The iterative approach uses Breadth-First Search (BFS) to traverse the tree level

by level. After traversal, it computes the heights of subtrees and checks for balance, ensuring that the tree adheres to the height-balanced property.

Why This Approach

- **Optimized Recursive Approach:** By immediately propagating the failure state (-1) when an unbalanced subtree is detected, the algorithm avoids redundant calculations, leading to improved efficiency.
- **Iterative Approach:** Provides an alternative to recursion, which can be beneficial in environments with limited stack space or when dealing with very deep trees.

Alternative Approaches

An alternative method involves using Depth-First Search (DFS) iteratively with a stack to simulate the recursive calls. This approach can help avoid potential stack overflow issues in languages with limited recursion depth. However, the recursive approach remains more intuitive and straightforward for this problem.

Similar Problems to This One

Similar tree-related problems include finding the minimum depth of a binary tree (Minimum Depth of Binary Tree), determining if a tree is symmetric (Symmetric Tree), and calculating the diameter of a binary tree (Diameter of Binary Tree).

Things to Keep in Mind and Tricks

- **Base Cases Are Crucial:** Always handle cases where the tree or subtree is empty to prevent unnecessary computations and potential errors.
- **Optimizing Recursive Calls:** Propagating failure states can significantly reduce the number of recursive calls, enhancing performance.
- **Choosing the Right Traversal:** Decide between depth-first and breadth-first approaches based on the specific requirements and constraints of the problem.
- **Understanding Tree Properties:** A solid grasp of tree properties and traversal methods enhances problem-solving efficiency.

Corner and Special Cases to Test When Writing the Code

- **Empty Tree:** A tree with no nodes should be considered balanced.
- **Single Node:** A tree with only one node is balanced.
- **Unbalanced Tree:** Trees that are skewed to the left or right should correctly identify as unbalanced.

- **Balanced Tree with Varying Subtree Heights:** Trees where the height difference between left and right subtrees is exactly one at some nodes.
- **Large Tree:** Testing with a large number of nodes ensures that the implementation handles depth and recursion efficiently.

Problem 17.6 Invert Binary Tree

Problem Statement

Invert a binary tree using recursion or iteration.

This problem involves inverting a binary tree. Inverting a binary tree means swapping the left and right children of every node in the tree. The goal is to transform the original tree into its mirror image.

Algorithmic Approach

To solve this problem, you can use either a recursive approach or an iterative approach using a queue. Below, both methods are described:

1. Recursive Approach (Depth-First Search):

- **Base Case:** If the current node is NULL, return NULL.
- **Recursive Case:**
 - Recursively invert the left subtree.
 - Recursively invert the right subtree.
 - Swap the left and right subtrees.
 - Return the current node.

2. Iterative Approach (Breadth-First Search):

- Use a queue to perform level-order traversal.
- Initialize the queue with the root node.
- While the queue is not empty:
 - Dequeue the current node.
 - Swap its left and right children.
 - Enqueue the non-NUL left and right children.

Complexities

- **Time Complexity:** The time complexity is $O(n)$, where n is the number of nodes in the tree. Each node is visited exactly once.
- **Space Complexity:**
 - **Recursive Approach:** $O(h)$, where h is the height of the tree, due to the call stack.

- **Iterative Approach:** $O(n)$, in the worst case, when the tree is completely unbalanced and all nodes are stored in the queue.

Python Implementation

```
# 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

# Recursive Approach
def invertTreeRecursive(root: TreeNode) -> TreeNode:
    if root is None:
        return None
    # Recursively invert the left and right subtrees
    left_inverted = invertTreeRecursive(root.left)
    right_inverted = invertTreeRecursive(root.right)
    # Swap the left and right subtrees
    root.left, root.right = right_inverted,
                           ↪ left_inverted
    return root

# Iterative Approach
from collections import deque

def invertTreeIterative(root: TreeNode) -> TreeNode:
    if root is None:
        return None
    queue = deque([root])
    while queue:
        current = queue.popleft()
        # Swap the left and right children
        current.left, current.right = current.right,
                                     ↪ current.left
        # Enqueue non-NULL children
        if current.left:
            queue.append(current.left)
        if current.right:
            queue.append(current.right)
    return root

# Example usage:
# Constructing the following binary tree
#      4
#     /   \
#    2     7
#   / \   / \
#  1   3 6   9
```

Implementing both recursive and iterative solutions for flexibility.

```

root = TreeNode(4)
root.left = TreeNode(2, TreeNode(1), TreeNode(3))
root.right = TreeNode(7, TreeNode(6), TreeNode(9))

inverted_recursive = invertTreeRecursive(root)
inverted_iterative = invertTreeIterative(root)

# The inverted tree should be:
#      4
#     / \
#    7   2
#   / \ / \
#  9  6 3  1

```

Explanation

The function `invertTree` transforms a binary tree into its mirror image by swapping the left and right children of every node.

- **Recursive Approach:**

- The function recursively inverts the left and right subtrees.
- After inverting the subtrees, it swaps the left and right children of the current node.
- This process continues until all nodes have been processed, resulting in an inverted tree.

- **Iterative Approach:**

- The function uses a queue to perform a level-order traversal of the tree.
- For each node dequeued, it swaps its left and right children.
- Non-NULL children are enqueue for subsequent processing.
- This continues until all nodes have been visited and their children swapped.

Why This Approach

- **Recursive Approach:** Aligns naturally with the hierarchical structure of trees, making the implementation straightforward and intuitive. It leverages the call stack to manage traversal.
- **Iterative Approach:** Provides an alternative to recursion, which can be beneficial in environments with limited stack space or when dealing with very deep trees. It uses a queue to manage traversal explicitly.

Alternative Approaches

An alternative method involves using Depth-First Search (DFS) iteratively with a stack to simulate the recursive calls. This approach can help avoid potential stack overflow issues in languages with limited recursion depth. However, the recursive and breadth-first iterative approaches are generally more intuitive and easier to implement for this problem.

Similar Problems to This One

Similar tree manipulation problems include inverting a binary tree (Invert Binary Tree), checking if a tree is symmetric (Symmetric Tree), and merging two binary trees (Merge Two Binary Trees).

Things to Keep in Mind and Tricks

- **Base Cases Are Crucial:** Always handle cases where the tree or subtree is empty to prevent unnecessary computations and potential errors.
- **Traversal Choice:** Decide between recursive and iterative approaches based on the specific requirements and constraints of the problem.
- **Space Optimization:** Be mindful of the space complexity, especially when dealing with very deep or very wide trees.
- **Understanding Tree Properties:** A solid grasp of tree properties and traversal methods enhances problem-solving efficiency.

Corner and Special Cases to Test When Writing the Code

- **Empty Tree:** A tree with no nodes should remain NULL after inversion.
- **Single Node:** A tree with only one node should remain unchanged after inversion.
- **Unbalanced Tree:** Trees that are skewed to the left or right should correctly invert their structure.
- **Balanced Tree:** Ensure that balanced trees are inverted correctly, maintaining their balanced property.
- **Large Tree:** Testing with a large number of nodes ensures that the implementation handles depth and breadth efficiently without performance degradation.

Problem 17.7 Binary Tree Level Order Traversal

Problem Statement

This problem involves performing a level order traversal on a binary tree. In level order traversal, nodes are visited level by level from left to right. The goal is to return

Traverse a binary tree level by level using BFS or DFS.

a list of lists, where each sublist contains the values of nodes at each respective level of the tree.

Algorithmic Approach

To solve this problem, you can use either an iterative approach with Breadth-First Search (BFS) or a recursive approach with Depth-First Search (DFS). Below, both methods are described:

1. Iterative Approach (Breadth-First Search):

- **Use a Queue:** Utilize a queue to keep track of nodes at the current level.
- **Level-by-Level Traversal:**
 - (a) Initialize the queue with the root node.
 - (b) While the queue is not empty:
 - i. Determine the number of nodes at the current level (size of the queue).
 - ii. Iterate through all nodes at this level:
 - A. Dequeue a node from the queue.
 - B. Add its value to the current level's list.
 - C. Enqueue its non-NULL left and right children.
 - iii. After processing all nodes at the current level, add the level's list to the result.

2. Recursive Approach (Depth-First Search):

- **Use a Helper Function:** Create a helper function that takes a node and its current level as arguments.
- **Traverse the Tree:**
 - (a) If the node is NULL, return.
 - (b) If the current level is equal to the size of the result list, append a new sublist for this level.
 - (c) Add the node's value to the corresponding level's sublist.
 - (d) Recursively traverse the left and right children, incrementing the level by one.

Complexities

- **Time Complexity:** The time complexity is $O(n)$, where n is the number of nodes in the tree. Each node is visited exactly once.
- **Space Complexity:**
 - **Iterative Approach (BFS):** $O(n)$, as in the worst case, the queue will hold all nodes at the last level.

- **Recursive Approach (DFS):** $O(n)$, due to the space required to store the result and the recursive call stack.

Python Implementation

```
# 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

# Iterative Approach (BFS)
from collections import deque

def levelOrderBFS(root: TreeNode) -> list:
    if not root:
        return []
    result = []
    queue = deque([root])
    while queue:
        level_size = len(queue)
        current_level = []
        for _ in range(level_size):
            node = queue.popleft()
            current_level.append(node.val)
            if node.left:
                queue.append(node.left)
            if node.right:
                queue.append(node.right)
        result.append(current_level)
    return result

# Recursive Approach (DFS)
def levelOrderDFS(root: TreeNode) -> list:
    def helper(node: TreeNode, level: int):
        if not node:
            return
        if level == len(result):
            result.append([])
        result[level].append(node.val)
        helper(node.left, level + 1)
        helper(node.right, level + 1)

    result = []
    helper(root, 0)
    return result

# Example usage:
# Constructing the following binary tree
#      3
#     / \
#    9  20
#       / \
#      15  7
```

Implementing both BFS and DFS solutions for flexibility.

```

#      / \
#     9   20
#       / \
#      15   7
root = TreeNode(3)
root.left = TreeNode(9)
root.right = TreeNode(20, TreeNode(15), TreeNode(7))

print(levelOrderBFS(root))  # Output: [[3], [9, 20],
                           ↪ [15, 7]]
print(levelOrderDFS(root))  # Output: [[3], [9, 20],
                           ↪ [15, 7]]

```

Explanation

The function `levelOrder` performs a level order traversal of a binary tree, returning a list of lists where each sublist contains the values of nodes at each level.

- **Iterative Approach (BFS):**

- Uses a queue to traverse the tree level by level.
- For each level, it records the number of nodes, processes each node by dequeuing it, and enqueues its children.
- After processing all nodes at the current level, it appends the collected values to the result list.

- **Recursive Approach (DFS):**

- Utilizes a helper function to traverse the tree depth-first while keeping track of the current level.
- If a new level is encountered (i.e., the current level is equal to the length of the result list), a new sublist is appended.
- The node's value is added to its corresponding level's sublist.
- Recursively processes the left and right children, incrementing the level.

Why This Approach

- **Iterative Approach (BFS):**

- Naturally fits the level order traversal requirement by exploring nodes level by level.
- Efficient in terms of time and space for balanced trees.

- **Recursive Approach (DFS):**

- Simplifies the traversal by leveraging the call stack to manage levels.

- Can be more intuitive for those familiar with recursive tree traversals.

Alternative Approaches

An alternative method involves using a stack to perform an iterative Depth-First Search (DFS) while keeping track of node levels. However, the BFS and recursive DFS approaches are generally more straightforward and easier to implement for level order traversal.

Similar Problems to This One

Similar tree traversal problems include finding the minimum depth of a binary tree (Minimum Depth of Binary Tree), checking if a tree is symmetric (Symmetric Tree), and performing a zigzag level order traversal (Binary Tree Zigzag Level Order Traversal).

Things to Keep in Mind and Tricks

- **Handling Empty Trees:** Always check if the root is NULL to handle empty trees gracefully.
- **Level Tracking:** In the iterative approach, keeping track of the current level's size helps in segregating nodes level by level.
- **Space Optimization:** Be mindful of the space used by the queue in BFS or the recursion stack in DFS, especially for very deep or wide trees.
- **Consistent Level Identification:** Ensure that nodes are correctly associated with their respective levels to maintain accurate traversal results.

Corner and Special Cases to Test When Writing the Code

- **Empty Tree:** Should return an empty list.
- **Single Node:** Should return a list containing one sublist with the single node's value.
- **Unbalanced Tree:** Trees that are skewed to the left or right should correctly return their respective levels.
- **Complete Binary Tree:** Ensure that all levels except possibly the last are fully filled.
- **Large Tree:** Testing with a large number of nodes ensures that the implementation handles depth and breadth efficiently without performance degradation.

Problem 17.8 Binary Tree Right Side View

Problem Statement

View a binary tree from the right side using BFS or DFS.

This problem involves determining the right side view of a binary tree. The right side view consists of the nodes visible when the tree is viewed from the right side. Specifically, you need to return a list of node values that are visible at each level of the tree from top to bottom.

Algorithmic Approach

To solve this problem, you can use either an iterative approach with Breadth-First Search (BFS) or a recursive approach with Depth-First Search (DFS). Below, both methods are described:

1. Iterative Approach (Breadth-First Search):

- **Use a Queue:** Utilize a queue to perform level-order traversal.
- **Level-by-Level Traversal:**
 - (a) Initialize the queue with the root node.
 - (b) While the queue is not empty:
 - i. Determine the number of nodes at the current level (size of the queue).
 - ii. Iterate through all nodes at this level:
 - A. Dequeue a node from the queue.
 - B. If it's the last node in the current level, add its value to the result list.
 - C. Enqueue its non-NULL left and right children.

2. Recursive Approach (Depth-First Search):

- **Use a Helper Function:** Create a helper function that takes a node and its current depth as arguments.
- **Traverse the Tree:**
 - (a) If the node is NULL, return.
 - (b) If the current depth equals the size of the result list, append the node's value to the result list.
 - (c) Recursively traverse the right subtree first, then the left subtree, incrementing the depth by one.

Complexities

- **Time Complexity:** The time complexity is $O(n)$, where n is the number of nodes in the tree. Each node is visited exactly once.
- **Space Complexity:**

- **Iterative Approach (BFS)**: $O(n)$, as in the worst case, the queue will hold all nodes at the last level.
- **Recursive Approach (DFS)**: $O(h)$, where h is the height of the tree, due to the call stack.

Python Implementation

```
# 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

# Iterative Approach (BFS)
from collections import deque

def rightSideViewBFS(root: TreeNode) -> list:
    if not root:
        return []
    result = []
    queue = deque([root])
    while queue:
        level_size = len(queue)
        for i in range(level_size):
            node = queue.popleft()
            # If it's the last node in the current
            # ↪ level, add to result
            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

# Recursive Approach (DFS)
def rightSideViewDFS(root: TreeNode) -> list:
    def helper(node: TreeNode, depth: int):
        if not node:
            return
        if depth == len(result):
            result.append(node.val)
        # Traverse right subtree first
        helper(node.right, depth + 1)
        helper(node.left, depth + 1)

    result = []
    helper(root, 0)
    return result
```

Implementing both BFS and DFS solutions for flexibility.

```

# Example usage:
# Constructing the following binary tree
#      1
#     / \
#    2   3
#   / \
#  5   4
root = TreeNode(1)
root.left = TreeNode(2, None, TreeNode(5))
root.right = TreeNode(3, None, TreeNode(4))

print(rightSideViewBFS(root))  # Output: [1, 3, 4]
print(rightSideViewDFS(root))  # Output: [1, 3, 4]

```

Explanation

The function `rightSideView` returns the values of the nodes that are visible when the binary tree is viewed from the right side.

- **Iterative Approach (BFS):**

- Utilizes a queue to perform level-order traversal.
- At each level, it records the last node's value, which is visible from the right side.
- Enqueues the left and right children of each node to traverse subsequent levels.

- **Recursive Approach (DFS):**

- Uses a helper function to traverse the tree depth-first, prioritizing the right subtree first.
- When visiting a node at a new depth, it adds the node's value to the result list.
- This ensures that the first node encountered at each depth is the rightmost node.

Why This Approach

- **Iterative Approach (BFS):**

- Efficiently handles level-by-level traversal, making it straightforward to identify the rightmost node at each level.
- Suitable for scenarios where breadth-wise information is essential.

- **Recursive Approach (DFS):**

- Recursively explores the right subtree first, ensuring that the first node encountered at each depth is the rightmost node.

- Simplifies the implementation by leveraging the call stack to manage traversal depth.

Alternative Approaches

An alternative method involves using Depth-First Search (DFS) iteratively with a stack, maintaining a record of depths and ensuring that the rightmost nodes are processed first. However, the BFS and recursive DFS approaches are generally more intuitive and easier to implement for this problem.

Similar Problems to This One

Similar tree traversal problems include finding the left side view of a binary tree (Binary Tree Left Side View), performing a zigzag level order traversal (Binary Tree Zigzag Level Order Traversal), and finding the maximum depth of a binary tree (Maximum Depth of Binary Tree).

Things to Keep in Mind and Tricks

- **Handling Empty Trees:** Always check if the root is NULL to handle empty trees gracefully.
- **Level Tracking:** In the iterative approach, keeping track of the current level's size helps in identifying the rightmost node.
- **Traversal Order:** Prioritizing the right subtree first in the recursive approach ensures that the first node encountered at each depth is the rightmost node.
- **Space Optimization:** Be mindful of the space used by the queue in BFS or the recursion stack in DFS, especially for very deep or wide trees.
- **Consistent Level Identification:** Ensure that nodes are correctly associated with their respective levels to maintain accurate traversal results.

Corner and Special Cases to Test When Writing the Code

- **Empty Tree:** Should return an empty list.
- **Single Node:** Should return a list containing the single node's value.
- **Left-Skewed Tree:** Trees where each node has only a left child should correctly return the last node at each level.
- **Right-Skewed Tree:** Trees where each node has only a right child should correctly return all node values.
- **Balanced Tree:** Ensure that balanced trees return the rightmost nodes at each level.
- **Large Tree:** Testing with a large number of nodes ensures that the implementation handles depth and breadth efficiently without performance degradation.

Problem 17.9 Lowest Common Ancestor of a Binary Search Tree

Problem Statement

Given a binary search tree (BST), 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).*”

Find the lowest common ancestor (LCA) of two nodes in a BST using BST properties.

Algorithmic Approach

To solve this problem efficiently, leverage the properties of the binary search tree. Since in a BST, for any node, all nodes in the left subtree have values less than the node’s value, and all nodes in the right subtree have values greater than the node’s value, we can determine the LCA by comparing the values of the current node with the values of p and q.

1. Iterative Approach:

- **Start at the Root:** Initialize a pointer to the root of the BST.
- **Traverse the Tree:**
 - (a) If both p and q are greater than the current node, move to the right child.
 - (b) If both p and q are less than the current node, move to the left child.
 - (c) If one of p or q is on one side and the other is on the opposite side (or equal to the current node), the current node is the LCA.

2. Recursive Approach:

- **Base Case:** If the current node is NULL, return NULL.
- **Compare Values:**
 - (a) If both p and q are greater than the current node’s value, recursively search the right subtree.
 - (b) If both p and q are less than the current node’s value, recursively search the left subtree.
 - (c) If p and q lie on different sides of the current node (or one is equal to the current node), the current node is the LCA.

Complexities

- **Time Complexity:** $O(h)$, where h is the height of the BST. In the best case of a balanced BST, $h = \log n$, and in the worst case of a skewed BST, $h = n$.
- **Space Complexity:**

- **Iterative Approach:** $O(1)$, as it uses constant extra space.
- **Recursive Approach:** $O(h)$, due to the recursive call stack.

Python Implementation

```
# 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

# Iterative Approach
def lowestCommonAncestorIterative(root: TreeNode, p:
    ↪ TreeNode, q: TreeNode) -> TreeNode:
    current = root
    while current:
        if p.val > current.val and q.val > current.val:
            current = current.right
        elif p.val < current.val and q.val < current.
            ↪ val:
            current = current.left
        else:
            return current
    return None

# Recursive Approach
def lowestCommonAncestorRecursive(root: TreeNode, p:
    ↪ TreeNode, q: TreeNode) -> TreeNode:
    if not root:
        return None
    if p.val > root.val and q.val > root.val:
        return lowestCommonAncestorRecursive(root.right
            ↪ , p, q)
    if p.val < root.val and q.val < root.val:
        return lowestCommonAncestorRecursive(root.left,
            ↪ p, q)
    return root

# Example usage:
# Constructing the following BST
#
#       6
#      / \
#     2   8
#    / \ / \
#   0  4 7  9
#      / \
#     3   5

root = TreeNode(6)
```

Implementing both iterative and recursive solutions leveraging BST properties.

```

root.left = TreeNode(2, TreeNode(0), TreeNode(4,
    ↪ TreeNode(3), TreeNode(5)))
root.right = TreeNode(8, TreeNode(7), TreeNode(9))

p = root.left      # Node with value 2
q = root.left.right # Node with value 4

lca_iterative = lowestCommonAncestorIterative(root, p,
    ↪ q)
print(lca_iterative.val) # Output: 2

lca_recursive = lowestCommonAncestorRecursive(root, p,
    ↪ q)
print(lca_recursive.val) # Output: 2

```

Explanation

The function `lowestCommonAncestor` identifies the lowest common ancestor (LCA) of two nodes in a binary search tree by utilizing the inherent properties of the BST.

- **Iterative Approach:**

- Begins at the root and traverses the tree based on the values of p and q.
- If both p and q are greater than the current node, it moves to the right subtree.
- If both are less, it moves to the left subtree.
- If they diverge (one is on the left and the other on the right) or one equals the current node, the current node is the LCA.

- **Recursive Approach:**

- Recursively navigates the tree in a similar manner to the iterative approach.
- The recursion continues until it finds the split point where p and q diverge, identifying the current node as the LCA.

Why This Approach

- **Efficiency:** Leveraging the BST properties allows for efficient traversal, reducing the search space at each step.
- **Simplicity:** Both iterative and recursive approaches are straightforward to implement and understand, making the solution elegant and maintainable.
- **Optimality:** The approaches achieve optimal time complexity by avoiding unnecessary traversal of irrelevant subtrees.

Alternative Approaches

An alternative method involves performing an inorder traversal to generate a sorted list of node values and then finding the LCA by analyzing this list. However, this approach is less efficient in terms of time and space compared to the iterative and recursive methods that directly utilize the BST properties.

Similar Problems to This One

Similar tree-related problems include finding the lowest common ancestor in a binary tree (Lowest Common Ancestor of a Binary Tree), determining if one tree is a subtree of another (Subtree of Another Tree), and finding the distance between two nodes in a tree (Distance Between Two Nodes in a Tree).

Things to Keep in Mind and Tricks

- **BST Properties:** Always utilize the BST properties to guide the traversal, as they significantly reduce the search space.
- **Edge Cases:** Consider scenarios where one of the nodes is the root or where one node is an ancestor of the other.
- **Handling Non-Existent Nodes:** Ensure that both nodes exist in the tree to avoid incorrect LCA identification.
- **Recursive Call Optimization:** In the recursive approach, short-circuit the recursion once the LCA is found to optimize performance.

Corner and Special Cases to Test When Writing the Code

- **Both Nodes are the Same:** When p and q are the same node, the LCA is the node itself.
- **One Node is the Ancestor of the Other:** When one node is an ancestor of the other, the ancestor node is the LCA.
- **Nodes on Different Subtrees:** When p and q are on different subtrees, the LCA is the split point where one is on the left and the other on the right.
- **Nodes Not Present in the Tree:** Handle cases where one or both nodes are not present in the tree.
- **Empty Tree:** When the tree is empty, there is no LCA.
- **Single Node Tree:** When the tree has only one node, it is the LCA if it matches one of the target nodes.

Problem 17.10 Validate Binary Search Tree

Problem Statement

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 **less than** the node's key.
- The right subtree of a node contains only nodes with keys **greater than** the node's key.
- Both the left and right subtrees must also be binary search trees.

Determine if a binary tree is a valid Binary Search Tree using recursive or iterative methods.

Algorithmic Approach

To solve this problem, you can use either a recursive approach or an iterative approach with in-order traversal. Below, both methods are described:

1. Recursive Approach (Using Bounds):

- **Use Helper Function with Bounds:** Create a helper function that takes a node and the allowable range of values for that node.
- **Validate Node Values:**
 - If the current node is NULL, it is valid by definition; return True.
 - Check if the current node's value is within the valid range ($\text{min_val} < \text{node.val} < \text{max_val}$). If not, return False.
 - Recursively validate the left subtree with updated max_val and the right subtree with updated min_val .

2. Iterative Approach (In-Order Traversal):

- **Use a Stack for Traversal:** Utilize a stack to perform an in-order traversal of the tree.
- **Maintain Previous Value:**
 - Initialize an empty stack and set the prev_val to None.
 - Traverse the tree:
 - Go as far left as possible, pushing nodes onto the stack.
 - Pop a node from the stack and compare its value with prev_val . If the current node's value is not greater, the BST property is violated; return False.
 - Update prev_val to the current node's value.
 - Move to the right subtree.

Complexities

- **Time Complexity:** $O(n)$, where n is the number of nodes in the tree. Each node is visited exactly once.
- **Space Complexity:**
 - **Recursive Approach:** $O(h)$, where h is the height of the tree, due to the recursive call stack.
 - **Iterative Approach:** $O(h)$, where h is the height of the tree, due to the stack used for traversal.

Python Implementation

```
# 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

# Recursive Approach (Using Bounds)
def isValidBSTRecursive(root: TreeNode) -> bool:
    def helper(node: TreeNode, min_val: float, max_val: float) -> bool:
        if not node:
            return True
        if not (min_val < node.val < max_val):
            return False
        return helper(node.left, min_val, node.val) and
               helper(node.right, node.val, max_val)

    return helper(root, float('-inf'), float('inf'))

# Iterative Approach (In-Order Traversal)
def isValidBSTIterative(root: TreeNode) -> bool:
    stack = []
    prev_val = None
    current = root

    while stack or current:
        while current:
            stack.append(current)
            current = current.left
        current = stack.pop()
        if prev_val is not None and current.val <=
            prev_val:
            return False
        prev_val = current.val
        current = current.right
```

Implementing both recursive and iterative solutions leveraging BST properties.

```

    return True

# Example usage:
# Constructing the following BST
#      5
#      / \
#     3   7
#    / \   \
#   2   4   8

root = TreeNode(5)
root.left = TreeNode(3, TreeNode(2), TreeNode(4))
root.right = TreeNode(7, None, TreeNode(8))

print(isValidBSTRecursive(root))  # Output: True
print(isValidBSTIterative(root))  # Output: True

# Constructing an invalid BST
#      5
#      / \
#     1   4
#      / \
#     3   6

invalid_root = TreeNode(5)
invalid_root.left = TreeNode(1)
invalid_root.right = TreeNode(4, TreeNode(3), TreeNode
    ↪ (6))

print(isValidBSTRecursive(invalid_root))  # Output:
    ↪ False
print(isValidBSTIterative(invalid_root))  # Output:
    ↪ False

```

Explanation

The function `isValidBST` determines whether a binary tree is a valid Binary Search Tree by leveraging the inherent properties of BSTs.

- **Recursive Approach (Using Bounds):**

- **Helper Function with Bounds**: The helper function checks whether each node's value lies within the valid range defined by `min_val` and `max_val`.
- **Validation Process**:
 1. If the current node is `NULL`, it is considered valid.
 2. If the current node's value does not satisfy `min_val < node.val < max_val`, the BST property is violated; return `False`.
 3. Recursively validate the left subtree with an updated `max_val` (current node's value) and the right subtree with an updated `min_val` (current

node's value).

- **Iterative Approach (In-Order Traversal):**

- ****In-Order Traversal**:** In-order traversal of a BST yields a sorted list of values in ascending order.
- ****Validation Process**:**
 1. Traverse the tree using a stack to simulate recursion.
 2. At each step, compare the current node's value with the previous node's value. If the current value is not greater, the BST property is violated; return `False`.
 3. Update the `prev_val` to the current node's value and continue traversal.

Why This Approach

- **Leverages BST Properties:** Both approaches utilize the fundamental properties of BSTs to efficiently validate the tree structure.
- **Efficiency:** The recursive approach optimally narrows down the valid range for each node, ensuring that each node is checked against appropriate boundaries. The iterative approach ensures that the in-order traversal strictly increases, maintaining the BST property.
- **Simplicity and Readability:** Both methods are straightforward to implement and understand, making the solution elegant and maintainable.

Alternative Approaches

An alternative method involves performing an in-order traversal to collect all node values into a list and then verifying if the list is strictly increasing. However, this approach requires additional space to store the list of node values, making it less space-efficient compared to the iterative in-order traversal method that validates on-the-fly.

Similar Problems to This One

Similar tree-related problems include finding the minimum depth of a binary tree (Minimum Depth of Binary Tree), checking if a tree is balanced (Balanced Binary Tree), and determining the lowest common ancestor in a binary tree (Lowest Common Ancestor of a Binary Tree).

Things to Keep in Mind and Tricks

- **Handling Edge Cases:** Always consider edge cases such as empty trees, single-node trees, and trees with duplicate values.
- **Using Bounds Correctly:** In the recursive approach, ensure that the bounds are correctly updated when traversing left and right subtrees.

- **In-Order Traversal Validity:** Remember that for a valid BST, in-order traversal should produce a strictly increasing sequence of values.
- **Avoiding Unnecessary Computations:** In the iterative approach, validate the BST property during traversal to avoid storing all node values.

Corner and Special Cases to Test When Writing the Code

- **Empty Tree:** Should return True as an empty tree is considered a valid BST.
- **Single Node:** Should return True as a single-node tree is a valid BST.
- **Valid BST:** Trees that satisfy all BST properties should return True.
- **Invalid BST Due to Left Subtree:** Trees where a node in the left subtree has a value greater than or equal to its parent.
- **Invalid BST Due to Right Subtree:** Trees where a node in the right subtree has a value less than or equal to its parent.
- **Large Tree:** Testing with a large number of nodes ensures that the implementation handles deep recursion and large stacks efficiently without performance degradation.
- **Trees with Duplicate Values:** Ensure that trees with duplicate values are handled according to the BST definition (typically, duplicates are not allowed or consistently placed in one subtree).

Problem 17.11 Kth Smallest Element in a Binary Search Tree

Problem Statement

Find the kth smallest element in a BST using in-order traversal.

Given the root of a binary search tree (BST) and an integer k , return the k th smallest element in the BST.

Algorithmic Approach

To efficiently find the k th smallest element in a BST, leverage the in-order traversal property of BSTs, which visits nodes in ascending order. You can implement this traversal either recursively or iteratively.

1. Recursive In-Order Traversal:

- **In-Order Traversal:** Traverse the left subtree, visit the current node, then traverse the right subtree.
- **Counter Mechanism:** Use a counter to keep track of the number of nodes visited. When the counter reaches k , record the current node's value as the result.

2. Iterative In-Order Traversal:

- **Use a Stack:** Utilize a stack to simulate the recursive in-order traversal.
- **Traversal Process:**
 - (a) Initialize an empty stack and set the current node to the root.
 - (b) While the stack is not empty or the current node is not NULL:
 - i. Traverse to the leftmost node, pushing each node onto the stack.
 - ii. Pop a node from the stack, decrement k , and check if it is the k th node.
 - iii. Move to the right subtree of the popped node.

Complexities

- **Time Complexity:** $O(n)$, where n is the number of nodes in the BST. In the worst case, you may need to traverse all nodes.
- **Space Complexity:**
 - **Recursive Approach:** $O(h)$, where h is the height of the tree, due to the recursive call stack.
 - **Iterative Approach:** $O(h)$, where h is the height of the tree, due to the stack used for traversal.

Python Implementation

```
# 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

# Recursive In-Order Traversal Approach
def kthSmallestRecursive(root: TreeNode, k: int) -> int:
    ↪ :
    def inorder(node: TreeNode):
        if node is None:
            return
        yield from inorder(node.left)
        yield node.val
        yield from inorder(node.right)

    gen = inorder(root)
    for _ in range(k):
        val = next(gen)
    return val

# Iterative In-Order Traversal Approach
```

Implementing both recursive and iterative in-order traversal solutions.

```

def kthSmallestIterative(root: TreeNode, k: int) -> int
    ↪ :
    stack = []
    current = root
    while stack or current:
        while current:
            stack.append(current)
            current = current.left
        current = stack.pop()
        k -= 1
        if k == 0:
            return current.val
        current = current.right
    return -1 # If k is out of bounds

# Example usage:
# Constructing the following BST
#      5
#     / \
#    3   6
#   / \
#  2   4
# /
# 1

root = TreeNode(5)
root.left = TreeNode(3, TreeNode(2, TreeNode(1)),
    ↪ TreeNode(4))
root.right = TreeNode(6)

k = 3
print(kthSmallestRecursive(root, k)) # Output: 3
print(kthSmallestIterative(root, k)) # Output: 3

```

Explanation

The function `kthSmallest` identifies the k th smallest element in a binary search tree by performing an in-order traversal, which naturally visits the nodes in ascending order.

- **Recursive In-Order Traversal:**

- **Generator Function**: The helper function `inorder` is a generator that yields node values in in-order sequence.
- **Iteration**: Iterate through the generator k times to retrieve the k th smallest value.

- **Iterative In-Order Traversal:**

- **Stack Utilization**: A stack is used to traverse the tree without recursion.

– ****Traversal Logic**:**

1. Traverse to the leftmost node, pushing each node onto the stack.
2. Pop a node from the stack, decrement k , and check if it is the k th node.
3. Move to the right subtree of the popped node and continue the process.

Why This Approach

- **Leveraging BST Properties:** In-order traversal exploits the BST's inherent property of ordered node values, making it an optimal choice for finding the k th smallest element.
- **Efficiency:** Both recursive and iterative in-order traversals ensure that each node is visited only once, achieving linear time complexity.
- **Flexibility:** Providing both recursive and iterative solutions caters to different programming preferences and system constraints (e.g., stack depth limitations).

Alternative Approaches

An alternative method involves augmenting the BST nodes with additional information, such as the count of nodes in their left subtree. This allows for $O(h)$ time complexity in finding the k th smallest element. However, this approach requires modifying the tree structure and maintaining the counts during insertions and deletions, which can add complexity to the implementation.

Similar Problems to This One

Similar tree-related problems include finding the minimum and maximum elements in a BST, performing in-order traversal, and finding the median in a BST.

Things to Keep in Mind and Tricks

- **BST Properties:** Always utilize the binary search tree properties to guide the traversal and optimize the search.
- **Handling Edge Cases:** Ensure that the value of k is within the valid range (1 to the number of nodes in the BST).
- **Generator Usage:** In the recursive approach, using a generator can make the implementation more elegant and memory-efficient.
- **Iterative Traversal Efficiency:** In the iterative approach, maintaining a stack ensures that the traversal does not exceed the space complexity related to the tree's height.

Corner and Special Cases to Test When Writing the Code

- **Empty Tree:** Should handle gracefully, possibly by returning an error or a sentinel value.
- **Single Node:** When the BST has only one node, and k is 1.
- **k Equals 1:** Finding the smallest element.
- **k Equals the Number of Nodes:** Finding the largest element.
- **k Out of Bounds:** When k is less than 1 or greater than the number of nodes in the BST.
- **Balanced vs. Unbalanced Trees:** Ensure that both balanced and skewed trees are handled correctly.
- **Large Tree:** Testing with a large number of nodes to ensure that the implementation scales and performs efficiently.

Problem 17.12 Lowest Common Ancestor of a Binary Tree

Problem Statement

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).*”

Find the lowest common ancestor (LCA) of two nodes in a binary tree using recursive depth-first search.

Algorithmic Approach

To solve this problem, we can utilize a recursive depth-first search (DFS) strategy that traverses the tree to locate both nodes and determine their LCA. The approach involves exploring each node’s subtrees and identifying the split point where one node resides in one subtree and the other node resides in the other subtree.

1. Recursive Depth-First Search (DFS):

- **Traverse the Tree:** Start from the root and recursively traverse the left and right subtrees.
- **Identify LCA:**
 - **Base Case**:** If the current node is NULL, return NULL. If the current node matches either p or q, return the current node.
 - **Recursive Search**:** Recursively search for p and q in the left and right subtrees.
 - **Determine LCA**:**
 - If both left and right recursive calls return non-NUL nodes, the current node is the LCA.

- If only one of the recursive calls returns a non-NULL node, propagate that node upward as a potential LCA.

Complexities

- **Time Complexity:** $O(N)$, where N is the number of nodes in the binary tree.
Each node is visited exactly once.
- **Space Complexity:** $O(H)$, where H is the height of the tree, due to the recursive call stack. In the worst case of a skewed tree, the space complexity becomes $O(N)$.

Python Implementation

```
# 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 lowestCommonAncestorBinaryTree(self, root:
        ↪ TreeNode, p: TreeNode, q: TreeNode) ->
        ↪ TreeNode:
        if not root:
            return None
        if root == p or root == q:
            return root

        left = self.lowestCommonAncestorBinaryTree(root
            ↪ .left, p, q)
        right = self.lowestCommonAncestorBinaryTree(
            ↪ root.right, p, q)

        if left and right:
            return root
        return left if left else right

# Example usage:
# Constructing the following binary tree
#
#      3
#     / \
#    5   1
#   / \ / \
#  6  2 0  8
#     / \
#    7  4

root = TreeNode(3)
```

Implementing a recursive DFS solution to find the lowest common ancestor in a binary tree.

```

root.left = TreeNode(5, TreeNode(6), TreeNode(2,
    ↪ TreeNode(7), TreeNode(4)))
root.right = TreeNode(1, TreeNode(0), TreeNode(8))

p = root.left      # Node with value 5
q = root.left.right.right # Node with value 4

solution = Solution()
lca = solution.lowestCommonAncestorBinaryTree(root, p,
    ↪ q)
print(lca.val)  # Output: 5

```

Explanation

The function `lowestCommonAncestorBinaryTree` identifies the lowest common ancestor (LCA) of two nodes in a binary tree by performing a recursive depth-first search (DFS).

- **Recursive Traversal:**

- ****Base Case**:** If the current node is `NULL`, it returns `NULL`, indicating that neither `p` nor `q` is found in this path. If the current node matches either `p` or `q`, it returns the current node as a potential LCA.
- ****Left and Right Search**:** The function recursively searches the left and right subtrees for `p` and `q`.
- ****Determining LCA**:**
 - * If both left and right recursive calls return non-`NULL` nodes, it implies that `p` and `q` are found in different subtrees, making the current node their LCA.
 - * If only one side returns a non-`NULL` node, it propagates that node upwards as a potential LCA.

Why This Approach

- **Comprehensive Traversal:** This recursive DFS approach ensures that all paths are explored, guaranteeing that the LCA is accurately identified regardless of the tree's structure.
- **Simplicity and Elegance:** The recursive nature of the solution aligns naturally with the hierarchical structure of binary trees, resulting in clear and maintainable code.
- **Flexibility:** Unlike BST-specific approaches, this method works for any binary tree, making it versatile for various applications.

Alternative Approaches

An alternative method involves using parent pointers and storing the ancestors of one node in a set, then traversing the ancestors of the second node to find the first common ancestor. However, this approach requires additional space to store ancestor information and is generally less efficient compared to the recursive DFS method, which utilizes the tree's structure without extra memory overhead.

Similar Problems to This One

Similar tree-related problems include finding the lowest common ancestor in a binary search tree (Lowest Common Ancestor of a Binary Search Tree), determining if one tree is a subtree of another (Subtree of Another Tree), and calculating the diameter of a binary tree (Diameter of a Binary Tree).

Things to Keep in Mind and Tricks

- **Edge Cases:** Consider scenarios where one node is the ancestor of the other, both nodes are the same, or one or both nodes do not exist in the tree.
- **Recursive Efficiency:** Ensure that the recursion is efficiently handled to prevent unnecessary computations, especially in large trees.
- **Null Checks:** Always check for NULL nodes to avoid runtime errors during traversal.
- **Tree Traversal Order:** Understanding different tree traversal orders (preorder, inorder, postorder) can aid in solving various tree-related problems.

Corner and Special Cases to Test When Writing the Code

- **Both Nodes Are the Same:** When p and q are the same node, the LCA is the node itself.
- **One Node Is the Ancestor of the Other:** When one node is an ancestor of the other, the ancestor node should be identified as the LCA.
- **Nodes on Different Subtrees:** When p and q are located in different subtrees, the LCA is the split point where their paths diverge.
- **Nodes Not Present in the Tree:** Handle cases where one or both nodes are not present in the tree gracefully, possibly by returning NULL.
- **Empty Tree:** If the tree is empty, there is no LCA to find, and the function should return NULL.
- **Single Node Tree:** In a tree with only one node, that node is the LCA if it matches one of the target nodes.
- **Skewed Trees:** Test with left or right skewed trees to ensure the algorithm handles deep recursion correctly without stack overflow issues.

How It Differs from the Lowest Common Ancestor of a Binary Search Tree

While both problems aim to find the Lowest Common Ancestor (LCA) of two nodes within a binary tree structure, the presence or absence of Binary Search Tree (BST) properties fundamentally changes the approach and efficiency of the solution.

Aspect	Lowest Common Ancestor of a Binary Tree	Lowest Common Ancestor of a Binary Search Tree
Tree Structure	Applies to any binary tree, which does not enforce any specific ordering of node values.	Applies specifically to Binary Search Trees (BSTs), which maintain an ordered structure where left children are less than the node and right children are greater.
Node Value Constraints	No constraints on node values; nodes can have any integer values, including duplicates.	Nodes follow BST properties: left subtree nodes have values less than the parent node, and right subtree nodes have values greater than the parent node.
Algorithmic Approach	Requires traversing the entire tree since there's no inherent order to exploit. Typically uses Depth-First Search (DFS) with recursion or iterative methods.	Can leverage the ordered nature of BSTs to optimize the search, often resulting in a more efficient solution that doesn't require traversing the entire tree.
Time Complexity	$O(N)$, where N is the number of nodes in the tree, as it may require visiting all nodes.	$O(h)$, where h is the height of the BST. In a balanced BST, this is $O(\log N)$, but in the worst case (skewed tree), it can degrade to $O(N)$.
Space Complexity	$O(H)$, where H is the height of the tree, due to the recursive call stack or iterative stack usage.	$O(1)$ for the iterative approach and $O(h)$ for the recursive approach, similar to the general binary tree but often more efficient in practice due to the ordered traversal.

Problem 17.13 Binary Tree Maximum Path Sum

Problem Statement

Given a binary tree, find the maximum path sum within it. The path can start and end at any node in the tree, must follow the parent-child connections, and must contain at least one node.

Find the maximum path sum in a binary tree using recursive depth-first search.

Algorithmic Approach

To solve this problem, we can use a recursive depth-first search (DFS) algorithm. The recursive function will compute two things for each node:

1. The maximum path sum that includes the current node and extends to one side (either left or right).
2. The maximum path sum that can be formed using the current node as the highest point (turning point) in a path.

1. Recursive Depth-First Search (DFS):

- **Define a Helper Function:** Create a helper function that returns the maximum gain from each node.
- **Compute Maximum Gains:**
 - (a) If the current node is NULL, return 0 as it contributes nothing to the path sum.
 - (b) Recursively calculate the maximum gain from the left and right subtrees. If a subtree's maximum gain is negative, consider it as 0 to avoid decreasing the overall path sum.
 - (c) Calculate the price of the new path that passes through the current node by adding the node's value to the left and right gains.
 - (d) Update the global maximum path sum if the new path's price is higher than the current maximum.
 - (e) Return the node's value plus the greater of the left or right gains to contribute to the parent node's path sum.

Complexities

- **Time Complexity:** $O(N)$, where N is the number of nodes in the tree. Each node is visited exactly once during the traversal.
- **Space Complexity:** $O(H)$, where H is the height of the tree, due to the recursive call stack. In the worst case of a skewed tree, the space complexity becomes $O(N)$.

Python Implementation

```
# 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:
        def max_gain(node: TreeNode) -> int:
            nonlocal max_path_sum
            if not node:
                return 0
            left_max = max(max_gain(node.left), 0)
            right_max = max(max_gain(node.right), 0)
            max_path_sum = max(max_path_sum, node.val + left_max + right_max)
            return node.val + max(left_max, right_max)
```

Implementing a recursive DFS solution to find the maximum path sum.

```

        return 0

    # Recursive call on children
    left_gain = max(max_gain(node.left), 0)
    right_gain = max(max_gain(node.right), 0)

    # Path sum with the current node as the
    #   ↪ highest point
    price_newpath = node.val + left_gain +
    #   ↪ right_gain

    # Update the global maximum path sum if the
    #   ↪ new path is better
    max_path_sum = max(max_path_sum,
    #   ↪ price_newpath)

    # Return the maximum gain the current node
    #   ↪ can contribute to the path
    return node.val + max(left_gain, right_gain
    #   ↪ )

max_path_sum = float('-inf')
max_gain(root)
return max_path_sum

# Example usage:
# Constructing the following binary tree
#      1
#     / \
#    2   3
#   / \
#  4   5

root = TreeNode(1)
root.left = TreeNode(2, TreeNode(4), TreeNode(5))
root.right = TreeNode(3)

solution = Solution()
print(solution.maxPathSum(root))  # Output: 11 (4 + 2 +
#   ↪ 5)

```

Explanation

The function `maxPathSum` calculates the maximum path sum in a binary tree by performing a recursive depth-first search (DFS).

- **Helper Function `max_gain`:**

- ****Base Case**:** If the current node is NULL, it contributes 0 to the path sum.
- ****Recursive Calls**:**

1. Recursively compute the maximum gain from the left child. If the gain is negative, consider it as 0 to avoid reducing the path sum.
 2. Recursively compute the maximum gain from the right child. Similarly, treat negative gains as 0.
- **Calculate New Path Sum**: The potential new path sum that passes through the current node is the sum of the node's value and the gains from both left and right children.
 - **Update Global Maximum**: If the new path sum is greater than the current global maximum, update it.
 - **Return Maximum Gain**: Return the node's value plus the greater of the left or right gain. This value is used to compute the path sums for parent nodes.

Why This Approach

- **Efficiency**: This recursive DFS approach ensures that each node is visited only once, achieving linear time complexity.
- **Leveraging Tree Properties**: By considering only positive gains from subtrees, the algorithm efficiently avoids paths that would decrease the overall sum.
- **Simplicity and Elegance**: The recursive nature of the solution aligns naturally with the hierarchical structure of trees, resulting in clear and maintainable code.

Alternative Approaches

An alternative method involves dynamic programming where each node stores additional information such as the maximum path sum that can be achieved through it. However, this approach typically requires more complex data structures and does not offer a better time complexity compared to the recursive DFS method.

Similar Problems to This One

Similar tree-related problems include finding the diameter of a binary tree, calculating the maximum depth of a binary tree, and determining the lowest common ancestor of two nodes in a binary tree.

Things to Keep in Mind and Tricks

- **Handling Negative Values**: By using `max(gain, 0)`, the algorithm effectively ignores paths that would decrease the overall path sum.
- **Global Variable Usage**: Utilizing a non-local or global variable to keep track of the maximum path sum encountered during traversal simplifies the update mechanism.

- **Recursive Call Stack:** Be mindful of the recursive depth, especially for very deep or skewed trees, to avoid stack overflow issues.
- **Base Cases Are Crucial:** Always handle NULL nodes to prevent incorrect calculations and potential runtime errors.

Corner and Special Cases to Test When Writing the Code

- **All Negative Node Values:** Ensure the algorithm correctly identifies the least negative value as the maximum path sum.
- **Single Node Tree:** The maximum path sum should be the value of the single node.
- **Left-Skewed or Right-Skewed Trees:** Test trees that are completely unbalanced to ensure the algorithm handles deep recursion and large stacks.
- **Multiple Paths with the Same Sum:** Verify that the algorithm correctly identifies one of the maximum paths.
- **Empty Tree:** Handle gracefully, possibly by returning 0 or an error, depending on the problem constraints.
- **Balanced Trees:** Ensure that the algorithm correctly computes the maximum path sum in perfectly balanced trees.
- **Large Trees:** Test with a large number of nodes to ensure that the implementation performs efficiently without exceeding memory limits.

Problem 17.14 Subtree of Another Tree

Problem Statement

Given two non-empty binary trees, s and t , check whether tree t has exactly the same structure and node values with a subtree of s . A subtree of s is a tree consisting of a node in s and all of its descendants. The tree s could also be considered as a subtree of itself.

Determine if one binary tree is a subtree of another using recursive strategies.

Algorithmic Approach

To solve this problem, we can utilize a recursive approach that traverses tree s and checks for the presence of tree t as a subtree at each node. The approach involves two main functions:

1. A function to traverse tree s and initiate the subtree comparison.
2. A helper function to compare two trees for identical structure and node values.

1. Recursive Traversal and Comparison:

- **Traverse Tree s:** Start from the root of s and traverse the tree in a depth-first manner.
- **Initiate Comparison:**
 - (a) At each node in s , check if the subtree rooted at that node matches tree t using the helper comparison function.
 - (b) If a match is found, return `True`.
 - (c) Otherwise, continue traversing the left and right subtrees of the current node.

2. Helper Function for Tree Comparison:

- **Compare Node Values:**
 - (a) If both nodes being compared are `NULL`, return `True` as empty trees are identical.
 - (b) If one node is `NULL` and the other is not, return `False`.
 - (c) If the values of the current nodes do not match, return `False`.
- **Recursively Compare Subtrees:**
 - (a) Recursively compare the left children of both nodes.
 - (b) Recursively compare the right children of both nodes.
 - (c) Return `True` only if both left and right subtree comparisons return `True`.

Complexities

- **Time Complexity:** $O(N \times M)$, where N is the number of nodes in tree s and M is the number of nodes in tree t . In the worst case, for each node in s , we may need to compare it with all nodes in t .
- **Space Complexity:** $O(H_s + H_t)$, where H_s is the height of tree s and H_t is the height of tree t . This accounts for the recursive call stacks of both the traversal and comparison functions.

Python Implementation

```
# 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:
```

Implementing a recursive solution to check for subtree existence.

```

def isSubtree(self, s: TreeNode, t: TreeNode) ->
    ↵ bool:
    if not s:
        return False
    if self.isSameTree(s, t):
        return True
    return self.isSubtree(s.left, t) or self.
        ↵ isSubtree(s.right, t)

def isSameTree(self, s: TreeNode, t: TreeNode) ->
    ↵ bool:
    if not s and not t:
        return True
    if not s or not t:
        return False
    if s.val != t.val:
        return False
    return self.isSameTree(s.left, t.left) and self
        ↵ .isSameTree(s.right, t.right)

# Example usage:
# Constructing tree s
#      3
#     / \
#    4   5
#   / \
#  1   2

s = TreeNode(3)
s.left = TreeNode(4, TreeNode(1), TreeNode(2))
s.right = TreeNode(5)

# Constructing tree t
#      4
#     /
#    1

t = TreeNode(4, TreeNode(1), TreeNode(2))

solution = Solution()
print(solution.isSubtree(s, t))  # Output: True

# Constructing tree t2
#      4
#     /
#    1

t2 = TreeNode(4, TreeNode(1), None)
print(solution.isSubtree(s, t2))  # Output: False

```

Explanation

The function `isSubtree` determines whether tree `t` is a subtree of tree `s` by recursively traversing tree `s` and comparing each node's subtree with tree `t`.

- **Main Function `isSubtree`:**

- ****Base Case**:** If the current node in `s` is `NULL`, return `False` as tree `t` cannot be a subtree.
- ****Subtree Check**:**
 1. Use the helper function `isSameTree` to check if the subtree rooted at the current node of `s` matches tree `t`.
 2. If a match is found, return `True`.
 3. Otherwise, recursively check the left and right subtrees of the current node.

- **Helper Function `isSameTree`:**

- ****Base Cases**:**
 1. If both nodes being compared are `NULL`, return `True` as empty trees are identical.
 2. If one node is `NULL` and the other is not, return `False`.
- ****Value Comparison**:** If the values of the current nodes do not match, return `False`.
- ****Recursive Comparison**:** Recursively compare the left children and the right children of both nodes.

Why This Approach

- **Leverages Tree Structure:** By utilizing the inherent structure of binary trees, the recursive approach efficiently navigates through tree `s` to find potential matching subtrees.
- **Simplicity and Readability:** The recursive functions are straightforward, making the implementation easy to understand and maintain.
- **Optimal Traversal:** Although the time complexity is $O(N \times M)$, where N and M are the number of nodes in `s` and `t` respectively, this approach avoids unnecessary comparisons by pruning branches that cannot contain the subtree.

Alternative Approaches

An alternative method involves serializing both trees (for example, using pre-order traversal with null markers) and then checking if the serialized string of tree `t` is a substring of the serialized string of tree `s`. This approach leverages string matching

Problem 17.15 Construct Binary Tree from Preorder and In-order Traversal

Problem Statement

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.

Reconstruct a binary tree given its preorder and inorder traversal sequences using recursive strategies.

Algorithmic Approach

To reconstruct the binary tree from its preorder and inorder traversal sequences, we can utilize the properties of these traversals:

1. Understanding Traversal Properties:

- **Preorder Traversal:** Visits nodes in the order of Root, Left, Right.
- **Inorder Traversal:** Visits nodes in the order of Left, Root, Right.

2. Recursive Reconstruction:

- **Identify Root:** The first element in the preorder traversal is the root of the tree.
- **Locate Root in Inorder Traversal:** Find the index of the root in the inorder traversal. This index divides the inorder list into left and right subtrees.
- **Determine Subtree Sizes:** The number of elements to the left of the root in inorder traversal corresponds to the number of nodes in the left subtree.
- **Recursively Construct Subtrees:**
 - (a) **Left Subtree:** Use the next set of elements in preorder corresponding to the left subtree and the left segment of inorder traversal.
 - (b) **Right Subtree:** Use the subsequent elements in preorder corresponding to the right subtree and the right segment of inorder traversal.

Complexities

- **Time Complexity:** $O(N)$, where N is the number of nodes in the tree. Each node is processed exactly once.
- **Space Complexity:** $O(N)$, due to the space required for the recursion stack and the hashmap storing inorder indices.

Python Implementation

```
# Definition for a binary tree node.
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
```

Implementing a recursive solution to reconstruct the binary tree from traversal sequences.

```

        self.val = val
        self.left = left
        self.right = right

class Solution:
    def buildTree(self, preorder: list, inorder: list)
        ↪ -> TreeNode:
        if not preorder or not inorder:
            return None

        # Map from value to its index in inorder
        ↪ traversal for quick lookup
        inorder_index_map = {value: idx for idx, value
            ↪ in enumerate(inorder)}

        # Recursive helper function
        def array_to_tree(pre_left, pre_right, in_left,
            ↪ in_right):
            if pre_left > pre_right:
                return None

            # Root value is the first element in the
            ↪ current preorder slice
            root_val = preorder[pre_left]
            root = TreeNode(root_val)

            # Index of the root in inorder traversal
            in_root_index = inorder_index_map[root_val]

            # Number of nodes in the left subtree
            left_tree_size = in_root_index - in_left

            # Recursively build the left and right
            ↪ subtrees
            root.left = array_to_tree(pre_left + 1,
                ↪ pre_left + left_tree_size, in_left,
                ↪ in_root_index - 1)
            root.right = array_to_tree(pre_left +
                ↪ left_tree_size + 1, pre_right,
                ↪ in_root_index + 1, in_right)

            return root

        # Initialize recursion boundaries
        return array_to_tree(0, len(preorder) - 1, 0,
            ↪ len(inorder) - 1)

# Example usage:
# Preorder traversal: [3,9,20,15,7]
# Inorder traversal: [9,3,15,20,7]
preorder = [3,9,20,15,7]

```

```

inorder = [9,3,15,20,7]

solution = Solution()
tree_root = solution.buildTree(preorder, inorder)

# Function to print inorder traversal of the
#   ↪ constructed tree
def print_inorder(node):
    if not node:
        return
    print_inorder(node.left)
    print(node.val, end=' ')
    print_inorder(node.right)

print_inorder(tree_root) # Output: 9 3 15 20 7

```

Explanation

The function `buildTree` reconstructs the binary tree from its preorder and inorder traversal lists by leveraging the following insights:

- **Root Identification**:** The first element in the preorder list is always the root of the tree or subtree being constructed.
- **Inorder Index Mapping**:** By creating a hashmap (`inorder_index_map`) that maps each value to its index in the inorder list, we can quickly determine the boundaries of left and right subtrees.
- **Recursive Construction**:**
 - **Base Case**:** If the current slice of the preorder list is empty (`pre_left > pre_right`), return `None`, indicating no subtree exists.
 - **Node Creation**:** Create a new `TreeNode` with the root value.
 - **Left Subtree Size**:** Calculate the number of nodes in the left subtree using the root's index in the inorder list.
 - **Recursive Calls**:**
 1. **Left Subtree**:** Construct the left subtree using the corresponding slices of preorder and inorder lists.
 2. **Right Subtree**:** Similarly, construct the right subtree.

Why This Approach

- **Efficiency:** By using a hashmap to store inorder indices, we reduce the time complexity of searching for root positions from $O(N)$ to $O(1)$, ensuring overall linear time complexity.

- **Simplicity and Clarity:** The recursive approach aligns naturally with the hierarchical structure of binary trees, making the code intuitive and easy to understand.
- **Optimal Space Utilization:** Although recursion introduces additional space complexity due to the call stack, this approach remains optimal for tree reconstruction problems.

Alternative Approaches

An alternative method involves using iterative tree construction techniques with stacks to simulate the recursion process. However, this approach can be more complex and less intuitive compared to the straightforward recursive method. Additionally, it may not offer significant performance benefits over the recursive approach.

Similar Problems to This One

Similar tree-related problems include:

- Binary Tree Preorder Traversal
- Binary Tree Inorder Traversal
- Binary Tree Postorder Traversal
- Validate Binary Search Tree
- Lowest Common Ancestor of a Binary Search Tree

Things to Keep in Mind and Tricks

- **Unique Elements Assumption:** This approach assumes that all elements in the tree are unique. If duplicates are allowed, additional handling is required to correctly identify subtree boundaries.
- **Preorder and Inorder Traversal Validity:** Ensure that the provided preorder and inorder traversal lists are valid and correspond to the same binary tree.
- **Recursive Boundaries:** Carefully manage the indices for the current slices of preorder and inorder lists to avoid incorrect subtree constructions.
- **Handling Edge Cases:** Consider scenarios where the tree is empty, has only one node, or is highly unbalanced (e.g., skewed trees).

Corner and Special Cases to Test When Writing the Code

- **Empty Tree:** Both preorder and inorder lists are empty. The function should return `None`.

- **Single Node Tree:** Preorder and inorder lists contain only one element. The function should correctly create a single-node tree.
- **Left-Skewed Tree:** All nodes have only left children. Verify that the tree is constructed correctly without missing nodes.
- **Right-Skewed Tree:** All nodes have only right children. Ensure accurate tree reconstruction.
- **Balanced Tree:** A perfectly balanced tree to confirm that both left and right subtrees are constructed accurately.
- **Invalid Traversals:** Preorder and inorder lists that do not correspond to the same tree. The function should handle such cases gracefully, potentially by returning an error or None.
- **Large Tree:** Test with a large number of nodes to assess the performance and recursion depth handling.
- **Duplicate Values:** If duplicates are allowed, ensure that the function correctly identifies the positions of duplicate elements in the inorder list.

Problem 17.16 Serialize and Deserialize Binary Tree

Problem Statement

Design an algorithm to serialize and deserialize a binary tree. Serialization is the process of converting a binary tree into a string representation, and deserialization is the reverse process of reconstructing the binary tree from the string. The serialized string should uniquely represent the original binary tree structure and node values.

Convert a binary tree to a string and back using traversal-based serialization and deserialization methods.

Algorithmic Approach

To serialize and deserialize a binary tree effectively, we can utilize **Breadth-First Search (BFS)** or **Depth-First Search (DFS)** traversal methods. Here, we'll focus on the BFS approach using level-order traversal, which is intuitive and handles trees with varying structures, including those with missing nodes.

1. Serialization (Tree to String):

- **Level-Order Traversal:** Traverse the tree level by level using a queue.
- **Handling Null Nodes:** Represent null (absent) children with a sentinel value (e.g., '#') to preserve the tree structure.
- **String Construction:** Append node values to the serialized string, separated by commas.

2. Deserialization (String to Tree):

- **String Splitting**: Split the serialized string by commas to retrieve node values.
- **Reconstruction Using Queue**: Use a queue to keep track of nodes whose children are to be assigned.
- **Node Assignment**: Iterate through the split values, creating child nodes or assigning nulls based on the sentinel values.

Complexities

- **Time Complexity**: $O(N)$, where N is the number of nodes in the binary tree. Both serialization and deserialization processes visit each node exactly once.
- **Space Complexity**: $O(N)$, due to the storage required for the serialized string and the queue used during traversal.

Python Implementation

```
from collections import deque

# 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 Codec:
    def serialize(self, root: TreeNode) -> str:
        """Encodes a tree to a single string using
           ↪ level-order traversal."""
        if not root:
            return ""

        serialized = []
        queue = deque([root])

        while queue:
            node = queue.popleft()
            if node:
                serialized.append(str(node.val))
                queue.append(node.left)
                queue.append(node.right)
            else:
                serialized.append("#")

        # Remove trailing "#" to optimize the
        # ↪ serialized string
        while serialized and serialized[-1] == "#":
            serialized.pop()
```

Implementing BFS-based serialization and deserialization for binary trees.


```

deserialized_root = codec.deserialize(serialized)

# Function to print level-order traversal of the tree
def print_level_order(node):
    if not node:
        print("Empty Tree")
        return
    queue = deque([node])
    result = []
    while queue:
        current = queue.popleft()
        if current:
            result.append(str(current.val))
            queue.append(current.left)
            queue.append(current.right)
        else:
            result.append("#")
    # Remove trailing "#" for clean output
    while result and result[-1] == "#":
        result.pop()
    print("Deserialized Tree Level-Order:", ', '.join(
        result))

print_level_order(deserialized_root)  # Output:
                                    # "1,2,3,#,#,4,5"

```

Explanation

The ‘Codec‘ class provides two primary methods: ‘serialize‘ and ‘deserialize‘.

- **Serialization:**

- ****Level-Order Traversal**:** The ‘serialize‘ method performs a level-order traversal of the binary tree using a queue (‘deque‘).
- ****Handling Null Nodes**:** When a node is ‘None‘, it appends a sentinel value (‘#‘) to the serialized list to indicate the absence of a child, preserving the tree structure.
- ****String Construction**:** After traversal, it joins the list into a comma-separated string. Trailing sentinel values are removed to optimize the string.

- **Deserialization:**

- ****String Splitting**:** The ‘deserialize‘ method splits the serialized string by commas to retrieve node values.
- ****Reconstruction Using Queue**:** It initializes the root node and uses a queue to keep track of nodes whose children need to be assigned.

- ****Node Assignment**:** Iterates through the split values, creating left and right child nodes or assigning ‘None’ based on the sentinel values. This reconstructs the original binary tree structure.

Why This Approach

- **Efficiency:** Both serialization and deserialization operations run in linear time relative to the number of nodes, ensuring scalability for large trees.
- **Simplicity and Clarity:** The BFS-based approach is straightforward, making the implementation easy to understand and maintain.
- **Preservation of Structure:** By including sentinel values for null nodes, the serialized string accurately captures the tree’s structure, ensuring faithful deserialization.

Alternative Approaches

An alternative method involves using **Depth-First Search (DFS)**, such as pre-order traversal with null markers, to serialize and deserialize the tree. While DFS can also achieve linear time complexity, BFS is often preferred for its intuitive handling of tree levels and straightforward reconstruction process. Additionally, DFS may require managing recursion depth, which can be a limitation for very deep trees.

Similar Problems to This One

Similar tree-related problems include:

- Construct Binary Tree from Preorder and Inorder Traversal
- Binary Tree Maximum Path Sum
- Subtree of Another Tree
- Lowest Common Ancestor of a Binary Tree

Things to Keep in Mind and Tricks

- **Handling Null Nodes:** Use a consistent sentinel value (e.g., ‘#’) to represent null nodes during serialization to maintain tree structure integrity.
- **Traversal Choice:** Choose BFS for a level-order approach or DFS for a depth-based approach based on the specific requirements and constraints of the problem.
- **Optimizing Serialized String:** Remove trailing sentinel values to optimize the serialized string without losing necessary structural information.

- **Hashmaps for Efficiency:** When reconstructing trees, using hashmaps to store inorder indices can significantly speed up node lookups and assignments.
- **Edge Cases:** Always consider edge cases such as empty trees, single-node trees, and highly unbalanced trees to ensure robustness.

Corner and Special Cases to Test When Writing the Code

- **Empty Tree:** Both preorder and inorder lists are empty. The function should return ‘None’.
- **Single Node Tree:** Preorder and inorder lists contain only one element. The function should correctly create a single-node tree.
- **Left-Skewed Tree:** All nodes have only left children. Verify that the tree is constructed correctly without missing nodes.
- **Right-Skewed Tree:** All nodes have only right children. Ensure accurate tree reconstruction.
- **Balanced Tree:** A perfectly balanced tree to confirm that both left and right subtrees are constructed accurately.
- **Invalid Traversals:** Preorder and inorder lists that do not correspond to the same tree. The function should handle such cases gracefully, potentially by returning an error or ‘None’.
- **Large Tree:** Test with a large number of nodes to assess the performance and recursion depth handling.
- **Duplicate Values:** If duplicates are allowed, ensure that the function correctly identifies the positions of duplicate elements in the inorder list.

Chapter 18

Graphs

Visual Representations

Directed Undirected Weighted

Graphs are fundamental data structures used to model pairwise relations between objects. They are widely applicable in various domains such as computer networks, social networks, and transportation systems.

Figure 18.1: Common Graph Types and Their Representations

Time and Space Complexity Analysis

Operation	Adjacency Matrix	Adjacency List	Edge List
Add Edge	O(1)	O(1)	O(1)
Remove Edge	O(1)	O(d)	O(E)
Query Edge	O(1)	O(d)	O(E)
Space	O(V ²)	O(V + E)	O(E)

Table 18.1: Time and Space Complexity Comparison

Implementation Examples

```
class Graph:
    def __init__(self, directed=False):
        self.graph = defaultdict(list)
        self.directed = directed

    def add_edge(self, u, v, weight=1):
        self.graph[u].append((v, weight))
        if not self.directed:
            self.graph[v].append((u, weight))

    def bfs(self, start):
        visited = set()
        queue = deque([start])
```

```

while queue:
    vertex = queue.popleft()
    if vertex not in visited:
        visited.add(vertex)
        queue.extend(v for v, _ in self.graph[
            ↪ vertex]
                     if v not in visited)
return visited

```

Common Graph Patterns

- Cycle Detection:*
- Using DFS with color marking
 - Using Union-Find data structure
 - Floyd's Cycle-Finding Algorithm

- Shortest Path Patterns:*
- Single-source shortest path

- All-pairs shortest path
- Bidirectional search

- Network Flow:*
- Ford-Fulkerson Algorithm

- Edmonds-Karp Algorithm
- Push-Relabel Algorithm

Advanced Topics

- Graph Coloring:*
- Vertex coloring algorithms
 - Edge coloring algorithms
 - Map coloring problems

- Maximum Flow:*
- Maximum flow-minimum cut theorem

- Residual graphs
- Augmenting paths

- Graph Matching:*
- Bipartite matching

- Maximum matching
- Perfect matching

Optimization Techniques

- Memory Optimization:*
- Bit manipulation for graph representation

- Compressed sparse row format
- Memory-efficient adjacency lists

Performance Optimization:

- Parallel graph algorithms
- Cache-friendly implementations
- Graph partitioning

Real-World Case Studies

Social Network Analysis:

- Friend recommendation systems
- Community detection
- Influence propagation

Route Planning:

- GPS navigation systems
- Traffic optimization
- Public transportation routing

Graphs consist of a set of nodes (also called vertices) and a set of edges connecting pairs of nodes. They can be categorized based on their properties:

- **Directed vs. Undirected Graphs:** In directed graphs, edges have a direction, indicating a one-way relationship. In undirected graphs, edges have no direction, representing a two-way relationship.
- **Weighted vs. Unweighted Graphs:** Weighted graphs have edges associated with weights or costs, while unweighted graphs do not.
- **Cyclic vs. Acyclic Graphs:** Cyclic graphs contain at least one cycle, whereas acyclic graphs do not.
- **Connected vs. Disconnected Graphs:** In connected graphs, there is a path between every pair of nodes. Disconnected graphs consist of multiple connected components.

Graph Representations

Graphs can be represented in various ways, each with its own advantages:

- **Adjacency Matrix:** A 2D array where each cell (i, j) indicates the presence (and possibly the weight) of an edge between nodes i and j . This representation allows for quick edge lookups but consumes $O(n^2)$ space.

- **Adjacency List:** An array of lists where each list contains the neighbors of a node. This representation is more space-efficient for sparse graphs and allows for efficient iteration over neighbors.
- **Edge List:** A list of all edges in the graph, typically represented as pairs (or triplets if weighted). This representation is simple but less efficient for certain operations.

Graph Traversal Algorithms

Graph traversal algorithms are essential for exploring the nodes and edges of a graph. The two primary traversal methods are:

1. **Depth-First Search (DFS):** Explores as far as possible along each branch before backtracking. DFS can be implemented using recursion or an explicit stack.
2. **Breadth-First Search (BFS):** Explores all neighbors at the current depth before moving to nodes at the next depth level. BFS is typically implemented using a queue.

Advanced Graph Algorithms

Beyond basic traversal, several advanced algorithms solve complex problems on graphs:

- **Dijkstra's Algorithm:** Finds the shortest path from a single source to all other nodes in a weighted graph with non-negative edge weights.
- **Bellman-Ford Algorithm:** Computes shortest paths from a single source in graphs that may have negative edge weights.
- **Floyd-Warshall Algorithm:** Finds shortest paths between all pairs of nodes in a weighted graph.
- **Kruskal's and Prim's Algorithms:** Used for finding the Minimum Spanning Tree (MST) of a weighted, undirected graph.
- **Topological Sort:** Orders the nodes of a directed acyclic graph (DAG) such that for every directed edge uv , node u comes before v in the ordering.
- **Tarjan's Algorithm:** Identifies strongly connected components in a directed graph.

Graph Theory Concepts

Understanding fundamental graph theory concepts is crucial for solving graph-related problems:

- **Degree of a Node:** The number of edges incident to a node. In directed graphs, this includes in-degree and out-degree.
- **Path and Cycle:** A path is a sequence of nodes where each consecutive pair is connected by an edge. A cycle is a path that starts and ends at the same node without repeating any edges or nodes.
- **Connected Components:** Subsets of nodes where each node is reachable from any other node in the same subset.
- **Bipartite Graph:** A graph whose nodes can be divided into two disjoint sets such that every edge connects a node from one set to the other.
- **Graph Isomorphism:** Determines whether two graphs are structurally identical, meaning there's a one-to-one correspondence between their node sets that preserves adjacency.

Applications of Graphs

Graphs are versatile structures used in various real-world applications:

- **Social Networks:** Modeling relationships between individuals.
- **Computer Networks:** Representing the connectivity of devices.
- **Transportation Systems:** Modeling routes and connections between locations.
- **Recommendation Systems:** Suggesting products or content based on user interactions.
- **Biological Networks:** Representing interactions within biological systems, such as protein-protein interaction networks.

Graph Problems

The following problems explore various graph concepts and algorithms. Each problem is introduced in its respective section.

18.1 Number of Islands

The **Number of Islands** problem is a classical algorithmic challenge that requires counting distinct clusters of connected components on a two-dimensional grid.

This problem is an excellent exercise for understanding how to perform depth-first search (DFS) or breadth-first search (BFS) on a grid.

This problem is a classic example of using depth-first search (DFS) or breadth-first search (BFS) to identify and count connected components in a grid.

Problem Statement

Given a 2D grid map of '1's (land) and '0's (water), the task is to count the number of islands. An island is defined as a group of adjacent lands connected horizontally or vertically. It is assumed that the four edges of the grid are surrounded by water.

Examples

Example 1:

Input :

```
11110
11010
11000
00000
```

Output: 1

Example 2:

Input :

```
11000
11000
00100
00011
```

Output: 3

LeetCode link: [Number of Islands](#)

[LeetCode Link]

[GeeksForGeeks Link]

[HackerRank Link]

[CodeSignal Link]

[InterviewBit Link]

[Educative Link]

[Codewars Link]

Algorithmic Approach

To count the number of islands, we can iterate over each cell in the grid. When we encounter a '1', we trigger a DFS or BFS to mark all adjacent land cells (also '1's). This process should recursively continue until it encounters water ('0') or the edge of the grid. Each distinct DFS or BFS traversal corresponds to one island, and thus we increment our count of islands accordingly.

DFS and BFS are effective for exploring all connected components in a grid, ensuring each island is counted exactly once.

Complexities

- **Time Complexity:** The overall time complexity is $O(M \times N)$, where M and N are the number of rows and columns in the grid, respectively. Each cell is visited once during the traversal.
- **Space Complexity:** The space complexity is $O(M \times N)$ in the worst-case scenario for the DFS recursion stack or the BFS queue when the grid is filled with land.

Python Implementation

Below is the complete Python code that uses depth-first search for the `number_of_islands` function:

```
def number_of_islands(grid):
    if not grid:
        return 0

    def dfs(grid, i, j):
        if i < 0 or i >= len(grid) or j < 0 or j >= len(grid[0]) or grid[i][j] == '0':
            return
        grid[i][j] = '0' # Mark as visited
        dfs(grid, i - 1, j) # Up
        dfs(grid, i + 1, j) # Down
        dfs(grid, i, j - 1) # Left
        dfs(grid, i, j + 1) # Right

        count = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == '1':
                    count += 1
                    dfs(grid, i, j)

        return count

    # Example usage:
    grid = [
        ['1', '1', '0', '0', '0'],
        ['1', '1', '0', '0', '0'],
        ['0', '0', '1', '0', '0'],
        ['0', '0', '0', '1', '1']
    ]
    print(number_of_islands(grid)) # Output: 3
```

Implementing DFS or BFS requires careful handling of grid boundaries and visited cells to avoid infinite loops and ensure accurate counting.

```
class Solution(object):
    def numIslands(self, grid):
        """
        :type grid: List[List[str]]
        :rtype: int
        """

        def dfs(i, j):
            if i < 0 or j < 0 or i >= len(grid) or j >= len(grid[0]) or grid[i][j] == '0':
                return
            grid[i][j] = '0' # mark as visited
            dfs(i + 1, j)
            dfs(i - 1, j)
```

```

    dfs(i, j + 1)
    dfs(i, j - 1)

if not grid:
    return 0

count = 0
for i in range(len(grid)):
    for j in range(len(grid[0])):
        if grid[i][j] == '1':
            dfs(i, j)
            count += 1

return count

```

This Python function performs DFS to explore all connecting lands for each unvisited island. Marking land as visited by replacing '1's with '0's prevents counting the same land twice.

Explanation

The provided Python implementation defines a function `number_of_islands` which takes a 2D grid as its parameter. Here's a step-by-step breakdown of the implementation:

- **Edge Case Handling:**
 - If the input grid is empty, return 0 as there are no islands.
- **Depth-First Search (DFS) Function:**
 - The nested `dfs` function takes the current cell indices `i` and `j`.
 - It checks for boundary conditions and whether the current cell is water ('0'). If so, it returns immediately.
 - Otherwise, it marks the current cell as visited by setting it to '0'.
 - It then recursively calls itself for all four adjacent cells (up, down, left, right).
- **Counting Islands:**
 - Initialize a counter `count` to 0.
 - Iterate through each cell in the grid using nested loops.
 - When a land cell ('1') is found, increment the `count` and initiate a DFS from that cell to mark all connected land cells as visited.
- **Return Value:**
 - After traversing the entire grid, return the `count`, which represents the total number of distinct islands.

Why This Approach

The DFS approach was chosen because it is a straightforward way to explore and mark connected components in a grid. It efficiently traverses nodes and their neighbors, using recursion to handle the navigation through adjacent lands, which makes the implementation concise and elegant. Additionally, DFS naturally fits the problem's requirement to explore all connected land cells starting from a given land cell, ensuring that each island is counted exactly once.

Alternative Approaches

An alternative approach to solving this problem could be using BFS. Instead of using recursion, a queue is used to store and visit the cells in level order. This approach might be easier to understand for individuals less comfortable with recursion and has the same time and space complexity as DFS.

Another alternative is to use the Union-Find (Disjoint Set Union) data structure to group connected land cells and count the number of distinct sets, which represent the islands. However, this method is generally more complex to implement and may not offer significant performance benefits over DFS or BFS for this particular problem.

Similar Problems to This One

Similar problems to "Number of Islands" include:

- **Max Area of Island:** Find the maximum area of an island in the grid.
- **Surrounded Regions:** Capture all regions surrounded by 'X's by flipping surrounded 'O's to 'X's.
- **Walls and Gates:** Fill each empty room with the distance to its nearest gate.
- **Pacific Atlantic Water Flow:** Determine the cells from which water can flow to both the Pacific and Atlantic oceans.
- **Clone Graph:** Clone an undirected graph.

Things to Keep in Mind and Tricks

When solving problems like "Number of Islands," keep the following tips in mind:

- **Marking Visited Cells:** To avoid revisiting the same cell, mark it as visited. This can be done by modifying the input grid or by maintaining a separate visited matrix.
- **Boundary Checks:** Always ensure that your DFS or BFS does not go out of the grid boundaries to prevent index errors.
- **Choosing DFS vs. BFS:** Both DFS and BFS are suitable for this problem. DFS can be implemented recursively or using a stack, while BFS uses a queue. Choose the one you are more comfortable with.
- **Iterative vs. Recursive DFS:** Recursive DFS is more concise but may lead to stack overflow for very large grids. Iterative DFS using a stack can be more robust.
- **Optimizing Space:** If modifying the input grid is allowed, it can save space by eliminating the need for an additional visited matrix.

Corner and Special Cases to Test When Writing the Code

When writing the code, consider testing the following corner cases to ensure robustness:

- **Empty Grid:** An empty grid should return 0 islands.
- **Single Cell:** Grids with only one cell, either land ('1') or water ('0').
- **All Water:** Grids with all cells as water should return 0 islands.
- **All Land:** Grids with all cells as land should return 1 island.
- **Non-Rectangular Grids:** Although the problem typically assumes a rectangular grid, ensure that your code can handle grids where rows have different lengths if not explicitly constrained.
- **Multiple Islands:** Grids with multiple distinct islands to verify correct counting.
- **Large Grids:** Very large grids to test the performance and stack limits if using recursive DFS.
- **Islands with Complex Shapes:** Islands that are not just rectangular or square but have L-shapes, T-shapes, etc., to ensure all connected cells are properly identified.
- **Edge Islands:** Islands that touch the borders of the grid to ensure boundary handling is correct.

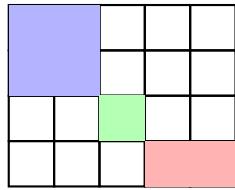


Figure 18.2: Island Identification in Grid

Visual representation of Example 2

Visual Representation

Implementation Variants

- **BFS Implementation:**

```
from collections import deque

def numIslands_bfs(grid):
    if not grid:
        return 0

    def bfs(i, j):
        queue = deque([(i, j)])
        while queue:
            i, j = queue.popleft()
            for ni, nj in [(i+1,j), (i-1,j), (i,j+1), (i,j-1)]:
                if (0 <= ni < len(grid) and 0 <= nj <
                    len(grid[0])) and grid[ni][nj] == '1':
                    grid[ni][nj] = '0'
                    queue.append((ni, nj))

        islands = 0
        for i in range(len(grid)):
            for j in range(len(grid[0])):
                if grid[i][j] == '1':
                    grid[i][j] = '0'
                    bfs(i, j)
                    islands += 1
    return islands
```

- **Union-Find Implementation:**

```
class UnionFind:
    def __init__(self, grid):
        m, n = len(grid), len(grid[0])
        self.parent = [-1] * (m * n)
        self.rank = [0] * (m * n)
```

```

self.count = 0
for i in range(m):
    for j in range(n):
        if grid[i][j] == '1':
            self.parent[i * n + j] = i * n + j
            self.count += 1

```

Performance Comparison

Approach	Time	Space	Best For
DFS	$O(mn)$	$O(mn)$	Simple implementation
BFS	$O(mn)$	$O(\min(m,n))$	Shortest path finding
Union-Find	$O(mn)$	$O(mn)$	Dynamic connectivity

Table 18.2: Comparison of Different Approaches

Corner and Special Cases

- **Empty Grid:** An empty grid should return 0 islands.
- **Single Cell:** Grids with only one cell, either land ('1') or water ('0').

Optimization Techniques

- **Memory Optimization:**
 - In-place modification of grid
 - Iterative DFS to avoid stack overflow
 - Bit manipulation for visited states
- **Performance Optimization:**
 - Direction array for neighbor checking
 - Early termination conditions
 - Cache-friendly traversal patterns

Common Pitfalls and Solutions

- **Stack Overflow:**
 - Problem: Deep recursion in large grids
 - Solution: Use iterative approach or tail recursion
- **Boundary Checking:**

- Problem: Index out of bounds errors
- Solution: Validate indices before accessing grid

Problem 18.1 Clone Graph

The **Clone Graph** problem revolves around creating an exact replica of a given undirected graph. A graph is composed of nodes connected by undirected edges. The objective is to produce a new graph where the structure—arrangement of nodes and edges—of the original graph is identically maintained.

This problem utilizes breadth-first search (BFS) and a hash map to efficiently clone an undirected graph while handling potential cycles.

Problem Statement

The task involves taking a reference to a node in a connected undirected graph and returning a deep copy (clone) of the entire graph. Every node within the graph contains a value ('val') and a list ('neighbors') indicating its adjacent nodes.

```
class Node {
    public int val;
    public List<Node> neighbors;
}
```

You are required to write a function that receives a node from a graph as input and returns a comprehensive copy of the graph. The duplication must preserve the structure of the original graph perfectly. You must ensure not to alter the original graph. Every node in the created graph must hold the same value as its counterpart in the original graph, and all connecting edges should be cloned to link the corresponding nodes in the copy.

Addressing this problem effectively requires managing potential cycles within the graph to prevent infinite loops during the cloning process. Utilizing a hash table or dictionary is a common strategy to track already cloned nodes during the copying process.

LeetCode link: [Clone Graph](#)

[LeetCode Link]
[\[GeeksForGeeks Link\]](#)
[\[HackerRank Link\]](#)
[\[CodeSignal Link\]](#)
[\[InterviewBit Link\]](#)
[\[Educative Link\]](#)
[\[Codewars Link\]](#)

Algorithmic Approach

Main Concept

Cloning a graph typically involves a breadth-first search (BFS) or depth-first search (DFS) traversal while keeping track of copied nodes to avoid infinite loops caused by cycles. A commonly used approach is to maintain a hash map where each original node's reference is mapped to its corresponding cloned node.

1. **Initialize:** Start by checking if the input node is `None`. If so, return `None` as there's nothing to clone.
2. **Hash Map Setup:** Create a hash map (e.g., a dictionary in Python) to store the mapping from original nodes to their cloned counterparts.
3. **BFS Traversal:**
 - Initialize a queue and enqueue the input node.
 - Clone the input node and add it to the hash map.
 - While the queue is not empty:
 - Dequeue a node from the queue.
 - Iterate through its neighbors.
 - For each neighbor:
 - * If the neighbor hasn't been cloned yet:
 - . Clone the neighbor.
 - . Add the cloned neighbor to the hash map.
 - . Enqueue the neighbor for further traversal.
 - * Link the cloned neighbor to the current node's clone by appending it to the 'neighbors' list.

This approach ensures that each node is cloned exactly once and that all connections (edges) between nodes are preserved in the cloned graph.

Using BFS ensures that nodes are cloned level by level, effectively handling graphs with cycles and preventing infinite loops.

Complexities

- **Time Complexity:** $O(N + E)$, where N is the number of nodes and E is the number of edges in the graph. Each node and each edge are visited exactly once during the traversal.
- **Space Complexity:** $O(N)$ for the hash map to store cloned nodes and the queue used for BFS traversal.

Python Implementation

Below is the complete Python code for the ‘Solution’ class, implementing the ‘cloneGraph’ method to clone a given undirected graph:

Implementing BFS with a hash map efficiently handles graph cloning by ensuring each node is visited and cloned exactly once.

```
from collections import deque

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':
        if not node:
            return node

        # Using a dictionary to keep track of copied nodes
        cloned_nodes = {node: Node(node.val)}

        # Use a queue for BFS
        queue = deque([node])

        while queue:
            current = queue.popleft()

            for neighbor in current.neighbors:
                if neighbor not in cloned_nodes:
                    # Clone the neighbor and add it to the queue
                    cloned_nodes[neighbor] = Node(neighbor.val)
                    queue.append(neighbor)

                    # Add the clone of the neighbor to the neighbors of the clone node
                    cloned_nodes[current].neighbors.append(cloned_nodes[neighbor])

        return cloned_nodes[node]
```

Implementation Details

Basic Setup • Create hash map for node mapping

- Initialize queue/stack for traversal
- Handle base case of null input

Node Processing • Clone current node if not already done

- Add to visited set
- Process neighbors

- Neighbor Handling*
- Create new nodes for unvisited neighbors
 - Update connections between nodes
 - Add to processing queue/stack

Implementation Approaches

BFS Implementation

```
def cloneGraph_bfs(self, node):
    if not node: return None
    cloned = {node: Node(node.val)}
    queue = deque([node])
    while queue:
        curr = queue.popleft()
        for neighbor in curr.neighbors:
            if neighbor not in cloned:
                cloned[neighbor] = Node(neighbor.val)
                queue.append(neighbor)
                cloned[curr].neighbors.append(cloned[neighbor]
                                              ↵ ])
    return cloned[node]
```

DFS Implementation

```
def cloneGraph_dfs(self, node):
    def dfs(node, visited):
        if not node: return None
        if node in visited: return visited[node]
        clone = Node(node.val)
        visited[node] = clone
        clone.neighbors = [dfs(n, visited) for n in node.
                           ↵ neighbors]
        return clone
    return dfs(node, {})
```

Edge Cases

Empty Graph Return null for null input

Single Node Create single node with no neighbors

Self Loop Handle node pointing to itself

Multiple Cycles Track visited nodes to prevent infinite loops

Large Graph Consider memory usage for large inputs

Common Issues

Memory Management

- Problem: Memory leaks in cyclic graphs
- Solution: Proper visited node tracking

Reference Handling

- Problem: Shallow vs. deep copying
- Solution: Ensure complete node recreation

Performance Analysis

Approach	Time	Space
BFS	$O(V + E)$	$O(V)$
DFS	$O(V + E)$	$O(V)$

Table 18.3: Performance Comparison

Similar Problems to This One

Similar problems that involve graph traversal and manipulation include:

- **Number of Islands:** Counting distinct islands in a grid.
- **Pacific Atlantic Water Flow:** Determining cells from which water can flow to both oceans.
- **Graph Valid Tree:** Checking if a given graph forms a valid tree.
- **Course Schedule:** Determining if it's possible to finish all courses based on prerequisites.
- **Alien Dictionary:** Deriving a valid character order from a sorted dictionary of alien words.
- **Word Ladder:** Finding the shortest transformation sequence from one word to another.
- **Word Ladder II:** Finding all shortest transformation sequences from one word to another.
- **Rotting Oranges:** Determining the minimum time required to rot all oranges in a grid.

- **Rotting Oranges Description:** A detailed explanation of the rotting oranges problem.
- **Clone Graph II:** Variations of the clone graph problem with different constraints.

Things to Keep in Mind and Tricks

- **Tracking Cloned Nodes:** Always use a hash map or dictionary to track already cloned nodes to prevent infinite loops and redundant cloning, especially in graphs with cycles.
- **Choosing Traversal Method:** Decide between BFS and DFS based on the graph's characteristics and your comfort level. Both methods are effective, but BFS is often preferred for its iterative nature and easier handling of cycles.
- **Handling Edge Cases:** Ensure that your implementation correctly handles edge cases such as an empty graph, a graph with a single node, or graphs with complex cycles.
- **Immutable vs. Mutable Graphs:** Be cautious about modifying the original graph. If the problem specifies that the original graph should remain unchanged, ensure that your traversal and cloning processes do not alter it.
- **Deep vs. Shallow Copies:** Understand the difference between deep and shallow copies. In this problem, a deep copy is required to ensure that all nodes and their connections are entirely independent of the original graph.
- **Optimizing Space:** While BFS typically requires additional space for the queue, ensure that your hash map does not grow unnecessarily by only storing necessary mappings.

Corner and Special Cases to Test When Writing the Code

- **Empty Graph:** Input node is `None`. The function should return `None` without errors.
- **Single Node:** Graph consists of a single node with no neighbors. The cloned graph should also consist of a single node with no neighbors.
- **Two Nodes with One Edge:** Graph with two nodes connected by a single edge. Ensure that both nodes are cloned correctly and that the connection is preserved.
- **Self-Loop:** A node that has an edge to itself. The cloned node should also have a self-loop.

- **Multiple Cycles:** Graphs with multiple cycles to ensure that the cloning process correctly handles complex cyclic structures without infinite recursion or duplication.
- **Disconnected Graph:** Although the problem specifies a connected graph, test with disconnected graphs to see how the function behaves. It should clone only the connected component that includes the input node.
- **Large Graph:** Graphs with a large number of nodes and edges to test the efficiency and performance of the cloning algorithm.
- **Graph with Varying Degrees:** Nodes with varying numbers of neighbors to ensure that the neighbors are cloned accurately.
- **Immutable Original Graph:** Verify that the original graph remains unchanged after cloning.
- **Non-Integer Node Values:** If the graph nodes contain non-integer values, ensure that the cloning process correctly handles different data types.

Problem 18.2 Pacific Atlantic Water Flow

The **Pacific Atlantic Water Flow** problem is a graph theory challenge that involves identifying matrix coordinates from which water can flow to both the Pacific and Atlantic oceans under specific conditions.

This problem leverages depth-first search (DFS) or breadth-first search (BFS) to determine cells from which water can flow to both the Pacific and Atlantic oceans.

Problem Statement

Imagine a continent as an $m \times n$ matrix of non-negative integers where each cell represents the height of a unit of terrain. The Pacific Ocean borders the continent to the west (left edge) and north (top edge), while the Atlantic Ocean borders it to the east (right edge) and south (bottom edge). Water flows from a cell to any adjacent cell with equal or lower elevation.

The task is to identify all the cells from where water can reach both the Pacific and Atlantic oceans. These cells are at crossroads where water, if allowed to flow naturally according to the rules, would be able to reach both oceans.

Examples

Example 1:

Input:

```
matrix = [
```

```
[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]]
```

Example 2:

```
Input:
matrix = [
    [2, 1],
    [1, 2]
]
Output:
[[0,0],[0,1],[1,0],[1,1]]
```

LeetCode link: Pacific Atlantic Water Flow

[[LeetCode Link](#)]
[GeeksForGeeks Link](#)
[HackerRank Link](#)
[CodeSignal Link](#)
[InterviewBit Link](#)
[Educative Link](#)
[Codewars Link](#)

Algorithmic Approach

Main Concept

To solve this problem, we can perform a search from the oceans inwards using either Depth-First Search (DFS) or Breadth-First Search (BFS), marking the cells that can be reached by water flowing from each ocean. Finally, we identify the intersection of the cells reachable from both oceans.

1. Initialize Reachability Sets:

- Create two sets, `pacific` and `atlantic`, to keep track of cells reachable by water flowing to the Pacific and Atlantic oceans, respectively.

2. Perform DFS/BFS for Both Oceans:

- Iterate through each row and perform DFS/BFS starting from the leftmost (Pacific) and rightmost (Atlantic) cells.
- Iterate through each column and perform DFS/BFS starting from the topmost (Pacific) and bottommost (Atlantic) cells.

3. Identify Common Reachable Cells:

- The cells present in both `pacific` and `atlantic` sets are the desired cells where water can flow to both oceans.

This approach ensures that each cell is visited at most twice (once for each ocean), leading to an efficient solution.

DFS and BFS efficiently explore all possible paths from the ocean boundaries, ensuring all reachable cells are accounted for without redundant processing.

Complexities

- **Time Complexity:** Let m be the number of rows and n be the number of columns in the matrix. The time complexity is $O(m \cdot n)$ for both DFS and BFS approaches, as each cell is processed once for each ocean.
- **Space Complexity:** The space complexity is $O(m \cdot n)$ due to the storage of two separate matrices (or equivalent data structures) to keep track of reachable cells from both oceans.

Python Implementation

Below is the complete Python code for the water flow problem using Depth-First Search (DFS):

```

class Solution(object):
    def pacificAtlantic(self, matrix):
        if not matrix or not matrix[0]: return []

        def dfs(x, y, visited, prevHeight):
            if (x, y) in visited or \
                x < 0 or x >= len(matrix) or y < 0 or y >= len(matrix[0]) or \
                matrix[x][y] < prevHeight:
                return
            visited.add((x, y))
            for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                dfs(x + dx, y + dy, visited, matrix[x][y])

        pacific = set()
        atlantic = set()

        for i in range(len(matrix)):
            dfs(i, 0, pacific, matrix[i][0])
            dfs(i, len(matrix[0]) - 1, atlantic, matrix[i][-1])
        for j in range(len(matrix[0])):
            dfs(0, j, pacific, matrix[0][j])
            dfs(len(matrix) - 1, j, atlantic, matrix[-1][j])

        return list(pacific & atlantic)

# You can call the function with any matrix input to test.
# For example:
# solution = Solution()
# matrix = [
#     [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]
# ]
# print(solution.pacificAtlantic(matrix)) # Output:
#   → [[0,4],[1,3],[1,4],[2,2],[3,0],[3,1],[4,0]]

```

Implementing DFS or BFS requires careful handling of grid boundaries and visited cells to ensure accurate and efficient traversal.

```

class Solution(object):
    def pacificAtlantic(self, matrix):
        if not matrix or not matrix[0]: return []

        def dfs(x, y, visited, prevHeight):
            if (x, y) in visited or \
                x < 0 or x >= len(matrix) or y < 0 or y >= len(matrix[0]) or \
                matrix[x][y] < prevHeight:
                return
            visited.add((x, y))
            for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
                dfs(x + dx, y + dy, visited, matrix[x][y])

```

```

        matrix[x][y] < prevHeight:
            return
        visited.add((x, y))
        for dx, dy in [(1, 0), (-1, 0), (0, 1), (0, -1)]:
            dfs(x + dx, y + dy, visited, matrix[x][y])

    pacific = set()
    atlantic = set()

    for i in range(len(matrix)):
        dfs(i, 0, pacific, matrix[i][0])
        dfs(i, len(matrix[0]) - 1, atlantic, matrix[i][-1])
    for j in range(len(matrix[0])):
        dfs(0, j, pacific, matrix[0][j])
        dfs(len(matrix) - 1, j, atlantic, matrix[-1][j])

    return list(pacific & atlantic)

```

This implementation uses Depth-First Search (DFS) from the edge cells adjacent to each ocean and marks cells that can be reached by water flowing from each ocean. By finding the intersection of cells reachable by both oceans, we obtain the desired cells where water can flow to both the Pacific and Atlantic oceans.

Explanation

The provided Python implementation defines a class `Solution` which contains the method `pacificAtlantic`. Here's a detailed breakdown of the implementation:

- **Edge Case Handling:**
 - If the input `matrix` is empty or contains no columns, the function returns an empty list, as there are no cells to process.
- **Depth-First Search (DFS) Function:**
 - The nested `dfs` function takes the current cell coordinates `x` and `y`, a `visited` set to track reachable cells, and `prevHeight` to ensure water can flow from higher or equal elevation to lower elevation.
 - It checks for the following conditions:
 - * If the current cell (x, y) has already been visited.
 - * If the current cell is out of the matrix boundaries.
 - * If the current cell's height is less than the previous cell's height, preventing water from flowing uphill.
 - If none of the above conditions are met, the cell is marked as `visited`.
 - The function then recursively calls itself for all four adjacent cells (up, down, left, right).

- **Initializing Reachability Sets:**

- Two sets, `pacific` and `atlantic`, are initialized to keep track of cells reachable by water flowing to the Pacific and Atlantic oceans, respectively.

- **Performing DFS from Ocean Boundaries:**

- Iterate through each row:
 - * Perform DFS starting from the first column (Pacific Ocean).
 - * Perform DFS starting from the last column (Atlantic Ocean).
- Iterate through each column:
 - * Perform DFS starting from the first row (Pacific Ocean).
 - * Perform DFS starting from the last row (Atlantic Ocean).

- **Identifying Common Reachable Cells:**

- The intersection of `pacific` and `atlantic` sets (`pacific & atlantic`) contains the cells from which water can flow to both oceans.
- Convert the intersection set to a list and return it as the result.

This approach ensures that each cell is visited at most twice (once for each ocean), leading to an efficient and optimized solution.

Why This Approach

The Depth-First Search (DFS) approach is chosen for its effectiveness in exploring all possible paths from the ocean boundaries inward. By starting DFS from the cells adjacent to each ocean, we can systematically mark all cells that are reachable by water flowing from that ocean. The intersection of these reachable cells from both oceans provides the desired solution. This method efficiently handles the constraints of water flow direction and elevation, ensuring that each relevant cell is accurately accounted for without redundant processing.

Alternative Approaches

An alternative approach involves using Breadth-First Search (BFS) instead of DFS. BFS can be implemented using a queue and may be more intuitive for some, as it explores cells level by level. Both DFS and BFS have similar time and space complexities for this problem, but BFS might offer better performance on graphs with a larger breadth. Additionally, one could optimize space by using a single visited matrix with different markers for each ocean, but maintaining separate sets or matrices tends to be clearer and less error-prone.

Another possibility is to use dynamic programming to compute reachable cells, but this is generally more complex and less straightforward compared to the DFS/BFS approaches.

Similar Problems to This One

Similar problems that involve traversal or pathfinding in a grid or matrix environment include:

- **Number of Islands:** Counting distinct islands in a grid.
- **Walls and Gates:** Filling each empty room with the distance to its nearest gate.
- **Longest Increasing Path in a Matrix:** Finding the longest path in a matrix where each step must go to a strictly higher value.
- **Surrounded Regions:** Capturing all regions surrounded by 'X's by flipping surrounded 'O's to 'X's.
- **Max Area of Island:** Finding the largest island's area in a grid.

Things to Keep in Mind and Tricks

- **Understanding Flow Directions:** Water can only flow from higher or equal elevation to lower elevation. Ensure that the traversal respects this constraint to avoid incorrect reachability.
- **Handling Edge Cells:** Cells on the borders of the matrix are adjacent to the oceans. Starting DFS/BFS from these cells ensures that all reachable cells are appropriately marked.
- **Avoiding Redundant Traversals:** Utilize visited sets or matrices to keep track of already processed cells, preventing unnecessary computations and infinite loops.
- **Efficient Data Structures:** Using sets for visited cells allows for $O(1)$ lookup times, enhancing the overall efficiency of the algorithm.
- **Optimizing Traversal Order:** While the order of traversing neighbors (up, down, left, right) doesn't affect the final result, maintaining a consistent order can aid in debugging and understanding the traversal process.
- **Intersection of Sets:** After marking reachable cells from both oceans, computing the intersection efficiently identifies the cells that satisfy both conditions.

Corner and Special Cases

To ensure robustness and correctness, consider testing the following corner cases:

- **Empty Matrix:** An empty matrix should return an empty list as there are no cells to process.
- **Single Cell:** A matrix with only one cell, which can either be land or water. Verify that the function handles this minimal case correctly.
- **All Cells Can Reach Both Oceans:** A matrix where all cells have high enough elevation to allow water to flow to both oceans.
- **No Cells Can Reach Both Oceans:** A matrix where no cell allows water to flow to both oceans simultaneously.
- **Cells Only Reach One Ocean:** A matrix where some cells can reach only the Pacific or only the Atlantic Ocean.
- **High Elevation Barriers:** Matrices with high elevation barriers that block water flow between certain regions, affecting reachability.
- **Large Matrix:** Very large matrices to test the performance and ensure that the algorithm scales efficiently without excessive memory usage or stack overflow (in the case of recursive DFS).
- **Matrix with Uniform Elevation:** A matrix where all cells have the same elevation, allowing unrestricted water flow.
- **Multiple Water Flow Paths:** Matrices that have multiple distinct paths for water to flow to both oceans.
- **Non-Rectangular Matrices:** Although the problem typically assumes a rectangular matrix, testing with non-uniform row lengths (if allowed) can ensure that the function handles such cases gracefully.

Visual Representation

Performance Optimization Techniques

Memory Usage

- Use bit manipulation for visited states
 - Reuse existing matrix for marking
 - Optimize set operations

Time Efficiency

- Early termination conditions
 - Direction-based pruning
 - Cached intermediate results

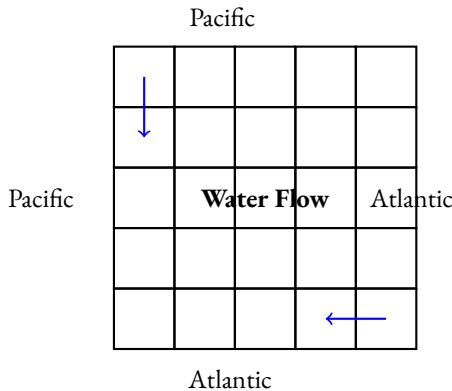


Figure 18.3: Water Flow Directions in the Grid

Problem 18.3 Graph Valid Tree

The **Graph Valid Tree** problem is a well-known question in computer science and competitive programming, focusing on determining whether a given graph constitutes a valid tree. A graph is defined by a set of nodes and edges connecting pairs of nodes. The objective is to verify that the graph is both fully connected and acyclic, which are the two fundamental properties that define a tree.

This problem utilizes the Union-Find (Disjoint Set Union) data structure to efficiently detect cycles and ensure graph connectivity, which are essential properties of a valid tree.

Problem Statement

Given n nodes labeled from 0 to $n - 1$ and a list of undirected edges (each edge is a pair of nodes), write a function to check whether these edges form a valid tree.

Inputs:

- n : An integer representing the total number of nodes in the graph.
- $edges$: A list of pairs of integers where each pair represents an undirected edge between two nodes.

Output:

- Return *true* if the given $edges$ constitute a valid tree, and *false* otherwise.

Examples:

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`

[LeetCode Link]
 [GeeksForGeeks Link]
 [HackerRank Link]
 [CodeSignal Link]
 [InterviewBit Link]
 [Educative Link]
 [Codewars Link]

Algorithmic Approach

Main Concept

To determine whether a graph is a valid tree, we need to verify two key properties:

1. **Acyclicity:** The graph must not contain any cycles.
2. **Connectivity:** The graph must be fully connected, meaning there is exactly one connected component.

The **Union-Find (Disjoint Set Union)** data structure is an efficient way to detect cycles and ensure connectivity in an undirected graph. By iterating through each edge and performing union operations, we can detect if adding an edge creates a cycle and verify if all nodes are connected.

1. Initialize Union-Find Structure:

- Create two arrays: `parent` and `rank`, where each node is initially its own parent, and the rank is initialized to 0.

2. Process Each Edge:

- For each edge (u, v) , perform the following:
 - Find the root parent of node u .
 - Find the root parent of node v .
 - If both nodes have the same root parent, a cycle is detected; return `false`.
 - Otherwise, union the two nodes by attaching the tree with the lower rank to the one with the higher rank.

3. Final Check for Connectivity:

- After processing all edges, ensure that the number of edges is exactly $n - 1$.
 This is a necessary condition for a tree.

This approach ensures that the graph remains acyclic and fully connected, thereby confirming it as a valid tree.

Using Union-Find efficiently detects cycles and ensures all nodes are interconnected, which are essential conditions for a valid tree.

Complexities

- **Time Complexity:** The time complexity of the Union-Find approach is $O(N \cdot \alpha(N))$, where N is the number of nodes and α is the inverse Ackermann function, which grows very slowly and is nearly constant for all practical purposes.
- **Space Complexity:** The space complexity is $O(N)$, required for storing the `parent` and `rank` arrays.

Python Implementation

Below is the complete Python code for checking if the given edges form a valid tree using the Union-Find algorithm:

```
class Solution:
    def validTree(self, n, edges):
        parent = list(range(n))
        rank = [0] * n

        def find(x):
            if parent[x] != x:
                parent[x] = find(parent[x]) # Path compression
            return parent[x]

        def union(x, y):
            xroot = find(x)
            yroot = find(y)
            if xroot == yroot:
                return False # Cycle detected
            # Union by rank
            if rank[xroot] < rank[yroot]:
                parent[xroot] = yroot
            elif rank[xroot] > rank[yroot]:
                parent[yroot] = xroot
            else:
                parent[yroot] = xroot
                rank[xroot] += 1
            return True

        for edge in edges:
            if not union(edge[0], edge[1]):
                return False # Cycle detected

        # Check if the number of edges is exactly n - 1
        return len(edges) == n - 1
```

Implementing the Union-Find data structure allows for efficient cycle detection and connectivity checks essential for validating the tree structure.

```
class Solution:
    def validTree(self, n, edges):
        parent = list(range(n))
        rank = [0] * n

        def find(x):
            if parent[x] != x:
                parent[x] = find(parent[x]) # Path compression
            return parent[x]

        def union(x, y):
            xroot = find(x)
            yroot = find(y)
```

```

        if xroot == yroot:
            return False # Cycle detected
        # Union by rank
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1
    return True

    for edge in edges:
        if not union(edge[0], edge[1]):
            return False # Cycle detected

    # Check if the number of edges is exactly n - 1
    return len(edges) == n - 1

```

This implementation uses the Union-Find algorithm to detect cycles and ensure that the graph is fully connected. Each node is initially its own parent, and as edges are processed, nodes are united into sets. If a cycle is detected (i.e., two nodes are already in the same set), the function returns *false*. Finally, it checks whether the number of edges is exactly $n - 1$, which is a necessary condition for a valid tree.

Explanation

The provided Python implementation defines a class `Solution` which contains the method `validTree`. Here's a detailed breakdown of the implementation:

- **Initialization:**
 - `parent`: An array where `parent[i]` represents the parent of node i . Initially, each node is its own parent.
 - `rank`: An array to keep track of the depth of trees for optimizing the Union-Find operations.
- **Find Function (`find(x)`):**
 - This function finds the root parent of node x .
 - Implements path compression by making each node on the path point directly to the root, thereby flattening the structure and optimizing future queries.
- **Union Function (`union(x, y)`):**
 - This function attempts to unite the sets containing nodes x and y .

- It first finds the root parents of both nodes.
- If both nodes have the same root parent, a cycle is detected, and the function returns *False*.
- Otherwise, it unites the two sets by attaching the tree with the lower rank to the one with the higher rank to keep the tree shallow.

- **Processing Edges:**

- Iterate through each edge in the `edges` list.
- For each edge, attempt to unite the two connected nodes.
- If the `union` function returns *False*, a cycle has been detected, and the function returns *False*.

- **Final Check:**

- After processing all edges, check if the number of edges is exactly $n - 1$. This is a necessary condition for the graph to be a tree.
- If this condition is met, return *True*; otherwise, return *False*.

This approach ensures that the graph is both acyclic and fully connected, thereby confirming it as a valid tree.

Why This Approach

The Union-Find algorithm is chosen for its efficiency in handling dynamic connectivity problems. It effectively detects cycles by determining if two nodes share the same root parent before performing a union operation. Additionally, by using path compression and union by rank, the algorithm optimizes the time complexity, making it highly suitable for large graphs. This method simplifies the process of verifying both acyclicity and connectivity in a single pass through the edges, providing a clear and concise solution to the problem.

Alternative Approaches

An alternative approach to solving the "Graph Valid Tree" problem is using Depth-First Search (DFS) or Breadth-First Search (BFS) to traverse the graph:

1. **DFS/BFS Traversal:**

- Start a DFS or BFS from an arbitrary node.
- Track visited nodes to ensure that each node is visited exactly once.
- After traversal, check if all nodes have been visited and that the number of edges is exactly $n - 1$.

2. Cycle Detection:

- During traversal, if a back-edge is detected (i.e., encountering an already visited node that is not the immediate parent), a cycle exists, and the graph cannot be a tree.

While DFS/BFS can also effectively determine if a graph is a valid tree, the Union-Find approach is often preferred for its simplicity and efficiency in handling both cycle detection and connectivity checks simultaneously.

Similar Problems to This One

Similar problems that involve graph traversal and validation include:

- **Number of Islands:** Counting distinct islands in a grid.
- **Graph Valid Tree II:** Variations of the graph valid tree problem with additional constraints.
- **Cycle Detection in Graph:** Determining whether a graph contains any cycles.
- **Connected Components in Graph:** Identifying all connected components within a graph.
- **Minimum Spanning Tree:** Finding the subset of edges that connects all vertices with the minimal total edge weight.

Things to Keep in Mind and Tricks

- **Edge Count Check:** For a graph to be a valid tree, it must have exactly $n - 1$ edges. This is a quick way to rule out invalid trees before performing more complex checks.
- **Union-Find Optimization:** Implement path compression and union by rank to optimize the performance of the Union-Find operations, especially for large graphs.
- **Handling Disconnected Graphs:** Ensure that after processing all edges, there is only one connected component. This guarantees that the graph is fully connected.
- **Cycle Detection:** Detecting a cycle early can save computation time by immediately returning *false* without needing to process the remaining edges.
- **Data Structures:** Choose appropriate data structures (e.g., lists for parent and rank arrays) that allow for efficient access and modification during the algorithm's execution.

- **Initialization:** Properly initialize the Union-Find structures to ensure that each node is its own parent at the start.

Corner and Special Cases

- **Empty Graph:** Input where $n = 0$ and $edges = []$. The function should handle this gracefully, typically by returning *false* as there are no nodes to form a tree.
- **Single Node:** Graph with $n = 1$ and $edges = []$. This should return *true* as a single node without edges is considered a valid tree.
- **Two Nodes with One Edge:** Graph with $n = 2$ and $edges = [[0, 1]]$. This should return *true*.
- **Two Nodes with Two Edges:** Graph with $n = 2$ and $edges = [[0, 1], [1, 0]]$. This should return *false* due to a cycle.
- **Multiple Components:** Graph where $n > 1$ but $edges$ do not connect all nodes, resulting in disconnected components. This should return *false*.
- **Cycle in Graph:** Graph with $n \geq 3$ and $edges$ forming a cycle. This should return *false*.
- **Extra Edges:** Graph where $\text{len}(edges) > n - 1$, which implies the presence of cycles. This should return *false*.
- **Large Graph:** Graph with a large number of nodes and edges to test the algorithm's performance and ensure it handles large inputs efficiently.
- **Self-Loops:** Graph containing edges where a node is connected to itself (e.g., $[0, 0]$). This should return *false* as self-loops introduce cycles.
- **Invalid Edge Indices:** Graph where edges contain node indices outside the range 0 to $n - 1$. The implementation should handle such cases appropriately, either by ignoring invalid edges or by returning *false*.

Problem 18.4 Course Schedule

The **Course Schedule** problem focuses on determining the possibility of completing a series of courses given their prerequisite requirements. This problem is analogous to understanding dependencies within systems and is commonly found as an algorithmic challenge that reinforces the understanding of graph theory concepts such as cycle detection and topological sorting.

This problem utilizes graph theory concepts such as cycle detection and topological sorting to determine course completion feasibility.

Problem Statement

Given `numCourses` courses labeled from 0 to `numCourses` – 1, and a list of prerequisite pairs (where each pair $[a, b]$ implies that you must take course b before course a), the goal is to ascertain whether it is achievable to complete all the courses.

The crux of the problem involves formulating an algorithm to determine if an order exists to take these courses such that for each course, all its prerequisites are met.

Inputs:

- `numCourses`: An integer representing the total number of courses.
- `prerequisites`: A list of pairs of integers where each pair $[a, b]$ indicates that course b is a prerequisite for course a .

Output:

- Return `True` if it is possible to finish all courses, and `False` otherwise.

Examples:

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

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

LeetCode link: Course Schedule

[\[LeetCode Link\]](#)

[\[GeeksForGeeks Link\]](#)

[\[HackerRank Link\]](#)

[\[CodeSignal Link\]](#)

[\[InterviewBit Link\]](#)

[\[Educative Link\]](#)

[\[Codewars Link\]](#)

Algorithmic Approach

To resolve this challenge, we can model the courses and prerequisites as a directed graph, with each course as a node and edges representing the dependency direction (from prerequisite course to the dependent course). The solution entails checking for cycles within this directed graph since the presence of a cycle indicates that there is no feasible way to complete all courses. This is because a cycle would mean a

course's prerequisites could never be fully satisfied. Detecting a cycle can be performed via Depth-First Search (DFS) or by attempting to perform a Topological Sort on the graph. If we successfully perform a topological sort with all nodes visited, it implies there is no cycle, and completing all courses is feasible.

Graph-based approaches like DFS and BFS are effective for handling dependencies and detecting cycles in prerequisite structures.

Complexities

- **Time Complexity:** In the scenario where we use Depth-First Search (DFS), the time complexity is $O(V + E)$ where V is the number of courses and E is the number of dependencies (prerequisites). This stems from the fact that every node and edge is visited in the worst case.
- **Space Complexity:** The space complexity is also $O(V + E)$ to store the graph data structure, as well as the recursion stack for the DFS procedure.

Python Implementation

Here is the complete Python solution utilizing DFS to check for the presence of cycles in the graph built from the course prerequisites:

Implementing DFS requires careful management of recursion to handle dependencies and detect cycles efficiently.

```
class Solution(object):
    def canFinish(self, numCourses, prerequisites):
        def dfs(course):
            if visited[course] == -1:
                return False
            if visited[course] == 1:
                return True
            visited[course] = -1
            for prereq in graph[course]:
                if not dfs(prereq):
                    return False
            visited[course] = 1
            return True

        graph = {i: [] for i in range(numCourses)}
        for course, prereq in prerequisites:
            graph[course].append(prereq)

        visited = [0] * numCourses
        for course in range(numCourses):
            if not dfs(course):
                return False
        return True
```

```
class Solution(object):
    def canFinish(self, numCourses, prerequisites):
        def dfs(course):
            if visited[course] == -1:
                return False
            if visited[course] == 1:
                return True
            visited[course] = -1
            for prereq in graph[course]:
                if not dfs(prereq):
                    return False
            visited[course] = 1
            return True

        graph = {i: [] for i in range(numCourses)}
        for course, prereq in prerequisites:
            graph[course].append(prereq)

        visited = [0] * numCourses
        for course in range(numCourses):
            if not dfs(course):
```

```

    return False
return True

```

This implementation begins by constructing a graph that represents the courses and their dependencies. It then utilizes a DFS approach to traverse this graph. For each course, it delves deeper into its prerequisites, using a `visited` list to track the state of each node (unvisited, visiting, visited). If a cycle is detected, indicated by finding a course in the 'visiting' state during the DFS, it returns `False`. If all courses can be traversed without encountering a cycle, the function returns `True`, indicating it is possible to finish all courses.

Explanation

The provided Python implementation defines a class `Solution` which contains the method `canFinish`. Here's a detailed breakdown of the implementation:

- **Edge Case Handling:**

- If the input `prerequisites` list is empty, it implies there are no dependencies, and all courses can be completed. The function returns `True`.

- **Graph Construction:**

- A dictionary named `graph` is initialized to represent the adjacency list of the graph. Each course is a key, and its value is a list of courses that are prerequisites for it.
- Iterate through each pair in `prerequisites` and populate the `graph` accordingly.

- **Visited List Initialization:**

- A list named `visited` is initialized with all elements set to 0. The values in this list represent the state of each course:
 - * 0: Unvisited
 - * -1: Visiting (currently in the recursion stack)
 - * 1: Visited (all prerequisites processed)

- **Depth-First Search (DFS) Function:**

- The nested `dfs` function takes a course as its parameter.
- It first checks if the course is currently being visited (`visited[course] == -1`). If so, a cycle is detected, and the function returns `False`.
- If the course has already been visited (`visited[course] == 1`), it returns `True` as this path has been processed.
- The course is marked as visiting (`visited[course] = -1`).

- The function then recursively calls itself for all prerequisites of the current course. If any recursive call returns `False`, it propagates the `False` value up the recursion stack.
- After all prerequisites are processed without detecting a cycle, the course is marked as visited (`visited[course] = 1`), and the function returns `True`.

- **Cycle Detection and Connectivity Check:**

- Iterate through each course. For each course, if it hasn't been visited, perform a DFS starting from that course.
- If any DFS call detects a cycle, return `False`.
- If all courses are processed without detecting a cycle, return `True`.

This approach ensures that all courses are checked for cyclic dependencies. If no cycles are found and all courses are connected appropriately, it confirms that it's possible to complete all courses.

Why This Approach

This graph-based approach offers an intuitive method for representing and processing course prerequisites. DFS is a classic algorithm for detecting cycles in directed graphs, making it an apt choice for this scenario. By using a visited list to track the state of each course, the implementation efficiently identifies cycles, ensuring that only feasible course schedules are considered valid. Additionally, this method provides a clear pathway for extending the solution to more complex variants, such as retrieving the actual order of courses (as in "Course Schedule II").

Alternative Approaches

Aside from the DFS-based cycle detection, another viable approach is to use **Breadth-First Search (BFS)** in conjunction with **Topological Sorting**. Specifically, Kahn's algorithm for topological sorting can be employed to determine if a valid course order exists. Here's how it works:

- **Compute In-Degrees:** Calculate the number of prerequisites (in-degrees) for each course.
- **Initialize Queue:** Enqueue all courses with an in-degree of 0 (i.e., courses with no prerequisites).
- **Process Courses:**
 - Dequeue a course from the queue and add it to the topological order.

- For each neighbor (dependent course), decrement its in-degree by 1.
- If a neighbor's in-degree becomes 0, enqueue it.
- **Check Completion:** If all courses are processed (i.e., the topological order contains all courses), return **True**. Otherwise, return **False**.

This method is particularly efficient for large graphs and provides a clear ordering of courses if one exists.

Similar Problems

Similar problems that students might encounter include:

- **Course Schedule II:** Extends the Course Schedule problem by asking for the actual order of courses to be taken.
- **Redundant Connection:** Detecting cycles in an undirected graph.
- **Minimum Height Trees:** Finding the roots of minimum height trees in a graph.
- **Alien Dictionary:** Deriving a valid character order from a sorted dictionary of alien words.
- **Longest Increasing Path in a Matrix:** Finding the longest path in a matrix where each step must go to a strictly higher value.

Things to Keep in Mind and Tricks

- **Cycle Detection:** When dealing with dependencies, always check for cycles to ensure that the dependencies do not form an impossible loop.
- **Graph Representation:** Efficiently represent the graph using adjacency lists or adjacency matrices based on the problem constraints. Adjacency lists are generally more space-efficient for sparse graphs.
- **Topological Sorting:** Understanding topological sorting is crucial for problems involving dependencies, as it provides an order of processing that respects all prerequisites.
- **Union-Find:** The Union-Find (Disjoint Set Union) data structure can be an effective tool for detecting cycles in undirected graphs.
- **Handling Edge Cases:** Always consider edge cases such as no prerequisites, all courses independent, or courses forming multiple disconnected components.

- **Optimizing Space:** If possible, modify the input data structure to save space, such as marking visited nodes directly in the graph.
- **Choosing the Right Traversal:** Decide between DFS and BFS based on the problem's requirements and the nature of the graph.

Corner and Special Cases

- **No Courses:** $n = 0$. The function should return `True` as there are no courses to complete.
- **Single Course:** $n = 1$ with no prerequisites. The function should return `True`.
- **All Courses Independent:** No prerequisites. The function should return `True`.
- **Chain of Prerequisites:** Courses form a linear dependency (e.g., $0 \rightarrow 1 \rightarrow 2 \rightarrow \dots \rightarrow n-1$). The function should return `True`.
- **Cycle in Prerequisites:** A cycle exists (e.g., $0 \rightarrow 1 \rightarrow 2 \rightarrow 0$). The function should return `False`.
- **Multiple Independent Cycles:** Multiple cycles exist in different components of the graph. The function should return `False` unless each component independently forms a valid tree with no cycles.
- **Disconnected Graph:** The graph has multiple disconnected components. The function should return `False` unless each component independently forms a valid tree with no cycles.
- **Large Number of Courses:** A large value of n with sparse or dense prerequisites to test the algorithm's performance and efficiency.
- **Self-Prerequisite:** A course that lists itself as a prerequisite (e.g., $[0, 0]$). The function should return `False`.
- **Redundant Prerequisites:** Multiple identical prerequisite pairs. The function should handle duplicates gracefully without affecting the outcome.

Problem 18.5 Alien Dictionary

The "Alien Dictionary" problem is an intriguing interview question that tests one's ability to infer the ordering of an unknown language's alphabet using its lexicon sorted according to alien linguistic rules. It is implied that the alien language utilizes the same alphabet as English but with a reorganized sequence.

Problem Statement

In the problem, you are given a list of sorted words from an alien language, with each word comprising a sequence of lowercase letters. Your task is to derive and return the ordering of the letters in the alien language as a string. If no valid order exists or if the input list is empty, the expected output is an empty string.

Example

Input: words = ["wrt", "wrf", "er", "ett", "rftt"]

Output: "wertf"

Explanation: Given the list of words from the alien language, we deduce the following order: - "t" precedes "f" (wrt -> wrf) - "w" precedes "e" (wrt -> er) - "r" precedes "t" (wrt -> ett) - "e" precedes "r" (er -> ett)

Thereby, the derived order is "wertf".

Algorithmic Approach

The solution to this problem is often tackled using graph theory concepts like directed acyclic graphs (DAGs). We can construct a graph where each vertex represents a character in the alien language, and directed edges define the lexicographic order between characters. Then we perform a topological sort on this graph to find the order of the characters. If we detect a cycle during this process, it would indicate that there is a conflict in the order, and no valid alphabet order can be determined, resulting in an empty string instead.

LeetCode link: Alien Dictionary

[ewpage](#)

Python Implementation

Below is the complete Python code that implements a solution to the "Alien Dictionary" problem:

```
from collections import defaultdict, deque

def alienOrder(words):
    # Create a graph (adjacency list)
    adj_list = defaultdict(set)
    # Count of how many times a character appears as 'second' in the alien
    # dictionary order
```

```

in_degree = {char: 0 for word in words for char in word}

# Populate the graph and in_degree
for first_word, second_word in zip(words, words[1:]):
    for char1, char2 in zip(first_word, second_word):
        if char1 != char2:
            if char2 not in adj_list[char1]:
                adj_list[char1].add(char2)
                in_degree[char2] += 1
            break
        else:
            # Check if second word is a prefix of the first word
            if len(second_word) < len(first_word): return ""

# Initialize a queue with letters that have in_degree of 0
queue = deque([char for char in in_degree if in_degree[char] == 0])
alien_alphabet = []

while queue:
    char = queue.popleft()
    alien_alphabet.append(char)
    # Reduce the in_degree for neighbors and add to queue if it becomes 0
    for neighbor in adj_list[char]:
        in_degree[neighbor] -= 1
        if in_degree[neighbor] == 0:
            queue.append(neighbor)

    if len(alien_alphabet) == len(in_degree):
        return "".join(alien_alphabet)
    else:
        return "" # Cycle or not all letters are connected through edges

# Example usage:
words = ["wrt", "wrf", "er", "ett", "rftt"]
print(alienOrder(words)) # Output should be 'wertf'

```

The provided implementation begins by establishing a graph in the form of an adjacency list.

Why this approach

This approach is very fitting for the "Alien Dictionary" problem because it essentially translates to finding a possible topological ordering of characters given partial order constraints. Utilizing graph traversal and topological sort principles allows us to systematically derive the full order, if one exists, or recognize the impossibility of such an order when a cycle is present in the graph.

Alternative approaches

One could approach the problem with different variations of graph traversal strategies or by examining the pairwise comparisons of consecutive words in a different manner, but any valid solution will likely utilize some form of topological sorting given the nature of the problem.

Similar problems to this one

Problems that also involve deducing orders or hierarchies based on constraints, such as "Course Schedule" where prerequisites dictate a valid order for taking courses, or tasks scheduling problems where dependencies determine task execution order, share a resemblance to the "Alien Dictionary" problem.

Things to keep in mind and tricks

Remember that graph-based algorithms and data structures such as in-degree representation and adjacency lists play a crucial role in implementing the solution. Recognizing a topological sort as the primary technique helps guide the overall strategy.

Corner and special cases to test when writing the code

Edge cases may include scenarios where a word is a prefix of another, or the input list has only one word or characters that don't follow any others. It is essential to account for these to avoid incorrect inferences about the alien language's alphabet order. ¹¹

Problem 18.6 Word Ladder

The **Word Ladder** problem is a classic algorithmic challenge that involves transforming a *beginWord* into an *endWord* by changing only one letter at a time. Each intermediate word in the transformation sequence must exist within a given *wordList*. The objective is to determine the length of the shortest possible transformation sequence that adheres to these constraints.

This problem leverages Breadth-First Search (BFS) and graph theory concepts to efficiently find the shortest transformation sequence between two words.

Problem Statement

Given two words, *beginWord* and *endWord*, and a dictionary *wordList*, find the length of the shortest transformation sequence from *beginWord* to *endWord*, such that:

- Only one letter can be changed at a time.
- Each transformed word must exist in the `wordList`.

Return the number of words in the shortest transformation sequence, or 0 if no such sequence exists.

Note:

- `beginWord` does not need to be in `wordList`.
- All words have the same length.
- All words contain only lowercase English letters.

Examples:

Example 1:

Input:

```
beginWord = "hit",
endWord = "cog",
wordList = ["hot", "dot", "dog", "lot", "log", "cog"]
```

Output: 5

Explanation: One shortest transformation is "hit" → "hot" → "dot" → "dog" → "cog", which has 5 words.

Example 2:

Input:

```
beginWord = "hit",
endWord = "cog",
wordList = ["hot", "dot", "dog", "lot", "log"]
```

Output: 0

Explanation: The `endWord` "cog" is not in `wordList`, so no possible transformation.

LeetCode link: Word Ladder

[LeetCode Link]

[GeeksForGeeks Link]

[HackerRank Link]

[CodeSignal Link]

[InterviewBit Link]

[Educative Link]

[Codewars Link]

Algorithmic Approach

The **Word Ladder** problem can be effectively solved using Breadth-First Search (BFS) by modeling each word as a node in a graph. An edge exists between two

nodes if the corresponding words differ by exactly one letter. The goal is to find the shortest path from the `beginWord` to the `endWord` within this graph.

To optimize the BFS process, we preprocess the `wordList` to create a mapping between generic intermediate states and the list of words that can be transformed into that state. For example, for the word "dog", the generic states would be "*og", "d*g", "do*". This preprocessing allows for efficient adjacency lookups during BFS.

1. Preprocessing:

- For each word in the `wordList`, generate all possible generic intermediate states by replacing each letter with a wildcard character "*".
- Create a dictionary (`all_combo_dict`) where each key is a generic intermediate state, and the value is a list of words matching that state.

2. Breadth-First Search (BFS):

- Initialize a queue with a tuple containing the `beginWord` and the initial level (1).
- Use a `visited` dictionary to keep track of visited words to prevent revisiting and cycles.
- While the queue is not empty:
 - Dequeue the first element to get the current word and its level.
 - For each character position in the current word, generate the corresponding generic intermediate state.
 - For each word in the list corresponding to the generic state:
 - * If the word is the `endWord`, return the current level plus one.
 - * If the word has not been visited, mark it as visited and enqueue it with an incremented level.
 - Clear the list of words for the current generic state to prevent redundant processing.

3. Termination:

- If the `endWord` is never reached during BFS, return 0 as no valid transformation sequence exists.

Complexities

- **Time Complexity:** $O(N \cdot K^2)$, where N is the number of words in `wordList` and K is the length of each word. This accounts for generating all generic intermediate states and performing BFS traversal.
- **Space Complexity:** $O(N \cdot K)$, due to storing the intermediate states in `all_combo_dict` and maintaining the `visited` dictionary.

Efficient adjacency lookups using generic intermediate states significantly reduce the number of operations during BFS, enhancing performance.

Python Implementation

Below is the complete Python code that implements a solution to the **Word Ladder** problem:

```
from collections import defaultdict, deque

def ladderLength(beginWord, endWord, wordList):
    if endWord not in wordList:
        return 0

    L = len(beginWord)
    all_combo_dict = defaultdict(list)
    for word in wordList:
        for i in range(L):
            intermediate_word = word[:i] + "*" + word[i+1:]
            all_combo_dict[intermediate_word].append(word)

    queue = deque([(beginWord, 1)])
    visited = {beginWord: True}

    while queue:
        current_word, level = queue.popleft()
        for i in range(L):
            intermediate_word = current_word[:i] + "*" + current_word[i+1:]
            for word in all_combo_dict[intermediate_word]:
                if word == endWord:
                    return level + 1
                if word not in visited:
                    visited[word] = True
                    queue.append((word, level + 1))
            all_combo_dict[intermediate_word] = [] # Clear to prevent re-
                # processing
    return 0

# Example usage:
print(ladderLength("hit", "cog", ["hot", "dot", "dog", "lot", "log", "cog"])) # Output
→ : 5
print(ladderLength("hit", "cog", ["hot", "dot", "dog", "lot", "log"])) # Output: 0
```

Implementing BFS with preprocessed intermediate states ensures that each transformation step is efficiently explored without unnecessary computations.

```
from collections import defaultdict, deque

def ladderLength(beginWord, endWord, wordList):
    if endWord not in wordList:
        return 0

    L = len(beginWord)
    all_combo_dict = defaultdict(list)
    for word in wordList:
```

```

for i in range(L):
    intermediate_word = word[:i] + "*" + word[i+1:]
    all_combo_dict[intermediate_word].append(word)

queue = deque([(beginWord, 1)])
visited = {beginWord: True}

while queue:
    current_word, level = queue.popleft()
    for i in range(L):
        intermediate_word = current_word[:i] + "*" + current_word[i+1:]
        for word in all_combo_dict[intermediate_word]:
            if word == endWord:
                return level + 1
            if word not in visited:
                visited[word] = True
                queue.append((word, level + 1))
    all_combo_dict[intermediate_word] = [] # Clear to prevent re-
        ↪ processing
return 0

```

This implementation begins by checking if the `endWord` exists in the `wordList`. If not, it immediately returns 0, as no valid transformation is possible. It then preprocesses the `wordList` to create a mapping of generic intermediate states to the corresponding words. Using BFS, the algorithm explores each word level by level, ensuring that the shortest transformation sequence is found. The `visited` dictionary keeps track of already processed words to prevent cycles and redundant processing.

Explanation

The provided Python implementation defines a function `ladderLength` which takes `beginWord`, `endWord`, and `wordList` as inputs and returns the length of the shortest transformation sequence from `beginWord` to `endWord`.

- **Edge Case Handling:**

- If the `endWord` is not present in the `wordList`, the function returns 0, as no transformation can lead to the desired word.

- **Preprocessing:**

- Determine the length L of the `beginWord`.
- Iterate through each word in the `wordList` and generate all possible generic intermediate states by replacing each character with “*”.
- Populate the `all_combo_dict` with these intermediate states mapping to the corresponding words.

- **BFS Initialization:**

- Initialize a queue with a tuple containing the `beginWord` and the initial transformation level 1.
- Initialize a `visited` dictionary to keep track of words that have already been processed.

- **BFS Traversal:**

- While the queue is not empty:
 - * Dequeue the first element to get the current word and its associated level.
 - * For each character position in the current word, generate the corresponding generic intermediate state.
 - * For each word associated with this intermediate state:
 - If the word matches the `endWord`, return the current level incremented by one, as the transformation is complete.
 - If the word has not been visited, mark it as visited and enqueue it with an incremented level.
 - * Clear the list of words for the current intermediate state in `all_combo_dict` to prevent re-processing in future iterations.

- **Termination:**

- If the BFS completes without finding the `endWord`, return 0, indicating that no valid transformation sequence exists.

This approach ensures that the shortest transformation sequence is found by exploring all possible one-letter transformations in a level-by-level manner, characteristic of BFS. The preprocessing step significantly optimizes the adjacency lookup, allowing the algorithm to efficiently navigate through potential transformation paths.

Why this approach

Breadth-First Search (BFS) is particularly well-suited for finding the shortest path in unweighted graphs, which aligns perfectly with the requirements of the **Word Ladder** problem. By treating each word as a node and establishing edges between words that differ by a single letter, BFS systematically explores all possible transformation sequences in order of increasing length. The preprocessing step of creating generic intermediate states facilitates rapid adjacency lookups, thereby enhancing the efficiency of the BFS traversal. This method guarantees the discovery of the shortest possible transformation sequence, if one exists.

Alternative Approaches

An alternative method to solving the **Word Ladder** problem is to employ **Bidirectional BFS**, which simultaneously initiates BFS from both the `beginWord` and the `endWord`. This technique can significantly reduce the search space and improve performance, especially in cases where the transformation sequence is long. Here's a brief overview of how Bidirectional BFS works:

1. Initialization:

- Initialize two queues, one starting from the `beginWord` and the other from the `endWord`.
- Maintain two separate `visited` dictionaries for both search fronts.

2. Traversal:

- Alternate between expanding the search frontiers from both ends.
- At each step, expand the smaller of the two queues to optimize performance.
- If a common word is found in both `visited` dictionaries, the shortest transformation sequence has been identified.

Bidirectional BFS can lead to faster convergence by effectively halving the search depth, which is particularly beneficial for large word lists.

Similar Problems to This One

Similar problems that involve finding the shortest transformation or path within a constrained space include:

- **Word Ladder II:** Extends the **Word Ladder** problem by requiring the enumeration of all shortest transformation sequences.
- **Sliding Puzzle Problems:** Such as the 8-puzzle, where the goal is to reach a target configuration through a series of valid moves.
- **Maze Solving Problems:** Finding the shortest path from a start point to an end point within a maze.
- **Minimum Genetic Mutation:** Determining the minimum number of mutations needed to mutate from a start gene string to an end gene string, with each mutation being valid.
- **Shortest Path in a Grid:** Finding the shortest path from the top-left corner to the bottom-right corner in a grid with obstacles.

These problems share the common theme of navigating through a space of possibilities to find an optimal or feasible path, often leveraging similar algorithmic strategies like BFS or DFS.

Things to Keep in Mind and Tricks

- **Preprocessing Intermediate States:** Efficiently generating and utilizing generic intermediate states can drastically reduce the number of operations during BFS.
- **Avoiding Reprocessing:** Mark words as visited immediately after enqueueing them to prevent multiple enqueues of the same word, which can lead to redundant computations.
- **Early Termination:** If the `endWord` is found during BFS, terminate immediately to ensure the shortest path is returned.
- **Bidirectional BFS:** Consider using Bidirectional BFS for large datasets to optimize search performance.
- **Handling Edge Cases:** Ensure that edge cases, such as an empty `wordList` or when the `beginWord` equals the `endWord`, are handled appropriately.
- **Optimizing Space:** Clearing the list of words for each intermediate state after processing helps in reducing memory usage and prevents unnecessary future processing.
- **Consistent Word Length:** All words must be of the same length. Validate this if the problem constraints are not explicitly guaranteed.

Corner and Special Cases to Test When Writing the Code

To ensure the robustness and correctness of the solution, consider testing the following corner cases:

- **Empty wordList:** No possible transformations should return 0.
- **Begin Word Equals End Word:** If the `beginWord` is the same as the `endWord`, the transformation sequence length is 1.
- **No Possible Transformation:** When no sequence can lead from `beginWord` to `endWord`, even if `endWord` is in the `wordList`.
- **Minimum Transformation:** Transformation is possible in one step.
- **Multiple Transformation Paths:** Ensure that the shortest path is returned even when multiple paths exist.

- **Large wordList:** Test the algorithm's performance and efficiency with a large number of words.
- **Words with No Common Letters:** Verify that the algorithm correctly identifies when no transformations are possible.
- **Prefix Words:** Words where one word is a prefix of another, ensuring that the algorithm handles such scenarios without errors.
- **Non-Alphabet Characters:** If allowed, ensure that words containing non-alphabet characters are handled correctly.
- **Case Sensitivity:** All words should be lowercase as per the problem statement; however, verify how the algorithm handles mixed case inputs.

Problem 18.7 Word Ladder II

The **Word Ladder II** problem builds upon the original Word Ladder challenge by not only finding the length of the shortest transformation sequence from a *beginWord* to an *endWord*, but also by enumerating all possible shortest transformation sequences that satisfy the transformation rules. This problem emphasizes advanced graph traversal methods and efficient backtracking strategies to handle multiple valid paths.

This problem extends the Word Ladder challenge by requiring the enumeration of all shortest transformation sequences using BFS and backtracking techniques.

Problem Statement

Given two words, *beginWord* and *endWord*, and a dictionary *wordList*, find all the shortest transformation sequences from *beginWord* to *endWord*, such that:

- Only one letter can be changed at a time.
- Each transformed word must exist in the *wordList*.

Note:

- *beginWord* does not need to be in *wordList*.
- All words have the same length.
- All words contain only lowercase English letters.

Output: A list of lists, where each inner list represents a valid shortest transformation sequence from *beginWord* to *endWord*. If no such sequence exists, return an empty list.

Examples:

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"]
]
```

Example 2:

Input:

```
beginWord = "hit",
endWord = "cog",
wordList = ["hot", "dot", "dog", "lot", "log"]
```

Output: []

Explanation: The endWord "cog" is not in wordList, so no possible transformation.

LeetCode link: Word Ladder II

[LeetCode Link]
[GeeksForGeeks Link]
[HackerRank Link]
[CodeSignal Link]
[InterviewBit Link]
[Educative Link]
[Codewars Link]

Algorithmic Approach

The **Word Ladder II** problem can be effectively solved by combining Breadth-First Search (BFS) to determine the shortest path lengths and Depth-First Search (DFS) for backtracking to construct all possible shortest transformation sequences. The primary challenge lies in efficiently handling multiple paths and ensuring that only the shortest sequences are captured.

1. BFS to Determine Levels:

- Treat each word as a node in a graph, with edges connecting words that differ by exactly one letter.
- Perform BFS starting from the `beginWord` to explore the graph level by level.
- During BFS, keep track of each word's level (distance from the `beginWord`).
- Use a dictionary (`level`) to store the earliest level at which each word is encountered.

2. Building the Graph:

- Use a dictionary (`parents`) to map each word to its predecessors in the BFS traversal.

- This mapping is essential for backtracking all possible shortest paths from `endWord` to `beginWord`.

3. DFS for Backtracking:

- Starting from the `endWord`, perform DFS to traverse back to the `beginWord` using the `parents` mapping.
- Accumulate the paths during DFS to collect all valid shortest transformation sequences.

4. Termination:

- BFS terminates once the `endWord` is reached, ensuring that only the shortest paths are considered.
- If the `endWord` is not reachable, return an empty list.

Complexities

Combining BFS for level determination with DFS for path construction efficiently captures all shortest transformation sequences without redundant computations.

- **Time Complexity:** $O(N \cdot K^2 + M)$, where:
 - N is the number of words in `wordList`.
 - K is the length of each word.
 - M is the number of shortest transformation sequences.

The $N \cdot K^2$ term accounts for generating all generic intermediate states and building the graph, while M represents the time taken to backtrack and construct all possible sequences.

- **Space Complexity:** $O(N \cdot K)$, due to storing the generic intermediate states in `all_combo_dict`, the BFS queue, and the `parents` mapping.

Python Implementation

Below is the complete Python code that implements a solution to the **Word Ladder II** problem:

```
from collections import defaultdict, deque

def findLadders(beginWord, endWord, wordList):
    wordSet = set(wordList)
    if endWord not in wordSet:
        return []

    # Initialize variables
    level = {beginWord: 0}
    parents = defaultdict(set)
```

Efficiently managing BFS levels and parent mappings is crucial for accurately backtracking all shortest paths without excessive memory usage.

```

queue = deque([beginWord])
word_len = len(beginWord)
found = False
current_level = 0

while queue and not found:
    current_level += 1
    for _ in range(len(queue)):
        word = queue.popleft()
        for i in range(word_len):
            for c in 'abcdefghijklmnopqrstuvwxyz':
                if c == word[i]:
                    continue
                next_word = word[:i] + c + word[i+1:]
                if next_word in wordSet:
                    if next_word not in level:
                        level[next_word] = current_level
                        queue.append(next_word)
                    if level[next_word] == current_level:
                        parents[next_word].add(word)
                    if next_word == endWord:
                        found = True
    # Optional: Remove words that have been visited to prevent revisiting
    # wordSet -= set(parents.keys())

if not found:
    return []

# Backtracking to build paths
res = []
path = [endWord]

def backtrack(word):
    if word == beginWord:
        res.append(path[::-1])
        return
    for parent in parents[word]:
        path.append(parent)
        backtrack(parent)
        path.pop()

backtrack(endWord)
return res

# Example usage:
print(findLadders("hit", "cog", ["hot", "dot", "dog", "lot", "log", "cog"]))
# Output: [[["hit", "hot", "dot", "dog", "cog"]], [{"hit": "hot", "hot": "dot", "dot": "dog", "dog": "log", "log": "cog"}]]

print(findLadders("hit", "cog", ["hot", "dot", "dog", "lot", "log"]))
# Output: []

```

```

from collections import defaultdict, deque

def findLadders(beginWord, endWord, wordList):
    wordSet = set(wordList)
    if endWord not in wordSet:
        return []

    # Initialize variables
    level = {beginWord: 0}
    parents = defaultdict(set)
    queue = deque([beginWord])
    word_len = len(beginWord)
    found = False
    current_level = 0

    while queue and not found:
        current_level += 1
        for _ in range(len(queue)):
            word = queue.popleft()
            for i in range(word_len):
                for c in 'abcdefghijklmnopqrstuvwxyz':
                    if c == word[i]:
                        continue
                    next_word = word[:i] + c + word[i+1:]
                    if next_word in wordSet:
                        if next_word not in level:
                            level[next_word] = current_level
                            queue.append(next_word)
                        if level[next_word] == current_level:
                            parents[next_word].add(word)
                        if next_word == endWord:
                            found = True
        # Optional: Remove words that have been visited to prevent revisiting
        # wordSet -= set(parents.keys())

    if not found:
        return []

    # Backtracking to build paths
    res = []
    path = [endWord]

    def backtrack(word):
        if word == beginWord:
            res.append(path[::-1])
            return
        for parent in parents[word]:
            path.append(parent)
            backtrack(parent)
            path.pop()

    backtrack(endWord)
    return res

```

```

backtrack(endWord)
return res

```

This implementation begins by converting the `wordList` into a set for efficient lookups and checks if the `endWord` is present. It then performs a BFS to determine the shortest path levels and simultaneously builds a `parents` mapping to track predecessors for each word. Once the `endWord` is found, it employs a DFS-based backtracking approach to construct all valid shortest transformation sequences by traversing the `parents` mapping from `endWord` back to `beginWord`.

Explanation

The provided Python implementation defines a function `findLadders` which takes `beginWord`, `endWord`, and `wordList` as inputs and returns a list of all shortest transformation sequences from `beginWord` to `endWord`.

- **Edge Case Handling:**

- Convert the `wordList` to a set (`wordSet`) for $O(1)$ lookups.
- If the `endWord` is not present in the `wordSet`, return an empty list as no transformation is possible.

- **BFS Traversal:**

- Initialize a `level` dictionary to store the level (distance from `beginWord`) of each word.
- Initialize a `parents` mapping using `defaultdict(set)` to track all possible predecessors of each word.
- Use a queue to perform BFS, starting with the `beginWord`.
- Iterate level by level, exploring all one-letter transformations.
- For each valid transformation, update the `level` and `parents` mappings.
- Terminate BFS early if the `endWord` is found to ensure only the shortest paths are considered.

- **Backtracking with DFS:**

- If the `endWord` is found, initiate backtracking to construct all valid shortest transformation sequences.
- Use a recursive `backtrack` function that traverses from the `endWord` to the `beginWord` using the `parents` mapping.
- Accumulate paths and append valid sequences to the result list `res`.

- **Termination:**

- If BFS completes without finding the `endWord`, return an empty list.
- Otherwise, return the list of all valid shortest transformation sequences.

This approach ensures that all shortest transformation sequences are captured by first identifying the minimum number of steps required using BFS and then systematically constructing all possible paths of that length using DFS-based backtracking.

Why This Approach

Combining BFS with DFS-based backtracking is a strategic choice for the **Word Ladder II** problem because:

- **BFS Guarantees Shortest Paths:** BFS explores the graph level by level, ensuring that the first time the `endWord` is encountered, it is reached via the shortest possible path.
- **Efficient Path Construction:** By tracking all possible parents during BFS, the algorithm can backtrack to construct every valid shortest sequence without redundant computations.
- **Handling Multiple Paths:** The `parents` mapping allows the algorithm to handle multiple predecessors for a single word, enabling the enumeration of all shortest sequences.
- **Optimal Time and Space Usage:** This method avoids exploring longer paths once the shortest paths are found, optimizing both time and space complexities.

Alternative Approaches

An alternative approach to solving the **Word Ladder II** problem is to implement **Bidirectional BFS** combined with backtracking. Here's how it can be structured:

1. **Initialize Two BFS Frontiers:**
 - One starting from the `beginWord`.
 - Another starting from the `endWord`.
2. **Expand the Smaller Frontier:**
 - At each step, expand the frontier with fewer nodes to optimize performance.
 - Update the `parents` mapping for both directions.
3. **Detect Intersection:**

- When the frontiers intersect, initiate backtracking from the intersection points to construct all valid shortest paths.

4. Termination:

- If no intersection is found, return an empty list.

Advantages of Bidirectional BFS:

- Reduces the search space by simultaneously exploring from both ends.
- Potentially lowers the time complexity, especially for large word lists.

Similar Problems to This One

Similar problems that involve finding all shortest paths or sequences in a graph-like structure include:

- **Word Ladder I:** Finding the length of the shortest transformation sequence.
- **Minimum Genetic Mutation:** Determining the minimum number of mutations to transform one gene string into another.
- **Sliding Puzzle Problems:** Such as the 8-puzzle, where the goal is to reach a target configuration through a series of valid moves.
- **Shortest Path in a Maze:** Finding the shortest path from a start point to an endpoint within a maze.
- **Course Schedule II:** Determining the order of courses to take based on prerequisites.

These problems share the common theme of navigating through a space of possibilities to find optimal paths or sequences, often leveraging similar algorithmic strategies like BFS, DFS, and backtracking.

Things to Keep in Mind and Tricks

- **Managing Visited States:** Carefully track visited words to prevent cycles and ensure that only the shortest paths are considered.
- **Backtracking with Parents Mapping:** Maintaining a comprehensive parents mapping during BFS is essential for accurately reconstructing all shortest transformation sequences.

- **Early Termination:** Once the `endWord` is found during BFS, halt further exploration to focus solely on constructing the shortest paths.
- **Handling Edge Cases:** Address scenarios where the `endWord` is not in the `wordList`, when `beginWord` equals `endWord`, or when no transformation is possible.
- **Bidirectional BFS:** For enhanced performance, especially with large datasets, consider implementing Bidirectional BFS to halve the search depth and reduce computational overhead.
- **Optimizing Space:** Clear the lists in `all_combo_dict` after processing to free up memory and prevent redundant checks.
- **Consistent Word Length:** Ensure that all words are of the same length to maintain consistency in generating intermediate states.

Corner and Special Cases to Test When Writing the Code

To ensure the robustness and correctness of the solution, consider testing the following corner cases:

- **End Word Not in Word List:** Verify that the function returns an empty list when the `endWord` is absent from the `wordList`.
- **Begin Word Equals End Word:** When the `beginWord` is identical to the `endWord`, the shortest transformation sequence is trivially the word itself.
- **Single Transformation:** Cases where the `beginWord` can transform into the `endWord` in one step.
- **Multiple Shortest Paths:** Ensure that all valid shortest transformation sequences are captured when multiple paths of equal length exist.
- **No Possible Transformation:** Scenarios where no sequence of valid transformations can lead from `beginWord` to `endWord`.
- **Large Word List:** Test the algorithm's performance and efficiency with a large number of words in the `wordList`.
- **Words with No Common Letters:** Ensure that the algorithm correctly identifies when no single-letter transformations are possible between words.
- **Prefix Words:** Words where one word is a prefix of another, ensuring the algorithm handles such scenarios without errors.
- **Self-Transformation:** Words that require transforming a letter to itself (e.g., transforming "a" to "a") should be handled appropriately.
- **Duplicate Words in Word List:** Ensure that the algorithm can handle duplicate entries in the `wordList` without affecting the outcome.

Problem 18.8 Rotting Oranges

The **Rotting Oranges** problem involves determining the minimum number of minutes that must elapse until no cell has a fresh orange in a given grid. Each cell in the grid can have one of three values:

- 0: Represents an empty cell.
- 1: Represents a fresh orange.
- 2: Represents a rotten orange.

An orange becomes rotten if it is adjacent (up, down, left, or right) to a rotten orange. The goal is to calculate the minimum time required for all fresh oranges to rot. If it is impossible to rot all oranges, the function should return -1 .

This problem utilizes Breadth-First Search (BFS) to efficiently simulate the spread of rot from rotten oranges to fresh ones, determining the minimum time required for all oranges to rot.

Problem Statement

Given a 2D grid where each cell can have values 0, 1, or 2, representing an empty cell, a fresh orange, or a rotten orange respectively, determine the minimum number of minutes that must elapse until no cell has a fresh orange. An orange becomes rotten if it is adjacent (up, down, left, or right) to a rotten orange. If it is impossible to rot all oranges, return -1 .

Inputs:

- `grid`: A list of lists of integers representing the grid.

Output:

- Return the minimum number of minutes required for all fresh oranges to rot, or -1 if it is impossible.

Examples:

Example 1:

Input:

```
grid = [
    [2,1,1],
    [1,1,0],
    [0,1,1]
]
```

Output: 4

Explanation:

```
Minute 0: [2,1,1], [1,1,0], [0,1,1]
Minute 1: [2,2,1], [2,1,0], [0,1,1]
Minute 2: [2,2,2], [2,2,0], [0,1,1]
Minute 3: [2,2,2], [2,2,0], [0,2,1]
Minute 4: [2,2,2], [2,2,0], [0,2,2]
All oranges rot in 4 minutes.
```

Example 2:

Input:

```
grid = [
    [2,1,1],
    [0,1,1],
    [1,0,1]
]
```

Output: -1

Explanation:

The orange at position (2,0) cannot be reached by any rotten orange, so it is impossible to rot all oranges.

Example 3:

Input:

```
grid = [
    [0,0,0],
    [0,0,0],
    [0,0,0]
]
```

Output: 0

Explanation: There are no oranges at all.

Example 4:

Input:

```
grid = [
    [2,2,2],
    [2,2,2],
    [2,2,2]
]
```

Output: 0

Explanation: All oranges are already rotten.

Example 5:

Input:

```
grid = [
    [1,2,1],
    [2,1,2],
    [1,2,1]
]
```

Output: 1

Explanation: All fresh oranges are adjacent to rotten ones and will rot in 1 minute.

LeetCode link: Rotting Oranges

[LeetCode Link]

[GeeksForGeeks Link]

[HackerRank Link]

[CodeSignal Link]

[InterviewBit Link]

[Educative Link]

[Codewars Link]

Algorithmic Approach

The **Rotting Oranges** problem uses Breadth-First Search (BFS). Here's how it works:

Step 1: Initialization

- Traverse the grid to find all initially rotten oranges (value 2)
- Count all fresh oranges (value 1)

Step 2: BFS Traversal

Define four possible directions: up, down, left, right.

For each minute of processing:

- Process all currently rotten oranges
- For each rotten orange:
 - Check all four adjacent cells
 - If a fresh orange is found, mark it as rotten

Step 3: Termination

- If all oranges rot: return elapsed time
- If some oranges can't rot: return -1

Utilizing BFS ensures that the rot spreads level by level, accurately tracking the minimum time required to rot all reachable fresh oranges.

Complexities

- **Time Complexity:** $\mathcal{O}(N \times M)$, where N is the number of rows and M is the number of columns in the grid. In the worst case, every cell is processed once.
- **Space Complexity:** $\mathcal{O}(N \times M)$, due to the space required to store the queue in BFS, which could potentially contain all cells in the grid.

Python Implementation

Below is the complete Python code that implements a solution to the **Rotting Oranges** problem:

```
from collections import deque

def orangesRotting(grid):
    rows, cols = len(grid), len(grid[0])
    rotten = deque()
    fresh_oranges = 0
    minutes = 0

    # Find all rotten oranges and count fresh oranges
    for r in range(rows):
        for c in range(cols):
            if grid[r][c] == 2:
                rotten.append((r, c))
            elif grid[r][c] == 1:
                fresh_oranges += 1

    # Directions: up, down, left, right
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]

    # If there are no fresh oranges, return 0
    if fresh_oranges == 0:
        return 0

    # BFS to spread rot
    while rotten and fresh_oranges > 0:
        minutes += 1
        for _ in range(len(rotten)):
            x, y = rotten.popleft()

            for dx, dy in directions:
                nx, ny = x + dx, y + dy

                # Check boundaries
                if nx < 0 or ny < 0 or nx >= rows or ny >= cols:
                    continue

                # If the orange is fresh, rot it
                if grid[nx][ny] == 1:
                    grid[nx][ny] = 2
                    fresh_oranges -= 1
                    rotten.append((nx, ny))

    return minutes if fresh_oranges == 0 else -1

# Example usage:
grid1 = [
```

Implementing BFS requires careful management of the queue and tracking of fresh oranges to ensure accurate time calculation and termination conditions.

```

[2,1,1],
[1,1,0],
[0,1,1]
]
print(orangesRotting(grid1)) # Output: 4

grid2 = [
[2,1,1],
[0,1,1],
[1,0,1]
]
print(orangesRotting(grid2)) # Output: -1

```

Explanation

The `orangesRotting` function takes a 2D grid as input and returns the minimum minutes required for all fresh oranges to rot.

Key Components

Initialization:

- Get grid dimensions (`rows, cols`)
- Create queue (`rotten`) for rotten orange positions
- Initialize fresh orange counter
- Initialize time tracker

Grid Processing:

- Traverse grid once to:
 - Record positions of rotten oranges
 - Count fresh oranges

BFS Implementation

Main Loop:

- Process while queue has elements and fresh oranges exist
- Increment minutes at each level

- Process all oranges at current level before moving to next

Orange Processing:

- Check four adjacent positions
- Validate grid boundaries
- Convert fresh oranges to rotten
- Update counters and queue

Termination

- Return minutes if all oranges rotted
- Return -1 if any fresh oranges remain

Why this approach

Breadth-First Search (BFS) is chosen for this problem because it naturally explores all possible paths of rot spread level by level, ensuring that the first time a fresh orange is rotted corresponds to the shortest possible path (i.e., minimum time). BFS guarantees that the minimum number of steps to reach any orange is found, making it the most suitable algorithm for determining the minimum time required to rot all oranges.

Alternative approaches

An alternative approach to solving the **Rotting Oranges** problem is using **Depth-First Search (DFS)** with backtracking. However, DFS is less efficient for this problem because it does not guarantee the discovery of the shortest path first. DFS would require exploring all possible paths to ensure that the minimum time is found, leading to higher time complexity compared to BFS.

Another alternative is using **Multi-Source BFS**, which treats all initially rotten oranges as sources and spreads the rot simultaneously from all of them. This is effectively what the standard BFS implementation does, making it inherently efficient for this problem.

Similar problems to this one

Similar problems that involve spreading or updating states in a grid include:

- **Flood Fill:** Changing the color of a region in a grid.
- **Number of Islands:** Counting the number of connected groups of land cells in a grid.
- **Walls and Gates:** Filling empty rooms with the distance to the nearest gate.
- **Sliding Puzzle Problems:** Solving puzzles by moving tiles to reach a target configuration.
- **Minimum Path Sum:** Finding the path from top-left to bottom-right of a grid with the minimum sum.

These problems share common themes of grid traversal, state updating, and efficient search strategies like BFS and DFS.

Things to keep in mind and tricks

- **Early Termination:** If there are no fresh oranges initially, return 0 immediately to avoid unnecessary processing.
- **Queue Management:** Use a queue data structure to implement BFS efficiently, ensuring FIFO order of processing.
- **Boundary Checks:** Always verify that adjacent cell indices are within grid boundaries to prevent index errors.
- **State Updates:** Update the state of fresh oranges to rotten immediately upon processing to avoid revisiting them.
- **Direction Vectors:** Utilize direction vectors to simplify the process of exploring adjacent cells.
- **Handling Isolated Oranges:** Recognize that some fresh oranges may be isolated and cannot be rotted, necessitating a return value of -1 .
- **Optimizing Space:** By enqueueing only the positions of rotten oranges and tracking the count of fresh oranges, space usage is optimized.

Common Mistakes to Avoid

- **Forgetting Base Cases:** Not handling empty grids or grids with no fresh oranges.
- **Incorrect Time Tracking:** Incrementing time counter inside the inner loop instead of once per level.

- **Missing Boundary Checks:** Not validating grid boundaries when exploring adjacent cells.
- **Queue Management:** Not processing all oranges at the current time step before moving to the next minute.
- **State Tracking:** Not keeping accurate count of remaining fresh oranges.

Corner and special cases to test when writing the code

To ensure robustness and correctness, consider testing the following corner cases:

- **No Oranges at All:** A grid with all cells empty (0), expecting a return value of 0.
- **All Oranges Rotten Initially:** A grid where all oranges are already rotten (2), expecting a return value of 0.
- **No Fresh Oranges:** If there are no fresh oranges to begin with, the function should return 0.
- **Single Fresh Orange:** A grid with only one fresh orange adjacent to a rotten orange, expecting a return value of 1.
- **Isolated Fresh Orange:** A fresh orange that is not adjacent to any rotten oranges, expecting a return value of -1.
- **All Fresh Oranges:** A grid with only fresh oranges and no rotten oranges, expecting a return value of -1.
- **Large Grid:** A large grid with multiple rotten and fresh oranges to test the algorithm's performance and efficiency.
- **Multiple Isolated Regions:** Multiple regions within the grid where some have fresh oranges that can be rotted and others have fresh oranges that cannot be rotted.
- **Edge Cells Rotting:** Fresh oranges located at the edges or corners of the grid to ensure proper boundary handling.
- **Self-Rotating Cells:** Cells that might rotate to themselves inadvertently, ensuring that such scenarios do not affect the outcome.

Problem 18.9 Heaps

Problem Statement

A **heap** is a specialized tree-based data structure that satisfies the heap property:

Heaps are a fundamental data structure used to implement priority queues and support efficient algorithms like heap sort and graph algorithms. This section delves into the properties of heaps, common operations, and their applications in solving various algorithmic problems.

- In a **max heap**, for any given node C , if P is a parent node of C , then the key (value) of P is greater than or equal to the key of C . This ensures that the largest key is at the root of the heap.
- In a **min heap**, the key of P is less than or equal to the key of C , ensuring that the smallest key is at the root.

Heaps are commonly implemented as binary heaps, where each node has at most two children. They are widely used in algorithms that require quick access to the largest or smallest element, such as priority queues, heap sort, and graph algorithms like Dijkstra's and Prim's.

Algorithmic Approach

Heaps support several fundamental operations efficiently:

1. **Insertion:** Add a new element to the heap while maintaining the heap property.
2. **Extraction:** Remove and return the root element (maximum in max heap, minimum in min heap) while maintaining the heap property.
3. **Heapify:** Convert an unordered array into a heap.

These operations are typically performed in $O(\log N)$ time, where N is the number of elements in the heap. The heap can be efficiently represented using an array, leveraging the properties of a complete binary tree.

Heap Operations

- **Insert Operation:**
 - Append the new element at the end of the heap.
 - Percolate (bubble) the element up by swapping it with its parent until the heap property is restored.
- **Extract Operation:**
 - Remove the root element.
 - Replace the root with the last element in the heap.
 - Percolate (bubble) the new root down by swapping it with its larger (max heap) or smaller (min heap) child until the heap property is restored.
- **Heapify Operation:**
 - Starting from the last non-leaf node, perform the extract operation to ensure each subtree satisfies the heap property.
 - Repeat this process iteratively or recursively for all parent nodes.

Complexities

- **Time Complexity:**

- Insertion: $O(\log N)$
- Extraction: $O(\log N)$
- Heapify: $O(N)$

- **Space Complexity:** $O(N)$, where N is the number of elements in the heap.

This space is used to store the heap in an array.

Python Implementation

Below is a complete Python implementation of a max heap, including insertion, extraction, and heapify operations:

```
class MaxHeap:
    def __init__(self):
        self.heap = []

    def parent(self, index):
        return (index - 1) // 2

    def left_child(self, index):
        return 2 * index + 1

    def right_child(self, index):
        return 2 * index + 2

    def insert(self, key):
        self.heap.append(key)
        self._heapify_up(len(self.heap) - 1)

    def extract_max(self):
        if not self.heap:
            return None
        if len(self.heap) == 1:
            return self.heap.pop()

        root = self.heap[0]
        self.heap[0] = self.heap.pop()
        self._heapify_down(0)
        return root

    def _heapify_up(self, index):
        while index > 0 and self.heap[self.parent(index)] < self.heap[index]:
            # Swap parent and current node
            self.heap[self.parent(index)], self.heap[index] = self.heap[index],
                                                       self.heap[self.parent(index)]
```

Implementing a heap involves maintaining the heap property through insertions and extractions. Below is a Python implementation of a max heap using a list.

```

index = self.parent(index)

def _heapify_down(self, index):
    largest = index
    left = self.left_child(index)
    right = self.right_child(index)

    if left < len(self.heap) and self.heap[left] > self.heap[largest]:
        largest = left
    if right < len(self.heap) and self.heap[right] > self.heap[largest]:
        largest = right
    if largest != index:
        # Swap and continue heapifying
        self.heap[index], self.heap[largest] = self.heap[largest], self.heap[
            ↪ index]
        self._heapify_down(largest)

def heapify(self, array):
    self.heap = array[:]
    start = self.parent(len(self.heap) - 1)
    for i in range(start, -1, -1):
        self._heapify_down(i)

def display(self):
    print(self.heap)

# Example usage:
if __name__ == "__main__":
    max_heap = MaxHeap()
    elements = [3, 1, 6, 5, 2, 4]
    for elem in elements:
        max_heap.insert(elem)
    print("Heap after insertions:")
    max_heap.display() # Output: [6, 5, 4, 1, 2, 3]

    print("Extracted max:", max_heap.extract_max()) # Output: 6
    max_heap.display() # Output: [5, 2, 4, 1, 3]

    # Heapify an existing array
    array = [3, 1, 6, 5, 2, 4]
    max_heap.heapify(array)
    print("Heap after heapify:")
    max_heap.display() # Output: [6, 5, 4, 1, 2, 3]

```

Explanation

The provided Python implementation defines a ‘MaxHeap’ class that encapsulates the behavior of a max heap using a list to store elements. Here’s a breakdown of the

implementation:

- **Initialization:**

- The heap is represented as a list ('self.heap').

- **Helper Methods:**

- 'parent(index)': Returns the parent index of a given node.
- 'left-child(index)': Returns the left child index.
- 'right-child(index)': Returns the right child index.

- **Insertion ("insert" method):**

- Appends the new key to the end of the heap.
- Calls '-heapify-up' to maintain the heap property by swapping the new element with its parent until it's in the correct position.

- **Extraction ("extract-max" method):**

- Removes and returns the root element (maximum value).
- Replaces the root with the last element in the heap.
- Calls '-heapify-down' to restore the heap property by swapping the new root with its largest child until it's correctly positioned.

- **Heapify ('heapify' method):**

- Converts an arbitrary array into a heap.
- Starts from the last non-leaf node and calls '-heapify-down' on each node to ensure the heap property is satisfied.

- **Display ('display' method):**

- Prints the current state of the heap.

Why This Approach

Implementing a heap using a list is efficient in terms of both time and space. The list representation allows for easy access to parent and child nodes through index calculations, enabling efficient insertion and extraction operations with $O(\log N)$ time complexity. This structure is particularly beneficial for implementing priority queues, where quick access to the highest or lowest priority element is essential.

Alternative Approaches

While the array-based implementation is the most common, heaps can also be implemented using tree-based structures with explicit node references. However, this approach typically incurs higher space overhead and less cache-friendly memory access patterns compared to the array-based implementation.

Additionally, Python's ‘heapq’ module provides a built-in heap implementation, which can be used for min heaps. To implement a max heap using ‘heapq’, one can invert the values (e.g., multiply by -1) during insertion and extraction.

Similar Problems to This One

Heaps are versatile and are used to solve a variety of algorithmic problems. Some similar problems include:

- **Kth Largest Element in an Array:** Finding the k -th largest element using a heap.
- **Merge K Sorted Lists:** Merging multiple sorted lists using a heap to efficiently find the smallest current element.
- **Top K Frequent Elements:** Identifying the top k most frequent elements in a dataset using a heap.
- **Sliding Window Maximum:** Finding the maximum value in each sliding window of size k in an array using a heap.
- **Task Scheduler:** Scheduling tasks with cooldown periods using a heap to prioritize tasks.
- **Find Median from Data Stream:** Maintaining a data stream to efficiently find the median using two heaps.

Things to Keep in Mind and Tricks

- **Heap Representation:** Utilize array-based representations for efficient parent and child node access.
- **Custom Comparators:** When implementing heaps that require custom ordering (e.g., max heap), consider inverting values or using tuples to control the comparison.
- **Heapify Efficiency:** The heapify operation can be performed in $O(N)$ time, which is more efficient than inserting elements one by one.

- **Use Built-in Libraries:** Leverage Python's ‘heapq’ module for efficient and tested heap implementations, especially for min heaps.
- **Balancing Heaps:** For problems like finding the median, maintaining two heaps (max heap and min heap) can help in balancing and achieving desired access patterns.
- **Lazy Deletion:** In some heap-based algorithms, it might be beneficial to implement lazy deletion to handle element removals efficiently.
- **Understanding Heap Properties:** A deep understanding of heap properties and behaviors is crucial for implementing efficient solutions.

Corner and Special Cases to Test When Writing the Code

- **Empty Heap:** Ensure that extraction operations handle empty heaps gracefully, possibly by returning ‘None’ or raising appropriate exceptions.
- **Single Element Heap:** Verify that operations work correctly when the heap contains only one element.
- **Duplicate Elements:** Test heaps with duplicate values to ensure that the heap property is maintained correctly.
- **All Elements Same:** A heap where all elements have the same value should maintain the heap property without any issues.
- **Large Number of Elements:** Test the heap implementation with a large dataset to assess performance and memory usage.
- **Negative Values:** Ensure that heaps correctly handle negative values, especially when implementing max heaps using min heaps with inverted values.
- **Non-Comparable Elements:** If implementing heaps that store objects, ensure that all elements are comparable or provide custom comparison logic.
- **Heapify with Unordered Arrays:** Test the heapify function with completely unordered arrays to confirm that it correctly builds the heap.
- **Repeated Insertions and Extractions:** Perform a sequence of insertions and extractions to ensure that the heap maintains its properties throughout.
- **Heap Operations After Exhaustion:** Attempt to extract from the heap after all elements have been removed to ensure proper handling.

Problem 18.10 Merge k Sorted Lists

The "Merge k Sorted Lists" problem is a classic algorithm problem that is often asked in coding interviews. The problem challenges the solver to efficiently combine multiple sorted data streams into a single sorted data stream, which has practical applications in various domains such as merging time-series data from multiple sources or combining sorted log files.

Problem Statement

You are given an array of k linked-lists lists, each linked-list is sorted in ascending order. Your task is to merge all the k sorted linked-lists into one sorted linked-list and return it.

Example:

Consider the following k sorted linked lists:

List 1: 1 → 4 → 5

List 2: 1 → 3 → 4

List 3: 2 → 6

The merged list should be:

1 → 1 → 2 → 3 → 4 → 4 → 5 → 6

Input: The input consists of an array of k pointers to the head nodes of each of the k sorted linked lists.

Output: The output should be the head of the single sorted linked list that is the result of merging the k sorted lists.

LeetCode link: <https://leetcode.com/problems/merge-k-sorted-lists/>

Algorithmic Approach

The solution to this problem can be approached by using a min-heap or a priority queue to efficiently manage the current smallest nodes from each linked list. This method takes advantage of the fact that the heads of each linked list are the smallest elements remaining in each list, so we can perform a similar operation to a merge in merge sort by always selecting the smallest head node.

Complexities

- **Time Complexity:** The total time complexity is $O(N \log k)$, where N is the total number of nodes and k is the number of linked lists.
- **Space Complexity:** The space complexity is $O(k)$ for storing the pointers in the heap at any given time.

ewpage

Python Implementation

Below is the complete Python code for the ‘Solution‘ class, which implements the ‘mergeKLists‘ method to merge k sorted linked lists using a min-heap for efficient retrieval of the smallest node at any step:

```
from Queue import PriorityQueue

class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def mergeKLists(self, lists):
        head = point = ListNode(0)
        pq = PriorityQueue()
        for l in lists:
            if l:
                pq.put((l.val, l))
        while not pq.empty():
            val, node = pq.get()
            point.next = ListNode(val)
            point = point.next
            node = node.next
            if node:
                pq.put((node.val, node))
        return head.next
```

This implementation initially sets up a dummy head and a point reference to track the merged list. The ‘PriorityQueue‘ is used to store the tuple ‘(val, node)‘ for each head of the linked lists. It ensures the heap property such that the smallest value is always at the top. By continuously extracting the smallest and inserting the next element from the same list, the algorithm merges all lists into one sorted linked list.

Why this approach

The min-heap approach was chosen for its optimal time complexity considering the need to frequently find and remove the smallest element from a collection of sorted arrays. This method is efficient because it maintains a heap of only k elements representing the heads of each list, and thus the extract-min and insert operations are $O(\log k)$.

Alternative approaches

Alternative approaches include the brute force method, which involves collecting all nodes into an array, sorting it, and then creating a new sorted list. Another approach is to compare nodes one by one or use divide and conquer to merge lists in pairs.

Similar problems to this one

Similar problems include "Merge Two Sorted Lists" and "Sort List," where the principles of merging or handling sorted data structures are central to the solution.

Things to keep in mind and tricks

Keep

Problem 18.11 Find Median from Data Stream

The **Find Median from Data Stream** problem involves designing a data structure that efficiently supports adding numbers from a data stream and finding the median of the current set of numbers. The median is the middle value in an ordered integer list. If the size of the list is even, the median is the average of the two middle numbers.

This problem utilizes two heaps (a max heap and a min heap) to efficiently track the median of a dynamically changing data stream.

Problem Statement

Design a class `MedianFinder` that supports the following two operations:

- `addNum(num)`: Add an integer number `num` from the data stream to the data structure.
- `findMedian()`: Return the median of all elements so far.

Note:

- The number of elements in the data structure will not exceed 10^5 .

Examples:

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
medianFinder.addNum(3);      // arr[1, 2, 3]
medianFinder.findMedian();  // return 2.0
```

LeetCode link: Find Median from Data Stream

[\[LeetCode Link\]](#)

[\[GeeksForGeeks Link\]](#)

[\[HackerRank Link\]](#)

[\[CodeSignal Link\]](#)

[\[InterviewBit Link\]](#)

[\[Educative Link\]](#)

[\[Codewars Link\]](#)

Algorithmic Approach

The **Find Median from Data Stream** problem can be efficiently solved using two heaps: a max heap to store the lower half of the numbers and a min heap to store the upper half of the numbers. The key idea is to maintain the heaps in such a way that:

1. The max heap contains the smaller half of the numbers.
2. The min heap contains the larger half of the numbers.
3. The sizes of the heaps differ by at most one.

This structure allows for quick retrieval of the median:

- If both heaps have the same number of elements, the median is the average of the top elements of both heaps.
- If one heap has more elements, the median is the top element of that heap.

Utilizing two heaps ensures that insertion and median retrieval operations can be performed in $O(\log n)$ time, maintaining efficiency even with a large number of elements.

Complexities

- **Time Complexity:**
 - `addNum`: $O(\log n)$ due to heap insertion.
 - `findMedian`: $O(1)$ as it involves accessing the top elements of the heaps.
- **Space Complexity:** $O(n)$, where n is the number of elements added to the data structure, as all elements are stored in the heaps.

Python Implementation

Below is the complete Python code that implements the **MedianFinder** class to solve the problem:

```
import heapq

class MedianFinder:
    def __init__(self):
        # Max heap for the lower half numbers
        self.small = []
        # Min heap for the upper half numbers
        self.large = []

    def addNum(self, num: int) -> None:
        # Add to max heap (invert the number for max heap simulation)
        heapq.heappush(self.small, -num)

        # Ensure every num in small is <= every num in large
        if self.small and self.large and (-self.small[0] > self.large[0]):
            val = -heapq.heappop(self.small)
            heapq.heappush(self.large, val)

        # Balance the sizes of the two heaps
        if len(self.small) > len(self.large) + 1:
            val = -heapq.heappop(self.small)
            heapq.heappush(self.large, val)
        if len(self.large) > len(self.small):
            val = heapq.heappop(self.large)
            heapq.heappush(self.small, -val)

    def findMedian(self) -> float:
        if len(self.small) > len(self.large):
            return -self.small[0]
        return (-self.small[0] + self.large[0]) / 2.0

# Example usage:
medianFinder = MedianFinder()
medianFinder.addNum(1)
```

Implementing two heaps (max heap and min heap) allows for efficient addition of numbers and median retrieval. Python's `heapq` module provides a min heap, so the max heap can be simulated by inserting negated values.

```
medianFinder.addNum(2)
print(medianFinder.findMedian()) # Output: 1.5
medianFinder.addNum(3)
print(medianFinder.findMedian()) # Output: 2.0
```

Explanation

The **MedianFinder** class is designed to efficiently handle dynamic data streams and provide quick median retrieval. Here's a detailed breakdown of the implementation:

Data Structures

- `self.small`: A max heap that stores the lower half of the numbers. Since Python's `heapq` module only provides a min heap, we simulate a max heap by inserting negated values.
- `self.large`: A min heap that stores the upper half of the numbers.

Adding a Number (`addNum`)

1. **Insert into `self.small`:** Add the negated number to the max heap to maintain the lower half.
2. **Balance Heaps:**
 - If the largest number in `self.small` (i.e., the smallest in the negated heap) is greater than the smallest number in `self.large`, move the number from `self.small` to `self.large`.
 - Ensure that the sizes of the heaps differ by at most one by moving elements between heaps if necessary.

Finding the Median (`findMedian`)

- If `self.small` has more elements, the median is the top element of `self.small`.
- If both heaps have the same number of elements, the median is the average of the top elements of both heaps.

This approach ensures that both adding a number and finding the median are performed in $O(\log n)$ and $O(1)$ time respectively, making it highly efficient for large data streams.

Why This Approach

Using two heaps allows for an efficient way to keep track of the median in a dynamic data stream:

- **Efficiency:** Both insertion and median retrieval operations are efficient, with `addNum` operating in $O(\log n)$ time and `findMedian` operating in $O(1)$ time.
- **Dynamic Updates:** The heap structure adapts dynamically as new numbers are added, ensuring that the median is always accurately maintained.
- **Balanced Heaps:** By maintaining the size difference between the two heaps to at most one, we ensure that the median can be easily derived from the top elements.

This method is optimal for the problem constraints, handling up to 10^5 elements efficiently.

Alternative Approaches

An alternative approach to solving the **Find Median from Data Stream** problem is to use a self-balancing binary search tree (BST) or a skip list to maintain the ordered list of numbers. While this allows for efficient insertion and median retrieval, it generally has higher constant factors compared to the two-heaps method and may be more complex to implement. The two-heaps approach is preferred due to its simplicity and lower time and space overhead.

Similar Problems to This One

Similar problems that involve dynamic median finding or maintaining dynamic order statistics include:

- **Sliding Window Median:** Finding medians in a sliding window over a data stream.
- **Dynamic Order Statistics:** Maintaining statistics (like median, percentile) on a dynamically changing dataset.
- **Continuous Median:** Similar to finding the median in a continuous data stream.
- **Find Mode from Data Stream:** Tracking the most frequent element in a data stream.
- **Dynamic Range Queries:** Performing range-based queries on a dynamically updating dataset.

These problems share the common theme of maintaining dynamic statistics on a data stream or a dynamic dataset, often requiring efficient data structures like heaps or trees.

Things to Keep in Mind and Tricks

- **Heap Balancing:** Always ensure that the two heaps remain balanced in size to facilitate accurate median calculation.
- **Max Heap Simulation:** Python's `heapq` module only provides a min heap. To simulate a max heap, insert negated values.
- **Efficient Median Retrieval:** By maintaining the heaps such that the median can be directly accessed from the top elements, we ensure constant-time median retrieval.
- **Handling Even and Odd Counts:** Correctly handling cases where the number of elements is even or odd is crucial for accurate median computation.
- **Optimizing Space:** Although both heaps may store all elements, their separation into lower and upper halves optimizes the space usage for median retrieval.
- **Edge Cases:** Consider cases with no elements, one element, or all elements being the same to ensure the algorithm handles them gracefully.
- **Thread Safety:** If implementing in a multithreaded environment, ensure that heap operations are thread-safe to prevent race conditions.
- **Memory Management:** Be mindful of memory usage when dealing with large data streams, consider implementing size limits if needed.
- **Numerical Precision:** When calculating averages for even-sized sets, be aware of potential floating-point precision issues.

Corner and Special Cases to Test When Writing the Code

- **No Elements Added:** Calling `findMedian` before any numbers have been added should handle gracefully, possibly by raising an exception or returning a default value.
- **Single Element:** After adding only one number, `findMedian` should return that number.
- **Even Number of Elements:** Ensure that the median is correctly calculated as the average of the two middle numbers.
- **Odd Number of Elements:** Ensure that the median is correctly identified as the middle number.

- **Duplicate Numbers:** Adding multiple identical numbers should not affect the median calculation incorrectly.
- **Negative Numbers:** Ensure that the algorithm correctly handles negative integers.
- **Large Numbers:** Test with very large integer values to ensure no overflow issues occur.
- **High Frequency of Median Retrieval:** Rapidly calling `findMedian` after numerous `addNum` operations to test performance.
- **Alternating Adds and Finds:** Interleave `addNum` and `findMedian` calls to simulate real-time data stream scenarios.
- **All Same Elements:** All added numbers are identical, testing whether the median remains consistent.
- **Floating Point Results:** Test cases where the median results in a floating-point number to verify precision.
- **Maximum Capacity:** Test behavior when reaching the maximum capacity of 10^5 elements.
- **Sequential vs Random:** Test both sequential and random number insertions to ensure consistent behavior.
- **Boundary Values:** Test with minimum and maximum possible integer values to check for overflow handling.

Implementation Considerations

- **Exception Handling:** Implement proper exception handling for edge cases:
 - Empty data structure
 - Memory allocation failures
 - Invalid input values
- **Performance Optimization:** Consider implementing:
 - Lazy rebalancing of heaps
 - Caching of median value
 - Batch processing for multiple additions
- **Memory Efficiency:** Consider:
 - Implementing size limits
 - Memory-efficient heap implementations
 - Garbage collection strategies

Conclusion

The two-heaps method offers an optimal solution for dynamically finding the median in a data stream. By maintaining a balanced partition of the data, it ensures efficient insertion and retrieval operations, making it suitable for handling large-scale data with up to 10^5 elements. This approach leverages the strengths of heap data structures to provide both speed and accuracy in median calculations, making it a preferred choice over more complex alternatives like self-balancing binary search trees.