

Leetcode2.0 Problem Solving Patterns

https://leetcodelv2.vercel.app/revision_sheet

~Sai Ashish

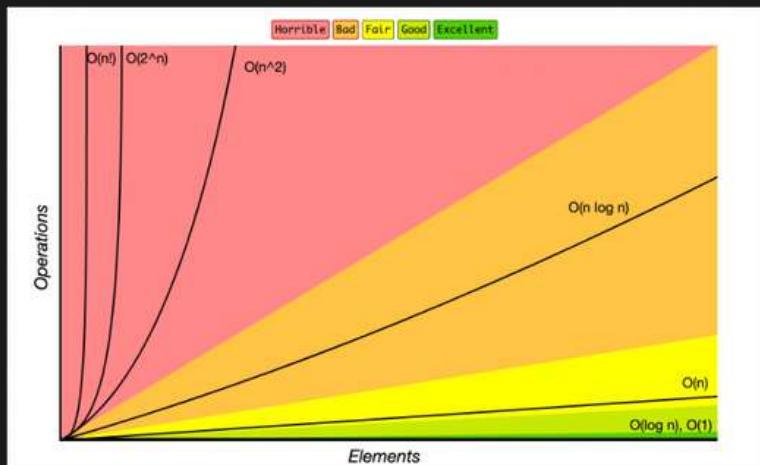
Table Of Contents

1. Two Pointers, Slow & Fast Pointers ($O(n)$)
2. Sliding Window ($O(n)$)
3. Binary Search
4. Prefix Sum
5. Monotonic Queue
6. Heap (Priority Queue)
7. Queue
8. Monotonic Stack
9. Stack
10. Sorting
11. Merge Sort
12. Quickselect
13. Bucket Sort
14. Counting Sort
15. Shell
16. Radix Sort
17. Binary Tree
18. Binary Search Tree
19. Ordered Set
20. Matrix
21. Dynamic Programming
22. Tree
23. Breadth-First Search
24. Depth-First Search
25. Binary Indexed Tree
26. Backtracking
27. Segment Tree
28. Graph
29. Topological Sort
30. Shortest Path
31. Union Find
32. Minimum Spanning Tree
33. Trie
34. Strongly Connected Component
35. Eulerian Circuit
36. Biconnected Component
37. Math
38. Recursion
39. Memoization
40. Divide and Conquer
41. Rolling Hash
42. Greedy
43. Hash Function
44. Combinatorics

- 38. Enumeration
- 39. Design
- 40. String Matching
- 41. Bitmask
- 42. Bit Manipulation
- 43. Counting
- 44. Hash Table
- 46. Database
- 47. Simulation
- 48. Linked List
- 49. Number Theory
- 50. Geometry
- 51. Game Theory
- 52. Interactive
- 53. Data Stream
- 54. Brainteaser
- 55. Randomized
- 57. Iterator
- 58. Concurrency
- 59. Doubly-Linked List
- 60. Probability and Statistics
- 63. Suffix Array
- 66. Line Sweep
- 67. Reservoir Sampling
- 69. Rejection Sampling
- 70. Array
- 71. String
- 72. Algorithms
- 72. C, C++ And Java
- 73. Competitive Programming Basics
- 74. SQL And MongoDB

Algorithms

- Two Pointers
- Fast And Slow Pointers
- Flyod's Cycle Detection (Tortoise and Hare algorithm)
- Manacher's Algorithm ($O(N)$)
- Kadane's Algorithm
- Morris InOrder Traversal Algorithm (Recover BST)
- Dijkstra's Algorithm - $O((V + E) \log V)$
- Bellman-Ford Algorithm - $O(VE)$
- Floyd-Warshall Algorithm - $O(V^3)$
- Kruskal's Algorithm - $O(E \log E)$
- Prim's Algorithm - $O((V + E) \log V)$
- Kahn's Algorithm (BFS)
- Kosaraju's Algorithm - $O(V + E)$
- Tarjan's Algorithm - $O(V + E)$
- Ford-Fulkerson Algorithm - $O(E \times \text{max flow})$
- Edmonds-Karp Algorithm - $O(VE^2)$
- Eulerian Path - Fleury's Algorithm - $O(V + E)$
- Hamiltonian Path - Backtracking - $O(2^V * V)$
- Greedy Algorithms
- Graph Coloring
- Red Black Trees
- Euclidean algorithm for GCD
- KMP algorithm (Knuth-Morris-Pratt)
- Boyer-Moore Algorithm
- Hierholzer's Algorithm (Eulerian Circuit)
- Branch And Bound
- Longest Palindromic Subsequence
- Deterministic And Non Deterministic Algorithms



Two Pointers, Fast & Slow Pointers O(n)

The two pointers technique is a fundamental approach used in solving array and string problems efficiently. It involves using two pointers (or indices) to traverse the data structure in a systematic way, reducing the time complexity compared to brute force solutions. It involve searching, sorting, or optimizing within arrays or strings.

Common Approaches

- Opposite Ends (Left & Right Pointers) → Sorted arrays or palindromes.
- Same Direction (Fast & Slow Pointers) → Used in linked lists or cyclic detection problems. (Floyd's Cycle Detection)
- Different Start Points → Intervals or merging sorted lists / arrays

When to Use Two Pointers?

- ✓ Sorted arrays (Binary Search-like problems)
- ✓ Optimized searching (Two Sum, Three Sum)
- ✓ Finding cycles (Linked list problems)
- ✓ String processing (Palindromes, Substrings)

When to Use Two Pointers in Unsorted Arrays

- ✓ Sorting first if order doesn't matter (Two Sum, Three Sum, etc.).
- ✓ Using a HashSet if sorting isn't ideal (Two Sum without index loss).
- ✓ Fast & Slow pointers for cycle detection (Linked List, Tortoise & Hare).
- ✓ Partitioning problems (Sorting Colors, Move Zeroes, Dutch National Flag).

The Tortoise and Hare algorithm (also known as Floyd's Cycle Detection Algorithm) is a two-pointer technique used to detect cycles in a sequence, most commonly in linked lists. It uses two pointers:

- Tortoise (slow pointer) moves one step at a time.
 - Hare (fast pointer) moves two steps at a time.
-
- If there is a cycle, the fast pointer will eventually meet the slow pointer.
 - If there is no cycle, the fast pointer will reach the end (NULL).

```
public class TwoSumWithoutIndex {  
    public static List<Integer> twoSumWithoutIndex(int[] nums, int target) {  
        Set<Integer> seen = new HashSet<>();  
        for (int num : nums) {  
            int complement = target - num;  
            if (seen.contains(complement)) {  
                return Arrays.asList(num, complement); // Found a valid pair  
            }  
            seen.add(num);  
        }  
        return Collections.emptyList(); // No pair found  
    }  
  
    public static void main(String[] args) {  
        int[] nums = {3, 1, 5, 7, 9};  
        int target = 10;  
        System.out.println(twoSumWithoutIndex(nums, target)); // Output: [3, 7] or  
    }  
}
```

Standard Two Pointers Q (Two Sum)

```
class Solution {  
    public int[] twoSum(int[] numbers, int target) {  
        int left = 0, right = numbers.length - 1;  
        while (left < right) {  
            int sum = numbers[left] + numbers[right];  
            if (sum == target) return new int[]{left + 1, right + 1};  
            else if (sum < target) left++;  
            else right--;  
        }  
        return new int[]{-1, -1}; // No solution  
    }  
}
```

Comparison of Approaches

Approach	Sorting Required?	Time Complexity	Space Complexity
Hash Set	✗ No	O(N)	O(N) (extra space for set)
Two Pointers	✓ Yes	O(N log N) (sorting) + O(N) (search)	O(1)

If the array is already **sorted**, the two-pointer approach is **better** (O(N)).

Otherwise, the **hash set approach** is **faster overall** (O(N)).

Two-pointer technique can be used on unsorted arrays, but it depends on the problem type and how you manipulate the pointers. Unlike the classical sorted two-pointer method, in unsorted arrays, you often need extra data structures or different strategies to make it work efficiently.

When Can Two Pointers Work on Unsorted Arrays?

✓ 1. Finding a Pair with a Given Difference (Variation of Two Sum)

- Instead of a sorted array, use hashing or sorting first to apply two pointers.

◆ Approach:

- Sort the array (if allowed) and use two pointers.
- If sorting isn't allowed, use hashing (set/map) for lookups.

```
import java.util.*;

public class FindPairWithDifference {
    public static boolean hasPairWithDiff(int[] nums, int k) {
        Arrays.sort(nums); // Sorting allows two-pointer usage
        int left = 0, right = 1;

        while (right < nums.length) {
            int diff = nums[right] - nums[left];
            if (diff == k) return true;
            else if (diff < k) right++; // Increase difference
            else left++; // Decrease difference
        }
        return false;
    }

    public static void main(String[] args) {
        int[] nums = {5, 20, 3, 2, 50, 80};
        int k = 78;
        System.out.println(hasPairWithDiff(nums, k)); // Output: true (80 - 2 = 78)
    }
}
```

✓ 2. Sorting-Based Two-Pointer for Subarray Problems

For problems like:

- Smallest subarray with sum $\geq K$
- Max sum subarray with constraints
- We can use two pointers (sliding window) on an unsorted array.

Comparison of Approaches

Approach	Sorting Required?	Time Complexity	Space Complexity
Hash Set	✗ No	O(N)	O(N) (extra space for set)
Two Pointers	✓ Yes	O(N log N) (sorting) + O(N) (search)	O(1)

If the array is already **sorted**, the two-pointer approach is **better** (O(N)).

Otherwise, the **hash set approach** is **faster overall** (O(N)).

Two-pointer technique can be used on unsorted arrays, but it depends on the problem type and how you manipulate the pointers. Unlike the classical sorted two-pointer method, in unsorted arrays, you often need extra data structures or different strategies to make it work efficiently.

When Can Two Pointers Work on Unsorted Arrays?

✓ 1. Finding a Pair with a Given Difference (Variation of Two Sum)

- Instead of a sorted array, use hashing or sorting first to apply two pointers.

◆ Approach:

- Sort the array (if allowed) and use two pointers.
- If sorting isn't allowed, use hashing (set/map) for lookups.

```
import java.util.*;

public class FindPairWithDifference {
    public static boolean hasPairWithDiff(int[] nums, int k) {
        Arrays.sort(nums); // Sorting allows two-pointer usage
        int left = 0, right = 1;

        while (right < nums.length) {
            int diff = nums[right] - nums[left];
            if (diff == k) return true;
            else if (diff < k) right++; // Increase difference
            else left++; // Decrease difference
        }
        return false;
    }

    public static void main(String[] args) {
        int[] nums = {5, 20, 3, 2, 50, 80};
        int k = 78;
        System.out.println(hasPairWithDiff(nums, k)); // Output: true (80 - 2 = 78)
    }
}
```

✓ 2. Sorting-Based Two-Pointer for Subarray Problems

For problems like:

- Smallest subarray with sum $\geq K$
- Max sum subarray with constraints
- We can use two pointers (sliding window) on an unsorted array.

- ✓ Two Pointers is a general technique that can sometimes expand and shrink but is not always used for contiguous subarrays.
- ✓ Sliding Window is a special case of two pointers, always working on contiguous subarrays or substrings to optimize a condition.

⌚ Time Complexity

Typically O(N), since each pointer moves in a linear fashion, avoiding nested loops.

Questions:

LeetCode 5 - Longest Palindromic Substring → Expand around center & Manachers Algorithm

LeetCode 658 - Find K Closest Elements → Binary search

LeetCode 11 - Container With Most Water → Max area using two pointers.

LeetCode 15 - 3Sum → Sort the array and use two pointers

LeetCode 42 - Trapping Rain Water → Track leftMax & rightMax to calculate trapped water.

LeetCode 26 - Remove Duplicates from Sorted Array → Slow & fast pointers for in-place modification.

LeetCode 125 - Valid Palindrome → Two pointers checking from opposite ends.

LeetCode 345 - Reverse Vowels of a String → Swap vowels using two pointers.

LeetCode 611 - Valid Triangle Number → Sort & use two pointers to find valid triangles.

10 LeetCode 977 - Squares of a Sorted Array → two pointers

11. LeetCode 141 - Linked List Cycle → Floyd's cycle detection using slow & fast pointers.

12. Sort Colors - Three Pointers and Swap

13. Move Zeros - Swap Fast & Slow Pointers

```
public class MergeIntervals {
    public int[][] merge(int[][] intervals) {
        if (intervals.length <= 1) return intervals;

        // Sort intervals by start time
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

        List<int[]> merged = new ArrayList<>();
        int[] current = intervals[0];

        for (int i = 1; i < intervals.length; i++) {
            int[] next = intervals[i];

            if (current[1] >= next[0]) {
                // Overlap: extend the current interval
                current[1] = Math.max(current[1], next[1]);
            } else {
                // No overlap: save current and move to next
                merged.add(current);
                current = next;
            }
        }

        // Add the last interval
        merged.add(current);

        return merged.toArray(new int[merged.size()][]);
    }

    public static void main(String[] args) {
        MergeIntervals solution = new MergeIntervals();
        int[][] input = {{1, 3}, {2, 4}, {5, 10}, {15, 18}};
        int[][] result = solution.merge(input);

        for (int[] interval : result) {
            System.out.println(Arrays.toString(interval));
        }
    }
}
```

Questions:

LeetCode 5 - Longest Palindromic Substring → Expand around center using two pointers.

Medium

Given a string `s`, return *the longest palindromic substring* in `s`.

Example 1:

```
Input: s = "babad"
Output: "bab"
Explanation: "aba" is also a valid answer.
```

Example 2:

```
Input: s = "cbbd"
Output: "bb"
```

Constraints:

- `1 <= s.length <= 1000`
- `s` consist of only digits and English letters.

Steps:

- Iterate through the string, treating each index as a potential center of a palindrome.
- Expand around the center to find both odd-length (`expandAroundCenter(i, i)`) and even-length (`expandAroundCenter(i, i + 1)`) palindromes.
- Calculate the palindrome length as right - left - 1, since left moves one step before the palindrome starts, and right moves one step after it ends.
- Update `maxLen` and `start` if the current palindrome length exceeds the previously found maximum.
- Extract the longest palindrome using `str.substring(start, start + maxLen)`.

The `expandAroundCenter` method utilizes **two pointers** (left and right) that move outward simultaneously to check for palindromes. **Sliding Window** is generally used when maintaining a fixed or variable-sized window (e.g., for substring problems like longest unique substring). Here, we are not maintaining a continuous range but rather expanding from a center dynamically.

$$(right - 1) - (left + 1) + 1 = right - left - 2 + 1 = right - left - 1$$

- After the last valid expansion:
 - `left` is one step before the palindrome starts.
 - `right` is one step after the palindrome ends.

- The length of the palindrome is given by:

$$right - left - 1$$

- This correctly accounts for all characters inside the valid palindrome.

```

public static void main(String[] args) {
    String str = "babad";
    int maxLen = 1;
    int start = 0;

    for (int i = 0; i < str.length(); i++) {
        int odd = expandAroundCenter(str, i, i); // odd-length palindromes
        int even = expandAroundCenter(str, i, right: i + 1); // even-length palindromes
        int len = Math.max(odd, even);
        if (len > maxLen) {
            maxLen = len;
            start = i - (maxLen - 1) / 2;
        }
    }

    String result = str.substring(start, start + maxLen);
    System.out.println(result);
}

private static int expandAroundCenter(String str, int left, int right) {
    while (left >= 0 && right < str.length() && str.charAt(left) == str.charAt(right)) {
        left--;
        right++;
    }
    return (right - 1) - (left + 1) + 1;
}
}

```

Time Complexity: O(n^2) (Nested Loops)

Space Complexity: O(1)

Example Walkthrough ("babad")

Step-by-Step Expansion

- $i = 0 \rightarrow "b"$ (odd), $"ba"$ (even) \rightarrow longest: `"b"`
- $i = 1 \rightarrow "bab"$ (odd), $"ba"$ (even) \rightarrow longest: `"bab"`
- $i = 2 \rightarrow "aba"$ (odd), $"ad"$ (even) \rightarrow longest remains `"bab"`
- $i = 3 \rightarrow "a"$ (odd), $"ad"$ (even) \rightarrow longest remains `"bab"`
- $i = 4 \rightarrow "d"$ (odd), $"d"$ (even) \rightarrow longest remains `"bab"`

Manacher's Algorithm (O(N))

Manacher's algorithm efficiently finds the longest palindromic substring in O(n) time using a transformed string and a center-expansion technique with a palindrome radius array (P[])

Approach

1. Preprocess the String

- Convert "babab" → "#b#a#b#a#d#" to handle even-length palindromes uniformly.
- This ensures all palindromes are odd-length.

2. Use a Center and Right Boundary (C and R)

- C : Center of the current longest palindrome.
- R : Rightmost boundary of the palindrome centered at C.

3. Iterate Through the Transformed String (T)

- Mirror property:** If i is inside R, use the value of P[mirror] (mirrored position) as an initial guess.
- Expand further if possible by checking surrounding characters.
- Update C and R when a new palindrome extends past R.

4. Find the Longest Palindrome

- Extract the longest palindrome from the processed array.

```
public static String longestPalindrome(String s) {
    // Step 1: Transform the string
    String t = preprocess(s);
    int n = t.length();
    int[] P = new int[n]; // Stores palindrome radii
    int C = 0, R = 0; // Center and right boundary
    int maxLen = 0, centerIndex = 0;

    // Step 2: Manacher's Algorithm
    for (int i = 0; i < n; i++) {
        int mirror = 2 * C - i; // Mirror of i around center C

        // Step 3: Initialize P[i] using the mirror property
        if (i < R) {
            P[i] = Math.min(R - i, P[mirror]);
        }

        // Step 4: Expand around center i
        while (i + P[i] + 1 < n && i - P[i] - 1 >= 0 &&
              t.charAt(i + P[i] + 1) == t.charAt(i - P[i] - 1)) {
            P[i]++;
        }

        // Step 5: Update Center and Right Boundary
        if (i + P[i] > R) {
            C = i;
            R = i + P[i];
        }

        // Step 6: Track maximum palindrome length
        if (P[i] > maxLen) {
            maxLen = P[i];
            centerIndex = i;
        }
    }
}
```

```

        // Step 7: Extract the longest palindromic substring
        int start = (centerIndex - maxLen) / 2;
        return s.substring(start, start + maxLen);
    }

    private static String preprocess(String s) {
        StringBuilder sb = new StringBuilder();
        sb.append('#');
        for (char c : s.toCharArray()) {
            sb.append(c).append('#');
        }
        return sb.toString();
    }
}

```

Time Complexity: O(n)

Space Complexity: O(1)

Populate => Initialize => Expand => Update => Track

LeetCode 658 - Find K Closest Elements → Binary search to find the starting index of k closest elements.

658. Find K Closest Elements

Medium

Given a sorted integer array `arr`, two integers `k` and `x`, return the `k` closest integers to `x` in the array. The result should also be sorted in ascending order.

An integer `a` is closer to `x` than an integer `b` if:

- $|a - x| < |b - x|$, or
- $|a - x| == |b - x|$ and $a < b$

Example 1:

Input: arr = [1,2,3,4,5], k = 4, x = 3
Output: [1,2,3,4]

Example 2:

Input: arr = [1,2,3,4,5], k = 4, x = -1
Output: [1,2,3,4]

Constraints:

- $1 \leq k \leq \text{arr.length}$
- $1 \leq \text{arr.length} \leq 10^4$
- `arr` is sorted in ascending order.
- $-10^4 \leq \text{arr}[i], x \leq 10^4$

Steps:

- Identify the leftmost starting index of the contiguous subarray and initialize right as `arr.length - k`.
- Use binary search to find the optimal starting index by comparing segment distances relative to `x`, adjusting left and right accordingly.
- Extract the subarray from left to `left + k` as the final result.

Memory Full ⚡

Input:

```
arr = [1,2,3,4,5], k = 4, x = 3
```

Step-by-step process:

- Compute the absolute differences:

```
|1 - 3| = 2
|2 - 3| = 1
|3 - 3| = 0
|4 - 3| = 1
|5 - 3| = 2
```

- Sorting by difference (and then by value if equal):

csharp

```
[3, 2, 4, 1, 5] (sorted by closeness to 3)
```

- Pick the first $k = 4$ elements: [3, 2, 4, 1]
- Sort the final output: [1, 2, 3, 4]

Output: [1,2,3,4] ↓

```

public class Solution {

    public static void main(String[] args) {
        int[] arr = {1, 2, 3, 4, 5};
        int k = 4;
        int x = 3;

        int left = 0;
        int right = arr.length - k;
        // We are trying to find the leftmost starting index of a contiguous subArray
        // of size k that contains the closest elements to x.

        while (left < right) {
            int mid = left + (right - left) / 2;
            if (x - arr[mid] <= arr[mid + k] - x) {
                // This means arr[mid] is closer to x than arr[mid + k] OR both distances are equal.
                right = mid;
            } else {
                left = mid + 1;
            }
        }

        List<Integer> result = Arrays
            .stream(arr)
            .boxed()
            .collect(Collectors.toList())
            .subList(left, k + left);
        System.out.println(result); // [1, 2, 3, 4]
    }
}

```

Time Complexity: $O(\log(n-k) + n)$

Space Complexity: $O(1)$

LeetCode 11 - Container With Most Water → Max area using two pointers moving inward.

11. Container With Most Water

Medium

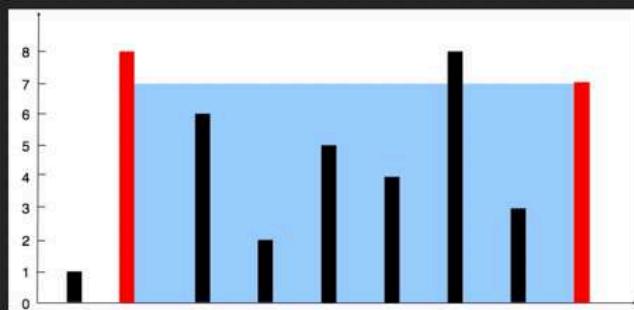
You are given an integer array `height` of length `n`. There are `n` vertical lines drawn such that the two endpoints of the `ith` line are `(i, 0)` and `(i, height[i])`.

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return the maximum amount of water a container can store.

Notice that you may not slant the container.

Example 1:



Input: `height = [1,8,6,2,5,4,8,3,7]`

Output: 49

Explanation: The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49.

```
public class Solution {

    public static void main(String[] args) {
        int[] heights = {1, 8, 6, 2, 5, 4, 8, 3, 7};
        int left = 0;
        int right = heights.length - 1;
        int maxArea = Integer.MIN_VALUE;

        while (left < right) {
            int area = (right - left) * Math.min(heights[left], heights[right]);
            maxArea = Math.max(maxArea, area);

            if (heights[left] < heights[right]) {
                left++;
            } else {
                right--;
            }
        }

        System.out.println(maxArea);
    }
}
```

The time complexity of this solution is O(n).

The space complexity is O(1)

Steps:

- Initialize two pointers, left at the beginning and right at the end of the heights array.
- Calculate the area using the minimum height between the two pointers and their distance.
- Update maxArea with the maximum found so far.
- Move the pointer pointing to the smaller height inward to explore a potentially larger area.
- Repeat until the pointers meet, and return maxArea as the final result.

LeetCode 15 - 3Sum → Fix one element, use two pointers for the remaining sum.

Medium

Given an integer array `nums`, return all the triplets `[nums[i], nums[j], nums[k]]` such that `i != j, i != k, and j != k`, and `nums[i] + nums[j] + nums[k] == 0`.

Notice that the solution set must not contain duplicate triplets.

Example 1:

Input: `nums = [-1,0,1,2,-1,-4]`
Output: `[[-1,-1,2],[-1,0,1]]`
Explanation:
`nums[0] + nums[1] + nums[2] = (-1) + 0 + 1 = 0.`
`nums[1] + nums[2] + nums[4] = 0 + 1 + (-1) = 0.`
`nums[0] + nums[3] + nums[4] = (-1) + 2 + (-1) = 0.`
The distinct triplets are `[-1,0,1]` and `[-1,-1,2]`.
Notice that the order of the output and the order of the triplets does not matter.

Example 2:

Input: `nums = [0,1,1]`
Output: `[]`
Explanation: The only possible triplet does not sum up to 0.

Example 3:

Input: `nums = [0,0,0]`
Output: `[[0,0,0]]`
Explanation: The only possible triplet sums up to 0.

```
private static List<List<Integer>> apply3Sum(int[] nums) {
    Arrays.sort(nums); // timeSort O(n log n)
    List<List<Integer>> result = new ArrayList<>();
    int n = nums.length;

    for (int i = 0; i < n - 2; i++) {
        if (!(i == 0 || (i > 0 && nums[i] != nums[i - 1]))) continue;

        int left = i + 1;
        int right = n - 1;
        int target = -nums[i];

        while (left < right) {
            if (nums[left] + nums[right] == target) {
                result.add(Arrays.asList(nums[left], nums[right], nums[i]));

                while (left < right && nums[left] == nums[left + 1]) {
                    left++;
                }

                while (left < right && nums[right] == nums[right - 1]) {
                    right--;
                }

                left++;
                right--;
            } else if (nums[left] + nums[right] < target) {
                left++;
            } else {
                right--;
            }
        }
    }
    return result;
}
```

The time complexity of this solution is $O(n \log n) + O(n^2) = O(n^2)$

The space complexity:

- Worst-case space complexity: $O(n^2)$ (for storing results)
- Best-case space complexity: $O(1)$ (if no valid triplets are found).

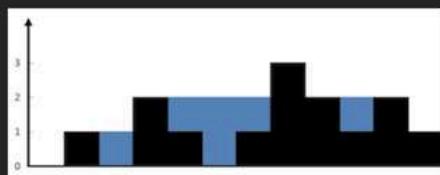
LeetCode 42 - Trapping Rain Water → Track leftMax & rightMax to calculate trapped water.

42. Trapping Rain Water

Hard

Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

Example 1:



Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]

Output: 6

Explanation: The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

Example 2:

Input: height = [4,2,0,3,2,5]

Output: 9

```
public class Solution {  
  
    private static int containerWithMostWater(int[] heights) {  
        int result = 0; // waterTrapped  
        int left_max = 0, right_max = 0;  
        int left = 0;  
        int right = heights.length - 1;  
  
        while (left < right) {  
            if (heights[left] < heights[right]) {  
                if (heights[left] >= left_max) {  
                    left_max = heights[left];  
                } else {  
                    result += left_max - heights[left];  
                }  
                left++;  
            } else {  
                if (heights[right] >= right_max) {  
                    right_max = heights[right];  
                } else {  
                    result += right_max - heights[right];  
                }  
                right--;  
            }  
        }  
  
        return result;  
    }  
  
    public static void main(String[] args) {  
        int[] heights = new int[]{0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1}; // 6  
        System.out.println(containerWithMostWater(heights));  
    }  
}
```

The time complexity of this solution is $O(n)$

The space complexity: $O(1)$

LeetCode 26 - Remove Duplicates from Sorted Array → Slow & fast pointers for in-place modification.

26. Remove Duplicates from Sorted Array

Easy

Given an integer array `nums` sorted in non-decreasing order, remove the duplicates in-place such that each unique element appears once. The relative order of the elements should be kept the same. Then return the number of unique elements in `nums`.

Consider the number of unique elements of `nums` to be `k`, to get accepted, you need to do the following things:

- Change the array `nums` such that the first `k` elements of `nums` contain the unique elements in the order they were present in `nums` initially. The remaining elements of `nums` are not important as well as the size of `nums`.
- Return `k`.

Custom Judge:

The judge will test your solution with the following code:

```
int[] nums = [...]; // Input array
int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;
for (int i = 0; i < k; i++) {
    assert nums[i] == expectedNums[i];
}
```

If all assertions pass, then your solution will be accepted.

Example 1:

```
Input: nums = [1,1,2]
Output: 2, nums = [1,2,_]
Explanation: Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively. It does not matter what you leave beyond the returned k (hence they are underscores).
```

```
class Solution {
    public int removeDuplicates(int[] nums) {
        if (nums.length == 0) return 0;

        int slow = 1; // slow pointer
        for (int fast = 1; fast < nums.length; fast++) { // fast pointer
            if (nums[fast] != nums[fast - 1]) {
                nums[slow++] = nums[fast];
            }
        }
        return slow;
    }
}
```

Steps:

- Check if the array is empty; return 0 if `nums.length == 0`.
- Initialize two pointers: `slow = 1` (tracks unique position) and `fast = 1` (iterates through the array).
- Iterate through the array using the fast pointer.
- If `nums[fast] != nums[fast - 1]`, assign `nums[slow] = nums[fast]`.
- Increment `slow` (`slow++`) to store the next unique element.
- Continue until the end of the array is reached.
- Return `slow` as the new length of the modified array.

The time complexity of this solution is O(n)

The space complexity: O(1)

LeetCode 125 - Valid Palindrome → Two pointers checking from opposite ends.

125. Valid Palindrome

Easy

A phrase is a palindrome if, after converting all uppercase letters into lowercase letters and removing all non-alphanumeric characters, it reads the same forward and backward. Alphanumeric characters include letters and numbers.

Given a string `s`, return `true` if it is a palindrome, or `false` otherwise.

Example 1:

Input: `s = "A man, a plan, a canal: Panama"`
Output: `true`
Explanation: `"amanaplanacanalpanama"` is a palindrome.

Example 2:

Input: `s = "race a car"`
Output: `false`
Explanation: `"raceacar"` is not a palindrome.

Example 3:

Input: `s = ""`
Output: `true`
Explanation: `s` is an empty string `""` after removing non-alphanumeric characters. Since an empty string reads the same forward and backward, it is a palindrome.

Constraints:

- $1 \leq s.length \leq 2 * 10^5$
- `s` consists only of printable ASCII characters.

String ▾ Auto

```
1  class Solution {
2      public boolean isPalindrome(String s) {
3          int left = 0, right = s.length() - 1;
4
5          while (left < right) {
6              char c1 = s.charAt(left);
7              char c2 = s.charAt(right);
8
9              if (!Character.isLetterOrDigit(c1)) {
10                  left++;
11              } else if (!Character.isLetterOrDigit(c2)) {
12                  right--;
13              } else {
14                  if (Character.toLowerCase(c1) != Character.toLowerCase(c2)) {
15                      return false;
16                  }
17                  left++;
18                  right--;
19              }
20          }
21
22          return true;
23      }
24  }
```

- If $s[left]$ is not a letter or digit, we increment $left$ by 1 (skip non-alphanumeric characters).
- If $s[right]$ is not a letter or digit, we decrement $right$ by 1 (skip non-alphanumeric characters).
- If both $s[left]$ and $s[right]$ are valid, we compare them:
- If they are not equal (ignoring case), return false.
- Otherwise, move both $left$ and $right$ inward.

The time complexity of this solution is $O(n)$

The space complexity: $O(1)$

LeetCode 345 - Reverse Vowels of a String → Swap vowels using two pointers.

345. Reverse Vowels of a String

Easy

Given a string s , reverse only all the vowels in the string and return it.

The vowels are 'a', 'e', 'i', 'o', and 'u', and they can appear in both lower and upper cases, more than once.

Example 1:

Input: $s = \text{"hello"}$
Output: "holle"

Example 2:

Input: $s = \text{"leetcode"}$
Output: "leotcede"

Constraints:

- $1 \leq s.length \leq 3 * 10^5$
- s consist of printable ASCII characters.

```
import java.util.HashSet;

public class Solution {
    public String reverseVowels(String s) {
        char[] chars = s.toCharArray();
        HashSet<Character> vowels = new HashSet<>();
        vowels.addAll(Set.of('a', 'e', 'i', 'o', 'u', 'A', 'E', 'I', 'O', 'U'));

        int left = 0, right = chars.length - 1;
        while (left < right) {
            while (left < right && !vowels.contains(chars[left])) {
                left++;
            }
            while (left < right && !vowels.contains(chars[right])) {
                right--;
            }
            if (left < right) {
                char temp = chars[left];
                chars[left] = chars[right];
                chars[right] = temp;
                left++;
                right--;
            }
        }

        return new String(chars);
    }
}
```

The time complexity of this solution is $O(n^2)$

The space complexity: $O(1)$

The key reason while is used instead of if is that multiple non-vowel characters may exist before reaching a vowel, and we need to skip all of them efficiently.

LeetCode 611 - Valid Triangle Number → Sort & use two pointers to find valid triangles.

611. Valid Triangle Number

Medium

Given an integer array `nums`, return the number of triplets chosen from the array that can make triangles if we take them as side lengths of a triangle.

Example 1:

```
Input: nums = [2,2,3,4]
Output: 3
Explanation: Valid combinations are:
2,3,4 (using the first 2)
2,3,4 (using the second 2)
2,2,3
```

Example 2:

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

Constraints:

- $1 \leq \text{nums.length} \leq 1000$
- $0 \leq \text{nums}[i] \leq 1000$

Two Pointers ▾ 🔍 Auto

```
1 import java.util.Arrays;
2
3 public class Solution {
4     public int triangleNumber(int[] nums) {
5         Arrays.sort(nums);
6         int count = 0;
7         for (int i = nums.length - 1; i >= 2; i--) {
8             int left = 0, right = i - 1;
9             while (left < right) {
10                 if (nums[left] + nums[right] > nums[i]) {
11                     count += right - left;
12                     right--;
13                 } else {
14                     left++;
15                 }
16             }
17         }
18     }
19     return count;
20 }
21 }
```

The time complexity of this solution is $O(n \log n) + O(n^2) = O(n^2)$

The space complexity: $O(1)$

Here, iterating from the back ensures we get the largest valid triangle perimeter as early as possible.

10 LeetCode 977 - Squares of a Sorted Array → Merge squares of negatives & positives using two pointers.

977. Squares of a Sorted Array

Easy

Given an integer array `nums`, sorted in non-decreasing order, return an array of the squares of each number sorted in non-decreasing order.

Example 1:

Input: `nums` = [-4,-1,0,3,10]
Output: [0,1,9,16,100]
Explanation: After squaring, the array becomes [16,1,0,9,100].
After sorting, it becomes [0,1,9,16,100].

Example 2:

Input: `nums` = [-7,-3,2,3,11]
Output: [4,9,9,49,121]

Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- `nums` is sorted in non-decreasing order.

Follow up: Squaring each element and sorting the new array is very trivial, could you find an $O(n)$ solution using a different approach?

Two Pointers ▾ Auto

```
1 import java.util.Arrays;
2
3 public class Solution {
4     public int[] sortedSquares(int[] nums) {
5         int n = nums.length;
6         int[] result = new int[n];
7         int left = 0, right = n - 1;
8
9         for (int i = n - 1; i >= 0; i--) {
10             if (Math.abs(nums[left]) > Math.abs(nums[right])) {
11                 result[i] = nums[left] * nums[left];
12                 left++;
13             } else {
14                 result[i] = nums[right] * nums[right];
15                 right--;
16             }
17         }
18
19         return result;
20     }
21 }
22
```

The time complexity of this solution is $O(n)$

The space complexity: $O(n)$

11. LeetCode 141 - Linked List Cycle → Floyd's cycle detection using slow & fast pointers.

141. Linked List Cycle

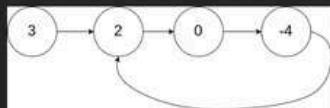
Easy

Given `head`, the head of a linked list, determine if the linked list has a cycle in it.

There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the `next` pointer. Internally, `pos` is used to denote the index of the node that tail's `next` pointer is connected to. Note that `pos` is not passed as a parameter.

Return `true` if there is a cycle in the linked list. Otherwise, return `false`.

Example 1:

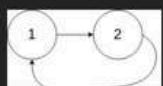


Input: `head = [3,2,0,-4], pos = 1`

Output: `true`

Explanation: There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

Example 2:



Input: `head = [1,2], pos = 0`

Output: `true`

Explanation: There is a cycle in the linked list, where the tail connects to the 0th node.

```
/*
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) {
            return false;
        }
        ListNode slow = head;
        ListNode fast = head.next;
        while (slow != fast) {
            if (fast == null || fast.next == null) {
                return false;
            }
            slow = slow.next;
            fast = fast.next.next;
        }
        return true;
    }
}
```

The time complexity of this solution is $O(n)$

The space complexity: $O(1)$

283. Move Zeroes

Easy

Given an integer array `nums`, move all `0`'s to the end of it while maintaining the relative order of the non-zero elements.

Note that you must do this `in-place` without making a copy of the array.

Example 1:

Input: `nums = [0,1,0,3,12]`
Output: `[1,3,12,0,0]`

Example 2:

Input: `nums = [0]`
Output: `[0]`

Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$

Follow up: Could you minimize the total number of operations done?

Two Pointers ▾ Auto

```
1  public class Solution {
2      public void moveZeroes(int[] nums) {
3          int slow = 0;
4
5          for (int fast = 0; fast < nums.length; fast++) {
6              if (nums[fast] != 0) {
7                  int temp = nums[slow];
8                  nums[slow] = nums[fast];
9                  nums[fast] = temp;
10                 slow++;
11             }
12         }
13     }
14 }
15 }
```

75. Sort Colors

Medium

Given an array `nums` with `n` objects colored red, white, or blue, sort them `in-place` so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers `0`, `1`, and `2` to represent the color red, white, and blue, respectively.

You must solve this problem without using the library's sort function.

Example 1:

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

Example 2:

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

Constraints:

- $n == \text{nums.length}$
- $1 \leq n \leq 300$
- $\text{nums}[i]$ is either `0`, `1`, or `2`.

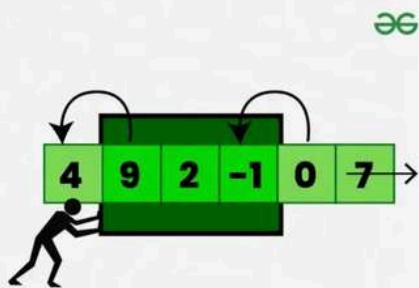
Follow up: Could you come up with a one-pass algorithm using only constant extra space?

Two Pointers ▾ Auto

```
1  class Solution {
2      public void sortColors(int[] nums) {
3          int low = 0;
4          int mid = 0;
5          int high = nums.length - 1;
6
7          while (mid <= high) {
8              if (nums[mid] == 0) {
9                  swap(nums, low, mid);
10                 low++;
11                 mid++;
12             } else if (nums[mid] == 1) {
13                 mid++;
14             } else { // nums[mid] == 2
15                 swap(nums, mid, high);
16                 high--;
17             }
18         }
19     }
20
21     private void swap(int[] nums, int i, int j) {
22         int temp = nums[i];
23         nums[i] = nums[j];
24         nums[j] = temp;
25     }
26
27     public static void main(String[] args) {
28         Solution sol = new Solution();
29         int[] nums = {2, 0, 2, 1, 1, 0};
30         sol.sortColors(nums);
31         for (int num : nums) {
32             System.out.print(num + " ");
33         }
34     }
35 }
36 }
```

Sliding Window

SLIDING WINDOW TECHNIQUE



Sliding Window problems are problems in which a fixed or variable-size window is moved through a data structure, typically an array or string, to solve problems efficiently based on continuous subsets of elements. This technique is used when we need to find subarrays or substrings according to a given set of conditions.

Sliding Window is a specialized form of the Two Pointers technique, mainly used for problems involving subarrays and substrings.

How to use Sliding Window Technique?

There are basically two types of sliding window:

1. Fixed Size Sliding Window:

The general steps to solve these questions by following below steps:

- Find the size of the window required, say K.
- Compute the result for 1st window, i.e. include the first K elements of the data structure.
- Then use a loop to slide the window by 1 and keep computing the result window by window.

2. Variable Size Sliding Window:

The general steps to solve these questions by following below steps:

- In this type of sliding window problem, we increase our right pointer one by one till our condition is true.
- At any step if our condition does not match, we shrink the size of our window by increasing left pointer.
- Again, when our condition satisfies, we start increasing the right pointer and follow step 1.
- We follow these steps until we reach to the end of the array.

How to Identify Sliding Window Problems:

- These problems generally require Finding Maximum/Minimum **Subarray, Substrings** which satisfy some specific condition.
- The size of the subarray or substring '**K**' will be given in some of the problems.
- These problems can easily be solved in $O(N^2)$ time complexity using nested loops, using sliding window we can solve these in $O(n)$ Time Complexity.
- **Required Time Complexity:** $O(N)$ or $O(N \log N)$
- **Constraints:** $N \leq 10^6$, If N is the size of the Array/String.

Sliding Window problems are problems in which a fixed or variable-size window is moved through a data structure, typically an array or string, to solve problems efficiently based on continuous subsets of elements. This technique is used when we need to find subarrays or substrings according to a given set of conditions.

Sliding Window is a specialized form of the Two Pointers technique, mainly used for problems involving subarrays and substrings.

Maximum sum of all subarrays of size K (Kadane's Algorithm)

2. 3. Longest Substring Without Repeating Characters – Expanding & contracting window, uses HashSet/HashMap.

3. 159. Longest Substring with At Most Two Distinct Characters – Variable-sized window, tracks at most two distinct characters.

4. 209. Minimum Size Subarray Sum – Shrinking window while maintaining $\text{sum} \geq \text{target}$, uses prefix sum.

5. 487. Max Consecutive Ones II – Flips one 0 to maximize consecutive 1s, expands until exceeded.

6. 1838. Frequency of the Most Frequent Element – Sorted window approach, expands while keeping the frequency valid.

7. 1248. Count Number of Nice Subarrays – Uses prefix sum + sliding window to count subarrays with exactly k odd numbers.

8. 1918. Kth Smallest Subarray Sum – Binary search + prefix sum sliding window,

9. 239. Sliding Window Maximum – Monotonic deque technique for tracking max in $O(n)$.

10, 713. Subarray Product Less Than K - Use Sliding Window

11. Run Length Coding

```
public class Solution {

    public static void main(String[] args) {
        String s = "aaaabbbccccc";

        StringBuilder encoded = new StringBuilder();
        int left = 0;
        int right = 0;
        int n = s.length();

        while (right < n) {
            while (right < n && s.charAt(right) == s.charAt(left)) {
                right++;
            }
            encoded.append(right - left).append(s.charAt(left));
            left = right;
        }

        System.out.println(encoded);
    }
}
```

Sliding Window problems are problems in which a fixed or variable-size window is moved through a data structure, typically an array or string, to solve problems efficiently based on continuous subsets of elements. This technique is used when we need to find subarrays or substrings according to a given set of conditions.

Sliding Window is a specialized form of the Two Pointers technique, mainly used for problems involving subarrays and substrings.

Maximum sum of all subarrays of size K:

```
private static int maxSubArraySum(int[] arr, int k) {
    int n = arr.length;
    int max_sum = 0;
    for (int i = 0; i < n - k + 1; i++) {
        int sum = 0;
        for (int j = 0; j < k; j++) {
            sum += arr[i + j];
        }
        max_sum = Math.max(max_sum, sum);
    }
    return max_sum;
} // O(n^2)

public class Solution {

    private static int maxSubArray(int[] nums, int k) {
        int n = nums.length;
        int maxSum;
        int windowSum = 0;
        for (int i = 0; i < k; i++) {
            windowSum += nums[i];
        }
        maxSum = windowSum;
        for (int i = k; i < n; i++) {
            windowSum += nums[i] - nums[i - k]; // right - left
            maxSum = Math.max(maxSum, windowSum);
        }

        return maxSum;
    }

    public static void main(String[] args) {
        int[] nums = {2, 2, 3, 4};
        int k = 2;
        System.out.println(maxSubArray(nums, k));
    }
}
```

```
public class MergeIntervals {
    public int[][] merge(int[][] intervals) {
        if (intervals.length <= 1) return intervals;

        // Sort intervals by start time
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[0], b[0]));

        List<int[]> merged = new ArrayList<>();
        int[] current = intervals[0];

        for (int i = 1; i < intervals.length; i++) {
            int[] next = intervals[i];

            if (current[1] >= next[0]) {
                // Overlap: extend the current interval
                current[1] = Math.max(current[1], next[1]);
            } else {
                // No overlap: save current and move to next
                merged.add(current);
                current = next;
            }
        }

        // Add the last interval
        merged.add(current);

        return merged.toArray(new int[merged.size()][]);
    }

    public static void main(String[] args) {
        MergeIntervals solution = new MergeIntervals();
        int[][] input = {{1, 3}, {2, 6}, {8, 10}, {15, 18}};
        int[][] result = solution.merge(input);

        for (int[] interval : result) {
            System.out.println(Arrays.toString(interval));
        }
    }
}
```

19:40 ✓

Kadane Algorithm:

Kadane's Algorithm is a dynamic programming approach used to find the maximum sum of a contiguous subarray in an array of integers.

```

private static int maxSubArraySum(int[] nums) {
    int n = nums.length;
    int curr_sum = 0, max_sum = Integer.MIN_VALUE;
    int start = 0, tempStart = 0, end = 0;

    for (int i = 0; i < n; i++) {
        curr_sum += nums[i];

        if (curr_sum > max_sum) {
            max_sum = curr_sum;
            start = tempStart;
            end = i;
        }

        if (curr_sum < 0) {
            curr_sum = 0;
            tempStart = i + 1;
        }
    }

    System.out.println(Arrays.stream(nums, start, endExclusive: end + 1).boxed()
        .collect(Collectors.toList()));

    return max_sum;
}

public static void main(String[] args) {
    int[] arr = {1, 4, 2, 10, 2, 3, 1, 0, 20};
    System.out.println(maxSubArraySum(arr)); // max_sum => 43, arr => 1, 4, 2, 10, 2, 3, 1, 0, 20
}

```

2.3. Longest Substring Without Repeating Characters – Expanding & contracting window, uses HashSet/HashMap.

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

Example 1:

Input: `s = "abcabcbb"`
Output: 3
Explanation: The answer is "abc", with the length of 3.

Example 2:

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

Example 3:

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

```

public class Solution {
    public int lengthOfLongestSubstring(String s) {
        int[] charIndex = new int[128];
        int left = 0, right = 0, maxLength = 0;
        int n = s.length();

        while (right < n) {
            char currentChar = s.charAt(right);
            left = Math.max(charIndex[currentChar], left);
            maxLength = Math.max(maxLength, right - left + 1);
            charIndex[currentChar] = right + 1;
            right++;
        }

        return maxLength;
    }
}

```

The time complexity of this solution is O(n)

The space complexity: O(1)

Steps:

- Initialize two pointers, left and right, both starting at the beginning of the string.
- As you iterate, increment right and update charIndex with the last occurrence of each character using its ASCII value.
- charIndex keeps track of the most recent index of each character.
- Retrieve the ASCII code of the current character, find its last occurrence, and compute maxLen as right - left + 1.

3. 159. Longest Substring with At Most Two Distinct Characters – Variable-sized window, tracks at most two distinct characters.

159. Longest Substring with At Most Two Distinct Characters

Medium

Given a string s , find the length of the longest substring t that contains at most 2 distinct characters.

Example 1:

```

Input: "eceba"
Output: 3
Explanation: t is "ece" which its length is 3.

```

Example 2:

```

Input: "ccaabbb"
Output: 5
Explanation: t is "aabbb" which its length is 5.

```

Steps:

- Traverse the string using two pointers, left and right, both starting from the beginning.
- Update the character count as the right pointer expands the window.
- If the number of unique characters in the map exceeds 2, decrement the count of the leftmost character and move left forward.
- Continuously update the maximum window length using left and right pointers.
- The final result is $\text{maxLength} = \text{right} - \text{left} + 1$.

The count map stores at most 3 distinct characters at any time (since we only remove when $\text{count.size()} > 2$).

The time complexity of this solution is O(n)

The space complexity: O(1)

```

public class Solution {

    public static int lengthOfLongestSubstringTwoDistinct(String s) {
        int n = s.length();
        int left = 0, right = 0;
        int maxLength = 0;
        Map<Character, Integer> count = new HashMap<>();

        while (right < n) {
            char ch = s.charAt(right);
            count.put(ch, count.getOrDefault(ch, defaultValue: 0) + 1);

            while (count.size() > 2) {
                ch = s.charAt(left);
                count.put(ch, count.get(ch) - 1);
                if (count.get(ch) == 0) {
                    count.remove(ch);
                }
                left++;
            }

            maxLength = Math.max(maxLength, right - left + 1);
            right++;
        }

        return maxLength;
    }

    public static void main(String[] args) {
        String s = "eceba";
        System.out.println(lengthOfLongestSubstringTwoDistinct(s)); // 3
    }
}

```

4. 209. Minimum Size Subarray Sum – Shrinking window while maintaining sum \geq target, uses prefix sum.

209. Minimum Size Subarray Sum

Medium

Given an array of positive integers `nums` and a positive integer `target`, return the *minimal length* of a subarray whose sum is greater than or equal to `target`. If there is no such subarray, return `0` instead.

Example 1:

`Input: target = 7, nums = [2,3,1,2,4,3]`
`Output: 2`
`Explanation: The subarray [4,3] has the minimal length under the problem constraint.`

Example 2:

`Input: target = 4, nums = [1,4,4]`
`Output: 1`

Example 3:

`Input: target = 11, nums = [1,1,1,1,1,1,1]`
`Output: 0`

Constraints:

- $1 \leq \text{target} \leq 10^9$
- $1 \leq \text{nums.length} \leq 10^5$
- $1 \leq \text{nums[i]} \leq 10^4$

Follow up: If you have figured out the $O(n)$ solution, try coding another solution of which the time complexity is $O(n \log(n))$.

```

public class Solution {

    public static int minSubArrayLen(int target, int[] nums) {
        int n = nums.length;
        int left = 0;
        int right = 0;
        int windowSum = 0;
        int result = Integer.MAX_VALUE;
        while (right < n) {
            windowSum += nums[right];
            while (windowSum >= target) {
                result = Math.min(result, right - left + 1);
                windowSum -= nums[left];
                left++;
            }
            right++;
        }
        return result == Integer.MAX_VALUE ? 0 : result;
    }

    public static void main(String[] args) {
        int[] arr = {2, 3, 1, 2, 4, 3};
        int target = 7;
        System.out.println(minSubArrayLen(target, arr)); // 2
    }
}

```

The time complexity of this solution is O(n)

The space complexity: O(1)

Steps:

- Traverse the array using two pointers, left and right, both starting at the beginning.
- Expand the window by adding `nums[right]` to `windowSum`.
- If `windowSum` meets or exceeds the target, shrink the window by subtracting `nums[left]` and moving left forward.
- Continuously update the minimum subarray length using `right - left + 1`.

5. 487. Max Consecutive Ones II – Flips one 0 to maximize consecutive 1s, expands until exceeded.

487. Max Consecutive Ones II

Medium

Given a binary array, find the maximum number of consecutive 1s in this array if you can flip at most one 0.

Example 1:

```

Input: [1,0,1,1,0]
Output: 4
Explanation: Flip the first zero will get the the maximum number of consecutive 1s. After
flipping, the maximum number of consecutive 1s is 4.

```

Note:

- The input array will only contain 0 and 1.
- The length of input array is a positive integer and will not exceed 10,000

Follow up:

What if the input numbers come in one by one as an **infinite stream**? In other words, you can't store all numbers coming from the stream as it's too large to hold in memory. Could you solve it efficiently?

```

public class Solution {

    public static int findMaxConsecutiveOnes(int[] nums) {
        int n = nums.length;
        int k = 1;

        int left = 0, right = 0;

        while (right < n) {
            if (nums[right++] == 0) {
                --k; // apply flip
            }
            if (k < 0 && nums[left++] == 0) {
                ++k; // reverse flip
            }
        }

        return right - left;
    }

    public static void main(String[] args) {
        int[] arr = {1, 0, 1, 1, 0};
        System.out.println(findMaxConsecutiveOnes(arr)); // 4
    }
}

```

The time complexity of this solution is O(n)

The space complexity: O(1)

Steps:

- Initialize two pointers, left and right, both starting at the beginning of the array.
- Move the right pointer forward in a loop.
- If the current element is 0, decrement k since a flip is required.
- If k becomes negative and the left pointer is at 0, increment left to adjust the window.
- The length of the current window is given by right - left.

6. 1838. Frequency of the Most Frequent Element – Sorted window approach, expands while keeping the frequency valid.

1838. Frequency of the Most Frequent Element

Medium

The frequency of an element is the number of times it occurs in an array.

You are given an integer array `nums` and an integer `k`. In one operation, you can choose an index of `nums` and increment the element at that index by 1.

Return the maximum possible frequency of an element after performing at most `k` operations.

Example 1:

Input: `nums = [1,2,4]`, `k = 5`
Output: 3
Explanation: Increment the first element three times and the second element two times to make `nums = [4,4,4]`. 4 has a frequency of 3.

Example 2:

Input: `nums = [1,4,8,13]`, `k = 5`
Output: 2
Explanation: There are multiple optimal solutions:
- Increment the first element three times to make `nums = [4,4,8,13]`. 4 has a frequency of 2.
- Increment the second element four times to make `nums = [1,8,8,13]`. 8 has a frequency of 2.
- Increment the third element five times to make `nums = [1,4,13,13]`. 13 has a frequency of 2.

Example 3:

Input: `nums = [3,9,6]`, `k = 2`
Output: 1

```

public class Solution {

    public static int maxFrequency(int[] nums, int k) {
        int left = 0;
        int right = 0;
        int n = nums.length;
        int sum = 0;
        int result = 0;

        while (right < n) {
            sum += nums[right];
            int windowSize = right - left + 1;
            while ((windowSize * nums[right]) - sum > k){ // 12 - 7 == 5 == 5
                sum -= nums[left];
                left++;
            }
            result = Math.max(result, windowSize);
            right++;
        }

        return result;
    }

    public static void main(String[] args) {
        int[] arr = {1, 2, 4};
        int k = 5;
        System.out.println(maxFrequency(arr, k)); // 3
    }
}

```

The time complexity of this solution is O(n)

The space complexity: O(1)

Steps:

- Sort the array.
- Initialize two pointers, left and right, both starting at the beginning.
- Move the right pointer forward in a loop, adding elements to the sum.
- If the difference between the desired sum (where all elements match `nums[right]`) and the current sum exceeds `k`, increment left and remove the starting element from the sum.
- The window size is `right - left + 1`, and the maximum window size encountered is the final answer.

7. 1248. Count Number of Nice Subarrays – Uses prefix sum + sliding window to count subarrays with exactly k odd numbers.

Medium

Given an array of integers `nums` and an integer `k`. A continuous subarray is called nice if there are `k` odd numbers on it.

Return the number of nice sub-arrays.

Example 1:

```

Input: nums = [1,1,2,1,1], k = 3
Output: 2
Explanation: The only sub-arrays with 3 odd numbers are [1,1,2,1] and [1,2,1,1].

```

Example 2:

```

Input: nums = [2,4,6], k = 1
Output: 0
Explanation: There are no odd numbers in the array.

```

Example 3:

```

Input: nums = [2,2,2,1,2,2,1,2,2,2], k = 2
Output: 16

```

```

public class Solution {

    public static int numberOfSubarrays(int[] nums, int k) {
        return atMost(nums, k) - atMost(nums, k - 1);
    }
    // atMost(nums, k): Counts the number of subarrays with at most k odd numbers.
    // atMost(nums, k - 1): Counts the number of subarrays with at most k - 1 odd numbers.
    // The difference between these two gives us the count of subarrays with exactly k odd numbers.
}

private static int atMost(int[] nums, int k) {
    int left = 0, right = 0, res = 0;
    int n = nums.length;

    while (right < n) {
        if (nums[right] % 2 == 1) {
            --k; // decrement odd numbers
        }
        while (k < 0) {
            if (nums[left] % 2 == 1) {
                ++k; // reverse flips
            }
            ++left;
        }
        int windowSize = right - left + 1;
        res += windowSize;
        right++;
    }

    return res;
}

public static void main(String[] args) {
    int[] arr = {1, 1, 2, 1, 1}; int k = 3;
    System.out.println(numberOfSubarrays(arr, k)); // 3
}
}

```

The time complexity of this solution is O(n)

The space complexity: O(1)

Steps:

- Sort the array.
- Initialize two pointers, left and right, both starting at the beginning.
- Move the right pointer forward in a loop, adding elements to the sum.
- If the difference between the desired sum (where all elements match nums[right]) and the current sum exceeds k, increment left and remove the starting element from the sum.
- The window size is right - left + 1, and the maximum window size encountered is the final answer.

8. 1918. Kth Smallest Subarray Sum – Binary search + prefix sum sliding window, useful for range queries & optimization.

Medium

Given an integer array `nums` of length `n` and an integer `k`, return the `k-th` smallest subarray sum.

A subarray is defined as a non-empty contiguous sequence of elements in an array. A subarray sum is the sum of all elements in the subarray.

Example 1:

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

Output: 3

Explanation: The subarrays of `[2,1,3]` are:

- `[2]` with sum 2
- `[1]` with sum 1
- `[3]` with sum 3
- `[2,1]` with sum 3
- `[1,3]` with sum 4
- `[2,1,3]` with sum 6

Ordering the sums from smallest to largest gives 1, 2, 3, 3, 4, 6. The 4th smallest is 3.

```

public class Solution {
    public static int kthSmallestSubarraySum(int[] nums, int k) {
        int min = Integer.MAX_VALUE, sum = 0;
        for (int num : nums) {
            min = Math.min(min, num);
            sum += num;
        }
        int low = min, high = sum;
        while (low < high) {
            int mid = (high - low) / 2 + low;
            int count = countSubarrays(nums, mid);
            if (count < k)
                low = mid + 1;
            else
                high = mid;
        }
        return low;
    }

    private static int countSubarrays(int[] nums, int target) {
        int left = 0, right = 0, res = 0;
        int n = nums.length;
        int windowSum = 0;
        while (right < n) {
            windowSum += nums[right];
            while (windowSum > target) {
                windowSum -= nums[left++];
            }
            intWindowSize = right - left + 1;
            res += windowSize;
            right++;
        }
        return res;
    }

    public static void main(String[] args) {
        int[] arr = {1, 1, 2, 1, 1}; int k = 3;
        System.out.println(kthSmallestSubarraySum(arr, k)); // 3
    }
}

```

The time complexity of this solution is $O(n \log(\text{sum} - \text{min}))$

The space complexity: $O(1)$

Steps:

- Compute the maxSum (sum of all elements) and the minimum value in the given nums array.
- Set left as the minimum value and right as maxSum.
- Perform a binary search:
 - If $\text{countSubarrays}(\text{nums}, \text{mid}) < k$, search in the right half.
 - Otherwise, search in the left half.
- The final value of low will be the answer.
- Subarray sums are naturally sorted in increasing order.

9. 239. Sliding Window Maximum – Monotonic deque technique for tracking max in O(n).

Hard

You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Example 1:

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:
Window position          Max
-----  
[1 3 -1] -3 5 3 6 7      <strong>3</strong>
1 [3 -1 -3] 5 3 6 7      <strong>3</strong>
1 3 [-1 -3 5] 3 6 7      <strong>5</strong>
1 3 -1 [-3 5 3] 6 7      <strong>5</strong>
1 3 -1 -3 [5 3 6] 7      <strong>6</strong>
1 3 -1 -3 5 [3 6 7]      <strong>7</strong>
```

Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

Sliding Window ▾ Auto

```
1 import java.util.*;
2
3 class Solution {
4     public int[] maxSlidingWindow(int[] nums, int k) {
5         int n = nums.length;
6         if (n * k == 0) return new int[0];
7         if (k == 1) return nums;
8         Deque<Integer> deque = new LinkedList<>();
9         int[] result = new int[n - k + 1];
10        for (int i = 0; i < n; i++) {
11            while (!deque.isEmpty() && deque.peek() < i - k + 1) {
12                deque.poll();
13            }
14            while (!deque.isEmpty() && nums[deque.peekLast()] <= nums[i]) {
15                deque.pollLast();
16            }
17            deque.offer(i);
18            if (i - k + 1 >= 0) {
19                result[i - k + 1] = nums[deque.peek()];
20            }
21        }
22        return result;
23    }
24 }
```

The time complexity of this solution is $O(n)$

The space complexity: $O(n)$

Steps:

- Iterate through the input array `nums`.
- If the deque is not empty, remove indices that are out of the current window.
- Remove elements from the back of the deque if they are smaller than the current element.
- Add the current index to the deque.
- Once the window reaches the required size, store the maximum value (front of the deque) and move to the next window.

10, 713. Subarray Product Less Than K

713. Subarray Product Less Than K

Medium

Given an array of integers `nums` and an integer `k`, return the number of contiguous subarrays where the product of all the elements in the subarray is strictly less than `k`.

Example 1:

Input: `nums = [10,5,2,6]`, `k = 100`
Output: 8
Explanation: The 8 subarrays that have product less than 100 are:
`[10], [5], [2], [6], [10, 5], [5, 2], [2, 6], [5, 2, 6]`
Note that `[10, 5, 2]` is not included as the product of 100 is not strictly less than k.

Example 2:

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

Constraints:

- $1 \leq \text{nums.length} \leq 3 * 10^4$
- $1 \leq \text{nums}[i] \leq 1000$
- $0 \leq k \leq 10^6$

```
class Solution {
    public int numSubarrayProductLessThanK(int[] nums, int k) {
        if (k <= 1) return 0;
        int left = 0;
        int product = 1;
        int count = 0;

        for (int right = 0; right < nums.length; right++) {
            product *= nums[right];

            while (product >= k) {
                product /= nums[left];
                left++;
            }

            count += right - left + 1;
        }

        return count;
    }
}
```

The time complexity of this solution is $O(n)$

The space complexity: $O(1)$

Binary Search

Binary Search is an efficient searching algorithm that operates on a sorted array by repeatedly dividing the search interval in half. It has a time complexity of $O(\log N)$.

1. Normal Binary Search:

```
public class Solution {

    private static int binarySearch(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
                return mid;
            }
            if (nums[mid] > target) {
                right = mid - 1;
            } else if (nums[mid] < target) {
                left = mid + 1;
            }
        }
        return -1;
    }

    public static void main(String[] args) {
        System.out.println(binarySearch(new int[]{1, 2, 3, 4}, target: 2));
    }
}
```

The time complexity of this solution is $O(\log n)$

The space complexity: $O(1)$

2. Next Greater Element (NGE, Upper Bound)

- Find the smallest element greater than the target in a sorted array.

```
public class Solution {

    private static int nextGreaterElement(int[] nums, int target){
        int result = 0;
        int left = 0, right = nums.length - 1;
        while (left <= right){
            int mid = left + (right - left) / 2;
            if(nums[mid] > target){
                result = nums[mid];
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }
        return result;
    }

    public static void main(String[] args) {
        System.out.println(nextGreaterElement(new int[]{1, 2, 3, 4}, target: 2));
    }
}
```

The time complexity of this solution is O(log n)

The space complexity: O(1)

Note:

- First, Consider the right half then the left half

3. Next Smaller Element (NSE, Lower Bound - 1)

Find the largest element smaller than the target in a sorted array.

```
public class Solution {

    private static int nextSmallerElement(int[] nums, int target) {
        int result = -1;
        int left = 0, right = nums.length - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] < target) {
                result = nums[mid];
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 3, 5, 7, 9};
        System.out.println(nextSmallerElement(nums, target: 5)); // Output: 3
    }
}
```

The time complexity of this solution is O(log n)

The space complexity: O(1)

Note:

- First, Consider the left half then the right half

4. Upper Bound

The upper bound in binary search is the smallest element that is strictly greater than the target.

```
public class Solution {

    private static int upperBound(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        int result = -1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] > target) {
                result = nums[mid];
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        return result;
    }
}
```

```

private static int upperBoundIndex(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    int index = -1;

    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (nums[mid] > target) {
            index = mid;
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return index;
}

public static void main(String[] args) {
    int[] nums = {1, 3, 5, 7, 9};
    System.out.println(upperBound(nums, target: 5)); // Output: 7
    System.out.println(upperBoundIndex(nums, target: 5)); // Output: 3 (index of 7)
}
}

```

The time complexity of this solution is O(log n)

The space complexity: O(1)

Note:

- First, Consider the right half then the left half

5. Lower Bound

Find the largest element smaller than or equal to the target in a sorted array.

```

public class Solution {

    private static int lowerBound(int[] nums, int target) {
        int left = 0, right = nums.length - 1;
        int result = -1;

        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] >= target) {
                result = nums[mid];
                right = mid - 1;
            } else {
                left = mid + 1;
            }
        }

        return result;
    }
}

```

```

private static int lowerBoundIndex(int[] nums, int target) {
    int left = 0, right = nums.length - 1;
    int index = -1;

    while (left <= right) {
        int mid = left + (right - left) / 2;

        if (nums[mid] >= target) {
            index = mid;
            right = mid - 1;
        } else {
            left = mid + 1;
        }
    }

    return index;
}

public static void main(String[] args) {
    int[] nums = {1, 3, 5, 7, 9};
    System.out.println(lowerBound(nums, target: 5)); // Output: 5
    System.out.println(lowerBoundIndex(nums, target: 5)); // Output: 2 (index of 5)
}
}

```

The time complexity of this solution is $O(\log n)$

The space complexity: $O(1)$

6. [33] Search in Rotated Sorted Array → (Binary Search on Rotated Array)

There is an integer array `nums` sorted in ascending order (with distinct values).

Prior to being passed to your function, `nums` is possibly rotated at an unknown pivot index `k` ($0 \leq k < \text{nums.length}$) such that the resulting array is `[nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]]` (0-indexed). For example, `[0,1,2,4,5,6,7]` might be rotated at pivot index `3` and become `[4,5,6,7,0,1,2]`.

Given the array `nums` after the possible rotation and an integer `target`, return the `index of target` if it is in `nums`, or `-1` if it is not in `nums`.

You must write an algorithm with $O(\log n)$ runtime complexity.

Example 1:

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

Example 2:

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

Example 3:

Input: `nums = [1]`, `target = 0`
 Output: `-1`

```

public class Solution {

    private static int search(int[] nums, int target) {
        int left = 0;
        int right = nums.length - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {
                return mid;
            }

            if (nums[left] <= nums[mid]) {
                if (nums[left] <= target && target < nums[mid]) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            } else {
                if (nums[mid] < target && target <= nums[right]) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }

        return -1;
    }

    public static void main(String[] args) {
        int[] nums = {4, 5, 6, 7, 0, 1, 2};
        int target = 0;
        System.out.println(search(nums, target)); // 4
    }
}

```

The time complexity of this solution is $O(\log n)$

The space complexity: $O(1)$

7. [34] Find First and Last Position of Element in Sorted Array → (Binary Search + Lower & Upper Bound)

34. Find First and Last Position of Element in Sorted Array

Medium

Given an array of integers `nums`, sorted in non-decreasing order, find the starting and ending position of a given `target` value.

If `target` is not found in the array, return `[-1, -1]`.

You must write an algorithm with $O(\log n)$ runtime complexity.

Example 1:

`Input: nums = [5,7,7,8,8,10], target = 8`
`Output: [3,4]`

Example 2:

`Input: nums = [5,7,7,8,8,10], target = 6`
`Output: [-1,-1]`

Example 3:

`Input: nums = [], target = 0`
`Output: [-1,-1]`

Constraints:

- $0 \leq \text{nums.length} \leq 10^5$

```

class Solution {
public:
    int binarySearchFirst(const vector<int>& nums, int target) {
        int low = 0, high = nums.size() - 1;
        int firstPos = -1;

        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (nums[mid] == target) {
                firstPos = mid;
                high = mid - 1;
            } else if (nums[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }

        return firstPos;
    }

    int binarySearchLast(const vector<int>& nums, int target) {
        int low = 0, high = nums.size() - 1;
        int lastPos = -1;

        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (nums[mid] == target) {
                lastPos = mid;
                low = mid + 1;
            } else if (nums[mid] < target) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }

        return lastPos;
    }
}

```

The time complexity of this solution is $O(\log n)$

The space complexity: $O(1)$

8. [69] Sqrt(x) → (Binary Search on a Mathematical Function)

69. Sqrt(x)

Easy

Given a non-negative integer x , return the square root of x rounded down to the nearest integer. The returned integer should be non-negative as well.

You must not use any built-in exponent function or operator.

- For example, do not use `pow(x, 0.5)` in c++ or `x ** 0.5` in python.

Example 1:

```

Input: x = 4
Output: 2
Explanation: The square root of 4 is 2, so we return 2.

```

Example 2:

```

Input: x = 8
Output: 2
Explanation: The square root of 8 is 2.82842..., and since we round it down to the nearest
integer, 2 is returned.

```

Constraints:

- $0 \leq x \leq 2^{31} - 1$

```

1: #include <iostream>
2: using namespace std;
3:
4: class Solution {
5: public:
6:     int mySqrt(int x) {
7:         if (x == 0 || x == 1) return x;
8:
9:         int low = 0, high = x, ans = 0;
10:        while (low <= high) {
11:            int mid = low + (high - low) / 2;
12:
13:            if (mid * mid <= x) {
14:                ans = mid;
15:                low = mid + 1;
16:            } else {
17:                high = mid - 1;
18:            }
19:        }
20:        return ans;
21:    }
22: };
23:
```

The time complexity of this solution is $O(\log x)$

The space complexity: $O(1)$

9. [74] Search a 2D Matrix → (Binary Search on 2D Grid)

74. Search a 2D Matrix

Medium

You are given an $m \times n$ integer matrix `matrix` with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Given an integer `target`, return `true` if `target` is in `matrix` or `false` otherwise.

You must write a solution in $O(\log(m * n))$ time complexity.

Example 1:

1	3	5	7
10	11	16	20
23	30	34	60

Input: `matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]]`, `target = 3`

Output: `true`

```

class Solution {
    public boolean searchMatrix(int[][] matrix, int target) {
        if (matrix == null || matrix.length == 0 || matrix[0].length == 0) {
            return false;
        }

        int m = matrix.length;
        int n = matrix[0].length;

        int left = 0;
        int right = m * n - 1;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            int midValue = matrix[mid / n][mid % n];

            if (midValue == target) {
                return true;
            } else if (midValue < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return false;
    }
}

```

The time complexity of this solution is $O(\log n)$

The space complexity: $O(1)$

10. [153] Find Minimum in Rotated Sorted Array → (Binary Search on Rotated Array)

Medium

Suppose an array of length n sorted in ascending order is rotated between 1 and n times. For example, the array `nums = [0,1,2,4,5,6,7]` might become:

- `[4,5,6,7,0,1,2]` if it was rotated 4 times.
- `[0,1,2,4,5,6,7]` if it was rotated 7 times.

Notice that rotating an array `[a[0], a[1], a[2], ..., a[n-1]]` 1 time results in the array `[a[n-1], a[0], a[1], a[2], ..., a[n-2]]`.

Given the sorted rotated array `nums` of unique elements, return *the minimum element of this array*.

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

Example 1:

```

Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.

```

Example 2:

```

Input: nums = [4,5,6,7,0,1,2]
Output: 0
Explanation: The original array was [0,1,2,4,5,6,7] and it was rotated 4 times.

```

Example 3:

```

Input: nums = [11,13,15,17]
Output: 11
Explanation: The original array was [11,13,15,17] and it was rotated 4 times.

```

```

class Solution {
public:
    int findMin(vector<int>& nums) {
        int low = 0, high = nums.size() - 1;

        while (low < high) {
            int mid = low + (high - low) / 2;

            if (nums[mid] > nums[high]) {
                low = mid + 1;
            } else {
                high = mid;
            }
        }

        return nums[low];
    }
};

```

Note: We're not skipping the mid here in the left half

The time complexity of this solution is O(log n)

The space complexity: O(1)

11. [162] Find Peak Element → (Binary Search for Peak Finding)

162. Find Peak Element

Medium

A peak element is an element that is strictly greater than its neighbors.

Given a 0-indexed integer array `nums`, find a peak element, and return its index. If the array contains multiple peaks, return the index to any of the peaks.

You may imagine that `nums[-1] = nums[n] = -∞`. In other words, an element is always considered to be strictly greater than a neighbor that is outside the array.

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

Example 1:

Input: `nums = [1,2,3,1]`
Output: 2
Explanation: 3 is a peak element and your function should return the index number 2.

Example 2:

Input: `nums = [1,2,1,3,5,6,4]`
Output: 5
Explanation: Your function can return either index number 1 where the peak element is 2, or index number 5 where the peak element is 6.

Constraints:

- $1 \leq \text{nums.length} \leq 1000$
- $-2^{31} \leq \text{nums}[i] \leq 2^{31} - 1$
- $\text{nums}[i] \neq \text{nums}[i + 1]$ for all valid i .

```

#include <iostream>
#include <vector>
using namespace std;

class Solution {
public:
    int findPeakElement(vector<int>& nums) {
        int low = 0, high = nums.size() - 1;

        while (low < high) {
            int mid = low + (high - low) / 2;

            if (nums[mid] > nums[mid + 1]) {
                high = mid;
            } else {
                low = mid + 1;
            }
        }

        return low;
    }
};

```

The time complexity of this solution is $O(\log n)$

The space complexity: $O(1)$

12. [2187] Minimum Time to Complete Trips → (Binary Search on Answer)

2187. Minimum Time to Complete Trips

Medium

You are given an array `time` where `time[i]` denotes the time taken by the i^{th} bus to complete one trip.

Each bus can make multiple trips successively; that is, the next trip can start immediately after completing the current trip. Also, each bus operates independently; that is, the trips of one bus do not influence the trips of any other bus.

You are also given an integer `totalTrips`, which denotes the number of trips all buses should make in total. Return the *minimum time required for all buses to complete at least `totalTrips` trips*.

Example 1:

Input: `time` = [1,2,3], `totalTrips` = 5
Output: 3

Explanation:

- At time $t = 1$, the number of trips completed by each bus are [1,0,0].
The total number of trips completed is $1 + 0 + 0 = 1$.
 - At time $t = 2$, the number of trips completed by each bus are [2,1,0].
The total number of trips completed is $2 + 1 + 0 = 3$.
 - At time $t = 3$, the number of trips completed by each bus are [3,1,1].
The total number of trips completed is $3 + 1 + 1 = 5$.
- So the minimum time needed for all buses to complete at least 5 trips is 3.

Example 2:

Input: `time` = [2], `totalTrips` = 1
Output: 2
Explanation:
There is only one bus, and it will complete its first trip at $t = 2$.
So the minimum time needed to complete 1 trip is 2.

Binary Search ✓ Auto

```
1 #include <vector>
2 #include <algorithm>
3 using namespace std;
4
5 class Solution {
6 public:
7     bool canCompleteTrips(const vector<int>& buses, long long time, int trips) {
8         long long completedTrips = 0;
9         for (int bus : buses) {
10             completedTrips += time / bus;
11             if (completedTrips >= trips) {
12                 return true;
13             }
14         }
15         return completedTrips >= trips;
16     }
17
18     long long minimumTime(vector<int>& buses, int trips) {
19         long long left = 1, right = (long long)*min_element(buses.begin(), buses.end()) * trips;
20
21         while (left < right) {
22             long long mid = left + (right - left) / 2;
23
24             if (canCompleteTrips(buses, mid, trips)) {
25                 right = mid;
26             } else {
27                 left = mid + 1;
28             }
29         }
30
31         return left;
32     }
33 };
34 
```

The time complexity of this solution is O(log n)

The space complexity: O(1)

1. Normal Binary Search

2. Next Greater Element (NGE, Upper Bound)

3. Next Smaller Element (NSE, Lower Bound - 1)

4. Upper Bound

5. Lower Bound

6. [33] Search in Rotated Sorted Array → (Binary Search on Rotated Array)

7. [34] Find First and Last Position of Element in Sorted Array → (Binary Search + Lower & Upper Bound)

8. [69] Sqrt(x) → (Binary Search on a Mathematical Function)

9. [74] Search a 2D Matrix → (Binary Search on 2D Grid)

10. [153] Find Minimum in Rotated Sorted Array → (Binary Search on Rotated Array)

11. [162] Find Peak Element → (Binary Search for Peak Finding)

12. [2187] Minimum Time to Complete Trips → (Binary Search on Answer)

Prefix Sum

Prefix sum is a powerful technique often used in algorithm problems to quickly compute the sum of elements in a subarray. By precomputing the cumulative sums of elements up to each index, we can answer range sum queries in constant time.

```
public class Solution {

    private static final int[] computePrefixSum(int[] nums) {
        int n = nums.length;
        int[] prefix = new int[n + 1];
        prefix[0] = nums[0];
        for (int i = 1; i < n; i++) {
            prefix[i] = prefix[i - 1] + nums[i];
        }
        return prefix;
    }

    public static void main(String[] args) {
        int[] nums = {5, 7, 7, 8, 8, 10};
        System.out.println(Arrays.toString(computePrefixSum(nums)));
        // [5, 12, 19, 27, 35, 45, 0]
    }
}
```

The time complexity of this solution is $O(n)$

The space complexity: $O(1)$

Problems:

238. Product of Array Except Self

560. Subarray Sum Equals K → (Prefix Sum + HashMap, Classic Subarray Problem)

862. Shortest Subarray with Sum at Least K → (Sliding Window + Monotonic Queue, Hard Level)

974. Subarray Sums Divisible by K → (Prefix Sum + Modulo Technique for Efficient Counting)

525. Contiguous Array → (Prefix Sum + HashMap for Tracking Equal 0s and 1s)

1.238. Product of Array Except Self

238. Product of Array Except Self

Medium

Given an integer array `nums`, return an array `answer` such that `answer[i]` is equal to the product of all the elements of `nums` except `nums[i]`.

The product of any prefix or suffix of `nums` is guaranteed to fit in a 32-bit integer.

You must write an algorithm that runs in $O(n)$ time and without using the division operation.

Example 1:

```
Input: nums = [1,2,3,4]
Output: [24,12,8,6]
```

Example 2:

```
Input: nums = [-1,1,0,-3,3]
Output: [0,0,9,0,0]
```

Constraints:

- $2 \leq \text{nums.length} \leq 10^5$
- $-30 \leq \text{nums}[i] \leq 30$
- The product of any prefix or suffix of `nums` is guaranteed to fit in a 32-bit integer.

Follow up: Can you solve the problem in $O(1)$ extra space complexity? (The output array does not count as extra space for space complexity analysis.)

```
public class Solution {
    public int[] productExceptSelf(int[] nums) {
        int n = nums.length;

        int[] answer = new int[n];

        for (int i = 0; i < n; i++) {
            answer[i] = 1;
        }

        int prefix = 1;
        for (int i = 0; i < n; i++) {
            answer[i] *= prefix;
            prefix *= nums[i];
        }

        int postfix = 1;
        for (int i = n - 1; i >= 0; i--) {
            answer[i] *= postfix;
            postfix *= nums[i];
        }

        return answer;
    }
}
```

The time complexity of this solution is $O(n)$

The space complexity: $O(1)$

2. 560. Subarray Sum Equals K → (Prefix Sum + HashMap, Classic Subarray Problem)

Medium

Given an array of integers `nums` and an integer `k`, return the total number of subarrays whose sum equals to `k`.

A subarray is a contiguous non-empty sequence of elements within an array.

Example 1:

```
Input: nums = [1,1,1], k = 2
Output: 2
```

Example 2:

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

Constraints:

- $1 \leq \text{nums.length} \leq 2 * 10^4$
- $-1000 \leq \text{nums}[i] \leq 1000$
- $-10^7 \leq k \leq 10^7$

```
import java.util.HashMap;

public class Solution {
    public int subarraySum(int[] nums, int k) {
        HashMap<Integer, Integer> prefixSumMap = new HashMap<>();
        
        // Initialize with the base case: prefix sum of 0 appears once
        prefixSumMap.put(0, 1);

        int prefixSum = 0;
        int count = 0;

        for (int num : nums) {
            prefixSum += num;

            if (prefixSumMap.containsKey(prefixSum - k)) {
                count += prefixSumMap.get(prefixSum - k);
            }
        }

        prefixSumMap.put(prefixSum, prefixSumMap.getOrDefault(prefixSum, 0) + 1);
    }

    return count;
}

public static void main(String[] args) {
    Solution solution = new Solution();
    int[] nums = {1, 1, 1};
    int k = 2;
    System.out.println(solution.subarraySum(nums, k)); // Output: 2
}
```

The time complexity of this solution is $O(n)$

The space complexity: $O(1)$

Sliding window does not work for arrays with negative numbers—which is why this approach is invalid for this problem.

862. Shortest Subarray with Sum at Least K → (Sliding Window + Monotonic Queue, Hard Level)

Given an integer array `nums` and an integer `K`, return the length of the shortest non-empty subarray of `nums` with a sum of at least `K`. If there is no such subarray, return `-1`.

A subarray is a contiguous part of an array.

Example 1:

Input: `nums = [1]`, `K = 1`
Output: 1

Example 2:

Input: `nums = [1, 2]`, `K = 4`
Output: -1

Example 3:

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

```
1 import java.util.Deque;
2 import java.util.LinkedList;
3
4 public class Solution {
5
6     public int shortestSubarray(int[] nums, int K) {
7         int n = nums.length;
8         long[] prefixSum = new long[n + 1];
9         for (int i = 0; i < n; i++) {
10             prefixSum[i + 1] = prefixSum[i] + nums[i];
11         }
12
13         Deque<Integer> deque = new LinkedList<>();
14         int minLength = Integer.MAX_VALUE;
15
16         for (int i = 0; i <= n; i++) {
17             while (!deque.isEmpty() && prefixSum[i] - prefixSum[deque.peekFirst()] >= K) {
18                 minLength = Math.min(minLength, i - deque.pollFirst());
19             }
20             while (!deque.isEmpty() && prefixSum[i] <= prefixSum[deque.peekLast()]) {
21                 deque.pollLast();
22             }
23
24             deque.addLast(i);
25         }
26
27         return minLength == Integer.MAX_VALUE ? -1 : minLength;
28     }
29
30     public static void main(String[] args) {
31         Solution solution = new Solution();
32         int[] nums = {2, -1, 2};
33         int K = 3;
34         System.out.println(solution.shortestSubarray(nums, K));
35     }
36 }
37 }
```

The time complexity of this solution is O(n)

The space complexity: O(1)

974. Subarray Sums Divisible by K → (Prefix Sum + Modulo Technique for Efficient Counting)

Medium

Given an integer array `nums` and an integer `k`, return the number of non-empty subarrays that have a sum divisible by `k`.

A subarray is a contiguous part of an array.

Example 1:

```
Input: nums = [4,5,0,-2,-3,1], k = 5
Output: 7
Explanation: There are 7 subarrays with a sum divisible by k = 5:
[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]
```

Example 2:

```
Input: nums = [5], k = 9
Output: 0
```

Constraints:

- $1 \leq \text{nums.length} \leq 3 * 10^4$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- $2 \leq k \leq 10^4$

```
import java.util.HashMap;

public class Solution {
    public int subarraysDivByK(int[] nums, int k) {
        HashMap<Integer, Integer> countMap = new HashMap<>();
        countMap.put(0, 1);

        int prefixSum = 0;
        int count = 0;

        for (int num : nums) {
            prefixSum += num;
            int remainder = (prefixSum % k + k) % k;
            count += countMap.getOrDefault(remainder, 0);
            countMap.put(remainder, countMap.getOrDefault(remainder, 0) + 1);
        }

        return count;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {23, 2, 4, 6, 7};
        int k = 6;
        System.out.println(solution.subarraysDivByK(nums, k)); // Output: 4
    }
}
```

The time complexity of this solution is $O(n)$

The space complexity: $O(1)$

525. Contiguous Array → (Prefix Sum + HashMap for Tracking Equal 0s and 1s)

525. Contiguous Array

Medium

Given a binary array `nums`, return the maximum length of a contiguous subarray with an equal number of `0` and `1`.

Example 1:

Input: `nums` = [0,1]
Output: 2
Explanation: [0, 1] is the longest contiguous subarray with an equal number of 0 and 1.

Example 2:

Input: `nums` = [0,1,0]
Output: 2
Explanation: [0, 1] (or [1, 0]) is a longest contiguous subarray with equal number of 0 and 1.

Constraints:

- `1 <= nums.length <= 105`
- `nums[i]` is either `0` or `1`.

```
import java.util.HashMap;

public class Solution {
    public int findMaxLength(int[] nums) {
        HashMap<Integer, Integer> prefixSumMap = new HashMap<>();
        
        // Initialize with the base case: sum 0 at index -1
        prefixSumMap.put(0, -1);

        int prefixSum = 0;
        int maxLength = 0;

        for (int i = 0; i < nums.length; i++) {
            prefixSum += (nums[i] == 1) ? 1 : -1;

            if (prefixSumMap.containsKey(prefixSum)) {
                maxLength = Math.max(maxLength, i - prefixSumMap.get(prefixSum));
            } else {
                prefixSumMap.put(prefixSum, i);
            }
        }

        return maxLength;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {0, 1, 0, 1, 1, 0};
        System.out.println(solution.findMaxLength(nums)); // Output: 6
    }
}
```

Note: Identify the index where the prefixSum last appeared and compute the maxLength by taking the difference between the indexes

The time complexity of this solution is O(n)

The space complexity: O(1)

Binary Search + Prefix Sum

Split Array Largest Sum

Given an integer array `nums` and an integer `k`, split `nums` into `k` non-empty subarrays such that the largest sum of any subarray is minimized.

Return the minimized largest sum of the split.

A **subarray** is a contiguous part of the array.

Example 1:

Input: `nums = [7,2,5,10,8]`, `k = 2`
Output: 18
Explanation: There are four ways to split `nums` into two subarrays.
The best way is to split it into `[7,2,5]` and `[10,8]`, where the largest sum among the two subarrays is only 18.

Example 2:

Input: `nums = [1,2,3,4,5]`, `k = 2`
Output: 9
Explanation: There are four ways to split `nums` into two subarrays.
The best way is to split it into `[1,2,3]` and `[4,5]`, where the largest sum among the two subarrays is only 9.

Constraints:

- `1 <= nums.length <= 1000`
- `0 <= nums[i] <= 106`
- `1 <= k <= min(50, nums.length)`

```
class Solution {  
    public int splitArray(int[] nums, int k) {  
        int n = nums.length;  
        long[] prefix = new long[n + 1];  
        int maxNum = 0;  
        long totalSum = 0;  
        for (int i = 0; i < n; i++) {  
            prefix[i + 1] = prefix[i] + nums[i];  
            maxNum = Math.max(maxNum, nums[i]);  
            totalSum += nums[i];  
        }  
        long low = maxNum, high = totalSum;  
        long answer = totalSum;  
        while (low <= high) {  
            long mid = low + (high - low) / 2;  
            if (canSplit(prefix, k, mid)) { // lower bound  
                answer = mid;  
                high = mid - 1;  
            } else {  
                low = mid + 1;  
            }  
        }  
        return (int) answer;  
    }  
}
```

```
private boolean canSplit(long[] prefix, int k, long maxSum) {  
    int n = prefix.length - 1;  
    int count = 0;  
    int start = 0;  
    for (int end = 1; end <= n; end++) {  
        long subarraySum = prefix[end] - prefix[start];  
        if (subarraySum > maxSum) {  
            start = end - 1;  
            count++;  
            if (prefix[end] - prefix[start] > maxSum) return false;  
        }  
    }  
    count++;  
    return count <= k;  
}
```

* If subarray sum exceeds `maxSum`:

```
if (subarraySum > maxSum) {  
    start = end - 1; // start new subarray from previous element  
    count++; // we made a split  
  
    if (prefix[end] - prefix[start] > maxSum) return false;
```

- Start a new subarray from index `end - 1`
- Count the split
- Extra check: the single number `nums[end - 1]` shouldn't exceed `maxSum`, else it's impossible

If not `<=`, use `<` and
`high = mid`

Binary Search + Prefix Sum

Koko Eating Bananas (LeetCode 875)

Koko loves to eat bananas. There are n piles of bananas, the i^{th} pile has $\text{piles}[i]$ bananas. The guards have gone and will come back in h hours.

Koko can decide her bananas-per-hour eating speed of k . Each hour, she chooses some pile of bananas and eats k bananas from that pile. If the pile has less than k bananas, she eats all of them instead and will not eat any more bananas during this hour.

Koko likes to eat slowly but still wants to finish eating all the bananas before the guards return.

Return the minimum integer k such that she can eat all the bananas within h hours.

Example 1:

```
Input: piles = [3,6,7,11], h = 8
Output: 4
```

Example 2:

```
Input: piles = [30,11,23,4,20], h = 5
Output: 30
```

Example 3:

```
Input: piles = [30,11,23,4,20], h = 6
Output: 23
```

Constraints:

- $1 \leq \text{piles.length} \leq 10^4$
- $\text{piles.length} \leq h \leq 10^9$
- $1 \leq \text{piles}[i] \leq 10^9$

```
class Solution {
    public int minEatingSpeed(int[] piles, int h) {
        int low = 1;
        int high = Arrays.stream(piles).max().getAsInt();
        int answer = high;

        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (canFinish(piles, mid, h)) {
                answer = mid;
                high = mid - 1;
            } else {
                low = mid + 1;
            }
        }

        return answer;
    }

    private boolean canFinish(int[] piles, int k, int h) {
        long hours = 0;
        for (int pile : piles) {
            hours += (pile + (long)k - 1) / k;
            if (hours > h) return false;
        }
        return true;
    }
}

// Check if Koko can eat all bananas with speed 'k' in 'h' hours
private boolean canFinish(int[] piles, int k, int h) {
    int hours = 0;
    for (int pile : piles) {
        hours += (pile + k - 1) / k; // ceil(pile / k)
    }
    return hours <= h;
}
```

Binary Search + Prefix Sum

Minimum Number of Days to Make m Bouquets (LeetCode 1482)

You are given an integer array `bloomDay`, an integer `m` and an integer `k`.

You want to make `m` bouquets. To make a bouquet, you need to use `k` adjacent flowers from the garden.

The garden consists of `n` flowers, the `ith` flower will bloom in the `bloomDay[i]` and then can be used in exactly one bouquet.

Return the minimum number of days you need to wait to be able to make `m` bouquets from the garden. If it is impossible to make `m` bouquets return `-1`.

Example 1:

```
Input: bloomDay = [1,10,3,10,2], m = 3, k = 1
Output: 3
Explanation: Let us see what happened in the first three days. x means flower bloomed and _ means flower did not bloom in the garden.
We need 3 bouquets each should contain 1 flower.
After day 1: [x, _, _, _, _] // we can only make one bouquet.
After day 2: [x, _, _, _, x] // we can only make two bouquets.
After day 3: [x, _, x, _, x] // we can make 3 bouquets. The answer is 3.
```

Example 2:

```
Input: bloomDay = [1,10,3,10,2], m = 3, k = 2
Output: -1
Explanation: We need 3 bouquets each has 2 flowers, that means we need 6 flowers. We only have 5 flowers so it is impossible to get the needed bouquets and we return -1.
```

Example 3:

```
Input: bloomDay = [7,7,7,7,12,7,7], m = 2, k = 3
Output: 12
Explanation: We need 2 bouquets each should have 3 flowers.
Here is the garden after the 7 and 12 days:
After day 7: [x, x, x, x, _, x, x]
We can make one bouquet of the first three flowers that bloomed. We cannot make another bouquet from the last three flowers that bloomed because they are not adjacent.
After day 12: [x, x, x, x, x, x, x]
It is obvious that we can make two bouquets in different ways.
```

```
class Solution {
    public int minDays(int[] bloomDay, int m, int k) {
        int n = bloomDay.length;
        if ((long)m * k > n) return -1; // Not enough flowers

        int low = 1, high = (int)1e9;
        int result = -1;

        while (low <= high) {
            int mid = low + (high - low) / 2;

            if (canMake(bloomDay, m, k, mid)) {
                result = mid;
                high = mid - 1; // Try earlier day
            } else {
                low = mid + 1; // Need more time
            }
        }

        return result;
    }

    private boolean canMake(int[] bloomDay, int m, int k, int day) {
        int bouquets = 0;
        int flowers = 0;

        for (int bloom : bloomDay) {
            if (bloom <= day) {
                flowers++;
                if (flowers == k) {
                    bouquets++;
                    flowers = 0;
                }
            } else {
                flowers = 0;
            }
        }

        return bouquets >= m;
    }
}
```

Minimum Time to Complete Trips (LeetCode 2187)**2187. Minimum Time to Complete Trips**

Medium

You are given an array `time` where `time[i]` denotes the time taken by the i^{th} bus to complete one trip.

Each bus can make multiple trips successively; that is, the next trip can start immediately after completing the current trip. Also, each bus operates independently; that is, the trips of one bus do not influence the trips of any other bus.

You are also given an integer `totalTrips`, which denotes the number of trips all buses should make in total. Return the *minimum time required for all buses to complete at least `totalTrips` trips*.

Example 1:

Input: `time = [1,2,3], totalTrips = 5`

Output: 3

Explanation:

- At time $t = 1$, the number of trips completed by each bus are $[1, 0, 0]$.
The total number of trips completed is $1 + 0 + 0 = 1$.
 - At time $t = 2$, the number of trips completed by each bus are $[2, 1, 0]$.
The total number of trips completed is $2 + 1 + 0 = 3$.
 - At time $t = 3$, the number of trips completed by each bus are $[3, 1, 1]$.
The total number of trips completed is $3 + 1 + 1 = 5$.
- So the minimum time needed for all buses to complete at least 5 trips is 3.

Example 2:

Input: `time = [2], totalTrips = 1`

Output: 2

Explanation:

- There is only one bus, and it will complete its first trip at $t = 2$.
So the minimum time needed to complete 1 trip is 2.

Binary Search ✓ Auto

```

1 #include <vector>
2 #include <algorithm>
3 using namespace std;
4
5 class Solution {
6 public:
7     bool canCompleteTrips(const vector<int>& buses, long long time, int trips) {
8         long long completedTrips = 0;
9         for (int bus : buses) {
10             completedTrips += time / bus;
11             if (completedTrips >= trips) {
12                 return true;
13             }
14         }
15         return completedTrips >= trips;
16     }
17
18     long long minimumTime(vector<int>& buses, int trips) {
19         long long left = 1, right = (long long) *min_element(buses.begin(), buses.end()) * trips;
20
21         while (left < right) {
22             long long mid = left + (right - left) / 2;
23
24             if (canCompleteTrips(buses, mid, trips)) {
25                 right = mid;
26             } else {
27                 left = mid + 1;
28             }
29         }
30
31         return left;
32     }
33 };

```

The time complexity of this solution is $O(\log n)$

The space complexity: $O(1)$

Monotonic Queue

Monotonic Queue is a data structure that maintains elements in a way that supports efficient querying of either the minimum or maximum value in the current set of elements, depending on whether it's a monotonic increasing or decreasing queue. This structure is useful for problems that require sliding window operations or maintaining order properties dynamically.

Feature	Monotonic Queue	Standard Queue	Priority Queue
Order	Maintains a specific monotonic order	FIFO order	Based on element priority
Complexity	O(1) amortized per operation	O(1) per operation	O(log n) for insertion and deletion
Use Case	Sliding window minimum/maximum	BFS, task scheduling	Dijkstra's algorithm, priority-based tasks
Data Structure	Deque	Queue (LinkedList, ArrayDeque)	Binary heap, Fibonacci heap

- **Monotonic Queue** is ideal for problems that involve maintaining a sorted order within a sliding window for efficient minimum/maximum queries.
- **Standard Queue** is useful for general FIFO operations like BFS and task processing.
- **Priority Queue** is suited for tasks where elements need to be processed based on priority, such as in shortest path algorithms or scheduling tasks with different priorities.

✓ Time Complexity Analysis

Operation	Method	Time Complexity	Explanation
Add to front	<code>addFirst()</code>	O(1)	Just a modular arithmetic update and direct array access.
Add to rear	<code>addLast()</code>	O(1)	Similar logic, efficient due to circular array.
Remove from front	<code>pollFirst()</code>	O(1)	Modular index update and fetch. No shifting involved.
Remove from rear	<code>pollLast()</code>	O(1)	Constant-time access and index shift.
Peek front	<code>peekFirst()</code>	O(1)	Direct access at <code>data[front]</code> .
Peek rear	<code>peekLast()</code>	O(1)	Direct access at <code>data[rear]</code> .
Print deque	<code>printDeque()</code>	O(n)	Loops through all <code>size</code> elements to display.

```
public class MyDeque {
    private int[] data;
    private int front;
    private int rear;
    private int size;
    private int capacity;

    public MyDeque(int capacity) {
        this.capacity = capacity;
        data = new int[capacity];
        front = -1;
        rear = 0;
        size = 0;
    }

    public boolean isFull() {
        return size == capacity;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    public void addFirst(int val) {
        if (isFull()) {
            throw new IllegalStateException("Deque is full");
        }
        if (isEmpty()) {
            front = rear = 0;
        } else {
            front = (front - 1 + capacity) % capacity;
        }
        data[front] = val;
        size++;
    }

    public void addLast(int val) {
        if (isFull()) {
            throw new IllegalStateException("Deque is full");
        }
        if (isEmpty()) {
            front = rear = 0;
        } else {
            rear = (rear + 1) % capacity;
        }
        data[rear] = val;
        size++;
    }

    public Integer pollFirst() {
        if (isEmpty()) {
            return null;
        }
        int val = data[front];
        front = (front + 1) % capacity;
        size--;
        if (isEmpty()) {
            front = -1;
            rear = 0;
        }
        return val;
    }
}
```

```
public Integer pollLast() {
    if (isEmpty()) {
        return null;
    }
    int val = data[rear];
    rear = (rear - 1 + capacity) % capacity;
    size--;
    if (isEmpty()) {
        front = -1;
        rear = 0;
    }
    return val;
}

public Integer peekFirst() {
    return isEmpty() ? null : data[front];
}

public Integer peekLast() {
    return isEmpty() ? null : data[rear];
}

public void printDeque() {
    if (isEmpty()) {
        System.out.println("Deque is empty");
        return;
    }
    System.out.print("Deque: ");
    for (int i = 0; i < size; i++) {
        System.out.print(data[(front + i) % capacity] + " ");
    }
    System.out.println();
}

public static void main(String[] args) {
    MyDeque deque = new MyDeque( capacity: 5);
    deque.addLast( val: 10);
    deque.addFirst( val: 20);
    deque.addLast( val: 30);
    deque.printDeque(); // Deque: 20 10 30
    System.out.println("pollFirst: " + deque.pollFirst()); // pollFirst: 20
    System.out.println("pollLast: " + deque.pollLast()); // pollLast: 30
    deque.printDeque(); // Deque: 10
    System.out.println("peekFirst: " + deque.peekFirst()); // peekFirst: 10
    System.out.println("peekLast: " + deque.peekLast()); // peekLast: 10
}
```

862. Shortest Subarray with Sum at Least K

Hard

Given an integer array `nums` and an integer `k`, return the length of the shortest non-empty subarray of `nums` with a sum of at least `k`. If there is no such subarray, return `-1`.

A subarray is a contiguous part of an array.

Example 1:

```
Input: nums = [1], k = 1
Output: 1
```

Example 2:

```
Input: nums = [1,2], k = 4
Output: -1
```

Example 3:

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

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^5 \leq \text{nums}[i] \leq 10^5$
- $1 \leq k \leq 10^9$

Monotonic Queue  Auto

```
1 import java.util.Deque;
2 import java.util.LinkedList;
3
4 public class Solution {
5
6     public int shortestSubarray(int[] nums, int K) {
7         int n = nums.length;
8         long[] prefixSum = new long[n + 1];
9         for (int i = 0; i < n; i++) {
10             prefixSum[i + 1] = prefixSum[i] + nums[i];
11         }
12
13         Deque<Integer> deque = new LinkedList<>();
14         int minLength = Integer.MAX_VALUE;
15
16         for (int i = 0; i <= n; i++) {
17             while (!deque.isEmpty() && prefixSum[i] - prefixSum[deque.peekFirst()] >= K) {
18                 minLength = Math.min(minLength, i - deque.pollFirst());
19             }
20             while (!deque.isEmpty() && prefixSum[i] <= prefixSum[deque.peekLast()]) {
21                 deque.pollLast();
22             }
23
24             deque.addLast(i);
25         }
26
27         return minLength == Integer.MAX_VALUE ? -1 : minLength;
28     }
29
30     public static void main(String[] args) {
31         Solution solution = new Solution();
32         int[] nums = {2, -1, 2};
33         int K = 3;
34         System.out.println(solution.shortestSubarray(nums, K));
35     }
36 }
37
```

The time complexity of this solution is O(n)

The space complexity: O(1)

239. Sliding Window Maximum

You are given an array of integers `nums`, there is a sliding window of size `k` which is moving from the very left of the array to the very right. You can only see the `k` numbers in the window. Each time the sliding window moves right by one position.

Return the max sliding window.

Example 1:

```
Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
Output: [3,3,5,5,6,7]
Explanation:
Window position          Max
-----  
[1 3 -1] -3 5 3 6 7      <strong>3</strong>
1 [3 -1 -3] 5 3 6 7      <strong>3</strong>
1 3 [-1 -3 5] 3 6 7      <strong>5</strong>
1 3 -1 [-3 5 3] 6 7      <strong>5</strong>
1 3 -1 -3 [5 3 6] 7      <strong>6</strong>
1 3 -1 -3 5 [3 6 7]      <strong>7</strong>
```

Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

Constraints:

- `1 <= nums.length <= 105`
- `-104 <= nums[i] <= 104`
- `1 <= k <= nums.length`

```
import java.util.*;

class Solution {
    public int[] maxSlidingWindow(int[] nums, int k) {
        int n = nums.length;
        if (n * k == 0) return new int[0];
        if (k == 1) return nums;
        Deque<Integer> deque = new LinkedList<>();
        int[] result = new int[n - k + 1];
        for (int i = 0; i < n; i++) {
            while (!deque.isEmpty() && deque.peek() < i - k + 1) {
                deque.poll();
            }
            while (!deque.isEmpty() && nums[deque.peekLast()] <= nums[i]) {
                deque.pollLast();
            }
            deque.offer(i);
            if (i - k + 1 >= 0) {
                result[i - k + 1] = nums[deque.peek()];
            }
        }
        return result;
    }
}
```

The time complexity of this solution is O(n)

The space complexity: O(1)

918. Maximum Sum Circular Subarray

Medium

Given a circular integer array `nums` of length `n`, return the maximum possible sum of a non-empty subarray of `nums`.

A circular array means the end of the array connects to the beginning of the array. Formally, the next element of `nums[i]` is `nums[(i + 1) % n]` and the previous element of `nums[i]` is `nums[(i - 1 + n) % n]`.

A subarray may only include each element of the fixed buffer `nums` at most once. Formally, for a subarray `nums[i], nums[i + 1], ..., nums[j]`, there does not exist $i \leq k_1, k_2 \leq j$ with $k_1 \% n == k_2 \% n$.

Example 1:

Input: `nums = [1,-2,3,-2]`
Output: 3
Explanation: Subarray [3] has maximum sum 3.

Example 2:

Input: `nums = [5,-3,5]`
Output: 10
Explanation: Subarray [5,5] has maximum sum 5 + 5 = 10.

Example 3:

Input: `nums = [-3,-2,-3]`
Output: -2
Explanation: Subarray [-2] has maximum sum -2.

Constraints:

- $n == \text{nums.length}$
- $1 \leq n \leq 3 * 10^4$
- $-3 * 10^4 \leq \text{nums}[i] \leq 3 * 10^4$

```
import java.util.LinkedList;

public class Solution {
    public int maxSubarraySumCircular(int[] nums) {
        int n = nums.length;
        int[] extendedNums = new int[2 * n];
        System.arraycopy(nums, 0, extendedNums, 0, n);
        System.arraycopy(nums, 0, extendedNums, n, n);
        long[] prefixSum = new long[2 * n + 1];
        for (int i = 0; i < 2 * n; i++) {
            prefixSum[i + 1] = prefixSum[i] + extendedNums[i];
        }

        Deque<Integer> deque = new LinkedList<>();
        deque.add(0);
        int maxSum = Integer.MIN_VALUE;

        for (int i = 1; i <= 2 * n; i++) {
            if (deque.peekFirst() < i - n) {
                deque.pollFirst();
            }
            maxSum = Math.max(maxSum, (int)(prefixSum[i] - prefixSum[deque.peekFirst()]));

            // Maintain the deque's monotonic property
            while (!deque.isEmpty() && prefixSum[i] <= prefixSum[deque.peekLast()]) {
                deque.pollLast();
            }

            deque.addLast(i);
        }

        return maxSum;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {1, -2, 3, -2};
        System.out.println(solution.maxSubarraySumCircular(nums)); // Output: 3
    }
}
```

The time complexity of this solution is $O(n)$

The space complexity: $O(1)$

Monotonic Increasing Queue

1696. Jump Game VI

Medium

You are given a 0-indexed integer array `nums` and an integer `k`.

You are initially standing at index `0`. In one move, you can jump at most `k` steps forward without going outside the boundaries of the array. That is, you can jump from index `i` to any index in the range `[i + 1, min(n - 1, i + k)]` inclusive.

You want to reach the last index of the array (index `n - 1`). Your score is the sum of all `nums[j]` for each index `j` you visited in the array.

Return the maximum score you can get.

Example 1:

Input: `nums = [1, -1, -2, 4, -7, 3]`, `k = 2`
Output: 7
Explanation: You can choose your jumps forming the subsequence `[1, -1, 4, 3]` (underlined above).
The sum is 7.

Example 2:

Input: `nums = [10, -5, -2, 4, 0, 3]`, `k = 3`
Output: 17
Explanation: You can choose your jumps forming the subsequence `[10, 4, 3]` (underlined above). The sum is 17.

Example 3:

Input: `nums = [1, -5, -20, 4, -1, 3, -6, -3]`, `k = 2`
Output: 0

Constraints:

- $1 \leq n \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

```
import java.util.Deque;
import java.util.LinkedList;

public class Solution {
    public int maxResult(int[] nums, int k) {
        int n = nums.length;
        Deque<Integer> deque = new LinkedList<>();
        int[] dp = new int[n];
        dp[0] = nums[0];
        deque.offerLast(0);

        for (int i = 1; i < n; i++) {
            while (!deque.isEmpty() && deque.peekFirst() < i - k) {
                deque.pollFirst();
            }

            dp[i] = nums[i] + dp[deque.peekFirst()];

            while (!deque.isEmpty() && dp[i] >= dp[deque.peekLast()]) {
                deque.pollLast();
            }

            deque.offerLast(i);
        }

        return dp[n - 1];
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {1, -1, -2, 4, -7, 3};
        int k = 2;
        System.out.println(solution.maxResult(nums, k)); // Output: 7
    }
}
```

The time complexity of this solution is $O(n)$

The space complexity: $O(1)$

Priority Queue

A Priority Queue is a special type of queue in which elements are served based on priority rather than just the order in which they arrive (like a normal queue).

A heap (priority queue) is a special tree-based data structure that satisfies the heap property:

- **Max-Heap:** The parent node is always greater than or equal to its child nodes.
- **Min-Heap:** The parent node is always smaller than or equal to its child nodes.

By default, Java's PriorityQueue is a min-heap, i.e., the smallest element is at the head.

Operation	Method	Time Complexity	Description
Insert / Push	<code>offer(E e)</code> / <code>add(E)</code>	$O(\log N)$	Inserts the element
Get Min / Max	<code>peek()</code>	$O(1)$	Returns the head without removing it
Remove Min / Max	<code>poll()</code>	$O(\log N)$	Removes and returns the head
Size	<code>size()</code>	$O(1)$	Number of elements in the queue
Check Empty	<code>isEmpty()</code>	$O(1)$	Checks if queue is empty

```
PriorityQueue<Integer> pq = new PriorityQueue<>();
pq.offer(10);
pq.offer(2);
pq.offer(5);

System.out.println(pq.peek()); // Output: 2 (min element)
```

To implement a **max-heap**, pass a custom comparator:

```
java                                     ⌂ Copy ⌂ Edit
PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);
```

Or with `Comparator.reverseOrder()`:

```
java                                     ⌂ Copy ⌂ Edit
PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Comparator.reverseOrder());
```

Use Cases

- Dijkstra's Algorithm (shortest path)
- A* Search
- Top K elements
- Median in a stream
- Huffman Coding
- Merging K sorted lists
- CPU Task Scheduling

Operations in a Heap:

Insertion ($O(\log n)$)

- Add the new element at the end (last index).
- Restore the heap property by bubbling up (heapify up).

Deletion ($O(\log n)$)

- Remove the root (max in a max-heap, min in a min-heap).
- Replace it with the last element.

Restore heap property by bubbling down (heapify down).

- Peek ($O(1)$)
- Returns the root element (max/min) without removal.

Heapify ($O(n)$)

- Convert an unsorted array into a heap.

```
import java.util.PriorityQueue;

class Student implements Comparable<Student> {
    String name;
    int marks;

    public Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }

    // Sort by marks in descending order (Max-Heap behavior)
    @Override
    public int compareTo(Student other) {
        return other.marks - this.marks;
    }

    public String toString() {
        return name + ": " + marks;
    }
}

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Student> pq = new PriorityQueue<>();

        pq.offer(new Student("Alice", 85));
        pq.offer(new Student("Bob", 95));
        pq.offer(new Student("Charlie", 80));

        while (!pq.isEmpty()) {
            System.out.println(pq.poll());
        }
    }
}
```

```

package com.sai.designPatterns.shortestPath;

import java.util.PriorityQueue;
import java.util.Comparator;

class Student {
    String name;
    int marks;

    public Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }

    public String toString() {
        return name + ": " + marks;
    }
}

public class Main {
    public static void main(String[] args) {
        PriorityQueue<Student> minHeap = new PriorityQueue<>(Comparator.comparingInt(s -> s.marks));

        minHeap.offer(new Student(name: "Alice", marks: 85));
        minHeap.offer(new Student(name: "Bob", marks: 95));
        minHeap.offer(new Student(name: "Charlie", marks: 80));

        while (!minHeap.isEmpty()) {
            System.out.println(minHeap.poll());
        }
    }
}

```

Recommended: Use static inner class or static comparator field

java

```

class Student {
    String name;
    int marks;

    public Student(String name, int marks) {
        this.name = name;
        this.marks = marks;
    }

    public static class MarksComparator implements Comparator<Student> {
        @Override
        public int compare(Student s1, Student s2) {
            return Integer.compare(s2.marks, s1.marks); // Descending
        }
    }

    @Override
    public String toString() {
        return name + ": " + marks;
    }
}

```

Usage:

java

```

PriorityQueue<Student> pq = new PriorityQueue<>(new Student.MarksComparator());

```

1. Dijkstra's Algorithm (shortest path)

It finds the shortest path from a single source node to all other nodes in a weighted graph (with non-negative edge weights).

```
public class Dijkstra {  
    public static void dijkstra(List<List<Edge>> graph, int source, int[] distances) {  
        PriorityQueue<Edge> pq = new PriorityQueue<Edge>();  
        pq.add(new Edge(source, source, weight: 0));  
  
        while (!pq.isEmpty()) {  
            Edge current = pq.poll();  
            int u = current.getDest();  
            int dist = current.getWeight();  
  
            for (Edge edge : graph.get(u)) {  
                int v = edge.getDest();  
                int weight = edge.getWeight();  
                if (dist + weight < distances[v]) {  
                    distances[v] = dist + weight;  
                    pq.add(new Edge(v, v, distances[v]));  
                }  
            }  
        }  
    }  
  
    public static void main(String[] args) {  
        List<Edge> edges = new ArrayList<Edge>();  
        edges.add(new Edge(src: 0, dest: 1, weight: 4));  
        edges.add(new Edge(src: 0, dest: 7, weight: 8));  
        edges.add(new Edge(src: 1, dest: 2, weight: 8));  
        edges.add(new Edge(src: 1, dest: 7, weight: 11));  
        edges.add(new Edge(src: 2, dest: 3, weight: 7));  
        edges.add(new Edge(src: 2, dest: 5, weight: 4));  
        edges.add(new Edge(src: 2, dest: 8, weight: 2));  
        edges.add(new Edge(src: 3, dest: 4, weight: 9));  
        edges.add(new Edge(src: 3, dest: 5, weight: 14));  
        edges.add(new Edge(src: 4, dest: 5, weight: 10));  
        edges.add(new Edge(src: 5, dest: 6, weight: 2));  
        edges.add(new Edge(src: 6, dest: 7, weight: 1));  
        edges.add(new Edge(src: 6, dest: 8, weight: 6));  
        edges.add(new Edge(src: 7, dest: 8, weight: 7));  
        int n = 9; // Number of vertices  
        List<List<Edge>> graph = new ArrayList<List<Edge>>();  
        for (int i = 0; i < n; i++) {  
            graph.add(new ArrayList<Edge>());  
        }  
        // Populate the graph with edges  
        for (Edge edge : edges) {  
            graph.get(edge.getSrc()).add(edge);  
        }  
        // Array to store distances from source vertex  
        int[] distances = new int[n];  
        Arrays.fill(distances, Integer.MAX_VALUE);  
        distances[0] = 0; // Distance from source to source is 0  
        // Run Dijkstra's algorithm  
        dijkstra(graph, source: 0, distances);  
        // Print shortest distances from source vertex  
        System.out.println("Shortest distances from source vertex:");  
        for (int i = 0; i < n; i++) {  
            System.out.println("Vertex " + i + ": " + distances[i]);  
        }  
    }  
}
```

Time Complexity	$O((V + E) \log V)$
Space Complexity	$O(V + E)$

```

@Data
public class Edge implements Comparable<Edge> {
    int src;
    int dest;
    int weight;

    public Edge(int src, int dest, int weight) {
        this.src = src;
        this.dest = dest;
        this.weight = weight;
    }

    @Override
    public int compareTo(Edge other) {
        return this.weight - other.weight; // descending
    }
}

```

2. A* Algorithm

Find the shortest path from a start node to a target node, using both:

- **Actual cost from the start** \rightarrow current node ($g(n)$)
- **Heuristic estimate from current node** \rightarrow target ($h(n)$)

It chooses the path that minimizes $f(n) = g(n) + h(n)$.

12 Components

$g(n)$ = Actual cost from the start to node n

$h(n)$ = Estimated cost from n to goal (heuristic)

$f(n) = g(n) + h(n)$ \rightarrow total estimated cost through n

The heuristic $h(n)$ should be:

- Admissible (never overestimates)
- Common choices: Manhattan distance, Euclidean, etc.

Pseudocode

```
text Copy Edit
1. Add start node to open set with  $f(\text{start}) = g(\text{start}) + h(\text{start})$ 
2. While open set is not empty:
   a. Pick node with lowest  $f(n)$ 
   b. If node is goal, reconstruct path
   c. Else, for each neighbor:
      - Calculate tentative  $g(n)$ 
      - If better path, update  $g(n)$ ,  $f(n)$ , and parent
3. If goal not found, return failure
```

```
package com.sai.designPatterns.shortestPath;

import java.util.*;

class Node {
    int row, col;
    int costFromStart; // g(n)
    int estimatedToGoal; // h(n)
    int totalEstimatedCost; // f(n) = g + h
    Node parent;

    Node(int row, int col, int costFromStart, int estimatedToGoal, Node parent) {
        this.row = row;
        this.col = col;
        this.costFromStart = costFromStart;
        this.estimatedToGoal = estimatedToGoal;
        this.totalEstimatedCost = costFromStart + estimatedToGoal;
        this.parent = parent;
    }
}

public class AStarPathfinder {
    // Directions: right, down, left, up
    static final int[][] DIRECTIONS = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    public static List<int[]> findPath(int[][] grid, int[] start, int[] goal) {
        int rows = grid.length, cols = grid[0].length;
        boolean[][] visited = new boolean[rows][cols];

        PriorityQueue<Node> openSet = new PriorityQueue<>(Comparator.comparingInt(node -> node.totalEstimatedCost));
        Node startNode = new Node(start[0], start[1], costFromStart: 0, calculateHeuristic(start, goal), parent: null);
        openSet.offer(startNode);

        while (!openSet.isEmpty()) {
            Node current = openSet.poll();

            if (current.row == goal[0] && current.col == goal[1]) {
                return reconstructPath(current);
            }

            if (visited[current.row][current.col]) continue;
            visited[current.row][current.col] = true;

            for (int[] direction : DIRECTIONS) {
                int newRow = current.row + direction[0];
                int newCol = current.col + direction[1];

                if (isValidCell(newRow, newCol, grid, visited)) {
                    int newCostFromStart = current.costFromStart + 1;
                    int estimatedToGoal = calculateHeuristic(new int[]{newRow, newCol}, goal);
                    Node neighbor = new Node(newRow, newCol, newCostFromStart, estimatedToGoal, current);
                    openSet.offer(neighbor);
                }
            }
        }

        return Collections.emptyList(); // No path found
    }

    private static boolean isValidCell(int row, int col, int[][] grid, boolean[][] visited) {
        return row >= 0 && col >= 0 && row < grid.length && col < grid[0].length
            && grid[row][col] == 0 && !visited[row][col];
    }
}
```

```

private static int calculateHeuristic(int[] from, int[] to) {
    // Manhattan distance heuristic
    return Math.abs(from[0] - to[0]) + Math.abs(from[1] - to[1]);
}

private static List<int[]> reconstructPath(Node node) {
    List<int[]> path = new ArrayList<>();
    while (node != null) {
        path.add(new int[]{node.row, node.col});
        node = node.parent;
    }
    Collections.reverse(path);
    return path;
}

public static void main(String[] args) {
    int[][] grid = {
        {0, 0, 0, 0},
        {1, 1, 0, 1},
        {0, 0, 0, 0},
        {0, 1, 1, 0},
        {0, 0, 0, 0}
    };

    int[] start = {0, 0};
    int[] goal = {4, 3};

    List<int[]> shortestPath = findPath(grid, start, goal);
    for (int[] cell : shortestPath) {
        System.out.println(Arrays.toString(cell));
    }
}
}

```

Time and Space Complexity:

Time O(E) or O(V log V) (with priority queue)

Space O(V) (for visited, queue, path tracking)

✓ Ideal Use Cases:

- Grid maps
- Game AI
- Pathfinding with weighted heuristics
- Real-world maps (Google Maps-style shortest path)

3. Top K Elements

Top K Largest Elements (Using Min-Heap)

Top K Frequent Elements (Using HashMap + Min-Heap)

```
import java.util.PriorityQueue;

public class TopKLargestElements {
    public static int[] findTopK(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int num : nums) {
            minHeap.offer(num);
            if (minHeap.size() > k) {
                minHeap.poll(); // Remove smallest
            }
        }

        // Result array
        int[] result = new int[k];
        int index = 0;
        for (int num : minHeap) {
            result[index++] = num;
        }
        return result;
    }

    public static void main(String[] args) {
        int[] nums = {3, 2, 1, 5, 6, 4};
        int k = 3;
        int[] topK = findTopK(nums, k);

        System.out.print("Top " + k + " elements: ");
        for (int num : topK) {
            System.out.print(num + " ");
        }
    }
}

import java.util.*;

public class TopKFrequentElements {
    public static List<Integer> topKFrequent(int[] nums, int k) {
        Map<Integer, Integer> frequencyMap = new HashMap<>();

        for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        }

        // Min-heap based on frequency
        PriorityQueue<Map.Entry<Integer, Integer>> minHeap =
            new PriorityQueue<>(Comparator.comparingInt(Map.Entry::getValue));

        for (Map.Entry<Integer, Integer> entry : frequencyMap.entrySet()) {
            minHeap.offer(entry);
            if (minHeap.size() > k) {
                minHeap.poll(); // Remove least frequent
            }
        }

        List<Integer> result = new ArrayList<>();
        for (Map.Entry<Integer, Integer> entry : minHeap) {
            result.add(entry.getKey());
        }
        return result;
    }

    public static void main(String[] args) {
        int[] nums = {1, 1, 1, 2, 2, 3};
        int k = 2;
        List<Integer> topK = topKFrequent(nums, k);

        System.out.println("Top " + k + " frequent elements: " + topK);
    }
}
```

Complexity

Operation	Time Complexity	Notes
Insertion into heap	$O(\log k)$ per element	Maintains size $\leq k$
Overall time	$O(n \log k)$	Efficient for large n and small k
For frequency-based	$O(n \log k + n)$	n for map creation, $\log k$ for heap

```
public class Solution {
    public static int[] findTopK(int[] nums, int k) {
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int num : nums) {
            minHeap.offer(num);
            if (minHeap.size() > k) {
                minHeap.poll(); // Remove smallest
            }
            System.out.println(minHeap);
        }
    }
}
```

```
Connected to the target VM, address: '127.0.0.1:54283', transport: F7
Step Into F7
> Task :Solution.main()
[3]
[2, 3]
[1, 3, 2]
[2, 3, 5]
[3, 6, 5]
[4, 6, 5]
Top 3 elements: 4 6 5
Deprecated Gradle features were used in this build, making it
```

4. Median in a Stream

Approach: Two Heaps

- Max Heap (`leftHalf`): stores the smaller half of the numbers.
- Min Heap (`rightHalf`): stores the larger half of the numbers.

We balance both heaps so that:

- The size difference is at most 1.
- If the total number of elements is odd, the extra element goes to the max heap.

```

public class Solution {
    private PriorityQueue<Integer> maxHeap; // left half (smaller numbers)
    private PriorityQueue<Integer> minHeap; // right half (larger numbers)

    public Solution() {
        // Max-heap for left half
        maxHeap = new PriorityQueue<>(Collections.reverseOrder());
        // Min-heap for right half
        minHeap = new PriorityQueue<>();
    }

    public void addNumber(int num) {
        // Step 1: Add to maxHeap
        maxHeap.offer(num);

        // Step 2: Balance - move largest from maxHeap to minHeap
        minHeap.offer(maxHeap.poll());

        // Step 3: Maintain size property
        if (maxHeap.size() < minHeap.size()) {
            maxHeap.offer(minHeap.poll());
        }
    }

    public double getMedian() {
        if (maxHeap.size() == minHeap.size()) {
            // Even count: average of middle two
            return (maxHeap.peek() + minHeap.peek()) / 2.0;
        } else {
            // Odd count: middle is top of maxHeap
            return maxHeap.peek();
        }
    }
}

```

```

public static void main(String[] args) {
    MedianFinder mf = new MedianFinder();
    int[] stream = {5, 15, 1, 3};

    for (int num : stream) {
        mf.addNum(num);
        System.out.println("Current Median: " + mf.findMedian());
    }
}

```

```

Current Median: 5.0
Current Median: 10.0
Current Median: 5.0
Current Median: 4.0

```

5. Huffman Coding

Huffman Coding is a greedy algorithm used for lossless data compression. It assigns variable-length binary codes to input characters, based on their frequencies – more frequent characters get shorter codes, while less frequent characters get longer codes.

✓ Steps in Huffman Coding:

1. Count frequency of each character.
2. Insert all characters into a **min-heap** (priority queue) by frequency.
3. While heap has more than one node:
 - Remove two nodes with the lowest frequency.
 - Create a new internal node with their sum as frequency.
 - Insert the new node back into the heap.
4. Generate codes by traversing the tree:
 - Left = **0**, Right = **1**.

```

class HuffmanNode {
    char character;
    int frequency;
    HuffmanNode left, right;

    HuffmanNode(char ch, int freq) {
        this.character = ch;
        this.frequency = freq;
    }

    HuffmanNode(int freq, HuffmanNode left, HuffmanNode right) {
        this.character = '\0'; // internal node
        this.frequency = freq;
        this.left = left;
        this.right = right;
    }
}

```

```

// Comparator for min-heap (priority queue)
class NodeComparator implements Comparator<HuffmanNode> {
    public int compare(HuffmanNode a, HuffmanNode b) {
        return a.frequency - b.frequency;
    }
}

public class Solution {
    private static void buildCodes(HuffmanNode root, String code, Map<Character, String> huffmanCodeMap) {
        if (root == null) return;

        // Leaf node
        if (root.left == null && root.right == null) {
            huffmanCodeMap.put(root.character, code);
            return;
        }

        buildCodes(root.left, code + "0", huffmanCodeMap);
        buildCodes(root.right, code + "1", huffmanCodeMap);
    }

    public static void main(String[] args) {
        String text = "abccdeeee";

        // Step 1: Count frequency
        Map<Character, Integer> freqMap = new HashMap<>();
        for (char ch : text.toCharArray()) {
            freqMap.put(ch, freqMap.getOrDefault(ch, defaultValue: 0) + 1);
        }

        // Step 2: Create min-heap
        PriorityQueue<HuffmanNode> pq = new PriorityQueue<>(new NodeComparator());

        for (Map.Entry<Character, Integer> entry : freqMap.entrySet()) {
            pq.offer(new HuffmanNode(entry.getKey(), entry.getValue()));
        }

        // Step 3: Build Huffman Tree
        while (pq.size() > 1) {
            HuffmanNode left = pq.poll();
            HuffmanNode right = pq.poll();

            HuffmanNode merged = new HuffmanNode(freq: left.frequency + right.frequency, left, right);
            pq.offer(merged);
        }

        // Root of the Huffman tree
        HuffmanNode root = pq.poll();

        // Step 4: Generate codes
        Map<Character, String> huffmanCodeMap = new HashMap<>();
        buildCodes(root, code: "", huffmanCodeMap);
    }
}

```

```

        System.out.println("Huffman Codes:");
        for (Map.Entry<Character, String> entry : huffmanCodeMap.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }

        // Encoding
        StringBuilder encoded = new StringBuilder();
        for (char ch : text.toCharArray()) {
            encoded.append(huffmanCodeMap.get(ch));
        }

        System.out.println("\nOriginal: " + text);
        System.out.println("Encoded: " + encoded.toString());
    }
}

```

Huffman Codes:

a: 110
 b: 1110
 c: 10
 d: 1111
 e: 0

Original: aabccdeeee

Encoded: 1101101110101011110000

6. Merge K Sorted Lists

Given k sorted linked lists, merge them into one sorted linked list.

Approach: Min Heap (Priority Queue)

- Use a **min-heap** to keep track of the smallest head node among all the k lists.
- Repeatedly extract the smallest node and add its next node to the heap.
- Efficiently maintains sorted order.

Time Complexity:

- $O(N \log k)$
 Where N is the total number of nodes and k is the number of linked lists.

```

import java.util.*;

class ListNode {
    int val;
    ListNode next;
    ListNode() {}
    ListNode(int val) { this.val = val; }
    ListNode(int val, ListNode next) { this.val = val; this.next = next; }
}

class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        PriorityQueue<ListNode> pq = new PriorityQueue<>(Comparator.comparingInt(a -> a.val));

        // Add head of each list to the min-heap
        for (ListNode list : lists) {
            if (list != null) pq.offer(list);
        }

        ListNode dummy = new ListNode(0);
        ListNode tail = dummy;

        while (!pq.isEmpty()) {
            ListNode min = pq.poll();
            tail.next = min;
            tail = tail.next;

            if (min.next != null) {
                pq.offer(min.next);
            }
        }

        return dummy.next;
    }
}

```

Input:

```

java
lists = [
    1 -> 4 -> 5,
    1 -> 3 -> 4,
    2 -> 6
]

```

Copy Edit

Heap flow:

- Insert 1, 1, 2 → min = 1
- Next: 1 (list 1) → insert 4
- Next: 1 (list 2) → insert 3
- Next: 2 → insert 6
- And so on...

Final Output:

```

java
1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 6

```

Copy Edit

7. CPU Task Scheduling

Given a list of CPU tasks represented by characters and a cooldown period n, schedule tasks so that the same task appears at least n units apart, minimizing total time.

✓ Key Idea:

- Always pick the most frequent task available.
- Use a **max heap** (priority queue with custom comparator) to simulate choosing the most needed task first.
- Use a **queue** to keep track of cooldown.

✓ Time Complexity:

- $O(N \log 26) \approx O(N)$
Where N = total number of tasks. Max 26 different tasks (A-Z).

```
public class Solution {  
    public int leastInterval(char[] tasks, int n) {  
        // Frequency map for each task  
        int[] frequencies = new int[26];  
        for (char task : tasks) {  
            frequencies[task - 'A']++;  
        }  
  
        // Max Heap: tasks with highest frequency first  
        PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());  
        for (int freq : frequencies) {  
            if (freq > 0) maxHeap.offer(freq);  
        }  
  
        // Queue to hold tasks during cooldown: (count, availableTime)  
        Queue<int[]> cooldownQueue = new LinkedList<>();  
  
        int time = 0;  
  
        while (!maxHeap.isEmpty() || !cooldownQueue.isEmpty()) {  
            time++;  
            if (!maxHeap.isEmpty()) {  
                int count = maxHeap.poll() - 1;  
                if (count > 0) {  
                    cooldownQueue.offer(new int[]{count, time + n});  
                }  
            }  
            // Restore task from cooldown if it's ready  
            if (!cooldownQueue.isEmpty() && cooldownQueue.peek()[1] == time) {  
                maxHeap.offer(cooldownQueue.poll()[0]);  
            }  
        }  
  
        return time;  
    }  
  
    public static void main(String[] args) {  
        Solution scheduler = new Solution();  
        char[] tasks = {'A', 'A', 'A', 'B', 'B', 'B'};  
        int cooldown = 2;  
        System.out.println(scheduler.leastInterval(tasks, cooldown)); // Output: 8  
    }  
}
```

8. 215. Kth Largest Element in an Array

Medium

Given an integer array `nums` and an integer `k`, return *the kth largest element in the array*.

Note that it is the `kth` largest element in the sorted order, not the `kth` distinct element.

Can you solve it without sorting?

Example 1:

Input: `nums = [3,2,1,5,6,4]`, `k = 2`
Output: 5

Example 2:

Input: `nums = [3,2,3,1,2,4,5,5,6]`, `k = 4`
Output: 4

Constraints:

- $1 \leq k \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

Heap (Priority Queue) Auto

```
1  class Solution {
2      public int findKthLargest(int[] nums, int k) {
3          Queue<Integer> minHeap = new PriorityQueue<>();
4
5          for (final int num : nums) {
6              minHeap.offer(num);
7              while (minHeap.size() > k)
8                  minHeap.poll();
9          }
10
11         return minHeap.peek();
12     }
13 }
```

Time and Space Complexity:

Time $O(E)$ or $O(V \log V)$ (with priority queue)

Space $O(V)$ (for visited, queue, path tracking)

9. 378. Kth Smallest Element in a Sorted Matrix

378. Kth Smallest Element in a Sorted Matrix

Medium

Given an $n \times n$ matrix where each of the rows and columns is sorted in ascending order, return the k^{th} smallest element in the matrix.

Note that it is the k^{th} smallest element in the sorted order, not the k^{th} distinct element.

You must find a solution with a memory complexity better than $O(n^2)$.

Example 1:

Input: matrix = [[1,5,9],[10,11,13],[12,13,15]], k = 8
Output: 13
Explanation: The elements in the matrix are [1,5,9,10,11,12,13,13,15], and the 8th smallest number is 13

Example 2:

Input: matrix = [[-5]], k = 1
Output: -5

Constraints:

- $n == \text{matrix.length} == \text{matrix}[i].length$
- $1 \leq n \leq 300$
- $-10^9 \leq \text{matrix}[i][j] \leq 10^9$
- All the rows and columns of `matrix` are guaranteed to be sorted in non-decreasing order.
- $1 \leq k \leq n^2$

Follow up:

- Could you solve the problem with a constant memory (i.e., $O(1)$ memory complexity)?
- Could you solve the problem in $O(n)$ time complexity? The solution may be too advanced for an interview but you may find reading [this paper](#) fun.

```
import java.util.PriorityQueue;

class Solution {
    public int kthSmallest(int[][] matrix, int k) {
        int n = matrix.length;
        PriorityQueue<int[]> minHeap = new PriorityQueue<>((a, b) -> a[0] - b[0]);

        for (int j = 0; j < n; j++) {
            minHeap.offer(new int[]{matrix[0][j], 0, j}); // value, row, col
        }

        for (int i = 0; i < k - 1; i++) {
            int[] current = minHeap.poll();
            int row = current[1];
            int col = current[2];

            if (row < n - 1) {
                minHeap.offer(new int[]{matrix[row + 1][col], row + 1, col});
            }
        }

        return minHeap.poll()[0];
    }
}
```

Time and Space Complexity:

Time $O(k \log n)$

Space $O(n)$

10. 692. Top K Frequent Words

Medium

Given an array of strings `words` and an integer `k`, return the `k` most frequent strings.

Return the answer sorted by the frequency from highest to lowest. Sort the words with the same frequency by their lexicographical order.

Example 1:

Input: `words` = ["i", "love", "leetcode", "i", "love", "coding"], `k` = 2
Output: ["i", "love"]
Explanation: "i" and "love" are the two most frequent words.
Note that "i" comes before "love" due to a lower alphabetical order.

Example 2:

Input: `words` = ["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], `k` = 4
Output: ["the", "is", "sunny", "day"]
Explanation: "the", "is", "sunny" and "day" are the four most frequent words, with the number of occurrence being 4, 3, 2 and 1 respectively.

Constraints:

- `1 <= words.length <= 500`
- `1 <= words[i].length <= 10`
- `words[i]` consists of lowercase English letters.
- `k` is in the range [1, The number of unique `words[i]`]

Follow-up: Could you solve it in $O(n \log(k))$ time and $O(n)$ extra space?

```
import java.util.*;

class Solution {
    public List<String> topKFrequent(String[] words, int k) {
        Map<String, Integer> frequencyMap = new HashMap<>();
        for (String word : words) {
            frequencyMap.put(word, frequencyMap.getOrDefault(word, 0) + 1);
        }

        PriorityQueue<String> minHeap = new PriorityQueue<>(
            (a, b) -> frequencyMap.get(a).equals(frequencyMap.get(b))
                ? b.compareTo(a)
                : frequencyMap.get(a) - frequencyMap.get(b));
        
        for (String word : frequencyMap.keySet()) {
            minHeap.offer(word);
            if (minHeap.size() > k) {
                minHeap.poll();
            }
        }

        List<String> result = new ArrayList<>();
        while (!minHeap.isEmpty()) {
            result.add(minHeap.poll());
        }
        Collections.reverse(result);
        return result;
    }
}
```

Time and Space Complexity:

Time $O(k \log n)$

Space $O(n)$

12. 786. K-th Smallest Prime Fraction

Medium

You are given a sorted integer array `arr` containing 1 and prime numbers, where all the integers of `arr` are unique. You are also given an integer `k`.

For every `i` and `j` where `0 <= i < j < arr.length`, we consider the fraction `arr[i] / arr[j]`.

Return the k^{th} smallest fraction considered. Return your answer as an array of integers of size 2, where `answer[0] == arr[i]` and `answer[1] == arr[j]`.

Example 1:

Input: `arr = [1,2,3,5], k = 3`

Output: `[2,5]`

Explanation: The fractions to be considered in sorted order are:

$1/5, 1/3, 2/5, 1/2, 3/5$, and $2/3$.

The third fraction is $2/5$.

Example 2:

Input: `arr = [1,7], k = 1`

Output: `[1,7]`

Constraints:

- $2 \leq \text{arr.length} \leq 1000$
- $1 \leq \text{arr}[i] \leq 3 \times 10^4$
- $\text{arr}[0] == 1$
- $\text{arr}[i]$ is a prime number for $i > 0$.
- All the numbers of `arr` are unique and sorted in strictly increasing order.
- $1 \leq k \leq \text{arr.length} * (\text{arr.length} - 1) / 2$

Follow up: Can you solve the problem with better than $O(n^2)$ complexity?

```
import java.util.PriorityQueue;

public class Solution {
    public int[] kthSmallestPrimeFraction(int[] arr, int k) {
        int n = arr.length;
        PriorityQueue<Fraction> minHeap = new PriorityQueue<>((a, b) -> Double.compare(a.value, b.value));

        for (int i = 0; i < n - 1; i++) {
            minHeap.offer(new Fraction(i, n - 1, (double) arr[i] / arr[n - 1]));
        }

        Fraction fraction = null;
        for (int i = 0; i < k; i++) {
            fraction = minHeap.poll();
            int numeratorIndex = fraction.numeratorIndex;
            int denominatorIndex = fraction.denominatorIndex;
            if (denominatorIndex - 1 > numeratorIndex) {
                minHeap.offer(new Fraction(numeratorIndex, denominatorIndex - 1,
                    (double) arr[numeratorIndex] / arr[denominatorIndex - 1]));
            }
        }

        return new int[] { arr[fraction.numeratorIndex], arr[fraction.denominatorIndex] };
    }

    private static class Fraction {
        int numeratorIndex;
        int denominatorIndex;
        double value;

        Fraction(int numeratorIndex, int denominatorIndex, double value) {
            this.numeratorIndex = numeratorIndex;
            this.denominatorIndex = denominatorIndex;
            this.value = value;
        }
    }
}
```

Time and Space Complexity:

Time $O(k \log n)$

Space $O(n)$

1337. The K Weakest Rows in a Matrix

You are given an $m \times n$ binary matrix mat of 1's (representing soldiers) and 0's (representing civilians). The soldiers are positioned in front of the civilians. That is, all the 1's will appear to the left of all the 0's in each row.

A row i is weaker than a row j if one of the following is true:

- The number of soldiers in row i is less than the number of soldiers in row j .
- Both rows have the same number of soldiers and $i < j$.

Return the indices of the k weakest rows in the matrix ordered from weakest to strongest.

Example 1:

```
Input: mat =  
[[1,1,0,0,0],  
[1,1,1,1,0],  
[1,0,0,0,0],  
[1,1,0,0,0],  
[1,1,1,1,1]], k = 3  
Output: [2,0,3]  
Explanation:  
The number of soldiers in each row is:  
- Row 0: 2  
- Row 1: 4  
- Row 2: 1  
- Row 3: 2  
- Row 4: 5  
The rows ordered from weakest to strongest are [2,0,3,1,4].
```

Example 2:

```
Input: mat =  
[[1,0,0,0],  
[1,1,1,1],  
[1,0,0,0],  
[1,0,0,0]], k = 2  
Output: [0,2]  
Explanation:  
The number of soldiers in each row is:  
- Row 0: 1  
- Row 1: 4  
- Row 2: 1  
- Row 3: 1  
The rows ordered from weakest to strongest are [0,2,3,1].
```

```
import java.util.*;  
  
class Solution {  
    public int[] kWeakestRows(int[][] mat, int k) {  
        int m = mat.length;  
        int n = mat[0].length;  
  
        PriorityQueue<int[]> pq = new PriorityQueue<>(  
            (a, b) -> a[1] == b[1] ? a[0] - b[0] : a[1] - b[1]  
        );  
  
        for (int i = 0; i < m; i++) {  
            int soldiers = countSoldiers(mat[i]);  
            pq.offer(new int[]{i, soldiers});  
        }  
  
        int[] result = new int[k];  
        for (int i = 0; i < k; i++) {  
            result[i] = pq.poll()[0];  
        }  
  
        return result;  
    }  
  
    private int countSoldiers(int[] row) {  
        int count = 0;  
        for (int num : row) {  
            if (num == 1) {  
                count++;  
            }  
        }  
        return count;  
    }  
}
```

Time and Space Complexity:

Time $O(n \log n)$

Space $O(n)$

1985. Find the Kth Largest Integer in the Array

Medium

You are given an array of strings `nums` and an integer `k`. Each string in `nums` represents an integer without leading zeros.

Return the string that represents the `kth` largest integer in `nums`.

Note: Duplicate numbers should be counted distinctly. For example, if `nums` is `["1", "2", "2"]`, `"2"` is the first largest integer, `"2"` is the second-largest integer, and `"1"` is the third-largest integer.

Example 1:

Input: `nums = ["3", "6", "7", "10"]`, `k = 4`

Output: `"3"`

Explanation:

The numbers in `nums` sorted in non-decreasing order are `["3", "6", "7", "10"]`.

The 4th largest integer in `nums` is `"3"`.

Example 2:

Input: `nums = ["2", "21", "12", "1"]`, `k = 3`

Output: `"2"`

Explanation:

The numbers in `nums` sorted in non-decreasing order are `["1", "2", "12", "21"]`.

The 3rd largest integer in `nums` is `"2"`.

Example 3:

Input: `nums = ["0", "0"]`, `k = 2`

Output: `"0"`

Explanation:

The numbers in `nums` sorted in non-decreasing order are `["0", "0"]`.

The 2nd largest integer in `nums` is `"0"`.

Constraints:

- `1 <= k <= nums.length <= 104`
- `1 <= nums[i].length <= 100`
- `nums[i]` consists of only digits.
- `nums[i]` will not have any leading zeros.

```
class Solution {  
  
    public String kthLargestNumber(String[] nums, int k) {  
        Queue<String> minHeap = new PriorityQueue<>(  
            (a, b) -> a.length() == b.length() ? a.compareTo(b) : a.length() - b.length());  
  
        for (final String num : nums) {  
            minHeap.offer(num);  
            if (minHeap.size() > k)  
                minHeap.poll();  
        }  
  
        return minHeap.poll();  
    }  
}
```

Time and Space Complexity:

Time O(k log n)

Space O(n)

2233. Maximum Product After K Increments

Medium

You are given an array of non-negative integers `nums` and an integer `k`. In one operation, you may choose any element from `nums`, and increment it by `1`.

Return the *maximum product* of `nums` after at most `k` operations. Since the answer may be very large, return it modulo $10^9 + 7$. Note that you should maximize the product before taking the modulo.

Example 1:

```
Input: nums = [0,4], k = 5
Output: 20
Explanation: Increment the first number 5 times.
Now nums = [5, 4], with a product of 5 * 4 = 20.
It can be shown that 20 is maximum product possible, so we return 20.
Note that there may be other ways to increment nums to have the maximum product.
```

Example 2:

```
Input: nums = [6,3,3,2], k = 2
Output: 216
Explanation: Increment the second number 1 time and increment the fourth number 1 time.
Now nums = [6, 4, 3, 3], with a product of 6 * 4 * 3 * 3 = 216.
It can be shown that 216 is maximum product possible, so we return 216.
Note that there may be other ways to increment nums to have the maximum product.
```

Constraints:

- $1 \leq n \leq 10^5$
- $0 \leq n[i] \leq 10^6$

```
class Solution {
    public int maximumProduct(int[] nums, int k) {
        int kMod = 1_000_000_007;
        long ans = 1;
        Queue<Integer> minHeap = new PriorityQueue<>();

        for (int num : nums)
            minHeap.offer(num);

        for (int i = 0; i < k; ++i) {
            int minNum = minHeap.poll();
            minHeap.offer(minNum + 1);    [4, 5]
        }

        while (!minHeap.isEmpty()) {
            ans *= minHeap.poll();
            ans %= kMod;           4 * 5 = 20
        }

        return (int) ans;
    }
}
```

Time and Space Complexity:

Time $O(k \log n)$

Space $O(n)$

"Minimum Cost to Merge Stones" or "Optimal Merge Pattern"

Example:

```
java
Input: [4, 3, 2, 1]

Step 1: merge 1 and 2 → [4, 3, 3], cost = 3
Step 2: merge 3 and 3 → [4, 6], cost = 6
Step 3: merge 4 and 6 → [10], cost = 10
Total Cost = 3 + 6 + 10 = 19
```

Optimal Strategy: Greedy + Min Heap

At each step, you merge the two smallest elements to get the minimum cost.

Java Code (Greedy using Min Heap):

```
java
import java.util.*;

class Solution {
    public int minCostToMerge(List<Integer> arr) {
        PriorityQueue<Integer> pq = new PriorityQueue<>(arr);
        int totalCost = 0;

        while (pq.size() > 1) {
            int a = pq.poll();
            int b = pq.poll();
            int cost = a + b;
            totalCost += cost;
            pq.add(cost);
        }

        return totalCost;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        List<Integer> arr = Arrays.asList(4, 3, 2, 1);
        System.out.println("Minimum cost: " + sol.minCostToMerge(arr)); // Output: 19
    }
}
```

11. 703. Kth Largest Element in a Stream

Easy

Design a class to find the k^{th} largest element in a stream. Note that it is the k^{th} largest element in the sorted order, not the k^{th} distinct element.

Implement `KthLargest` class:

- `KthLargest(int k, int[] nums)` Initializes the object with the integer `k` and the stream of integers `nums`.
- `int add(int val)` Appends the integer `val` to the stream and returns the element representing the k^{th} largest element in the stream.

Example 1:

Input ["KthLargest", "add", "add", "add", "add", "add"]

[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]

Output

[null, 4, 5, 5, 8, 8]

Explanation

```
KthLargest kthLargest = new KthLargest(3, [4, 5, 8, 2]);
kthLargest.add(3); // return 4
kthLargest.add(5); // return 5
kthLargest.add(10); // return 5
kthLargest.add(9); // return 8
kthLargest.add(4); // return 8
```

Constraints:

- $1 \leq k \leq 10^4$
- $0 \leq \text{nums.length} \leq 10^4$
- $-10^4 \leq \text{nums}[i] \leq 10^4$
- $-10^4 \leq \text{val} \leq 10^4$
- At most 10^4 calls will be made to `add`.
- It is guaranteed that there will be at least `k` elements in the array when you search for the k^{th} element.

```
import java.util.PriorityQueue;
```

```
class KthLargest {  
    private PriorityQueue<Integer> minHeap;  
    private int k;  
  
    public KthLargest(int k, int[] nums) {  
        this.k = k;  
        this.minHeap = new PriorityQueue<>(k);  
  
        for (int num : nums) {  
            add(num);  
        }  
    }  
  
    public int add(int val) {  
        if (minHeap.size() < k) {  
            minHeap.offer(val);  
        } else if (val > minHeap.peek()) {  
            minHeap.poll();  
            minHeap.offer(val);  
        }  
        return minHeap.peek();  
    }  
}
```

```
public class TopKFrequentWords {  
    public List<String> topKFrequent(String[] words, int k) {  
        Map<String, Integer> freqMap = new HashMap<>();  
        for (String word : words) {  
            freqMap.put(word, freqMap.getOrDefault(word, 0) + 1);  
        }  
  
        PriorityQueue<String> heap = new PriorityQueue<>((w1, w2) -> {  
            int freqCompare = freqMap.get(w1) - freqMap.get(w2);  
            if (freqCompare == 0) {  
                return w2.compareTo(w1);  
            }  
            return freqCompare;  
        });  
  
        for (String word : freqMap.keySet()) {  
            heap.offer(word);  
            if (heap.size() > k) {  
                heap.poll();  
            }  
        }  
  
        List<String> result = new ArrayList<>();  
        while (!heap.isEmpty()) {  
            result.add(heap.poll());  
        }  
        Collections.reverse(result);  
        return result;  
    }  
}
```

Time and Space Complexity:

Time $O(k \log n)$

Space $O(n)$

Queue

A Queue is a linear data structure that follows the FIFO (First In, First Out) principle:

- The first element added is the first one to be removed.
- Think of a queue at a ticket counter – the person who comes first is served first.

Core Operations of a Queue

Operation	Description
<code>add()</code>	Adds an element to the <code>rear</code> of the queue
<code>offer()</code>	Same as <code>add()</code> but returns <code>false</code> if it fails
<code>remove()</code>	Removes and returns the <code>front</code> element
<code>poll()</code>	Removes and returns the front element (returns <code>null</code> if empty)
<code>element()</code>	Returns the front element without removing (throws exception if empty)
<code>peek()</code>	Returns the front element without removing (returns <code>null</code> if empty)

```
public class QueueExample {  
    public static void main(String[] args) {  
        Queue<Integer> queue = new LinkedList<>();  
  
        // Adding elements  
        queue.add(10);  
        queue.add(20);  
        queue.offer(30); // safe add  
  
        // Viewing the front element  
        System.out.println("Front: " + queue.peek()); // 10  
  
        // Removing elements  
        System.out.println("Removed: " + queue.remove()); // 10  
        System.out.println("Removed: " + queue.poll()); // 20  
  
        System.out.println("Current Queue: " + queue); // [30]  
    }  
}
```

Implementation of Queue in Java

Class	Description
LinkedList	Standard Queue using doubly linked list
PriorityQueue	Elements are ordered based on natural/comparator order (not FIFO)
ArrayDeque	Double-ended queue (can be used as both stack & queue)

💡 Thread-safe Queues (in `java.util.concurrent`)

Class	Description
ConcurrentLinkedQueue	High-performance, non-blocking queue
BlockingQueue	Interface for queues that wait if full/empty
ArrayBlockingQueue	Bounded blocking queue using array
LinkedBlockingQueue	Optionally-bounded, thread-safe queue

💡 When to Use

- Task scheduling (e.g., printer jobs)
- Breadth-first Search (BFS) in graphs
- Order processing
- Thread-safe producer-consumer problems

Operation	Description	Time Complexity
Enqueue (push)	Inserts an element at the rear of the queue	O(1)
Dequeue (pop)	Removes and returns the front element	O(1)
Front (peek)	Returns the front element without removing it	O(1)
Rear (back)	Returns the last element without removing it	O(1)
isEmpty()	Checks if the queue is empty	O(1)

✓ 3. Using `ArrayDeque` (Faster than `LinkedList`)

```
java
import java.util.*;

public class ArrayDequeQueue {
    public static void main(String[] args) {
        Queue<String> queue = new ArrayDeque<>();

        queue.offer("A");
        queue.offer("B");
        queue.offer("C");

        System.out.println("Queue: " + queue);      // [A, B, C]
        System.out.println("Front: " + queue.peek()); // A
        queue.poll(); // remove A
        System.out.println("Queue after dequeue: " + queue); // [B, C]
    }
}
```

```
class MyQueue {  
    private int[] arr;  
    private int front, rear, size, capacity;  
  
    public MyQueue(int capacity) {  
        this.capacity = capacity;  
        arr = new int[capacity];  
        front = 0;  
        rear = -1;  
        size = 0;  
    }  
  
    public void enqueue(int data) {  
        if (size == capacity) {  
            System.out.println("Queue is full!");  
            return;  
        }  
        rear = (rear + 1) % capacity;  
        arr[rear] = data;  
        size++;  
    }  
  
    public int dequeue() {  
        if (isEmpty()) {  
            System.out.println("Queue is empty!");  
            return -1;  
        }  
        int item = arr[front];  
        front = (front + 1) % capacity;  
        size--;  
        return item;  
    }  
  
    public int peek() {  
        if (isEmpty()) return -1;  
        return arr[front];  
    }  
  
    public boolean isEmpty() {  
        return size == 0;  
    }  
  
    public int getSize() {  
        return size;  
    }  
}  
  
public class CustomQueueDemo {  
    public static void main(String[] args) {  
        MyQueue queue = new MyQueue( capacity: 5);  
        queue.enqueue( data: 1);  
        queue.enqueue( data: 2);  
        queue.enqueue( data: 3);  
  
        System.out.println("Front: " + queue.peek()); // 1  
        System.out.println("Removed: " + queue.dequeue()); // 1  
        System.out.println("Size: " + queue.getSize()); // 2  
    }  
}
```

225. Implement Stack using Queues

Easy

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (`push`, `top`, `pop`, and `empty`).

Implement the `MyStack` class:

- `void push(int x)` Pushes element `x` to the top of the stack.
- `int pop()` Removes the element on the top of the stack and returns it.
- `int top()` Returns the element on the top of the stack.
- `boolean empty()` Returns `true` if the stack is empty, `false` otherwise.

Notes:

- You must use only standard operations of a queue, which means that only `push` to back, `peek/pop` from front, `size` and `is empty` operations are valid.
- Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

Example 1:

```
Input
["MyStack", "push", "push", "top", "pop", "empty"]
[], [1], [2], [], [1], []
Output
[null, null, null, 2, 2, false]
Explanation
MyStack myStack = new MyStack();
myStack.push(1);
myStack.push(2);
myStack.top(); // return 2
myStack.pop(); // return 2
myStack.empty(); // return False

import java.util.LinkedList;
import java.util.Queue;

class MyStack {

    private Queue<Integer> queue1;
    private Queue<Integer> queue2;

    public MyStack() {
        queue1 = new LinkedList<>();
        queue2 = new LinkedList<>();
    }

    public void push(int x) {
        queue2.offer(x);
        while (!queue1.isEmpty()) {
            queue2.offer(queue1.poll());
        }
        Queue<Integer> temp = queue1;
        queue1 = queue2;
        queue2 = temp;
    }

    public int pop() {
        if (queue1.isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        return queue1.poll();
    }

    public int top() {
        if (queue1.isEmpty()) {
            throw new RuntimeException("Stack is empty");
        }
        return queue1.peek();
    }

    public boolean empty() {
        return queue1.isEmpty();
    }
}

/**
 * Your MyStack object will be instantiated and called as such:
 * MyStack obj = new MyStack();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.top();
 */
```

232. Implement Queue using Stacks

Easy

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (`push`, `peek`, `pop`, and `empty`).

Implement the `MyQueue` class:

- `void push(int x)` Pushes element `x` to the back of the queue.
- `int pop()` Removes the element from the front of the queue and returns it.
- `int peek()` Returns the element at the front of the queue.
- `boolean empty()` Returns `true` if the queue is empty, `false` otherwise.

Notes:

- You must use only standard operations of a stack, which means only `push` to `top`, `peek/pop` from `top`, `size`, and `is empty` operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

Example 1:

```
Input ["MyQueue", "push", "push", "peek", "pop", "empty"]
[], [1], [2], [], [1], []
Output
[null, null, null, 1, 1, false]
Explanation MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false
```

```
import java.util.Stack;

class MyQueue {

    private Stack<Integer> stack1;
    private Stack<Integer> stack2;

    public MyQueue() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }

    public void push(int x) {
        stack1.push(x);
    }

    public int pop() {
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.pop();
    }

    public int peek() {
        if (stack2.isEmpty()) {
            while (!stack1.isEmpty()) {
                stack2.push(stack1.pop());
            }
        }
        return stack2.peek();
    }

    public boolean empty() {
        return stack1.isEmpty() && stack2.isEmpty();
    }
}

/**
 * Your MyQueue object will be instantiated and called as such:
 * MyQueue obj = new MyQueue();
 * obj.push(x);
 * int param_2 = obj.pop();
 * int param_3 = obj.peek();
 * boolean param_4 = obj.empty();
 */
```

Monotonic Stack

A monotonic stack is a stack data structure where the elements are ordered in a strictly increasing or decreasing manner. This ordering property helps efficiently solve problems related to finding the next or previous greater/smaller elements in an array, among other applications.

Types of Monotonic Stacks:

- **Monotonically Increasing Stack:** The elements in the stack are in increasing order, with the smallest element at the bottom.
- **Monotonically Decreasing Stack:** The elements in the stack are in decreasing order, with the largest element at the bottom.

```
public int[] nextGreaterElement(int[] nums) {  
    int[] result = new int[nums.length];  
    Stack<Integer> stack = new Stack<>();  
  
    for (int i = nums.length - 1; i >= 0; i--) {  
        while (!stack.isEmpty() && stack.peek() <= nums[i]) {  
            stack.pop();  
        }  
        result[i] = stack.isEmpty() ? -1 : stack.peek();  
        stack.push(nums[i]);  
    }  
  
    return result;  
}  
  
public int[] nextSmallerElement(int[] nums) {  
    int[] result = new int[nums.length];  
    Stack<Integer> stack = new Stack<>();  
  
    for (int i = nums.length - 1; i >= 0; i--) {  
        while (!stack.isEmpty() && stack.peek() >= nums[i]) {  
            stack.pop();  
        }  
        result[i] = stack.isEmpty() ? -1 : stack.peek();  
        stack.push(nums[i]);  
    }  
  
    return result;  
}
```

- Push Operation:
 - Time Complexity: (O(1))
 - Explanation: Adding an element to the top of the stack is constant-time.
- Pop Operation:
 - Time Complexity: (O(1))
 - Explanation: Removing the top element from the stack is constant-time.
- Peek Operation:
 - Time Complexity: (O(1))
 - Explanation: Viewing the top element without removing it is constant-time.
- Search Operation (contains):
 - Time Complexity: (O(n))
 - Explanation: Checking if an element exists in the stack requires iterating through the stack, resulting in linear time complexity.
- Size Operation:
 - Time Complexity: (O(1))
 - Explanation: Getting the number of elements in the stack is constant-time.

1. **Next Greater Elements**
2. **Next Smaller Elements**
3. **Prev Greater Elements**
4. **Prev Smaller Elements**
5. **84. Largest Rectangle in Histogram**
6. **456. 132 Pattern**
7. **496. Next Greater Element I**
8. **503. Next Greater Element II**
9. **901. Online Stock Span**
10. **1019. Next Greater Node In Linked List**
11. **1130. Minimum Cost Tree From Leaf Values**
12. **768. Max Chunks To Make Sorted II**
13. **769. Max Chunks To Make Sorted**
14. **316. Remove Duplicate Letters**
15. **402. Remove K Digits**

```

public class MonotonicStack {

    // Find Next Greater Element (Increasing Stack)
    public static int[] nextGreaterElements(int[] nums) {
        int[] res = new int[nums.length];
        Stack<Integer> stack = new Stack<>(); // stores indices

        for (int i = nums.length - 1; i >= 0; i--) {
            // maintain decreasing order
            while (!stack.isEmpty() && nums[stack.peek()] <= nums[i]) {
                stack.pop();
            }
            res[i] = stack.isEmpty() ? -1 : nums[stack.peek()];
            stack.push(i);
        }
        return res;
    }

    // Find Next Smaller Element (Decreasing Stack)
    public static int[] nextSmallerElements(int[] nums) {
        int[] res = new int[nums.length];
        Stack<Integer> stack = new Stack<>();

        for (int i = nums.length - 1; i >= 0; i--) {
            // maintain increasing order
            while (!stack.isEmpty() && nums[stack.peek()] >= nums[i]) {
                stack.pop();
            }
            res[i] = stack.isEmpty() ? -1 : nums[stack.peek()];
            stack.push(i);
        }
        return res;
    }
}

```

```

// Previous Greater Element
public static int[] prevGreaterElements(int[] nums) {
    int[] res = new int[nums.length];
    Stack<Integer> stack = new Stack<>();

    for (int i = 0; i < nums.length; i++) {
        while (!stack.isEmpty() && nums[stack.peek()] <= nums[i]) {
            stack.pop();
        }
        res[i] = stack.isEmpty() ? -1 : nums[stack.peek()];
        stack.push(i);
    }
    return res;
}

// Previous Smaller Element
public static int[] prevSmallerElements(int[] nums) {
    int[] res = new int[nums.length];
    Stack<Integer> stack = new Stack<>();

    for (int i = 0; i < nums.length; i++) {
        while (!stack.isEmpty() && nums[stack.peek()] >= nums[i]) {
            stack.pop();
        }
        res[i] = stack.isEmpty() ? -1 : nums[stack.peek()];
        stack.push(i);
    }
    return res;
}

```

```

// Utility function to print array
private static void printArray(int[] arr, String label) {
    System.out.print(label + ": ");
    for (int val : arr) {
        System.out.print(val + " ");
    }
    System.out.println();
}

public static void main(String[] args) {
    int[] nums = {2, 1, 5, 3, 4};

    printArray(nextGreaterElements(nums), label: "Next Greater"); // 5 5 -1 4 -1
    printArray(nextSmallerElements(nums), label: "Next Smaller"); // 1 -1 3 -1 -1
    printArray(prevGreaterElements(nums), label: "Previous Greater"); // -1 2 -1 5 5
    printArray(prevSmallerElements(nums), label: "Previous Smaller"); // -1 -1 1 1 3
}
}

```

7.496. Next Greater Element I

The next greater element of some element x in an array is the first greater element that is to the right of x in the same array.

You are given two distinct 0-indexed integer arrays nums1 and nums2 , where nums1 is a subset of nums2 .

For each $0 \leq i < \text{nums1.length}$, find the index j such that $\text{nums1}[i] == \text{nums2}[j]$ and determine the next greater element of $\text{nums2}[j]$ in nums2 . If there is no next greater element, then the answer for this query is -1 .

Return an array ans of length nums1.length such that $\text{ans}[i]$ is the next greater element as described above.

Example 1:

Input: $\text{nums1} = [4,1,2]$, $\text{nums2} = [1,3,4,2]$

Output: $[-1,3,-1]$

Explanation: The next greater element for each value of nums1 is as follows:

- 4 is underlined in $\text{nums2} = [1,\underline{3},\underline{4},2]$. There is no next greater element, so the answer is -1 .
- 1 is underlined in $\text{nums2} = [\underline{1},3,4,2]$. The next greater element is 3.
- 2 is underlined in $\text{nums2} = [1,3,\underline{4},2]$. There is no next greater element, so the answer is -1 .

Example 2:

Input: $\text{nums1} = [2,4]$, $\text{nums2} = [1,2,3,4]$

Output: $[3,-1]$

Explanation: The next greater element for each value of nums1 is as follows:

- 2 is underlined in $\text{nums2} = [1,2,\underline{3},4]$. The next greater element is 3.
- 4 is underlined in $\text{nums2} = [1,2,3,\underline{4}]$. There is no next greater element, so the answer is -1 .

```
class Solution {
    public int[] nextGreaterElement(int[] nums1, int[] nums2) {
        Map<Integer, Integer> map = new HashMap<>();
        int m = nums1.length;
        int n = nums2.length;
        int[] nums = nextGreaterElement(nums2);

        for (int i = 0; i < n; i++) {
            map.put(nums2[i], nums[i]);
        }

        int[] result = new int[m];

        for (int i = 0; i < m; i++) {
            result[i] = map.get(nums1[i]);
        }

        return result;
    }

    public int[] nextGreaterElement(int[] nums) {
        int[] result = new int[nums.length];
        Stack<Integer> stack = new Stack<>();

        for (int i = nums.length - 1; i >= 0; i--) {
            while (!stack.isEmpty() && stack.peek() <= nums[i]) {
                stack.pop();
            }
            result[i] = stack.isEmpty() ? -1 : stack.peek();
            stack.push(nums[i]);
        }

        return result;
    }
}
```

Time and Space Complexity:

Time $O(n)$

Space $O(n)$

8. 503. Next Greater Element II

Medium

Given a circular integer array `nums` (i.e., the next element of `nums[nums.length - 1]` is `nums[0]`), return the next greater number for every element in `nums`.

The next greater number of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, return `-1` for this number.

Example 1:

```
Input: nums = [1,2,1]
Output: [2,-1,2]
Explanation: The first 1's next greater number is 2;
The number 2 can't find next greater number.
The second 1's next greater number needs to search circularly, which is also 2.
```

Example 2:

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

Constraints:

- $1 \leq n \leq 10^4$
- $-10^9 \leq nums[i] \leq 10^9$

```
class Solution {
    public int[] nextGreaterElements(int[] nums) {
        int n = nums.length;
        int[] result = new int[n];
        Stack<Integer> stack = new Stack<>();

        for (int i = 2 * n - 1; i >= 0; i--) {
            while (!stack.isEmpty() && stack.peek() <= nums[i % n]) {
                stack.pop();
            }
            result[i % n] = stack.isEmpty() ? -1 : stack.peek();
            stack.push(nums[i % n]);
        }

        return result;
    }
}
```

Time and Space Complexity:

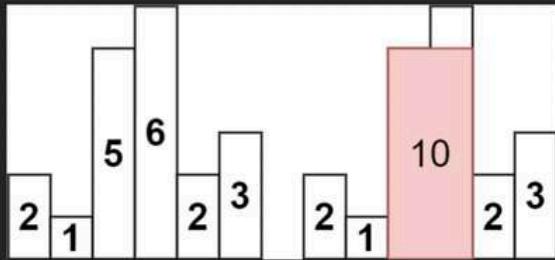
Time $O(n)$

Space $O(n)$

5. 84. Largest Rectangle in Histogram

Given an array of integers `heights` representing the histogram's bar height where the width of each bar is 1, return *the area of the largest rectangle in the histogram*.

Example 1:



Input: `heights = [2,1,5,6,2,3]`

Output: 10

Explanation: The above is a histogram where width of each bar is 1. The largest rectangle is shown in the red area, which has an area = 10 units.

```
class Solution {
    public int largestRectangleArea(int[] heights) {
        Stack<Integer> stack = new Stack<>();
        int maxArea = 0;
        int index = 0;

        while (index < heights.length) {
            if (stack.isEmpty() || heights[index] >= heights[stack.peek()]) {
                stack.push(index++);
            } else {
                int top = stack.pop();
                int height = heights[top];
                int width = stack.isEmpty() ? index : index - stack.peek() - 1;
                maxArea = Math.max(maxArea, height * width);
            }
        }

        while (!stack.isEmpty()) {
            int top = stack.pop();
            int height = heights[top];
            int width = stack.isEmpty() ? index : index - stack.peek() - 1;
            maxArea = Math.max(maxArea, height * width);
        }

        return maxArea;
    }
}
```

Time and Space Complexity:

Time $O(n)$

Space $O(n)$

Note: Second while loop is needed in the case array ends while bars were still increasing (stack not empty).

6. 456. 132 Pattern

Medium

Given an array of n integers nums , a 132 pattern is a subsequence of three integers $\text{nums}[i]$, $\text{nums}[j]$ and $\text{nums}[k]$ such that $i < j < k$ and $\text{nums}[i] < \text{nums}[k] < \text{nums}[j]$.

Return `true` if there is a 132 pattern in nums , otherwise, return `false`.

Example 1:

Input: $\text{nums} = [1, 2, 3, 4]$
Output: `false`
Explanation: There is no 132 pattern in the sequence.

Example 2:

Input: $\text{nums} = [3, 1, 4, 2]$
Output: `true`
Explanation: There is a 132 pattern in the sequence: $[1, 4, 2]$.

Example 3:

Input: $\text{nums} = [-1, 3, 2, 0]$
Output: `true`
Explanation: There are three 132 patterns in the sequence: $[-1, 3, 2]$, $[-1, 3, 0]$ and $[-1, 2, 0]$.

```
class Solution {
    public static boolean find132pattern(int[] nums) {
        int n = nums.length;
        if (n < 3) {
            return false;
        }

        Stack<Integer> stack = new Stack<>();
        int[] min = new int[n];
        min[0] = nums[0];

        for (int i = 1; i < n; ++i) {
            min[i] = Math.min(min[i - 1], nums[i]);
        }
        System.out.println(Arrays.toString(min)); // [1, 1, 1, 1]

        for (int j = n - 1; j >= 0; --j) {
            if (nums[j] > min[j]) {
                while (!stack.isEmpty() && stack.peek() <= min[j]) {
                    stack.pop();
                }
                if (!stack.isEmpty() && stack.peek() < nums[j]) {
                    return true;
                }
                stack.push(nums[j]);
            }
        }
        System.out.println(stack); // [4, 3, 2]

        return false;
    }

    public static void main(String[] args) {
        System.out.println(find132pattern(new int[]{1, 2, 3, 4})); // false
    }
}
```

Time and Space Complexity:

Time $O(n)$

Space $O(n)$

9. 901. Online Stock Span

Design an algorithm that collects daily price quotes for some stock and returns the span of that stock's price for the current day.

The span of the stock's price in one day is the maximum number of consecutive days (starting from that day and going backward) for which the stock price was less than or equal to the price of that day.

- For example, if the prices of the stock in the last four days is [7, 2, 1, 2] and the price of the stock today is 2, then the span of today is 4 because starting from today, the price of the stock was less than or equal 2 for 4 consecutive days.
- Also, if the prices of the stock in the last four days is [7, 34, 1, 2] and the price of the stock today is 8, then the span of today is 3 because starting from today, the price of the stock was less than or equal 8 for 3 consecutive days.

Implement the `StockSpanner` class:

- `StockSpanner()` Initializes the object of the class.
- `int next(int price)` Returns the span of the stock's price given that today's price is `price`.

Example 1:

Input
["StockSpanner", "next", "next", "next", "next", "next", "next", "next"]
[], [100], [80], [60], [70], [60], [75], [85]
Output
[null, 1, 1, 1, 2, 1, 4, 6]
Explanation
StockSpanner stockSpanner = new StockSpanner();
stockSpanner.next(100); // return 1
stockSpanner.next(80); // return 1
stockSpanner.next(60); // return 1
stockSpanner.next(70); // return 2
stockSpanner.next(60); // return 1
stockSpanner.next(75); // return 4, because the last 4 prices (including today's price of 75)
were less than or equal to today's price.
stockSpanner.next(85); // return 6

```
import java.util.Stack;

class StockSpanner {
    Stack<int[]> stack;

    public StockSpanner() {
        stack = new Stack<>();
    }

    public int next(int price) {
        int span = 1;
        while (!stack.isEmpty() && stack.peek()[0] <= price) {
            span += stack.pop()[1];
        }
        stack.push(new int[]{price, span});
        return span;
    }

    /**
     * Your StockSpanner object will be instantiated and called as such:
     * StockSpanner obj = new StockSpanner();
     * int param_1 = obj.next(price);
     */
}
```

Time and Space Complexity:

Time O(n)

Space O(n)

11. 402. Remove K Digits

Given string num representing a non-negative integer, and an integer k, return the smallest possible integer after removing k digits from num.

Example 1:

```
Input: num = "1432219", k = 3
Output: "1219"
Explanation: Remove the three digits 4, 3, and 2 to form the new number 1219 which is the smallest.
```

Example 2:

```
Input: num = "10200", k = 1
Output: "200"
Explanation: Remove the leading 1 and the number is 200. Note that the output must not contain leading zeroes.
```

Example 3:

```
Input: num = "10", k = 2
Output: "0"
Explanation: Remove all the digits from the number and it is left with nothing which is 0.
```

Constraints:

- $1 \leq k \leq \text{num.length} \leq 10^5$
- num consists of only digits.
- num does not have any leading zeros except for the zero itself.

```
class Solution {
    public String removeKdigits(String num, int k) {
        if (k == num.length()) {
            return "0";
        }

        Stack<Character> stack = new Stack<>();

        for (char digit : num.toCharArray()) {
            while (!stack.isEmpty() && k > 0 && stack.peek() > digit) {
                stack.pop();
                k--;
            }
            stack.push(digit);
        }

        while (k > 0) {
            stack.pop();
            k--;
        }

        StringBuilder result = new StringBuilder();
        while (!stack.isEmpty()) {
            result.append(stack.pop());
        }

        result.reverse();
        while (result.length() > 1 && result.charAt(0) == '0') {
            result.deleteCharAt(0);
        }

        return result.toString();
    }
}
```

Time and Space Complexity:

Time O(n)

Space O(n)

10. 316. Remove Duplicate Letters

316. Remove Duplicate Letters

Medium

Given a string s , remove duplicate letters so that every letter appears once and only once. You must make sure your result is the smallest in lexicographical order among all possible results.

Example 1:

Input: $s = "bcabc"$
Output: "abc"

Example 2:

Input: $s = "cbacdcbc"$
Output: "acdb"

Constraints:

- $1 \leq s.length \leq 10^4$
- s consists of lowercase English letters.

Note: This question is the same as 1081: <https://leetcode.com/problems/smallest-subsequence-of-distinct-characters/>

```
import java.util.Stack;

class Solution {
    public String removeDuplicateLetters(String s) {
        int[] count = new int[26];
        for (char c : s.toCharArray()) {
            count[c - 'a']++;
        }

        boolean[] visited = new boolean[26];
        Stack<Character> stack = new Stack<>();

        for (char c : s.toCharArray()) {
            count[c - 'a']--;
            if (visited[c - 'a']) {
                continue;
            }

            while (!stack.isEmpty() && stack.peek() > c
                  && count[stack.peek() - 'a'] > 0) {
                visited[stack.pop() - 'a'] = false;
            }

            stack.push(c);
            visited[c - 'a'] = true;
        }

        StringBuilder result = new StringBuilder();
        while (!stack.isEmpty()) {
            result.insert(0, stack.pop());
        }

        return result.toString();
    }
}
```

Time and Space Complexity:

Time $O(n)$

Space $O(n)$

12. 1130. Minimum Cost Tree From Leaf Values

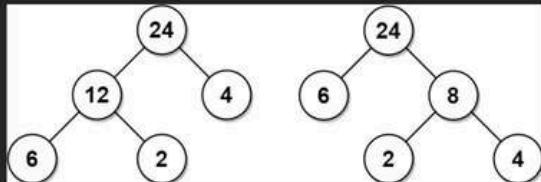
Given an array `arr` of positive integers, consider all binary trees such that:

- Each node has either 0 or 2 children;
- The values of `arr` correspond to the values of each leaf in an in-order traversal of the tree.
- The value of each non-leaf node is equal to the product of the largest leaf value in its left and right subtree, respectively.

Among all possible binary trees considered, return *the smallest possible sum of the values of each non-leaf node*. It is guaranteed this sum fits into a 32-bit integer.

A node is a leaf if and only if it has zero children.

Example 1:



Input: `arr = [6,2,4]`

Output: 32

Explanation: There are two possible trees shown. The first has a non-leaf node sum 36, and the second has non-leaf node sum 32.

```
class Solution {
    public int mctFromLeafValues(int[] arr) {
        int ans = 0;
        Deque<Integer> stack = new ArrayDeque<>();
        stack.push(Integer.MAX_VALUE);

        for (int a : arr) {
            while (stack.peek() <= a) {
                int mid = stack.pop();
                ans += Math.min(stack.peek(), a) * mid;
            }
            stack.push(a);
        }

        while (stack.size() > 2)
            ans += stack.pop() * stack.peek();

        return ans;
    }
}
```

Time and Space Complexity:

Time $O(n)$

Space $O(n)$

Stack

A Stack is a Last In, First Out (LIFO) data structure.

- You push an item to insert.
- You pop an item to remove the most recently added item.

Think of it like a stack of plates – you take the top one off first.

```
public class CustomStack {  
    private int[] arr;  
    private int top;  
    private int capacity;  
  
    public CustomStack(int size) {  
        arr = new int[size];  
        top = -1; // means stack is empty  
        capacity = size;  
    }  
  
    public void push(int x) {  
        if (top == capacity - 1) {  
            System.out.println("Stack Overflow");  
            return;  
        }  
        arr[++top] = x;  
    }  
  
    public int pop() {  
        if (isEmpty()) {  
            System.out.println("Stack Underflow");  
            return -1;  
        }  
        return arr[top--];  
    }  
  
    public int peek() {  
        if (isEmpty()) {  
            System.out.println("Stack is Empty");  
            return -1;  
        }  
        return arr[top];  
    }  
}
```

Basic Operations

Push: O(1) - Insert an element at the top.

Pop: O(1) - Remove the top element.

Peek: O(1) - Access the top element.

isEmpty: O(1) - Check if the stack is empty.

```
public boolean isEmpty() {
    return top == -1;
}

public int size() {
    return top + 1;
}

public void display() {
    if (isEmpty()) {
        System.out.println("Stack is Empty");
        return;
    }
    System.out.print("Stack elements: ");
    for (int i = 0; i <= top; i++) {
        System.out.print(arr[i] + " ");
    }
    System.out.println();
}
}

public static void main(String[] args) {
    CustomStack stack = new CustomStack( size: 5);

    stack.push( x: 10);
    stack.push( x: 20);
    stack.push( x: 30);

    stack.display(); // Output: 10 20 30

    System.out.println("Top: " + stack.peek()); // 30

    stack.pop();
    stack.display(); // Output: 10 20

    System.out.println("Is Empty? " + stack.isEmpty()); // false
    System.out.println("Size: " + stack.size()); // 2
}
```

1, 20. Valid Parentheses

2, 32. Longest Valid Parentheses

3, 71. Simplify Path

4, 155. Min Stack

5, 341. Flatten Nested List Iterator

6, 388. Longest Absolute File Path

7, 589. N-ary Tree Preorder Traversal

8, 716. Max Stack

9, 735. Asteroid Collision

10, 895. Maximum Frequency Stack

11, 2390. Removing Stars From a String

12, 2751. Robot Collisions

1, 20. Valid Parentheses

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

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.
- Every close bracket has a corresponding open bracket of the same type.

Example 1:

```
Input: s = "()"
Output: true
```

Example 2:

```
Input: s = "()[]"
Output: true
```

Example 3:

```
Input: s = "[]"
Output: false
```

Constraints:

- $1 \leq s.length \leq 10^4$
- `s` consists of parentheses only '`()[]`'.

```
import java.util.*;

class Solution {
    public boolean isValid(String s) {
        Deque<Character> stack = new ArrayDeque<>();

        for (char ch : s.toCharArray()) {
            if (ch == '(' || ch == '{' || ch == '[') {
                stack.push(ch);
            } else {
                if (stack.isEmpty())
                    return false;
                char open = stack.pop();
                if (!isMatchingPair(open, ch))
                    return false;
            }
        }

        return stack.isEmpty();
    }

    private boolean isMatchingPair(char open, char close) {
        return (open == '(' && close == ')') ||
               (open == '{' && close == '}') ||
               (open == '[' && close == ']');
    }
}
```

Time and Space Complexity:

Time $O(n)$

Space $O(n)$

2, 32. Longest Valid Parentheses

32. Longest Valid Parentheses

Hard

Given a string containing just the characters `'('` and `')'`, return *the length of the longest valid (well-formed) parentheses substring*.

Example 1:

```
Input: s = "()"
Output: 2
Explanation: The longest valid parentheses substring is "()".
```

Example 2:

```
Input: s = "())()()"
Output: 4
Explanation: The longest valid parentheses substring is "())".
```

Example 3:

```
Input: s = ""
Output: 0
```

Constraints:

- $0 \leq s.length \leq 3 * 10^4$
- $s[i]$ is `'('` or `)'`.

```
import java.util.*;

class Solution {
    public int longestValidParentheses(String s) {
        Deque<Integer> stack = new ArrayDeque<>();
        stack.push(-1);
        int maxLength = 0;

        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);
            if (ch == '(') {
                stack.push(i);
            } else {
                stack.pop();
                if (stack.isEmpty()) {
                    stack.push(i);
                } else {
                    maxLength = Math.max(maxLength, i - stack.peek());
                }
            }
        }

        return maxLength;
    }
}
```

Time and Space Complexity:

Time $O(n)$

Space $O(n)$

Note: If the stack is empty, start computing largest valid parentheses from that index

3, 71. Simplify Path

Medium

Given an absolute path for a Unix-style file system, which begins with a slash `'/'`, transform this path into its simplified canonical path.

In Unix-style file system context, a single period `'.'` signifies the current directory, a double period `'..'` denotes moving up one directory level, and multiple slashes such as `"//"` are interpreted as a single slash. In this problem, treat sequences of periods not covered by the previous rules (like `"..."`) as valid names for files or directories.

The simplified canonical path should adhere to the following rules:

- It must start with a single slash `'/'`.
- Directories within the path should be separated by only one slash `'/'`.
- It should not end with a slash `'/'`, unless it's the root directory.
- It should exclude any single or double periods used to denote current or parent directories.

Return the new path.

Example 1:

```
Input: path = "/home//"
Output: "/home"
Explanation:
The trailing slash should be removed.
```

Example 2:

```
Input: path = "/home//foo//"
Output: "/home/foo"
Explanation:
Multiple consecutive slashes are replaced by a single one.
```

```
import java.util.*;

class Solution {
    public String simplifyPath(String path) {
        Deque<String> stack = new ArrayDeque<>();
        String[] components = path.split("/");

        for (String component : components) {
            if (component.equals("") || component.equals(".")) {
                continue;
            } else if (component.equals("..")) {
                if (!stack.isEmpty()) {
                    stack.pop();
                }
            } else {
                stack.push(component);
            }
        }

        StringBuilder simplifiedPath = new StringBuilder();
        while (!stack.isEmpty()) {
            simplifiedPath.insert(0, "/" + stack.pop());
        }

        return simplifiedPath.length() > 0 ? simplifiedPath.toString() : "/";
    }
}
```

Time and Space Complexity:

Time $O(n)$

Space $O(n)$

4, 155. Min Stack

Medium

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the `MinStack` class:

- `MinStack()` initializes the stack object.
- `void push(int val)` pushes the element `val` onto the stack.
- `void pop()` removes the element on the top of the stack.
- `int top()` gets the top element of the stack.
- `int getMin()` retrieves the minimum element in the stack.

You must implement a solution with $O(1)$ time complexity for each function.

Example 1:

Input["MinStack","push","push","push","getMin","pop","top","getMin"]
[[],[-2],[0],[-3],[],[],[],[]]

Output
[null,null,null,null,-3,null,0,-2]

Explanation

```
MinStack minStack = new MinStack();
minStack.push(-2);
minStack.push(0);
minStack.push(-3);
minStack.getMin(); // return -3
minStack.pop();
minStack.top(); // return 0
minStack.getMin(); // return -2
```

Constraints:

- $-2^{31} \leq val \leq 2^{31} - 1$
- Methods `pop`, `top` and `getMin` operations will always be called on non-empty stacks.
- At most $3 * 10^4$ calls will be made to `push`, `pop`, `top`, and `getMin`.

```
import java.util.Stack;

class MinStack {
    private Stack<Integer> stack;
    private Stack<Integer> minStack;

    public MinStack() {
        stack = new Stack<>();
        minStack = new Stack<>();
    }

    public void push(int val) {
        stack.push(val);
        if (minStack.isEmpty() || val <= minStack.peek()) {
            minStack.push(val);
        }
    }

    public void pop() {
        if (stack.peek().equals(minStack.peek())) {
            minStack.pop();
        }
        stack.pop();
    }

    public int top() {
        return stack.peek();
    }

    public int getMin() {
        return minStack.peek();
    }
}
```

 Time and Space Complexity:

Time $O(n)$
Space $O(n)$

5, 341. Flatten Nested List Iterator

You are given a nested list of integers `nestedList`. Each element is either an integer or a list whose elements may also be integers or other lists. Implement an iterator to flatten it.

Implement the `NestedIterator` class:

- `NestedIterator(List<NestedInteger> nestedList)` Initializes the iterator with the nested list `nestedList`.
- `int next()` Returns the next integer in the nested list.
- `boolean hasNext()` Returns `true` if there are still some integers in the nested list and `false` otherwise.

Your code will be tested with the following pseudocode:

```
initialize iterator with nestedList res = [] while
iterator.hasNext() append iterator.next() to the end of res
return res
```

If `res` matches the expected flattened list, then your code will be judged as correct.

Example 1:

Input: `nestedList = [[1,1],2,[1,1]]`

Output: `[1,1,2,1,1]`

Explanation: By calling `next` repeatedly until `hasNext` returns `false`, the order of elements returned by `next` should be:
`[1,1,2,1,1]`.

```
public class NestedIterator implements Iterator<Integer> {
    private Stack<NestedInteger> stack = new Stack<>();

    public NestedIterator(List<NestedInteger> nestedList) {
        addInteger(nestedList);
    }

    @Override
    public Integer next() {
        return stack.pop().getInteger();
    }

    @Override
    public boolean hasNext() {
        while (!stack.isEmpty() && !stack.peek().isInteger()) {
            NestedInteger ni = stack.pop();
            addInteger(ni.getList());
        }
        return !stack.isEmpty();
    }

    private void addInteger(final List<NestedInteger> nestedList) {
        for (int i = nestedList.size() - 1; i >= 0; --i)
            stack.push(nestedList.get(i));
    }
}
```

Time and Space Complexity:

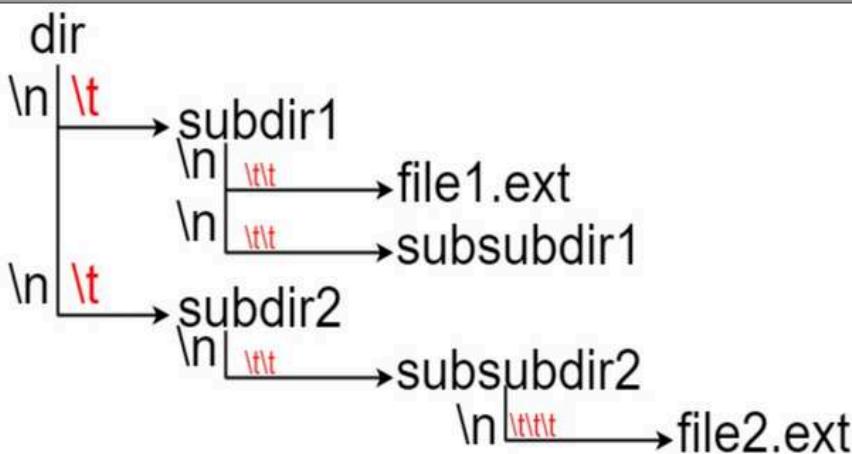
Time $O(n)$

Space $O(n)$

6, 388. Longest Absolute File Path

Medium

Suppose we have a file system that stores both files and directories. An example of one system is represented in the following picture:



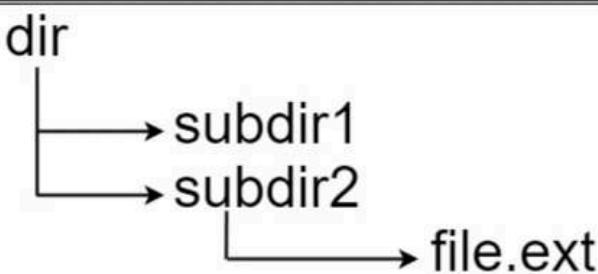
Here, we have `dir` as the only directory in the root. `dir` contains two subdirectories, `subdir1` and `subdir2`. `subdir1` contains a file `file1.ext` and subdirectory `subsubdir1`. `subdir2` contains a subdirectory `subsubdir2`, which contains a file `file2.ext`.

In text form, it looks like this (with \rightarrow representing the tab character):

Given a string `input` representing the file system in the explained format, return *the length of the longest absolute path to a file in the abstracted file system*. If there is no file in the system, return `0`.

Note that the testcases are generated such that the file system is valid and no file or directory name has length 0.

Example 1:



Input: `input = "dir\n\tsubdir1\n\tsubdir2\n\t\tfile.ext"`

Output: `20`

Explanation: We have only one file, and the absolute path is "`dir/subdir2/file.ext`" of length 20.

Time and Space Complexity:

Time $O(n)$

Space $O(n)$

```

import java.util.HashMap;

public class Solution {
    public int lengthLongestPath(String input) {
        HashMap<Integer, Integer> pathLengths = new HashMap<>();
        pathLengths.put(0, 0);
        int maxLength = 0;

        for (String part : input.split("\n")) {
            int level = part.lastIndexOf(' ') + 1;
            int len = part.length() - level;

            if (part.contains(".")) {
                maxLength = Math.max(maxLength, pathLengths.get(level) + len);
            } else {
                pathLengths.put(level + 1, pathLengths.get(level) + len + 1);
            }
        }

        return maxLength;
    }
}

```

Time and Space Complexity:

Time O(n)

Space O(n)

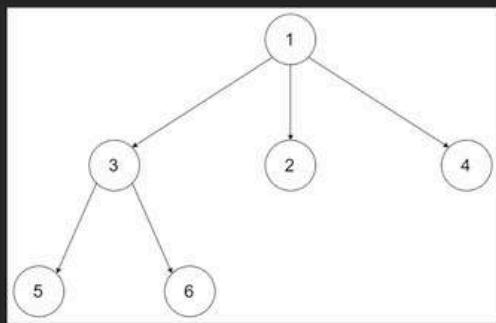
7, 589. N-ary Tree Preorder Traversal

Easy

Given the `root` of an n-ary tree, return *the preorder traversal of its nodes' values*.

Nary-Tree input serialization is represented in their level order traversal. Each group of children is separated by the null value (See examples)

Example 1:



Input: `root = [1,null,3,2,4,null,5,6]`
 Output: `[1,3,5,6,2,4]`

Time and Space Complexity:

Time O(n)

Space O(n)

```

/*
class Node {
    public int val;
    public List<Node> children;
    public Node() {}
    public Node(int _val) { val = _val; }
    public Node(int _val, List<Node> _children) {
        val = _val;
        children = _children;
    }
};

class Solution {
    public List<Integer> preorder(Node root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) { return result; }
        Stack<Node> stack = new Stack<>();
        stack.push(root);
        while (!stack.isEmpty()) {
            Node node = stack.pop();
            result.add(node.val);
            for (int i = node.children.size() - 1; i >= 0; i--) {
                stack.push(node.children.get(i));
            }
        }
        return result;
    }
}

```

8. 716. Max Stack

Hard

Design a max stack that supports push, pop, top, peekMax and popMax.

- `push(x)` -- Push element `x` onto stack.
- `pop()` -- Remove the element on top of the stack and return it.
- `top()` -- Get the element on the top.
- `peekMax()` -- Retrieve the maximum element in the stack.
- `popMax()` -- Retrieve the maximum element in the stack, and remove it. If you find more than one maximum elements, only remove the top-most one.

Example 1:

```

MaxStack stack = new MaxStack();
stack.push(5);
stack.push(1);
stack.push(5);
stack.top(); -> 5
stack.popMax(); -> 5
stack.top(); -> 1
stack.peekMax(); -> 5
stack.pop(); -> 1
stack.top(); -> 5

```

```
class Node {
    public int val;
    public Node prev, next;
    public Node() { }
    public Node(int val) {
        this.val = val;
    }
}

class DoubleLinkedList {
    private final Node head = new Node();
    private final Node tail = new Node();

    public DoubleLinkedList() {
        head.next = tail;
        tail.prev = head;
    }

    public Node append(int val) {
        Node node = new Node(val);
        node.next = tail;
        node.prev = tail.prev;
        tail.prev = node;
        node.prev.next = node;
        return node;
    }

    public static Node remove(Node node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
        return node;
    }

    public Node pop() {
        return remove(tail.prev);
    }

    public int peek() {
        return tail.prev.val;
    }
}

class MaxStack {
    private DoubleLinkedList stack = new DoubleLinkedList();
    private TreeMap<Integer, List<Node>> map = new TreeMap<>();

    public MaxStack() {
    }

    public void push(int x) {
        Node node = stack.append(x);
        map.computeIfAbsent(x, k -> new ArrayList<>()).add(node);
    }
}
```

```

public int pop() {
    Node node = stack.pop();
    List<Node> nodes = map.get(node.val);
    int x = nodes.remove(nodes.size() - 1).val;
    if (nodes.isEmpty()) {
        map.remove(node.val);
    }
    return x;
}

public int top() {
    return stack.peek();
}

public int peekMax() {
    return map.lastKey();
}

public int popMax() {
    int x = peekMax();
    List<Node> nodes = map.get(x);
    Node node = nodes.remove(nodes.size() - 1);
    if (nodes.isEmpty()) {
        map.remove(x);
    }
    DoubleLinkedList.remove(node);
    return x;
}
}

```

Time and Space Complexity:

Time O(n)

Space O(n)

9. 735. Asteroid Collision

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.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

Example 1:

Input: `asteroids = [5,10,-5]`
Output: `[5,10]`
Explanation: The 10 and -5 collide resulting in 10. The 5 and 10 never collide.

Example 2:

Input: `asteroids = [8,-8]`
Output: `[]`
Explanation: The 8 and -8 collide exploding each other.

```

// [5,10,-5]
class Solution {
    public int[] asteroidCollision(int[] asteroids) {
        Stack<Integer> stack = new Stack<>();
        for (final int a : asteroids)
            if (a > 0)
                stack.push(a);
            else {
                while (!stack.isEmpty() && stack.peek() > 0
                    && stack.peek() < -a)
                    stack.pop();
                if (stack.isEmpty() || stack.peek() < 0)
                    stack.push(a);
                else if (stack.peek() == -a)
                    stack.pop();
            }
        int[] ans = new int[stack.size()];
        for (int i = ans.length - 1; i >= 0; --i)
            ans[i] = stack.pop();
        return ans;
    }
}

```

Time and Space Complexity:

Time O(n)

Space O(n)

10. 895. Maximum Frequency Stack

Design a stack-like data structure to push elements to the stack and pop the most frequent element from the stack.

Implement the `FreqStack` class:

- `FreqStack()` constructs an empty frequency stack.
- `void push(int val)` pushes an integer `val` onto the top of the stack.
- `int pop()` removes and returns the most frequent element in the stack.
- If there is a tie for the most frequent element, the element closest to the stack's top is removed and returned.

Example 1:

Input
`["FreqStack", "push", "push", "push", "push", "push", "push", "pop", "pop"]`
`[[], [5], [7], [5], [7], [4], [5], [], [], [], []]`

Output
`[null, null, null, null, null, null, null, 5, 7, 5, 4]`

Explanation
`FreqStack freqStack = new FreqStack();`
`freqStack.push(5); // The stack is [5]`
`freqStack.push(7); // The stack is [5,7]`
`freqStack.push(5); // The stack is [5,7,5]`
`freqStack.push(7); // The stack is [5,7,5,7]`
`freqStack.push(4); // The stack is [5,7,5,7,4]`
`freqStack.push(5); // The stack is [5,7,5,7,4,5]`
`freqStack.pop(); // return 5, as 5 is the most frequent. The stack becomes [5,7,5,7,4].`
`freqStack.pop(); // return 7, as 5 and 7 is the most frequent, but 7 is closest to the top. The stack becomes [5,7,5,4].`
`freqStack.pop(); // return 5, as 5 is the most frequent. The stack becomes [5,7,4].`
`freqStack.pop(); // return 4, as 4, 5 and 7 is the most frequent, but 4 is closest to the top. The stack becomes [5,7].`

```

class FreqStack {
    public void push(int val) {
        count.merge(val, 1, Integer::sum);
        countToStack.putIfAbsent(count.get(val), new ArrayDeque<>());
        countToStack.get(count.get(val)).push(val);
        maxFreq = Math.max(maxFreq, count.get(val));
    }

    public int pop() {
        final int val = countToStack.get(maxFreq).pop();
        count.merge(val, -1, Integer::sum);
        if (countToStack.get(maxFreq).isEmpty())
            --maxFreq;
        return val;
    }

    private int maxFreq = 0;
    private Map<Integer, Integer> count = new HashMap<>();
    private Map<Integer, Deque<Integer>> countToStack = new HashMap<>();
}

/**
 * Your FreqStack object will be instantiated and called as such:
 * FreqStack obj = new FreqStack();
 * obj.push(val);
 * int param_2 = obj.pop();
 */

```

Time and Space Complexity:

Time O(n)

Space O(1)

11. 224. Basic Calculator

Given a string `s` representing a valid expression, implement a basic calculator to evaluate it, and return *the result of the evaluation*.

Note: You are not allowed to use any built-in function which evaluates strings as mathematical expressions, such as `eval()`.

Example 1:

Input: `s = "1 + 1"`
Output: 2

Example 2:

Input: `s = " 2-1 + 2 "`
Output: 3

Example 3:

Input: `s = "(1+(4+5+2)-3)+(6+8)"`
Output: 23

Time and Space Complexity:

Time O(n)

Space O(1)

```

import java.util.Stack;

public class Solution {
    public int calculate(String s) {
        Stack<Integer> stack = new Stack<>();
        int result = 0;
        int number = 0;
        int sign = 1;
        for (int i = 0; i < s.length(); i++) {
            char c = s.charAt(i);
            if (Character.isDigit(c)) {
                number = number * 10 + (c - '0');
            } else if (c == '+') {
                result += sign * number;
                number = 0;
                sign = 1;
            } else if (c == '-') {
                result += sign * number;
                number = 0;
                sign = -1;
            } else if (c == '(') {
                stack.push(result);
                stack.push(sign);
                result = 0;
                sign = 1;
            } else if (c == ')') {
                result += sign * number;
                number = 0;
                result *= stack.pop();
                result += stack.pop();
            }
        }
        if (number != 0) {
            result += sign * number;
        }
        return result;
    }
}

```

12. 2390. Removing Stars From a String

Medium

You are given a string `s`, which contains stars `*`.

In one operation, you can:

- Choose a star in `s`.
- Remove the closest non-star character to its left, as well as remove the star itself.

Return the string after all stars have been removed.

Note:

- The input will be generated such that the operation is always possible.
- It can be shown that the resulting string will always be unique.

Example 1:

```

Input: s = "leet**cod*e"
Output: "lecoe"
Explanation: Performing the removals from left to right:
- The closest character to the 1st star is 't' in "leet**cod*e". s becomes
"lee*cod*e".
- The closest character to the 2nd star is 'e' in "lee*cod*e". s becomes
"lecod*e".
- The closest character to the 3rd star is 'd' in "lecod*e". s becomes "lecoe".
There are no more stars, so we return "lecoe".

```

Time and Space Complexity:

Time O(n)

Space O(1)

```

import java.util.*;

class Solution {
    public String removeStars(String s) {
        Deque<Integer> stack = new ArrayDeque<>();

        for (int i = 0; i < s.length(); i++) {
            char ch = s.charAt(i);
            if (ch == '*') {
                if (!stack.isEmpty()) {
                    stack.pop();
                }
            } else {
                stack.push(i);
            }
        }

        StringBuilder result = new StringBuilder();
        while (!stack.isEmpty()) {
            result.append(s.charAt(stack.pop()));
        }

        return result.reverse().toString();
    }
}

```

13. 2751. Robot Collisions

There are n 1-indexed robots, each having a position on a line, health, and movement direction.

You are given 0-indexed integer arrays `positions`, `healths`, and a string `directions` (`directions[i]` is either 'L' for left or 'R' for right). All integers in `positions` are unique.

All robots start moving on the line simultaneously at the same speed in their given directions. If two robots ever share the same position while moving, they will collide.

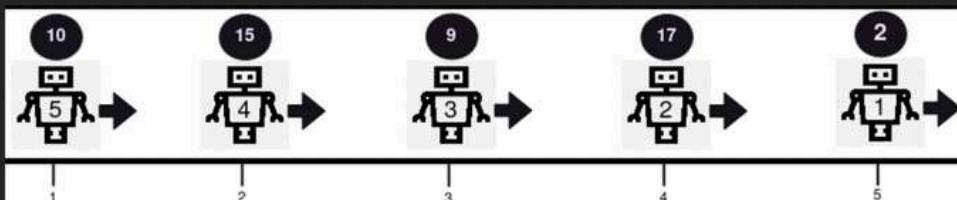
If two robots collide, the robot with lower health is removed from the line, and the health of the other robot decreases by one. The surviving robot continues in the same direction it was going. If both robots have the same health, they are both removed from the line.

Your task is to determine the health of the robots that survive the collisions, in the same order that the robots were given, i.e. final health of robot 1 (if survived), final health of robot 2 (if survived), and so on. If there are no survivors, return an empty array.

Return an array containing the health of the remaining robots (in the order they were given in the input), after no further collisions can occur.

Note: The positions may be unsorted.

Example 1:



Input: `positions = [5,4,3,2,1]`, `healths = [2,17,9,15,10]`, `directions = "RRRRR"`

Output: `[2,17,9,15,10]`

Explanation: No collision occurs in this example, since all robots are moving in the same direction. So, the health of the robots in order from the first robot is returned, `[2, 17, 9, 15, 10]`.

```

class Robot {
    public int index;
    public int position;
    public int health;
    public char direction;
}

class Solution {
    public List<Integer> survivedRobotsHealths(int[] positions, int[] healths,
                                                String directions) {
        List<Integer> ans = new ArrayList<>();
        Robot[] robots = new Robot[positions.length];
        List<Robot> stack = new ArrayList<>();
        for (int i = 0; i < positions.length; ++i)
            robots[i] = new Robot(i, positions[i], healths[i], directions.charAt(i));

        Arrays.sort(robots, (a, b) -> a.position - b.position);

        for (Robot robot : robots) {
            if (robot.direction == 'R') {
                stack.add(robot);
                continue;
            }
            while (!stack.isEmpty() && stack.get(stack.size() - 1).direction == 'R'
                   && robot.health > 0) {
                if (stack.get(stack.size() - 1).health == robot.health) {
                    stack.remove(stack.size() - 1);
                    robot.health = 0;
                } else if (stack.get(stack.size() - 1).health < robot.health) {
                    stack.remove(stack.size() - 1);
                    robot.health -= 1;
                } else {
                    stack.get(stack.size() - 1).health -= 1;
                    robot.health = 0;
                }
            }
            if (robot.health > 0)
                stack.add(robot);
        }

        stack.sort((a, b) -> a.index - b.index);

        for (Robot robot : stack)
            ans.add(robot.health);
    }

    return ans;
}

```

Time and Space Complexity:

Time O(n)

Space O(1)

10. Sorting

Comparison Based Sorting

- $O(n^2)$
- 1. Bubble
- 2. Insertion
- 3. Selection
- $O(n \log n)$
- 4. Merge
- 5. Quick
- 6. Heap
- 7. Tim
- 8. Tree
- $O(n^{1.5})$
- 9. Shell
- $O(n)$

10. Quick Select (selection version of Quick Sort)

Comparison-based sorting algorithms are those that determine the order of elements by comparing them to one another, typically using relational operators like $<$, $>$, \leq , \geq .

Index Based Sorting

- $O(n)$
- 1. Count
- 2. Bucket/Bin
- 3. Radix

Criteria For Analysis:

1. Number of comparisons
2. Number of swaps
3. Adaptive
4. Stable
5. Extra Memory

Algorithm	Stable	Adaptive	Best Case Time
Insertion Sort	Yes	Yes	$O(n)$
Tim Sort (used in Python, Java)	Yes	Yes	$O(n)$
Merge Sort (optimized)	Yes	With natural merges	$O(n \log n)$ to $O(n)$
Bubble Sort (optimized)	Yes	Yes	$O(n)$

Comparison Based Sorting

1. Bubble Sort

It is one of the simplest sorting algorithms. It repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated until the list is sorted.

Number of passes: $(n-1)$

Number of comparisons: $n*(n-1)/2$: $O(n^2)$

Maximum number of swaps: $n*(n-1)/2$: $O(n^2)$

Stone is heavier, will settle at the bottom, bubbles are lighter they will raise up. Bubble sort is adaptive and stable.

Stable means relative order will be maintained in the sorting process

Adaptive: If no swaps happen in a pass, it means the array is already sorted, and we can stop early — avoiding unnecessary work.

With the adaptive check (using a swapped flag), Bubble Sort will:

- Perform only one pass and exit early.
- Time complexity becomes $O(n)$ instead of $O(n^2)$.

```
public class Solution {  
  
    public static void bubbleSort(int[] arr) {  
        int n = arr.length;  
        boolean swapped;  
  
        for (int i = 0; i < n - 1; i++) {  
            swapped = false;  
  
            for (int j = 0; j < n - i - 1; j++) {  
                if (arr[j] > arr[j + 1]) {  
                    int temp = arr[j];  
                    arr[j] = arr[j + 1];  
                    arr[j + 1] = temp;  
                    swapped = true;  
                }  
            }  
  
            if (!swapped) break;  
        }  
  
        public static void main(String[] args) {  
            int[] arr = {5, 3, 8, 4, 2};  
            System.out.println("Original array:");  
            System.out.println(Arrays.toString(arr)); // [5, 3, 8, 4, 2]  
            bubbleSort(arr);  
            System.out.println("Sorted array:");  
            System.out.println(Arrays.toString(arr)); // [2, 3, 4, 5, 8]  
        }  
    }  
}
```

Bubble Sort Time & Space Complexity

Scenario	Time Complexity	Explanation
Best Case	<input checked="" type="checkbox"/> $O(n)$	If the array is already sorted. The <code>swapped</code> flag avoids extra passes.
Average Case	<input type="checkbox"/> $O(n^2)$	Every element might be compared with every other.
Worst Case	<input type="checkbox"/> $O(n^2)$	Reversed array – max number of swaps/comparisons.

Time and Space Complexity:

Time $O(n^2)$

Space $O(1)$

2. Insertion Sort

Insertion Sort ($O(n^2)$, best-case $O(n)$): Builds a sorted array by inserting elements at the right position.

Insertion Sort Time Complexity

It is **Adaptive** and **Stable**

K passes Usage: No

Linked List Usage: Yes

Min Comparisons: $O(n)$

Max Comparisons: $O(n^2)$

Min Swaps: $O(1)$

Max Swaps: $O(n^2)$

Case	Time Complexity	Explanation
<input checked="" type="checkbox"/> Best Case	$O(n)$	When the array is already sorted. Only comparisons, no shifting.
<input type="checkbox"/> Worst Case	$O(n^2)$	When the array is reverse sorted. Each new element must be compared and shifted all the way to the front.
<input type="checkbox"/> Average Case	$O(n^2)$	Random order – about half of the previous elements need shifting.

Number of passes: (n-1)

Number of comparisons: $n*(n-1)/2$: O(n^2)

Maximum number of swaps: $n*(n-1)/2$: O(n^2)

Insertion Sort is a simple sorting algorithm that builds the final sorted array one element at a time – just like sorting cards in your hand.

Place the elements greater than key to the right of window under consideration and move the element to the start

```
public class Solution {  
  
    public static void insertionSort(int[] arr) {  
        int n = arr.length;  
  
        for (int i = 1; i < n; i++) {  
            int key = arr[i];  
            int j = i - 1;  
  
            // Move elements greater than key one position ahead  
            while (j >= 0 && arr[j] > key) {  
                arr[j + 1] = arr[j];  
                j--;  
            }  
  
            // Place the key in its correct position  
            arr[j + 1] = key;  
        }  
    }  
  
    public static void main(String[] args) {  
        int[] arr = {5, 3, 8, 4, 2};  
        System.out.println("Original array:");  
        System.out.println(Arrays.toString(arr)); // [5, 3, 8, 4, 2]  
        insertionSort(arr);  
        System.out.println("Sorted array:");  
        System.out.println(Arrays.toString(arr)); // [2, 3, 4, 5, 8]  
    }  
}
```

Time and Space Complexity:

Time O(n^2)

Space O(1)

3. Selection Sort

Find the minimum element from the unsorted part of the array and swap it with the first element of that unsorted part.

Time and Space Complexity:

Time $O(n^2)$

Space $O(1)$

```
public class Solution {

    public static void selectionSort(int[] arr) {
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            int minIndex = i;

            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }

            int temp = arr[i];
            arr[i] = arr[minIndex];
            arr[minIndex] = temp;
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 3, 8, 4, 2};
        System.out.println("Original array:");
        System.out.println(Arrays.toString(arr)); // [5, 3, 8, 4, 2]
        selectionSort(arr);
        System.out.println("Sorted array:");
        System.out.println(Arrays.toString(arr)); // [2, 3, 4, 5, 8]
    }
}
```

How it Works

For `arr = [64, 25, 12, 22, 11]`

Step-by-step:

- $i=0$: $\min=11 \rightarrow \text{swap } 11 \& 64 \rightarrow [11, 25, 12, 22, 64]$
- $i=1$: $\min=12 \rightarrow \text{swap } 12 \& 25 \rightarrow [11, 12, 25, 22, 64]$
- $i=2$: $\min=22 \rightarrow \text{swap } 22 \& 25 \rightarrow [11, 12, 22, 25, 64]$
- $i=3$: already sorted
- $i=4$: done!

Case	Time Complexity	Explanation
✗ Best Case	$O(n^2)$	Still scans the unsorted array for the minimum every time, even if already sorted.
✗ Worst Case	$O(n^2)$	Same as above – selection sort always does $(n-1)/2$ comparisons regardless of order.
▢ Average Case	$O(n^2)$	Constant comparison pattern across all cases.

Stable ✗

Adaptive ✗

O (n. log n)

4. **Merge Sort (O(n log n))**: Uses divide and conquer to split the array and merge sorted halves.

```
public class Solution {

    public static void mergeSort(int[] arr, int left, int right) {
        if (left < right) {
            int mid = (left + right) / 2;
            mergeSort(arr, left, mid);
            mergeSort(arr, left: mid + 1, right);
            merge(arr, left, mid, right);
        }
    }

    public static void merge(int[] arr, int left, int mid, int right) {
        int n1 = mid - left + 1;
        int n2 = right - mid;

        int[] L = new int[n1];
        int[] R = new int[n2];

        for (int i = 0; i < n1; ++i)
            L[i] = arr[left + i];
        for (int j = 0; j < n2; ++j)
            R[j] = arr[mid + 1 + j];

        int i = 0, j = 0, k = left;
        while (i < n1 && j < n2) {
            if (L[i] <= R[j]) {
                arr[k++] = L[i++];
            } else {
                arr[k++] = R[j++];
            }
        }

        while (i < n1) arr[k++] = L[i++];
        while (j < n2) arr[k++] = R[j++];
    }

    public static void main(String[] args) {
        int[] arr = {38, 27, 43, 3, 9, 82, 10};
        System.out.println("Original array:");
        System.out.println(Arrays.toString(arr)); // [38, 27, 43, 3, 9, 82, 10]
        mergeSort(arr, left: 0, right: arr.length - 1);
        System.out.println("Sorted array:");
        System.out.println(Arrays.toString(arr)); // [3, 9, 10, 27, 38, 43, 82]
    }
}
```

✓ Time Complexity

Best Case: O(n)

Average Case: O(n log n)

Worst Case: O(n log n)

T(n) = O(n log n)

✓ Space Complexity

Space Complexity = O(n)

Merge Sort is a Divide and Conquer algorithm. It divides the array into halves, sorts each half recursively, and then merges the sorted halves.

```
public static void mergeSort(int[] arr, int left, int right) {  
    if (left < right) {  
        int mid = (left + right) / 2;  
        mergeSort(arr, left, mid);          // Recursive call on left half  
        mergeSort(arr, mid + 1, right);    // Recursive call on right half  
        merge(arr, left, mid, right);      // Merge the sorted halves  
    }  
}  
  
mergeSort(0, 3)  
└── mergeSort(0, 1)  
    ├── mergeSort(0, 0)  // single element, return  
    ├── mergeSort(1, 1)  // single element, return  
    └── merge(0, 0, 1)   // merge [4] and [2] => [2, 4]  
  
└── mergeSort(2, 3)  
    ├── mergeSort(2, 2)  // single element, return  
    ├── mergeSort(3, 3)  // single element, return  
    └── merge(2, 2, 3)   // merge [1] and [3] => [1, 3]  
  
└── merge(0, 1, 3)      // merge [2, 4] and [1, 3] => [1, 2, 3, 4]
```

Merge left and right half and merge the sorted halves.

Two Sorted Arrays:

```
public class Solution {  
  
    public static int[] mergeSortedArrays(int[] A, int[] B) {  
        int n = A.length, m = B.length;  
        int[] result = new int[n + m];  
        int i = 0, j = 0, k = 0;  
  
        while (i < n && j < m) {  
            if (A[i] <= B[j]) {  
                result[k++] = A[i++];  
            } else {  
                result[k++] = B[j++];  
            }  
        }  
  
        while (i < n) result[k++] = A[i++];  
        while (j < m) result[k++] = B[j++];  
  
        return result;  
    }  
  
    public static void main(String[] args) {  
        int[] A = {1, 3, 5, 7};  
        int[] B = {2, 4, 6, 8};  
        int[] merged = mergeSortedArrays(A, B);  
  
        System.out.println("Merged array:");  
        for (int num : merged) {  
            System.out.print(num + " ");  
        }  
    }  
}
```

✓ Time Complexity

$O(n + m)$

✓ Space Complexity

$O(n + m)$

```

public class Solution {

    public static int[] mergeSort(int[] arr) {
        int n = arr.length;
        if (n <= 1) return arr;

        int mid = n / 2;

        int[] left = Arrays.copyOfRange(arr, 0, mid);
        int[] right = Arrays.copyOfRange(arr, mid, n);
        left = mergeSort(left);
        right = mergeSort(right);
        return merge(left, right);
    }

    public static int[] merge(int[] left, int[] right) {
        int[] merged = new int[left.length + right.length];
        int i = 0, j = 0, k = 0;

        while (i < left.length && j < right.length) {
            if (left[i] <= right[j]) {
                merged[k++] = left[i++];
            } else {
                merged[k++] = right[j++];
            }
        }

        while (i < left.length) merged[k++] = left[i++];
        while (j < right.length) merged[k++] = right[j++];

        return merged;
    }
}

```

Four Sorted Arrays:

```

public class Solution {

    public static int[] mergeTwoSortedArrays(int[] A, int[] B) {
        int n = A.length, m = B.length;
        int[] result = new int[n + m];
        int i = 0, j = 0, k = 0;

        while (i < n && j < m) {
            result[k++] = (A[i] <= B[j]) ? A[i++] : B[j++];
        }
        while (i < n) result[k++] = A[i++];
        while (j < m) result[k++] = B[j++];
        return result;
    }

    public static void main(String[] args) {
        int[] A = {1, 5, 9};
        int[] B = {2, 6, 10};
        int[] C = {3, 7, 11};
        int[] D = {4, 8, 12};

        int[] mergedAB = mergeTwoSortedArrays(A, B);
        int[] mergedCD = mergeTwoSortedArrays(C, D);
        int[] finalMerged = mergeTwoSortedArrays(mergedAB, mergedCD);

        System.out.println("Final Merged Array:");
        System.out.println(Arrays.toString(finalMerged)); // [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
    }
}

```

✓ Java Code Snippet (Merge Sort with Optimization)

```

java

public static void mergeSort(int[] arr, int left, int right) {
    if (left < right) {
        int mid = (left + right) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        // Optimization: skip merge if already sorted
        if (arr[mid] <= arr[mid + 1]) {
            return;
        }

        merge(arr, left, mid, right);
    }
}

```

✓ When Does This Help?

- When the array is **already sorted** or **nearly sorted**.
- Example:
 - Input: [1, 2, 3, 4, 5]
 - Without optimization: still $O(n \log n)$
 - With optimization: only one pass needed $\Rightarrow O(n)$

5. QuickSort (Selection Exchange/Partition Exchange Sort) ($O(n \log n)$, worst-case $O(n^2)$):

Uses divide and conquer to split the array and merge sorted halves.

Line of members with different heights standing in a straight line.

Quick Sort is another Divide and Conquer algorithm. It picks a pivot element, partitions the array around the pivot (placing smaller elements on the left and larger on the right), and then recursively sorts the partitions.

```
quickSort(0, 4)
└─ partition with pivot → returns pivotIndex = 2
└─ quickSort(0, 1)
    └─ partition → pivotIndex = 0
    └─ quickSort(0, -1) → base case
    └─ quickSort(1, 1) → base case
└─ quickSort(3, 4)
    └─ partition → pivotIndex = 4
    └─ quickSort(3, 3) → base case
    └─ quickSort(5, 4) → base case
```

```
public class Solution {

    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pivotIndex = partition(arr, low, high);
            quickSort(arr, low, high: pivotIndex - 1); // Sort left part
            quickSort(arr, low: pivotIndex + 1, high); // Sort right part
        }
    }

    private static int partition(int[] arr, int low, int high) {
        int pivot = arr[high]; // take the last element as pivot
        int i = low - 1; // pointer for the smaller element
        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                i++;
                // swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        // swap pivot to its correct sorted position
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
        arr[high] = temp;
        return i + 1; // return index of pivot after sorting
    }

    public static void main(String[] args) {
        int[] arr = { 4, 2, 7, 1, 3, 6, 5 };
        System.out.println("Original array:");
        System.out.println(Arrays.toString(arr));
        quickSort(arr, low: 0, high: arr.length - 1);
        System.out.println("Sorted array:");
        System.out.println(Arrays.toString(arr)); // [1, 2, 3, 4, 5, 6, 7]
    }
}
```

💡 What it does:

- Picks **pivot = arr[high]**.
- **i** tracks the boundary between smaller and larger elements.
- For every **j** from **low** to **high - 1**, it checks if **arr[j] <= pivot**:
 - If yes, increment **i**, swap **arr[i]** and **arr[j]**.
- Finally, place pivot at the correct spot **i + 1**.

Case	Complexity	Explanation
Best Case	$O(n \log n)$	Pivot divides array into two roughly equal halves.
Average Case	$O(n \log n)$	Random distribution of elements, balanced partitions on average.
Worst Case	$O(n^2)$	Occurs when the pivot is the smallest or largest element every time (e.g., sorted or reverse-sorted array).

- At each level of recursion, we process all **n** elements for partitioning.
- The number of levels = height of recursion tree:
 - Best/average: $\log_2(n)$
 - Worst: n (skewed recursion tree)

To avoid worst case, randomized pivot or median-of-three pivot selection is used.

Move elements smaller than pivot to the left and larger than pivot to the right and keep on doing the same

🧠 Dry Run Example

Initial array:

```
csharp Copy Edit  
[4, 2, 7, 1, 3, 6, 5]
```

Step 1:

- **pivot = 5** (last element)
- Elements $<= 5$ are moved to the left → after partition: [4, 2, 1, 3, 5, 6, 7]
- Pivot **5** placed at index **4**

Then recursive calls:

- **quickSort(arr, 0, 3)** on [4, 2, 1, 3]
- **quickSort(arr, 5, 6)** on [6, 7]

Continue recursively until all subarrays are sorted.

6. Randomized Quick Sort $(O(n \log n))$

🎲 Randomized Quick Sort Enhancement

Instead of picking a fixed pivot (e.g., first or last element), pick a random index between **low** and **high**, and use that as the pivot. This randomization ensures:

- The input pattern doesn't cause degradation
- Average time complexity remains $O(n \log n)$ in practice

🧠 Intuition Behind Random Pivot

Imagine sorting this array:

```
java Copy Edit  
int[] arr = {4, 5, 1, 2, 3};
```

If you always pick the **first element** as pivot ($\rightarrow 4$), you might end up with unbalanced partitions repeatedly.

But if you pick a **random pivot**, say **3**, it divides the array into a more balanced partition.

```

import java.util.Random;
public class RandomizedQuickSort {
    Random rand = new Random();

    public void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            int pivotIndex = randomizedPartition(arr, low, high);
            quickSort(arr, low, pivotIndex - 1);
            quickSort(arr, pivotIndex + 1, high);
        }
    }

    private int randomizedPartition(int[] arr, int low, int high) {
        int pivotIdx = low + rand.nextInt(high - low + 1); // random index between low
        swap(arr, pivotIdx, high); // move random pivot to end
        return partition(arr, low, high); // normal partition
    }

    private int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = low - 1; // smaller element index

        for (int j = low; j < high; j++) {
            if (arr[j] < pivot) {
                i++;
                swap(arr, i, j);
            }
        }

        swap(arr, i + 1, high); // place pivot in correct position
        return i + 1;
    }

    private void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}

```

⌚ Time Complexity

Case	Time
Best / Avg	$O(n \log n)$
Worst	$O(n^2)$ (rare due to randomization)

But due to randomness, the **worst case is unlikely**, making it more robust.

✓ Advantages

- Avoids worst-case performance on sorted or nearly sorted data
- Performs better on average than deterministic quicksort

6. Heap Sort (O(n log n))

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to sort elements. It's an in-place algorithm with O(n log n) time complexity for all cases (best, average, and worst). It's not stable, but it's efficient and widely used in system-level sorting.

How Heap Sort Works

1. Build a Max Heap (or Min Heap for descending order):

- Convert the unsorted array into a max heap where the largest element is at the root.

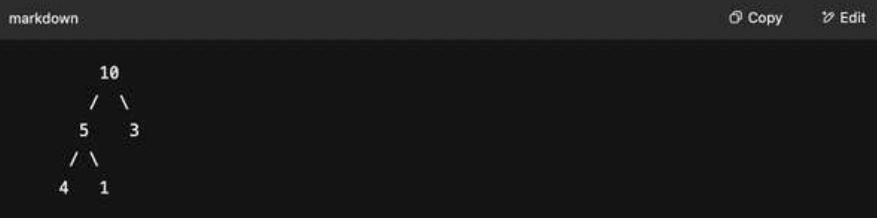
2. Heapify and Extract:

- Swap the root (maximum element) with the last element.
- Reduce the heap size by one (exclude the last element from the heap).
- Heapify the root again to maintain the heap property.
- Repeat until the heap is empty.

Step-by-Step Example

Input: [4, 10, 3, 5, 1]

Convert to Max Heap:



Array form: [10, 5, 3, 4, 1]

Sorting Steps:

- Swap 10 and 1 → [1, 5, 3, 4, 10]
- Heapify → [5, 4, 3, 1, 10]
- Swap 5 and 1 → [1, 4, 3, 5, 10]
- Heapify → [4, 1, 3, 5, 10]
- Swap 4 and 3 → [3, 1, 4, 5, 10]
- Heapify → [3, 1, 4, 5, 10]
- Swap 3 and 1 → [1, 3, 4, 5, 10]

Final Sorted Array: [1, 3, 4, 5, 10]

```
public class Solution {  
  
    public static void heapSort(int[] arr) {  
        int n = arr.length;  
  
        // Step 1: Build Max Heap  
        for (int i = n / 2 - 1; i >= 0; i--) {  
            heapify(arr, n, i);  
        }  
  
        // Step 2: Extract elements from heap one by one  
        for (int i = n - 1; i > 0; i--) {  
            // Move current root (largest) to end  
            int temp = arr[0];  
            arr[0] = arr[i];  
            arr[i] = temp;  
  
            // Call max heapify on the reduced heap  
            heapify(arr, i, 0);  
        }  
    }  
}
```

```

// Heapify a subtree rooted at index 'i' in an array of size 'n'
private static void heapify(int[] arr, int n, int i) {
    int largest = i;           // Initialize largest as root
    int left = 2 * i + 1;      // left = 2*i + 1
    int right = 2 * i + 2;     // right = 2*i + 2

    // If left child is larger than root
    if (left < n && arr[left] > arr[largest]) {
        largest = left;
    }

    // If right child is larger than largest so far
    if (right < n && arr[right] > arr[largest]) {
        largest = right;
    }

    // If largest is not root
    if (largest != i) {
        int swap = arr[i];
        arr[i] = arr[largest];
        arr[largest] = swap;

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

public static void main(String[] args) {
    int[] arr = { 4, 2, 7, 1, 3, 6, 5 };
    System.out.println("Original array:");
    System.out.println(Arrays.toString(arr));
    heapSort(arr);
    System.out.println("Sorted array:");
    System.out.println(Arrays.toString(arr)); // [1, 2, 3, 4, 5, 6, 7]
}
}

```

✗ Time and Space Complexity

Case	Time Complexity
Best	$O(n \log n)$
Average	$O(n \log n)$
Worst	$O(n \log n)$

- **Space:** $O(1)$ (in-place)
- **Stable:** ✗ No

⌚ How heapify works (Max-Heap)

- Given a node at index i in array $\text{arr}[]$, and assuming its subtrees are already heaps, you:
- Compare $\text{arr}[i]$ with its left and right children.
- Swap with the largest of the three if needed.
- Recursively heapify the affected subtree.

⚡ What's happening:

- This loop converts the array into a **max heap**.
- It starts from the last **non-leaf node** ($n / 2 - 1$) and goes up to the root ($i = 0$).
- For each node, it calls the `heapify` function, which ensures that the subtree rooted at that index satisfies the max heap property.

✗ Why $n / 2 - 1$?

- In a binary heap stored as an array:
 - All elements from index $n/2$ to $n-1$ are leaf nodes (no children), so we skip them.
 - We only need to heapify the **internal nodes**.

⚡ What's happening:

- The root of the max heap (`arr[0]`) is the largest element.
- It is **swapped with the last element** (`arr[i]`) and then **excluded** from the heap by reducing the size (i becomes the new size).
- After swapping, the heap may be **broken**, so we **heapify** from the root (`index 0`) to fix it.

⌚ This repeats until the entire array is sorted.

7. Tim Sort (O(n log n))

Tim Sort is a hybrid sorting algorithm used in Python (`sorted()` function) and Java (`Arrays.sort` for objects). It's designed to perform well on real-world data by combining:

- **Insertion Sort** – for small, nearly sorted segments (very efficient on such data).
- **Merge Sort** – for combining the segments efficiently.

✓ Key Idea:

It divides the array into small pieces (called **runs**) and:

1. Sorts each **run** using Insertion Sort.
2. Merges the runs together using Merge Sort logic.

☰ Why Use Both Insertion Sort and Merge Sort?

1. Insertion Sort for Small Runs (Chunks)

- Reason: Insertion Sort is very fast for small arrays, especially when the array is already partially sorted, which often happens in real-world data.
- Tim Sort breaks the array into small blocks (e.g., size 32) and sorts each block using Insertion Sort first.

Benefits of Insertion Sort:

- "Fast overhead (no recursion or extra memory)"
- "Cache-friendly"
- "Best case O(n) when array is nearly sorted"

2. Merge Sort for Combining Runs

- Once all small chunks are individually sorted via Insertion Sort,
- Merge Sort is used to combine them in a bottom-up fashion.

Benefits of Merge Sort:

- "Stable sort"
- "Handles large datasets efficiently"
- "Combines sorted runs in O(n log n) time"

⚡ Why use Insertion Sort on small pieces?

Because:

- Insertion Sort is super fast on **small or nearly sorted arrays**.
- Real-world data often has some **pre-existing order**.

⚙ How Tim Sort works:

1. Define a **Run**: Usually of size **32 or 64** (configurable).
2. Sort all **runs** using **Insertion Sort**.
3. Merge runs using **Merge Sort** until the whole array is sorted.

✓ Key Idea:

It divides the array into small pieces (called **runs**) and:

1. Sorts each **run** using Insertion Sort.
2. Merges the runs together using Merge Sort logic.

↖ Why use Insertion Sort on small pieces?

Because:

- Insertion Sort is super fast on **small or nearly sorted arrays**.
- Real-world data often has some **pre-existing order**.

⚙ How Tim Sort works:

1. Define a Run: Usually of size **32 or 64** (configurable).
2. Sort all runs using Insertion Sort.
3. Merge runs using Merge Sort until the whole array is sorted.

🕒 Time Complexity:

Case	Time
Best Case	$O(n)$
Average	$O(n \log n)$
Worst Case	$O(n \log n)$

✓ Tim Sort is:

Property	Status
Stable	✓ Yes
Adaptive	✓ Yes
In-place	✗ No (needs extra space during merge)

💡 Fun Fact:

Tim Sort was invented by Tim Peters (a Python core developer) and is the default sort algorithm in Python, Java (for objects), and Android.

```

import java.util.Arrays;

public class TimSort {

    static final int RUN_SIZE = 32;

    // Insertion sort for small chunks
    public static void insertionSort(int[] array, int left, int right) {
        for (int current = left + 1; current <= right; current++) {
            int temp = array[current];
            int j = current - 1;

            while (j >= left && array[j] > temp) {
                array[j + 1] = array[j];
                j--;
            }
            array[j + 1] = temp;
        }
    }

    // Merge two sorted subarrays: array[left..mid] and array[mid+1..right]
    public static void merge(int[] array, int left, int mid, int right) {
        int length1 = mid - left + 1;
        int length2 = right - mid;

        int[] leftPart = new int[length1];
        int[] rightPart = new int[length2];

        for (int i = 0; i < length1; i++) {
            leftPart[i] = array[left + i];
        }
        for (int j = 0; j < length2; j++) {
            rightPart[j] = array[mid + 1 + j];
        }

        int i = 0, j = 0, k = left;

        while (i < length1 && j < length2) {
            if (leftPart[i] <= rightPart[j]) {
                array[k++] = leftPart[i++];
            } else {
                array[k++] = rightPart[j++];
            }
        }
    }

    // TimSort main function
    public static void timSort(int[] array) {
        int n = array.length;

        // Step 1: Sort small blocks using insertion sort
        for (int start = 0; start < n; start += RUN_SIZE) {
            int end = Math.min(start + RUN_SIZE - 1, n - 1);
            insertionSort(array, start, end);
        }

        // Step 2: Merge blocks of size RUN_SIZE, doubling each pass
        for (int size = RUN_SIZE; size < n; size *= 2) {
            for (int left = 0; left < n; left += 2 * size) {
                int mid = left + size - 1;
                int right = Math.min(left + 2 * size - 1, n - 1);

                if (mid < right) {
                    merge(array, left, mid, right);
                }
            }
        }
    }

    public static void main(String[] args) {
        int[] array = {5, 21, 7, 23, 19, 18, 12, 11, 6, 9, 8};

        System.out.println("Original array:");
        System.out.println(Arrays.toString(array));
        timSort(array);

        System.out.println("Sorted array:");
        System.out.println(Arrays.toString(array));
    }
}

```

8. Tree Sort (n O(log n)))

Tree Sort is a comparison-based sorting algorithm that builds a Binary Search Tree (BST) from the input elements and performs an in-order traversal to get the sorted order.

🧠 Key Idea

1. Insert all elements into a **Binary Search Tree**.
2. **In-order traversal** of BST gives the sorted array in ascending order.

✅ Properties

Feature	Value
Stable	✗ No
Adaptive	✗ No
Best Time	$O(n \log n)$ (balanced BST)
Average Time	$O(n \log n)$
Worst Time	$O(n^2)$ (unbalanced BST)
Space	$O(n)$ (due to tree nodes)

You can improve worst-case to $O(n \log n)$ by using a self-balancing BST like AVL Tree or Red-Black Tree.

```
public class Solution {

    static class Node {
        int val;
        Node left, right;

        Node(int val) {
            this.val = val;
        }
    }

    static Node insert(Node root, int val) {
        if (root == null) return new Node(val);
        if (val < root.val) root.left = insert(root.left, val);
        else root.right = insert(root.right, val);
        return root;
    }

    static void inOrder(Node root, ArrayList<Integer> result) {
        if (root == null) return;
        inOrder(root.left, result);
        result.add(root.val);
        inOrder(root.right, result);
    }
}
```

```

    public static void treeSort(int[] arr) {
        Node root = null;
        for (int val : arr) {
            root = insert(root, val);
        }

        ArrayList<Integer> sorted = new ArrayList<>();
        inOrder(root, sorted);

        for (int i = 0; i < arr.length; i++) {
            arr[i] = sorted.get(i);
        }
    }

    public static void main(String[] args) {
        int[] arr = {5, 3, 7, 2, 4, 6, 8};
        System.out.println("Original: ");
        for (int i : arr) System.out.print(i + " "); // 5 3 7 2 4 6 8

        treeSort(arr);

        System.out.println("\nSorted: ");
        for (int i : arr) System.out.print(i + " "); // 2 3 4 5 6 7 8
    }
}

```

- Construct the tree
- Perform the inorder traversal

9. QuickSelect Sort (O(n)))

The Quick Select algorithm is a selection algorithm to find the k-th smallest (or largest) element in an unsorted array – faster than sorting the entire array.

🚀 Quick Select Overview

- Based on Quick Sort's partitioning logic.
- Instead of sorting both sides, we only recurse into the side where the k-th element **must** lie.
- **Average Time Complexity:** $O(n)$
- **Worst Time Complexity:** $O(n^2)$ (when poor pivots are chosen)
- **Space Complexity:** $O(1)$ (in-place) + recursion stack

🧠 Time and Space Complexity

Case	Time	Space
Best / Average	$O(n)$	$O(\log n)$ (recursion)
Worst	$O(n^2)$	$O(n)$
In-place	<input checked="" type="checkbox"/> Yes	

```

public class Solution {

    public static int quickSelect(int[] arr, int low, int high, int k) {
        if (low == high) {
            return arr[low];
        }

        int pivotIndex = partition(arr, low, high);
        int numLeft = pivotIndex - low + 1;

        if (k == numLeft) {
            return arr[pivotIndex];
        } else if (k < numLeft) {
            return quickSelect(arr, low, high: pivotIndex - 1, k);
        } else {
            return quickSelect(arr, low: pivotIndex + 1, high, k: k);
        }
    }

    private static int partition(int[] arr, int low, int high) {
        int pivot = arr[high];
        int i = low;
        for (int j = low; j < high; j++) {
            if (arr[j] <= pivot) {
                swap(arr, i, j);
                i++;
            }
        }
        swap(arr, i, high);
        return i;
    }

    private static void swap(int[] arr, int i, int j) {
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }

    public static void main(String[] args) {
        int[] arr = {7, 10, 4, 3, 20, 15};
        int k = 3;
        int kthSmallest = quickSelect(arr, low: 0, high: arr.length - 1, k);
        System.out.println(k + "rd smallest element is " + kthSmallest); // Output: 7
    }
}

public class QuickSelect {

    public int quickSelect(int[] arr, int k) {
        return quickSelect(arr, 0, arr.length - 1, k);
    }

    private int quickSelect(int[] arr, int left, int right, int k) {
        if (left == right) return arr[left];

        int pivotIndex = partition(arr, left, right);

        if (k == pivotIndex) {
            return arr[k];
        } else if (k < pivotIndex) {
            return quickSelect(arr, left, pivotIndex - 1, k);
        } else {
            return quickSelect(arr, pivotIndex + 1, right, k);
        }
    }

    private int partition(int[] arr, int left, int right) {
        int pivot = arr[right];
        int i = left;

        for (int j = left; j < right; j++) {
            if (arr[j] < pivot) {
                swap(arr, i, j);
                i++;
            }
        }

        swap(arr, i, right);
        return i;
    }

    private void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
}

```

Randomized Quick Select

```

import java.util.Random;

public class RandomizedQuickSelect {

    Random rand = new Random();

    public int quickSelect(int[] arr, int k) {
        return quickSelect(arr, 0, arr.length - 1, k);
    }

    private int quickSelect(int[] arr, int left, int right, int k) {
        if (left == right) return arr[left];

        int pivotIndex = left + rand.nextInt(right - left + 1);
        pivotIndex = partition(arr, left, right, pivotIndex);

        if (k == pivotIndex) {
            return arr[k];
        } else if (k < pivotIndex) {
            return quickSelect(arr, left, pivotIndex - 1, k);
        } else {
            return quickSelect(arr, pivotIndex + 1, right, k);
        }
    }

    private int partition(int[] arr, int left, int right, int pivotIndex) {
        int pivot = arr[pivotIndex];
        swap(arr, pivotIndex, right); // Move pivot to end
        int storeIndex = left;

        for (int i = left; i < right; i++) {
            if (arr[i] < pivot) {
                swap(arr, storeIndex, i);
                storeIndex++;
            }
        }

        swap(arr, storeIndex, right); // Move pivot to final place
        return storeIndex;
    }

    private void swap(int[] arr, int i, int j) {
        int tmp = arr[i];
        arr[i] = arr[j];
        arr[j] = tmp;
    }
}

```

Time Complexity	
Case	Time
Average	$O(n)$
Worst-case	$O(n^2)$ (rare due to randomization)

10. Shell Sort (Insertion Sort With Reduced Gap) ($O(n^{1.5})$)

Shell Sort is an in-place comparison-based sorting algorithm that improves upon Insertion Sort by comparing elements that are far apart first, then gradually reducing the gap between compared elements.

It was invented by Donald Shell in 1959.

Is Shell Sort Adaptive?

Yes, but partially.

Shell Sort is more adaptive than simple sorts like Bubble or Insertion Sort, because it takes advantage of partial order in the input.

It becomes faster on data that's partially sorted, especially for good gap sequences.

⌚ Time and Space Complexity

Case	Time Complexity
Best	$O(n \log n)$ (depends on gap sequence)
Average	$O(n^{1.25})$ (depends on gap)
Worst	$O(n^2)$ (basic gap sequence)
Space	$O(1)$ (in-place)
Stable	✗ No

```

public class ShellSort {
    Saved memory full ⓘ
    ⌂ Copy ⌂ Edit

    public static void shellSort(int[] arr) {
        int n = arr.length;

        // Start with a big gap, then reduce the gap
        for (int gap = n / 2; gap > 0; gap /= 2) {
            // Do a gapped insertion sort
            for (int i = gap; i < n; i++) {
                int temp = arr[i];
                int j = i;

                // Shift earlier gap-sorted elements up until correct location found
                while (j >= gap && arr[j - gap] > temp) {
                    arr[j] = arr[j - gap];
                    j -= gap;
                }

                // Put temp (the original arr[i]) in its correct location
                arr[j] = temp;
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {9, 8, 3, 7, 5, 6, 4, 1};
        System.out.println("Original: " + Arrays.toString(arr));
        shellSort(arr);
        System.out.println("Sorted: " + Arrays.toString(arr));
    }
}

```

Example Dry Run (gap = n/2 approach):

For `arr = {9, 8, 3, 7, 5, 6, 4, 1}`

- `gap = 4` → compare and sort elements 4 apart
- `gap = 2` → compare and sort elements 2 apart
- `gap = 1` → final insertion sort pass

```

public class Solution {

    public static void insertionSort(int[] arr) {
        int n = arr.length;

        for (int i = 1; i < n; i++) {
            int key = arr[i];
            int j = i - 1;

            // Move elements greater than key one position ahead
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }

            // Place the key in its correct position
            arr[j + 1] = key;
        }
    }
}

```

Index Based Sorting

Index-based sorting refers to sorting techniques where elements are rearranged based on their positions (indices), often without comparing the actual values directly. Instead of comparing elements (like in comparison-based sorting), it uses the value as a key to place it at a specific index or bucket.

11. Count Sort

Counting Sort is a non-comparison, index-based sorting algorithm that works best for non-negative integers in a known small range.

Instead of comparing elements, it counts the number of occurrences of each value and uses those counts to determine their correct positions in the sorted array.

- Counting sort is fast for integers with small range.
- Doesn't use comparisons (non-comparison-based).
- Can be made stable, so it's used in Radix Sort as a subroutine.
- Not in-place by default due to extra space usage.

```
public static void countingSort(int[] arr) {  
    if (arr.length == 0) return;  
  
    int max = Arrays.stream(arr).max().getAsInt();  
    int min = Arrays.stream(arr).min().getAsInt();  
    int range = max - min + 1;  
    int[] count = new int[range]; // Frequency of each element  
    int[] output = new int[arr.length]; // Output sorted array  
    // Count occurrences  
    for (int i = 0; i < arr.length; i++) {  
        count[arr[i] - min]++;  
    }  
    System.out.println(Arrays.toString(count)); // [1, 2, 2, 1, 0, 0, 0, 1]  
    // Cumulative count (for stable sort)  
    for (int i = 1; i < count.length; i++) {  
        count[i] += count[i - 1];  
    }  
    System.out.println(Arrays.toString(count)); // [1, 3, 5, 6, 6, 6, 6, 7]  
    // Build the output array (stable version)  
    for (int i = arr.length - 1; i >= 0; i--) {  
        output[count[arr[i] - min] - 1] = arr[i]; // [1, 2, 2, 3, 3, 4, 8]  
        count[arr[i] - min]--;  
    }  
    System.out.println(Arrays.toString(count)); // [0, 1, 3, 5, 6, 6, 6, 6]  
    // Copy back to original array  
    for (int i = 0; i < arr.length; i++) {  
        arr[i] = output[i];  
    }  
}  
  
public static void main(String[] args) {  
    int[] arr = {4, 2, 2, 8, 3, 3, 1};  
    countingSort(arr);  
    System.out.println("Sorted Array: " + Arrays.toString(arr));  
}
```

Time & Space Complexity

Metric	Complexity
Time (Best/Average/Worst)	$O(n+k)$
Space	$O(n+k)$

n = number of elements

k = range of input values = $\max - \min + 1$

Stable Logic in Code

This part makes it stable ✨

java

Copy

Edit

```
for (int i = arr.length - 1; i >= 0; i--) {  
    output[count[arr[i] - min] - 1] = arr[i];  
    count[arr[i] - min]--;  
}
```

If we did it from left to right ($i = 0$ to $n-1$), equal elements would get overwritten in reverse order, making it unstable.

12. Bin/Bucket Sort

Bucket Sort is an index-based, non-comparison sorting algorithm that:

- Divides the input into buckets (bins) based on value ranges.
- Sorts each bucket individually (often using insertion sort or another method).
- Combines all buckets to get the final sorted array.

It works best when:

- Input is uniformly distributed across a range.
- You want $O(n)$ average time performance.

Concept

1. Create buckets (arrays or lists).
2. Distribute elements into the appropriate bucket based on a mapping function.
3. Sort each bucket individually.
4. Concatenate all buckets in order.

Time and Space Complexity

Case	Time Complexity
Best	$O(n + k)$
Average	$O(n + k)$
Worst	$O(n^2)$ (if all go to one bucket)

 = number of buckets

- **Space:** $O(n + k)$
- **Stable:** Yes, if the bucket sort inside is stable
- **Comparison-based:**  Not in essence, but uses comparisons inside buckets

```

public class Solution {

    public static void bucketSort(float[] arr) {
        int n = arr.length;
        if (n <= 0)
            return;

        List<Float>[] buckets = new ArrayList[n];
        for (int i = 0; i < n; i++)
            buckets[i] = new ArrayList<>();

        for (float val : arr) {
            int bucketIndex = (int)(val * n);
            buckets[bucketIndex].add(val);
        }

        for (List<Float> bucket : buckets) {
            Collections.sort(bucket);
        }

        int index = 0;
        for (List<Float> bucket : buckets) {
            for (float val : bucket) {
                arr[index++] = val;
            }
        }
    }

    public static void main(String[] args) {
        float[] arr = { 0.42f, 0.32f, 0.23f, 0.52f, 0.25f, 0.47f, 0.51f };
        System.out.println("Original: " + Arrays.toString(arr));
        bucketSort(arr);
        System.out.println("Sorted: " + Arrays.toString(arr));
    }
}

```

13. Radix Sort

Radix Sort is a non-comparison-based integer sorting algorithm that:

- Processes numbers digit by digit (starting from least significant digit to most significant).
- Uses a stable sort (typically Counting Sort) as a subroutine for sorting digits.

✓ Key Points:

- Stable: ✓ (if the digit sort is stable)
- Adaptive: ✗
- Comparison-based: ✗ (uses digit indexing)
- Best used for: integers or fixed-length strings

💡 How it works (LSD - Least Significant Digit first)

For each digit position (starting from units):

1. Use a **stable sorting algorithm** (like Counting Sort) to sort by that digit.
2. Move to the next significant digit (tens, hundreds...).
3. Repeat until the highest digit is sorted.

⌚ Time and Space Complexity

Case	Time Complexity
Best	$O(nk)$
Average	$O(nk)$
Worst	$O(nk)$

Where:

- n = number of elements
- k = number of digits (logarithmic in max value)

Space: $O(n + k)$

```
public class Solution {

    static int getMax(int[] arr) {
        int max = arr[0];
        for (int value : arr)
            if (value > max)
                max = value;
        return max;
    }

    static void countingSort(int[] arr, int exp) {
        int n = arr.length;
        int[] output = new int[n];
        int[] count = new int[10];

        for (int i = 0; i < n; i++) {
            int digit = (arr[i] / exp) % 10;
            count[digit]++;
        }

        for (int i = 1; i < 10; i++)
            count[i] += count[i - 1];

        for (int i = n - 1; i >= 0; i--) {
            int digit = (arr[i] / exp) % 10;
            output[count[digit] - 1] = arr[i];
            count[digit]--;
        }

        for (int i = 0; i < n; i++)
            arr[i] = output[i];
    }
}
```

```

public static void radixSort(int[] arr) {
    int max = getMax(arr);
    for (int exp = 1; max / exp > 0; exp *= 10)
        countingSort(arr, exp);
}

public static void main(String[] args) {
    int[] arr = {170, 45, 75, 90, 802, 24, 2, 66};
    System.out.println("Original array: " + Arrays.toString(arr));
    radixSort(arr);
    System.out.println("Sorted array: " + Arrays.toString(arr));
}
}

```

Algorithm	Best	Average	Worst	Space	Stable	Adaptive	Linked List Friendly	Use Cases / When to Use
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓	✗	Educational purposes, small or nearly sorted arrays
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	✗	✗	✗	Simplicity, tight memory, small datasets
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$	✓	✓	✓	Best for small/partially sorted or linked lists
Tim Sort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	✓	✓	✗	Java/Python built-in sort; real-world datasets
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	✗	✗	✗	General purpose, best average-case performance
Shell Sort	$O(n \log n)$	$\sim O(n^{1.3})$	$O(n^2)$	$O(1)$	✗	✓ (partially)	✗	Larger datasets, low space requirement
Quick Select	$O(n)$	$O(n)$	$O(n^2)$	$O(1)$	✗	✗	✗	Find kth smallest/largest efficiently
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	✓	✗	✓	Linked lists, external sorting, stable sort required
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	✗	✗	✗	Memory-constrained systems, priority queues
Count Sort	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	✓*	✗	✗	Integers in limited range, can be made stable
Bucket Sort	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	✓	✗	✗	Uniformly distributed decimal/floating-point numbers
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n + k)$	✓	✗	✗	Fixed-length integers, when counting sort is suitable
Tree Sort (BST)	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(n)$	✗	✗	✓ (with self-balancing)	BST logic reused, searching + sorting in one pass

Binary Tree

Advanced Operations

- **Binary Search Tree (BST) Operations**

- **Insert:** Similar to binary search, insert at the appropriate position.
- **Delete:** Replace the node with either its largest left subtree node or smallest right subtree node.
- **Search:** Similar to binary search on a sorted array.

- **Tree Transformations**

- **Balanced Tree:** AVL Tree, Red-Black Tree.
- **Binary Heap:** Min-Heap, Max-Heap used in priority queues.
- **Expression Tree:** Used in compilers for evaluating expressions.

1. General Binary Tree Complexities

- **Traversal (Inorder, Preorder, Postorder)**: $O(n)$ time, $O(h)$ space
- **Insertion (Level Order)**: $O(n)$ time, $O(1)$ space
- **Deletion (Deepest Node Replacement)**: $O(n)$ time, $O(1)$ space
- **Searching**: $O(n)$ time, $O(1)$ space

2. Binary Search Tree (BST) Complexities

- **Best Case (Balanced BST)** - $O(\log n)$
- **Worst Case (Skewed BST)** - $O(n)$
- **Search**: $O(\log n)$ best, $O(n)$ worst
- **Insertion**: $O(\log n)$ best, $O(n)$ worst
- **Deletion**: $O(\log n)$ best, $O(n)$ worst
- **Traversal**: $O(n)$ time

3. Balanced Binary Search Tree (AVL Tree, Red-Black Tree)

- **Search**: $O(\log n)$
- **Insertion**: $O(\log n)$
- **Deletion**: $O(\log n)$

4. Complete Binary Tree (Binary Heap)

- **Insertion**: $O(\log n)$
- **Delete (Heapify)**: $O(\log n)$
- **Search**: $O(n)$ (No ordering like BST)

5. Space Complexities

- **Recursive Traversal**: $O(h)$ (h = tree height)
- **Iterative Traversal (Using Stack/Queue)**: $O(n)$
- **Balanced BST Storage**: $O(n)$

Key Takeaways

- BST is better than a general binary tree due to faster search, insert, and delete ($O(\log n)$).
- Balanced BST (AVL, Red-Black Tree) ensures $O(\log n)$ operations, avoiding worst-case skewed tree issues.
- Binary Heaps are great for priority queues but lack fast searches.

Property	Formula
Maximum number of nodes in a binary tree of height h	$2^{h+1} - 1$
Minimum number of nodes in a binary tree of height h	$h + 1$ (for skewed tree)
Height of a full binary tree with n nodes	$\lceil \log_2(n) \rceil$
Number of leaf nodes in a full binary tree with n nodes	$(n + 1) / 2$
Number of internal (non-leaf) nodes in a full binary tree	$(n - 1) / 2$
In a perfect binary tree, number of leaf nodes	2^h
In a perfect binary tree, total number of nodes	$2^{h+1} - 1$

2. Definitions

- Height of a tree = number of edges on the longest path from root to leaf.
- Depth of a node = number of edges from root to that node.
- Leaf node = node with 0 children.
- Internal node = node with at least 1 child.
- Perfect binary tree = all internal nodes have 2 children, and all leaves are at same level.
- Full binary tree = each node has either 0 or 2 children.
- Complete binary tree = all levels filled except possibly the last, which is filled left to right.

3. Tree Traversals

Traversal Type	Order
Inorder	Left → Root → Right
Preorder	Root → Left → Right
Postorder	Left → Right → Root
Level Order	Level by level (BFS)

Saved memory till ⏺

4. Time & Space Complexity

Operation	Time Complexity
Traversals (pre/in/post/level)	$O(n)$
Search in BST (avg)	$O(\log n)$
Search in BST (worst - skewed)	$O(n)$
Insert/Delete in BST	$O(\log n)$ average, $O(n)$ worst
Heapify (Max/Min Heap)	$O(\log n)$
Building Heap from Array	$O(n)$

5. Binary Tree vs BST vs Heap Summary

Feature	Binary Tree	BST	Heap
Structure	Hierarchical	Hierarchical	Complete Binary Tree
Ordering	No specific order	$\text{Left} < \text{Root} < \text{Right}$	Root is max/min
Use case	General trees	Fast lookup	Priority queue, Heap Sort

1. Inorder Traversal (Left → Root → Right)

When to Use:

- To get sorted order from a Binary Search Tree (BST).
- Useful for range queries in BSTs.
- When you want to traverse in increasing order.

Use-Cases:

- Printing all nodes in ascending order (in BST).
- Verifying if a tree is a valid BST (values must be strictly increasing in inorder).
- Problems like:
 - Validate BST
 - Kth Smallest in BST

2. Preorder Traversal (Root → Left → Right)

When to Use:

- To copy a tree, serialize/deserialize it.
- When you want to process the root node before its children.
- Building expression trees.

Use-Cases:

- Cloning a binary tree.
- Creating a prefix expression from an expression tree.
- Problems like:
 - Binary Tree Preorder Traversal
 - Serialize and Deserialize Binary Tree

3. Postorder Traversal (Left → Right → Root)

When to Use:

- To delete a tree (children must be deleted before parent).
- Useful for bottom-up processing, such as evaluating expressions or computing subtree values.
- Dynamic Programming on Trees.

Use-Cases:

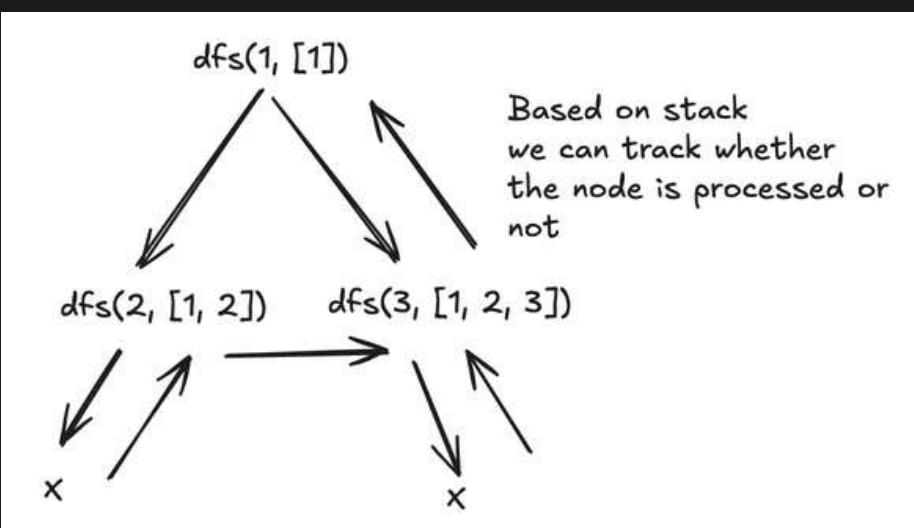
- Deleting nodes or freeing memory.
- Calculating height/depth of subtrees.
- Problems like:
 - Diameter of Binary Tree
 - Binary Tree Maximum Path Sum

Summary Table

Traversal	Order	Use Case
Inorder	Left → Root → Right	Sorted data (BST), validation
Preorder	Root → Left → Right	Tree serialization, building trees
Postorder	Left → Right → Root	Deletion, dynamic programming

- Preorder Traversal (Root → Left → Right)
 - Constructing BSTs from a given sequence.
 - Serialization/deserialization of a BST.
 - Prefix Expression Trees
 - Sum Calculations
 - Process Root before its children
- Inorder Traversal (Left → Root → Right)
 - Sorted Increasing order.
 - K-th smallest/largest element.
 - BST validity
- Postorder Traversal (Left → Right → Root)
 - Deleting nodes (ensuring children are deleted first).
 - Evaluating expressions in an expression tree.
 - DP on trees
 - Process Root after its children
- Level Order Traversal
 - Kth Largest Sum Of Binary Tree
 - Add One Row to Tree
 - Even Odd Tree
- Direct Check:
 - 2236. Root Equals Sum of Children

1. Preorder Traversal



```

public class Solution {

    static class TreeNode {
        private final int data;
        private TreeNode left;
        private TreeNode right;

        public TreeNode(int data) {
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }

    private static List<Integer> preOrder(TreeNode root, List<Integer> result) {
        if (root != null) {
            result.add(root.data);
            preOrder(root.left, result);
            preOrder(root.right, result);
        }
        return result;
    }
}

```

```

private static List<Integer> preOrderStack(TreeNode root, List<Integer> result) {
    Stack<TreeNode> stack = new Stack<>();
    stack.add(root);

    while (!stack.isEmpty()) {
        TreeNode curr = stack.pop();
        result.add(curr.data);
        // Push right first so that left is processed first
        if (curr.right != null) {
            stack.add(curr.right);
        }
        if (curr.left != null) {
            stack.add(curr.left);
        }
    }
    return result;
}

public static void main(String[] args) {
    TreeNode treeNode = new TreeNode( data: 1 );
    treeNode.left = new TreeNode( data: 2 );
    treeNode.right = new TreeNode( data: 3 );
    List<Integer> result = new ArrayList<>();
    preOrder(treeNode, result);
    System.out.println(result); // [1, 2, 3]
    result = new ArrayList<>();
    preOrderStack(treeNode, result);
    System.out.println(result); // [1, 2, 3]
}
}

```

Saved memory full ⓘ

⌚ 1. Time Complexity

- **Time Complexity:** $O(n)$
- **Reason:** Every node in the binary tree is visited **exactly once**.

Where n is the number of nodes in the tree.

🧠 2. Space Complexity

Recursive Version:

- **Best Case (Balanced Tree):** $O(\log n)$ → height of tree
- **Worst Case (Skewed Tree):** $O(n)$ → every node is a left or right child
- **Reason:** The call stack grows up to the height of the tree

Iterative Version (Using Stack):

- **Best Case (Balanced Tree):** $O(\log n)$ → height of tree
- **Worst Case (Skewed Tree):** $O(n)$
- **Reason:** Stack stores nodes along the path; maximum height of tree nodes may be in stack at once

📋 Summary Table:

Version	Time Complexity	Space Complexity (Avg)	Space Complexity (Worst)
Recursive	$O(n)$	$O(\log n)$	$O(n)$
Iterative	$O(n)$	$O(\log n)$	$O(n)$

✓ 1. 1022. Sum of Root To Leaf Binary Numbers

1022. Sum of Root To Leaf Binary Numbers

Easy

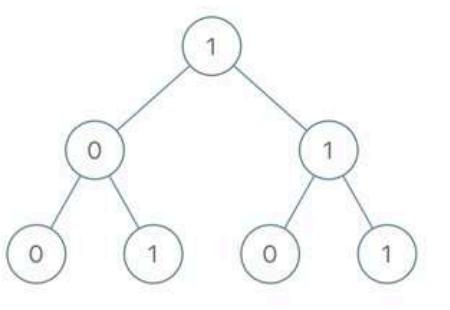
You are given the `root` of a binary tree where each node has a value `0` or `1`. Each root-to-leaf path represents a binary number starting with the most significant bit.

- For example, if the path is `0 -> 1 -> 1 -> 0 -> 1`, then this could represent `01101` in binary, which is `13`.

For all leaves in the tree, consider the numbers represented by the path from the root to that leaf. Return *the sum of these numbers*.

The test cases are generated so that the answer fits in a 32-bits integer.

Example 1:



Input: `root = [1,0,1,0,1,0,1]`

Output: `22`

Explanation: $(100) + (101) + (110) + (111) = 4 + 5 + 6 + 7 = 22$

```
→ current = 0 * 2 + 1 = 1

└── sumFromRootToLeaves(treeNode.left, 1)
    → current = 1 * 2 + 0 = 2
    → leaf node (left & right are null), return 2

└── sumFromRootToLeaves(treeNode.right, 1)
    → current = 1 * 2 + 1 = 3
    → leaf node, return 3

→ return 2 + 3 = 5
```

✓ Traversal Used: Preorder

Why?

In this problem, we:

- Start from the `root`,
- Traverse down to `leaf nodes`,
- Carry a `running path value` with us as we move `top to bottom`.

That's exactly what `preorder traversal` does:

- Visit `root first`, then left, then right — while passing the `current path value` down.

```

public class Solution {

    static class TreeNode {
        private final int data;
        private TreeNode left;
        private TreeNode right;

        public TreeNode(int data) {
            this.data = data;
            this.left = null;
            this.right = null;
        }
    }

    private static int sumFromRootToLeaves(TreeNode node, int current) {
        if (node == null) return 0;
        current = current * 2 + node.data; // current << 2

        if (node.left == null && node.right == null) {
            return current;
        }

        int leftSum = sumFromRootToLeaves(node.left, current); // left subtree sum
        int rightSum = sumFromRootToLeaves(node.right, current); // right subtree sum
        return leftSum + rightSum;
    }

    public static void main(String[] args) {
        TreeNode treeNode = new TreeNode( data: 1);
        treeNode.left = new TreeNode( data: 0);
        treeNode.right = new TreeNode( data: 1);
        int result = 0;
        System.out.println(sumFromRootToLeaves(treeNode, result)); // 10 + 11 = 5
    }
}

```

→ current = $0 * 2 + 1 = 1$

```

|--- sumFromRootToLeaves(treeNode.left, 1)
|   → current =  $1 * 2 + 0 = 2$ 
|   → leaf node (left & right are null), return 2

|--- sumFromRootToLeaves(treeNode.right, 1)
|   → current =  $1 * 2 + 1 = 3$ 
|   → leaf node, return 3

```

→ return $2 + 3 = 5$

✓ Time Complexity: O(N)

✓ Space Complexity: O(H)

Process the left subtree and accumulate the sum until reaching the leaf level, then backtrack to compute the right subtree's sum.

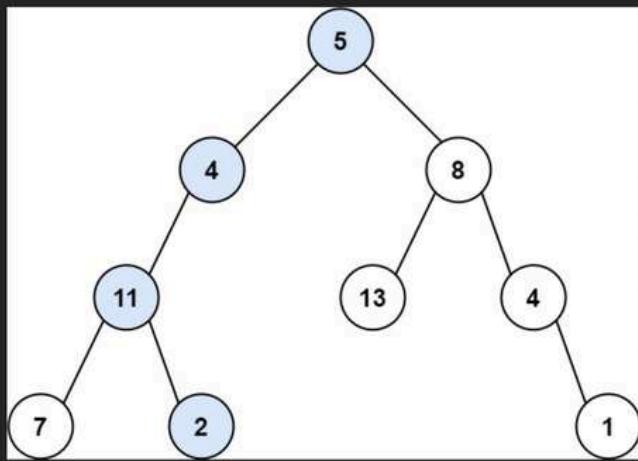
✓ 2.112. Path Sum

Easy

Given the `root` of a binary tree and an integer `targetSum`, return `true` if the tree has a root-to-leaf path such that adding up all the values along the path equals `targetSum`.

A leaf is a node with no children.

Example 1:



Input: `root = [5,4,8,11,null,13,4,7,2,null,null,null,1]`, `targetSum = 22`

Output: `true`

Explanation: The root-to-leaf path with the target sum is shown.

```
private static boolean hasPathSum(TreeNode root, int targetSum) {  
    if (root == null) {  
        return false;  
    }  
  
    if (root.left == null && root.right == null && targetSum == root.data) {  
        return true;  
    }  
  
    int remainingSum = targetSum - root.data;  
    boolean hasLeftPathSum = hasPathSum(root.left, remainingSum);  
    boolean hasRightPathSum = hasPathSum(root.right, remainingSum);  
    return hasLeftPathSum || hasRightPathSum;  
}  
  
public static void main(String[] args) {  
    TreeNode treeNode = new TreeNode( data: 1 );  
    treeNode.left = new TreeNode( data: 0 );  
    treeNode.right = new TreeNode( data: 1 );  
    int result = 1;  
    System.out.println(hasPathSum(treeNode, result)); // true  
}
```

✓ Time Complexity: $O(n)$

✓ Space Complexity: $O(h)$

Case	Time Complexity	Space Complexity
Worst Case	$O(n)$	$O(n)$
Average Case	$O(n)$	$O(\log n)$

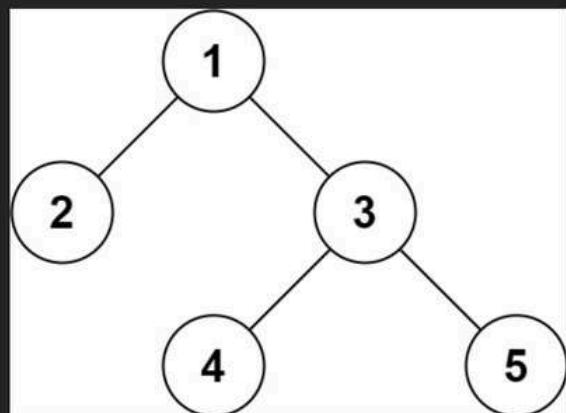
✓ 3.297. Serialize and Deserialize Binary Tree

Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

Clarification: The input/output format is the same as [how LeetCode serializes a binary tree](#). You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

Example 1:



Input: root = [1,2,3,null,null,4,5]
Output: [1,2,3,null,null,4,5]

```
private static String serialize(TreeNode root) {  
    if (root == null) return "null";  
  
    Queue<TreeNode> queue = new LinkedList<>();  
    StringBuilder sb = new StringBuilder();  
    queue.offer(root);  
  
    while (!queue.isEmpty()) {  
        TreeNode curr = queue.poll();  
        if (curr == null) {  
            sb.append("null,");  
        } else {  
            sb.append(curr.data).append(",");  
            queue.offer(curr.left);  
            queue.offer(curr.right);  
        }  
    }  
  
    return sb.toString();  
}
```

```

private static TreeNode deserialize(String data) {
    if (data.equals("null")) return null;

    String[] nodes = data.split( regex: "\\" );
    TreeNode root = new TreeNode(Integer.parseInt(nodes[0]));
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    int i = 1;

    while (!queue.isEmpty()) {
        TreeNode current = queue.poll();

        if (!nodes[i].equals("null")) {
            current.left = new TreeNode(Integer.parseInt(nodes[i]));
            queue.offer(current.left);
        }
        i++;

        if (!nodes[i].equals("null")) {
            current.right = new TreeNode(Integer.parseInt(nodes[i]));
            queue.offer(current.right);
        }
        i++;
    }

    return root;
}

```

```

public static void main(String[] args) {
    TreeNode treeNode = new TreeNode( data: 1 );
    treeNode.left = new TreeNode( data: 0 );
    treeNode.right = new TreeNode( data: 1 );
    String serialized = serialize(treeNode);
    System.out.println("Serialized Tree: " + serialized); // 1,0,1,null,null,null,null,
    TreeNode deserialized = deserialize(serialized);
    System.out.print("Inorder Traversal of Deserialized Tree: "); // 0 1 1
    inorderPrint(deserialized);
}
}

```

```

private static String serialize(TreeNode root) {
    if (root == null) return "null";
    return root.data + "," + serialize(root.left) + serialize(root.right);
}

private static int index = 0;

private static TreeNode deserialize(String data) {
    String[] nodes = data.split( regex: "\\", );
    return deserializeHelper(nodes);
}

private static TreeNode deserializeHelper(String[] nodes) {
    if (index >= nodes.length || nodes[index].equals("null")) {
        index++;
        return null;
    }

    TreeNode node = new TreeNode(Integer.parseInt(nodes[index++]));
    node.left = deserializeHelper(nodes);
    node.right = deserializeHelper(nodes);
    return node;
}

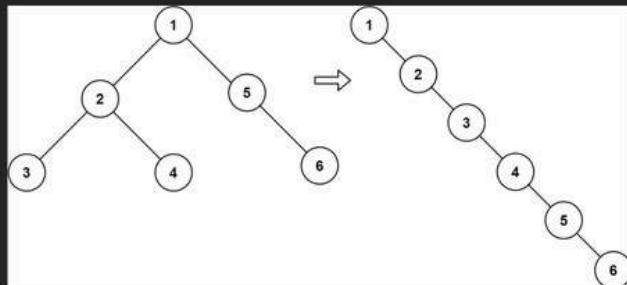
```

4. 114. Flatten Binary Tree to Linked List

Given the `root` of a binary tree, flatten the tree into a "linked list":

- The "linked list" should use the same `TreeNode` class where the `right` child pointer points to the next node in the list and the `left` child pointer is always `null`.
- The "linked list" should be in the same order as a pre-order traversal of the binary tree.

Example 1:



Input: `root = [1,2,5,3,4,null,6]`
Output: `[1,null,2,null,3,null,4,null,5,null,6]`

```
@Override
public String toString() {
    return inorderString( node: this);
}

private String inorderString(TreeNode node) {
    if (node == null) return "";
    return inorderString(node.left) + node.data + " " + inorderString(node.right);
}

private static TreeNode prev = null;

private static void flatten(TreeNode root) {
    if (root == null) return;
    flatten(root.right);
    flatten(root.left);
    root.right = prev;
    root.left = null;
    prev = root;
}

public static void main(String[] args) {
    TreeNode treeNode = new TreeNode( data: 1);
    treeNode.left = new TreeNode( data: 2);
    treeNode.left.left = new TreeNode( data: 3);
    treeNode.left.right = new TreeNode( data: 4);
    treeNode.right = new TreeNode( data: 5);
    treeNode.right.right = new TreeNode( data: 6);
    flatten(treeNode);
    System.out.println(treeNode); // 1 2 3 4 5 6
}
```



● Why right first, then left?

Because this solution is doing **reverse preorder traversal**:

`right → left → root`, and using a global `prev` pointer to link nodes from **back to front**.

⌚ Think in reverse:

- You're building the linked list **from the tail (6) to the head (1)**.
- You need to visit **6 before 5**, and **5 before 4**, etc.
- So the right subtree must be processed first.

💡 Brilliant Observation

What if... instead of going `root -> left -> right` (preorder), we went in **reverse preorder**:

`right -> left -> root`

Why?

Because then we could **build the list backward**, by rewiring from the last node (`6`) up to the root (`1`).

Reverse Preorder traversal: `6 -> 5 -> 4 -> 3 -> 2 -> 1`

Now wiring from tail to head:

sql

Copy Edit

```
prev = null
flatten(6) -> 6.right = null -> prev = 6
flatten(5) -> 5.right = 6 -> prev = 5
flatten(4) -> 4.right = 5 -> prev = 4
flatten(3) -> 3.right = 4 -> prev = 3
flatten(2) -> 2.right = 3 -> prev = 2
flatten(1) -> 1.right = 2 -> prev = 1
```

Final tree (flattened list):

Copy Edit

```
1 -> 2 -> 3 -> 4 -> 5 -> 6
```

All `.left = null`

💡 How was the logic discovered?

1. Try **preorder**, fail to wire correctly because you visit nodes before their children are flattened.
2. Realize you need to *flatten subtrees first*, then wire the root.
3. Since in **preorder** we do `root -> left -> right`, do **reverse**: `right -> left -> root`.
4. Build the result in **reverse order** using a `prev` pointer — elegant and avoids extra space.

✨ Key Patterns Learned

- Traversing in **reverse of the desired order** can help when building output from end to start.
- Maintaining a `prev` pointer is useful in linked-list-style problems on trees.
- This is similar to **postorder** but adjusted for a specific traversal order need.

Call Stack Trace (Reverse Preorder: right → left → root)

```
kotlin
flatten(1)
└ flatten(5)
  └ flatten(6)
    └ flatten(null) -> return
    └ flatten(null) -> return
    └ root.right = null
    └ root.left = null
    └ prev = 6
  └ root.right = 6
  └ root.left = null
  └ prev = 5
└ flatten(2)
  └ flatten(4)
    └ flatten(null) -> return
    └ flatten(null) -> return
    └ root.right = 5
    └ root.left = null
    └ prev = 4
  └ flatten(3)
    └ flatten(null) -> return
    └ flatten(null) -> return
    └ root.right = 4
    └ root.left = null
    └ prev = 3
  └ root.right = 3
  └ root.left = null
  └ prev = 2
└ root.right = 2
└ root.left = null
└ prev = 1
```

Transformation Steps (Final pointers after each node processed):

1. 6.right = null, prev = 6
2. 5.right = 6, prev = 5
3. 4.right = 5, prev = 4
4. 3.right = 4, prev = 3
5. 2.right = 3, prev = 2
6. 1.right = 2, prev = 1

Final Structure (Right-skewed "linked list"):

```
rust
3 -> 2 -> 3 -> 4 -> 5 -> 6
```

Process the right subtree , maintain prev and update right of current element using prev

its the reverse of preorder traversal where we start from the rightmost lead node, and start linking backwards

5. 129. Sum Root to Leaf Numbers

You are given the `root` of a binary tree containing digits from `0` to `9` only.

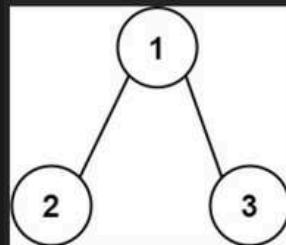
Each root-to-leaf path in the tree represents a number.

- For example, the root-to-leaf path `1 -> 2 -> 3` represents the number `123`.

Return the *total sum of all root-to-leaf numbers*. Test cases are generated so that the answer will fit in a 32-bit integer.

A leaf node is a node with no children.

Example 1:



Input: `root = [1,2,3]`

Output: `25`

Explanation:

The root-to-leaf path `1->2` represents the number `12`.

The root-to-leaf path `1->3` represents the number `13`.

Therefore, sum = `12 + 13 = 25`.

```
private String inorderString(TreeNode node) {
    if (node == null) return "";
    return inorderString(node.left) + node.data + " " + inorderString(node.right);
}

private static int sum(TreeNode root, int curr) {
    if (root == null) return 0;
    curr = curr * 10 + root.data;
    if (root.left == null && root.right == null){
        return curr;
    }
    int leftSum = sum(root.left, curr);
    int rightSum = sum(root.right, curr);
    return leftSum + rightSum;
}

public static void main(String[] args) {
    TreeNode treeNode = new TreeNode( data: 1);
    treeNode.left = new TreeNode( data: 2);
    treeNode.left.left = new TreeNode(3);
    treeNode.left.right = new TreeNode(4);
    treeNode.right = new TreeNode( data: 5);
    treeNode.right.right = new TreeNode(6);
    System.out.println(sum(treeNode, curr: 0)); // 27
}
```

Traverse the left subtree to compute the sum down to each leaf, return the accumulated value, and then process the right subtrees similarly. Multiply the sum by 10 at each level of traversal to account for place value.

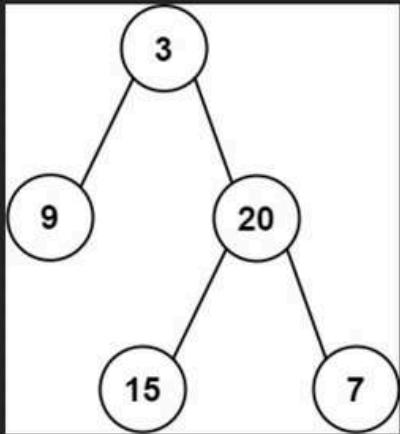
6. 404. Sum of Left Leaves

Easy

Given the `root` of a binary tree, return *the sum of all left leaves*.

A leaf is a node with no children. A **left leaf** is a leaf that is the left child of another node.

Example 1:



Input: `root = [3,9,20,null,null,15,7]`

Output: 24

Explanation: There are two left leaves in the binary tree with values 15 and 7 respectively.

Example 2:

Input: `root = [1]`

Output: 0

✓ Java BFS (Level Order) Solution

```
java
Copy Edit

import java.util.*;

public class Solution {
    public int sumOfLeftLeaves(TreeNode root) {
        if (root == null) return 0;

        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        int sum = 0;

        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();

            if (node.left != null) {
                // Check if it's a left leaf
                if (node.left.left == null && node.left.right == null) {
                    sum += node.left.val;
                }
                queue.offer(node.left);
            }

            if (node.right != null) {
                queue.offer(node.right);
            }
        }
        return sum;
    }
}
```

```
private String inorderString(TreeNode node) {
    if (node == null) return "";
    return inorderString(node.left) + node.data + " " + inorderString(node.right);
}

private static int sumOfLeftLeaves(TreeNode root) {
    if (root == null) return 0;
    int sum = 0;
    if (root.left != null && root.left.left == null && root.left.right == null) {
        sum += root.left.data;
    }
    sum += sumOfLeftLeaves(root.left);
    sum += sumOfLeftLeaves(root.right);
    return sum;
}

public static void main(String[] args) {
    TreeNode treeNode = new TreeNode( data: 3 );
    treeNode.left = new TreeNode( data: 9 );
    treeNode.right = new TreeNode( data: 20 );
    treeNode.right.left = new TreeNode( data: 15 );
    treeNode.right.right = new TreeNode( data: 7 );
    System.out.println(sumOfLeftLeaves(treeNode)); // 24
}
```

7.116. Populating Next Right Pointers in Each Node

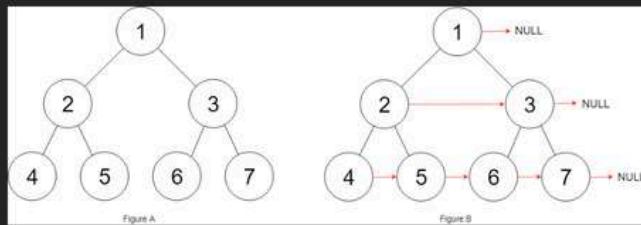
You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. The binary tree has the following definition:

```
struct Node {  
    int val;  
    Node *left;  
    Node *right;  
    Node *next;  
};
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to `NULL`.

Initially, all next pointers are set to `NULL`.

Example 1:



Input: `root = [1,2,3,4,5,6,7]`
Output: `[1,#,2,3,#,4,5,6,7,#]`

Explanation: Given the above perfect binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level.

```
private static TreeNode connect(TreeNode root) {  
    if (root == null || root.left == null || root.right == null) return null;  
  
    root.left.next = root.right;  
  
    if(root.next != null){  
        root.right.next = root.next.left;  
    }  
  
    connect(root.left);  
    connect(root.right);  
  
    return root;  
}  
  
public static void main(String[] args) {  
    TreeNode treeNode = new TreeNode( data: 1);  
    treeNode.left = new TreeNode( data: 2);  
    treeNode.left.left = new TreeNode( data: 4);  
    treeNode.left.right = new TreeNode( data: 5);  
    treeNode.right = new TreeNode( data: 3);  
    treeNode.right.left = new TreeNode( data: 6);  
    treeNode.right.right = new TreeNode( data: 7);  
    System.out.println(connect(treeNode));  
}
```

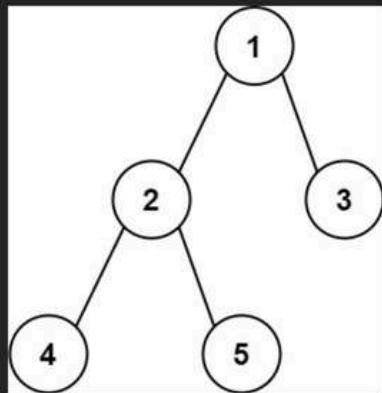
8. 543. Diameter of Binary Tree

Given the `root` of a binary tree, return *the length of the diameter of the tree*.

The diameter of a binary tree is the length of the longest path between any two nodes in a tree. This path may or may not pass through the `root`.

The length of a path between two nodes is represented by the number of edges between them.

Example 1:



Input: `root = [1,2,3,4,5]`

Output: 3

Explanation: 3 is the length of the path [4,2,1,3] or [5,2,1,3].

```
private static int diameterOfBinaryTree(TreeNode root) {
    int[] diameter = new int[1];
    dfs(root, diameter);
    return diameter[0];
}

private static int dfs(TreeNode node, int[] diameter){
    if (node == null) return 0;

    int leftDepth = dfs(node.left, diameter);
    int rightDepth = dfs(node.right, diameter);
    diameter[0] = Math.max(diameter[0], leftDepth + rightDepth);

    return Math.max(leftDepth, rightDepth) + 1;
}

public static void main(String[] args) {
    TreeNode treeNode = new TreeNode( data: 1 );
    treeNode.left = new TreeNode( data: 2 );
    treeNode.left.left = new TreeNode( data: 4 );
    treeNode.left.right = new TreeNode( data: 5 );
    treeNode.right = new TreeNode( data: 3 );
    System.out.println(diameterOfBinaryTree(treeNode)); // 3
}
```

💡 Why `leftDepth + rightDepth`?

At any node, the longest path that passes through that node would be:

sql

Copy Edit

```
Longest path through current node = left subtree depth + right subtree depth
```

This is because we go down to the deepest leaf on the left, and deepest leaf on the right, forming the longest path that includes the current node.

Example:

markdown

Copy Edit

```
1
 / \
2   3
 \
4
```

- From node 4 to node 3 is the longest path: 4 → 2 → 1 → 3
- `leftDepth = 2` (from node 1 to 4)
- `rightDepth = 1` (from node 1 to 3)
- So, `leftDepth + rightDepth = 3` (number of edges)

Return the max of left and right subTree, and at each iteration compute the max diameter

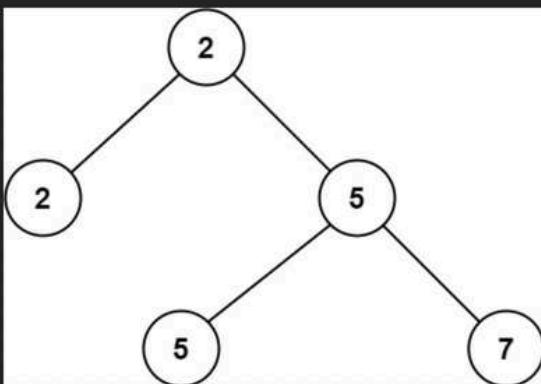
10. 671. Second Minimum Node In a Binary Tree

Given a non-empty special binary tree consisting of nodes with the non-negative value, where each node in this tree has exactly two or zero sub-node. If the node has two sub-nodes, then this node's value is the smaller value among its two sub-nodes. More formally, the property `root.val = min(root.left.val, root.right.val)` always holds.

Given such a binary tree, you need to output the **second minimum** value in the set made of all the nodes' value in the whole tree.

If no such second minimum value exists, output -1 instead.

Example 1:



Input: `root = [2,2,5,null,null,5,7]`

Output: 5

Explanation: The smallest value is 2, the second smallest value is 5.

```

private static void dfs(TreeNode node, int minVal, int[] secondMin) {
    if (node == null) return;

    if (minVal < node.data) { // secondMin
        if (secondMin[0] == -1) {
            secondMin[0] = node.data;
        } else {
            secondMin[0] = Math.min(secondMin[0], node.data);
        }
    }

    dfs(node.left, minVal, secondMin);
    dfs(node.right, minVal, secondMin);
}

public static void main(String[] args) {
    TreeNode treeNode = new TreeNode(data: 2);
    treeNode.left = new TreeNode(data: 2);
    treeNode.right = new TreeNode(data: 5);
    treeNode.right.left = new TreeNode(data: 5);
    treeNode.right.right = new TreeNode(data: 7);
    System.out.println(findSecondMinimumValue(treeNode)); // 5
}

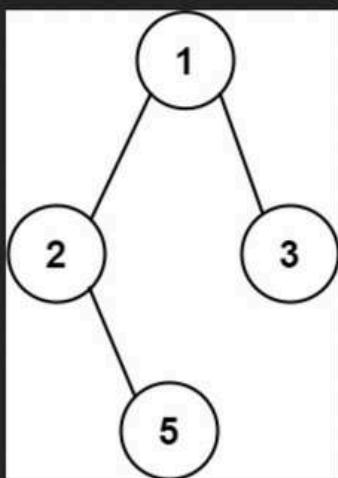
```

11. 257. Binary Tree Paths

Given the `root` of a binary tree, return *all root-to-leaf paths in any order*.

A **leaf** is a node with no children.

Example 1:



Input: root = [1,2,3,null,5]
Output: ["1->2->5","1->3"]

Example 2:

Input: root = [1]
Output: ["1"]

```

private static void dfs(TreeNode node, String curr, List<String> result) {
    if (node == null) return;
    curr += node.data;

    if (node.left == null && node.right == null) {
        result.add(curr);
        return;
    }

    dfs(node.left, curr + "->", result);
    dfs(node.right, curr + "->", result);
}

public static void main(String[] args) {
    TreeNode treeNode = new TreeNode( val: 1 );
    treeNode.left = new TreeNode( val: 2 );
    treeNode.right = new TreeNode( val: 3 );
    List<String> result = new ArrayList<>();
    dfs(treeNode, curr: "", result);
    System.out.println(result);
}

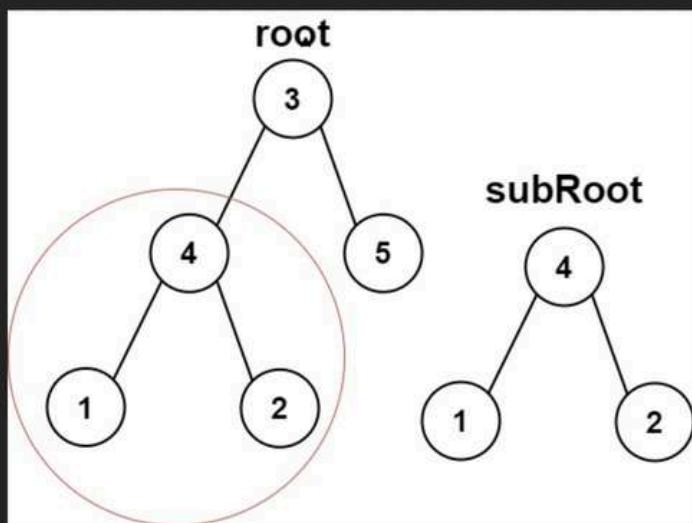
```

12. 572. Subtree of Another Tree

Given the roots of two binary trees `root` and `subRoot`, return `true` if there is a subtree of `root` with the same structure and node values of `subRoot` and `false` otherwise.

A subtree of a binary tree `tree` is a tree that consists of a node in `tree` and all of this node's descendants. The tree `tree` could also be considered as a subtree of itself.

Example 1:



Input: `root = [3,4,5,1,2]`, `subRoot = [4,1,2]`
Output: `true`

```

private static boolean isSubtree(TreeNode root, TreeNode subRoot) {
    if (root == null) return false;

    if (isSameTree(root, subRoot)) return true;

    boolean isLeftSubTree = isSubtree(root.left, subRoot);
    boolean isRightSubTree = isSubtree(root.right, subRoot);
    return isLeftSubTree || isRightSubTree;
}

private static boolean isSameTree(TreeNode t1, TreeNode t2) {
    if (t1 == null && t2 == null) {
        return true;
    }
    if (t1 == null || t2 == null) {
        return false;
    }
    if (t1.data != t2.data) {
        return false;
    }

    return isSameTree(t1.left, t2.left) && isSameTree(t1.right, t2.right);
}

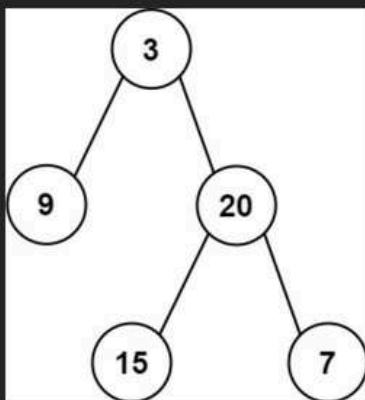
public static void main(String[] args) {
    TreeNode treeNode = new TreeNode( data: 3 );
    treeNode.left = new TreeNode( data: 4 );
    treeNode.left.left = new TreeNode( data: 1 );
    treeNode.left.right = new TreeNode( data: 2 );
    treeNode.right = new TreeNode( data: 5 );
    TreeNode subTree = new TreeNode( data: 4 );
    subTree.left = new TreeNode( data: 1 );
    subTree.right = new TreeNode( data: 2 );
    System.out.println(isSubtree(treeNode, subTree));
}

```

13.105. Construct Binary Tree from Preorder and Inorder Traversal

Given two integer arrays `preorder` and `inorder` where `preorder` is the preorder traversal of a binary tree and `inorder` is the inorder traversal of the same tree, construct and return the *binary tree*.

Example 1:



Input: preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]
Output: [3,9,20,null,null,15,7]

Example 2:

Input: preorder = [-1], inorder = [-1]
Output: [-1]

```

import java.util.*;

class Solution {
    public TreeNode buildTree(int[] preorder, int[] inorder) {
        Map<Integer, Integer> inorderIndexMap = new HashMap<>();
        for (int i = 0; i < inorder.length; i++) {
            inorderIndexMap.put(inorder[i], i);
        }
        int[] preorderIndex = new int[]{0}; // Wrap in array to pass by reference
        return dfs(preorder, inorderIndexMap, preorderIndex, 0, inorder.length - 1);
    }

    private TreeNode dfs(int[] preorder, Map<Integer, Integer> inorderIndexMap,
                         int[] preorderIndex, int inorderStart, int inorderEnd) {
        if (inorderStart > inorderEnd) return null;
        int rootValue = preorder[preorderIndex[0]++];
        TreeNode root = new TreeNode(rootValue);

        int inorderIndex = inorderIndexMap.get(rootValue);

        root.left = dfs(preorder, inorderIndexMap, preorderIndex, inorderStart, inorderIndex - 1);
        root.right = dfs(preorder, inorderIndexMap, preorderIndex, inorderIndex + 1, inorderEnd);

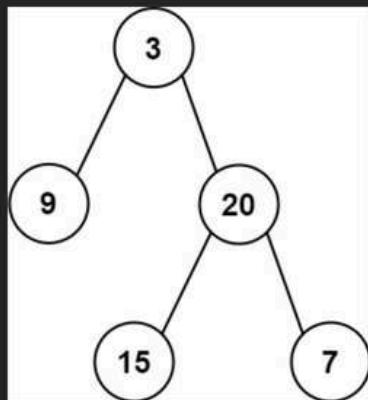
        return root;
    }
}

```

14. 106. Construct Binary Tree from Inorder and Postorder Traversals

Given two integer arrays `inorder` and `postorder` where `inorder` is the inorder traversal of a binary tree and `postorder` is the postorder traversal of the same tree, construct and return the binary tree.

Example 1:



Input: `inorder = [9,3,15,20,7]`, `postorder = [9,15,7,20,3]`
Output: `[3,9,20,null,null,15,7]`

Example 2:

Input: `inorder = [-1]`, `postorder = [-1]`
Output: `[-1]`

```

private int postorderIndex;
private Map<Integer, Integer> inorderIndexMap;

public TreeNode buildTree(List<Integer> inorder, List<Integer> postorder) {
    // Initialize index map
    inorderIndexMap = new HashMap<>();
    for (int i = 0; i < inorder.size(); i++) {
        inorderIndexMap.put(inorder.get(i), i);
    }

    // Start from the end of postorder
    postorderIndex = postorder.size() - 1;

    return constructTree(postorder, 0, inorder.size() - 1);
}

private TreeNode constructTree(List<Integer> postorder, int inorderStart, int inorderEnd) {
    if (inorderStart > inorderEnd) return null;

    int rootValue = postorder.get(postorderIndex--);
    TreeNode root = new TreeNode(rootValue);

    int inorderIndex = inorderIndexMap.get(rootValue);

    // Construct right subtree first (postorder goes left->right->root, reverse is root->right->left)
    root.right = constructTree(postorder, inorderIndex + 1, inorderEnd);
    root.left = constructTree(postorder, inorderStart, inorderIndex - 1);

    return root;
}

```

Recursive DFS Solution (Top-down)

```
java Copy Edit  
  
public int maxDepth(TreeNode root) {  
    if (root == null) return 0;  
    return 1 + Math.max(maxDepth(root.left), maxDepth(root.right));  
}
```

- Time Complexity: $O(n)$ — visit each node once

Java Code – DFS Approach

```
java Copy Edit  
  
class TreeNode {  
    int val;  
    TreeNode left, right;  
    TreeNode(int x) { val = x; }  
}  
  
public class BinaryTreeDepth {  
    public int getDepth(TreeNode root, int target) {  
        return dfs(root, target, 0);  
    }  
  
    private int dfs(TreeNode node, int target, int depth) {  
        if (node == null) return -1;  
        if (node.val == target) return depth;  
  
        int left = dfs(node.left, target, depth + 1);  
        if (left != -1) return left;  
  
        return dfs(node.right, target, depth + 1);  
    }  
}
```

2. Inorder Traversal

```
public class InorderTraversal {  
    public static List<Integer> inorderTraversal(Node root) {  
        List<Integer> result = new ArrayList<>();  
        Stack<Node> stack = new Stack<>();  
        Node current = root;  
  
        // Traverse the tree  
        while (current != null || !stack.isEmpty()) {  
            // Reach the leftmost Node of the current Node  
            while (current != null) {  
                stack.push(current);  
                current = current.left;  
            }  
  
            // Current must be null at this point  
            current = stack.pop();  
            result.add(current.val); // Visit the node  
  
            // Now visit the right subtree  
            current = current.right;  
        }  
  
        return result;  
    }  
}
```

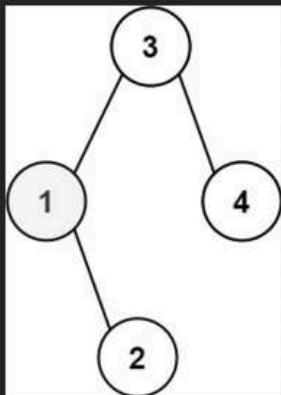
1.230. Kth Smallest Element in a BST

230. Kth Smallest Element in a BST

Medium

Given the `root` of a binary search tree, and an integer `k`, return the k^{th} smallest value (1-indexed) of all the values of the nodes in the tree.

Example 1:



Input: `root = [3,1,4,null,2]`, `k = 1`
Output: 1

```

*/
class Solution {
    private int count = 0;
    private int result = 0;

    public int kthSmallest(TreeNode root, int k) {
        inorder(root, k);
        return result;
    }

    private void inorder(TreeNode root, int k) {
        if (root == null) {
            return;
        }

        inorder(root.left, k);

        count++;
        if (count == k) {
            result = root.val;
            return;
        }

        inorder(root.right, k);
    }
}

```

2. 230. Kth Smallest Element in a BST

98. Validate Binary Search Tree

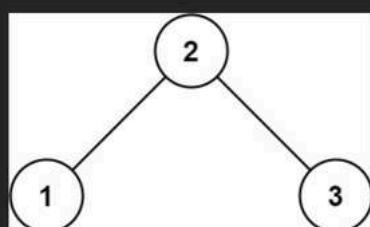
Medium

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.

Example 1:



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

```

public boolean isValidBST(TreeNode root) {
    return inorder(root);
}

private boolean inorder(TreeNode node) {
    if (node == null) return true;

    // Left
    if (!inorder(node.left)) return false;

    // Current
    if (prev != null && node.val <= prev.val) return false;
    prev = node;

    // Right
    return inorder(node.right);
}

```

Left should be valid and if current node val is lesser than the prev set prev as the node

3. PostOrder Traversal

```

class Solution {
    public List<Integer> postorderTraversal(TreeNode root) {
        List<Integer> result = new ArrayList<>();
        if (root == null) return result;

        Stack<TreeNode> s1 = new Stack<>();
        Stack<TreeNode> s2 = new Stack<>();

        s1.push(root);
        while (!s1.isEmpty()) {
            TreeNode node = s1.pop();
            s2.push(node);

            if (node.left != null) {
                s1.push(node.left);
            }
            if (node.right != null) {
                s1.push(node.right);
            }
        }

        while (!s2.isEmpty()) {
            result.add(s2.pop().val);
        }

        return result;
    }
}

```

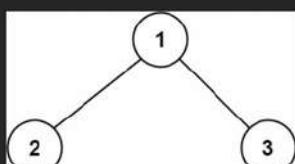
124. Binary Tree Maximum Path Sum

A path in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence at most once. Note that the path does not need to pass through the root.

The path sum of a path is the sum of the node's values in the path.

Given the `root` of a binary tree, return the maximum path sum of any non-empty path.

Example 1:



`Input: root = [1,2,3]`

`Output: 6`

`Explanation: The optimal path is 2 → 1 → 3 with a path sum of 2 + 1 + 3 = 6.`

```

class Solution {
    public:
        int maxPathSum(TreeNode* root) {
            int maxSum = INT_MIN;
            dfs(root, maxSum);
            return maxSum;
        }

    private:
        int dfs(TreeNode* node, int& maxSum) {
            if (!node) return 0;

            int left = max(dfs(node->left, maxSum), 0);
            int right = max(dfs(node->right, maxSum), 0);

            int currentMax = node->val + left + right;
            maxSum = max(maxSum, currentMax);

            return node->val + max(left, right);
        };
}

```

Return the max of left and right subTree + current val, and at each iteration compute the max of left and right subtree and update maxSum using currentSum

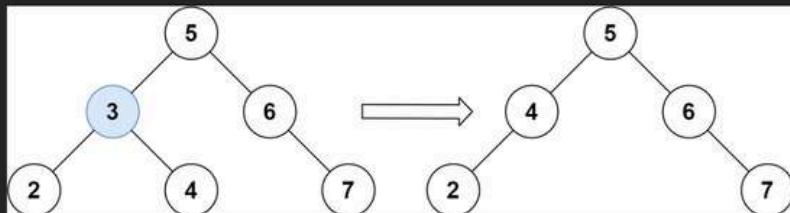
1.450. Delete Node in a BST

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return *the root node reference (possibly updated) of the BST*.

Basically, the deletion can be divided into two stages:

- Search for a node to remove.
- If the node is found, delete the node.

Example 1:



Input: root = [5,3,6,2,4,null,7], key = 3

Output: [5,4,6,2,null,null,7]

Explanation: Given key to delete is 3. So we find the node with value 3 and delete it.

One valid answer is [5,4,6,2,null,null,7], shown in the above BST.

Please notice that another valid answer is [5,2,6,null,4,null,7] and it's also accepted.

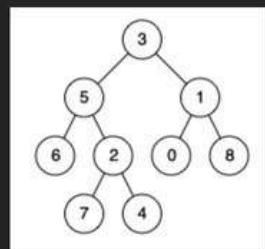
```
class Solution {  
    public TreeNode deleteNode(TreeNode root, int key) {  
        if (root == null) {  
            return null;  
        }  
  
        if (key < root.val) {  
            root.left = deleteNode(root.left, key);  
        } else if (key > root.val) {  
            root.right = deleteNode(root.right, key);  
        } else {  
            if (root.left == null) {  
                return root.right;  
            } else if (root.right == null) {  
                return root.left;  
            }  
            root.val = findMin(root.right);  
            root.right = deleteNode(root.right, root.val);  
        }  
  
        return root;  
    }  
  
    private int findMin(TreeNode node) {  
        while (node.left != null) {  
            node = node.left;  
        }  
        return node.val;  
    }  
}
```

2.236. Lowest Common Ancestor of a Binary Tree

Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the [definition of LCA on Wikipedia](#): "The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself)."

Example 1:



Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

Output: 3

Explanation: The LCA of nodes 5 and 1 is 3.

```
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if (!root) return nullptr; // Base case: if the node is null, return nullptr

        // If the current node is either p or q, return it
        if (root == p || root == q) return root;

        // Recursively search in the left and right subtrees
        TreeNode* left = lowestCommonAncestor(root->left, p, q);
        TreeNode* right = lowestCommonAncestor(root->right, p, q);

        // If both left and right are non-null, the current node is the LCA
        if (left && right) return root;

        // If one of the subtrees is non-null, return the non-null subtree's result
        return left ? left : right;
    }

    // Find LCA of two nodes
    private static TreeNode findLCA(TreeNode root, int n1, int n2) {
        if (root == null) return null;

        if (root.data == n1 || root.data == n2) return root;

        TreeNode left = findLCA(root.left, n1, n2);
        TreeNode right = findLCA(root.right, n1, n2);

        if (left != null && right != null) return root;

        return (left != null) ? left : right;
    }

    // Find distance from root to target node
    private static int findDistance(TreeNode root, int target, int level) {
        if (root == null) return -1;

        if (root.data == target) return level;

        int left = findDistance(root.left, target, level + 1);
        if (left != -1) return left;

        return findDistance(root.right, target, level + 1);
    }

    public static int distanceBetweenNodes(TreeNode root, int n1, int n2) {
        TreeNode lca = findLCA(root, n1, n2);

        int d1 = findDistance(lca, n1, 0);
        int d2 = findDistance(lca, n2, 0);

        return d1 + d2;
    }

    public static int findDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }

        int leftDepth = findDepth(root.left);
        int rightDepth = findDepth(root.right);

        return 1 + Math.max(leftDepth, rightDepth);
    }
}
```

Max distance from root to leaf

Amazon allows sellers to create parent-child relationships for digital products to help customers find a version of a digital download they are looking for. You decide for each version of your digital product. You currently have a rooted tree consisting of n versions of your digital product.

The versions are numbered from 1 to n. The root of the tree is the versioned number 1.

Each version has an integer value assigned to it, which is the amount of downloads that version received. Your goal is to find the price value for each version of the product. The price value for version i is defined to be: 1.

If version i has no child versions (a leaf node), then it is 0.

Otherwise it is the maximum product of values assigned to two distinct versions in the subtree of version i. You need to return an array of integers where ith integer denotes the price value of version i. The subtree of version i is the tree that contains version i along with all of its descendants in the tree. Note: versions with negative assigned values are present in the test cases for testing purposes.

Example

Input:

```
python
n = 5
values = [3, -2, 4, 5, -6]
edges = [
    [1, 2],
    [1, 3],
    [3, 4],
    [3, 5]
]
```

Tree:

```
markdown
      1(3)
     /   \
    2(-2) 3(4)
       / \
      4(5) 5(-6)
```

Subtree of node 3: [3, 4, 5] → values = [4, 5, -6] → possible products = [4×5, 4×-6, 5×-6] = [20, -24, -30]
→ max = 20
So price of node 3 is 20.

```
public class Solution {

    private static class TreeNode {
        int data; // download count
        TreeNode left;
        TreeNode right;
        int version; // index from 1 to n

        public TreeNode(int data, int version) {
            this.data = data;
            this.version = version;
        }
    }

    int[] price;

    public int[] findPrice(TreeNode root, int n) {
        price = new int[n + 1]; // version 1 to n
        dfs(root);
        return Arrays.copyOfRange(price, from: 1, to: n + 1); // skip index 0
    }
}
```


3. Remove all nodes which lie on path having sum less than k

```
private static TreeNode removeNodesLessThanK(TreeNode root, int curr, int k) {  
    if (root == null) return null;  
  
    // Recurse for left and right subtrees  
    root.left = removeNodesLessThanK(root.left, curr + root.data, k);  
    root.right = removeNodesLessThanK(root.right, curr + root.data, k);  
  
    // If this is a leaf and path sum is less than k, prune it  
    if (root.left == null && root.right == null && curr + root.data < k) {  
        return null;  
    }  
  
    return root;  
}  
  
public static void main(String[] args) {  
    TreeNode root = new TreeNode(data: 1);  
    root.left = new TreeNode(data: 2);  
    root.right = new TreeNode(data: 3);  
    root.left.left = new TreeNode(data: 4);  
    root.left.right = new TreeNode(data: 5);  
    System.out.println(root); // 4 2 5 1 3  
    System.out.println(removeNodesLessThanK(root, curr: 0, k: 7)); // 4 2 5 1  
}
```

Level Order Traversal Questions

- 1.Kth Largest Sum Of Binary Tree
- 2.Mirror Of Binary Tree
- 3.Add One Row to Tree
- 4.Height Of A Binary Tree
- 5.Depth Of A Binary Tree
- 6.Even Odd Tree
- 7.Flip Binary Tree

```
private static List<Integer> levelOrder(TreeNode root) {  
    List<Integer> result = new ArrayList<>();  
    Queue<TreeNode> queue = new ArrayDeque<>();  
    queue.add(root);  
    while (!queue.isEmpty()) {  
        TreeNode curr = queue.poll();  
        result.add(curr.data);  
  
        if (curr.left != null) {  
            queue.offer(curr.left);  
        }  
  
        if (curr.right != null) {  
            queue.offer(curr.right);  
        }  
    }  
  
    return result;  
}  
  
public static void main(String[] args) {  
    TreeNode root = new TreeNode(data: 1);  
    root.left = new TreeNode(data: 2);  
    root.right = new TreeNode(data: 3);  
    root.left.left = new TreeNode(data: 4);  
    root.left.right = new TreeNode(data: 5);  
    System.out.println(levelOrder(root)); // [1, 2, 3, 4, 5]  
}
```

1. Kth Largest Sum Of Binary Tree

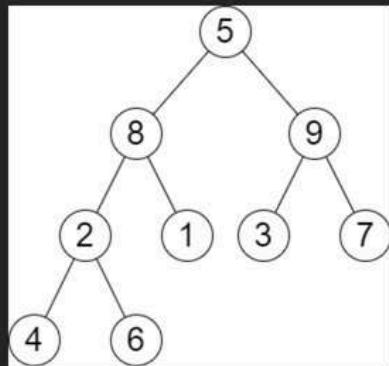
You are given the `root` of a binary tree and a positive integer `k`.

The level sum in the tree is the sum of the values of the nodes that are on the same level.

Return the `kth` largest level sum in the tree (not necessarily distinct). If there are fewer than `k` levels in the tree, return `-1`.

Note that two nodes are on the same level if they have the same distance from the root.

Example 1:



Input: `root = [5,8,9,2,1,3,7,4,6], k = 2`

Output: 13

Explanation: The level sums are the following:

- Level 1: 5.
- Level 2: $8 + 9 = 17$.
- Level 3: $2 + 1 + 3 + 7 = 13$.

```
class Solution {  
    public long kthLargestLevelSum(TreeNode root, int k) {  
        if (root == null) return -1;  
  
        Queue<TreeNode> queue = new LinkedList<>();  
        queue.offer(root);  
  
        PriorityQueue<Long> minHeap = new PriorityQueue<>(); // min-heap  
  
        while (!queue.isEmpty()) {  
            int size = queue.size();  
            long levelSum = 0;  
  
            for (int i = 0; i < size; i++) {  
                TreeNode curr = queue.poll();  
                levelSum += curr.val;  
  
                if (curr.left != null) queue.offer(curr.left);  
                if (curr.right != null) queue.offer(curr.right);  
            }  
  
            minHeap.offer(levelSum);  
            if (minHeap.size() > k) {  
                minHeap.poll(); // Remove smallest sum to maintain top K largest  
            }  
        }  
  
        return minHeap.size() == k ? minHeap.peek() : -1;  
    }  
}
```

2. Mirror Of Binary Tree

```
public static void mirror(TreeNode root) {  
    Queue<TreeNode> queue = new ArrayDeque<>();  
    queue.offer(root);  
    while (!queue.isEmpty()) {  
        TreeNode curr = queue.poll();  
        TreeNode temp = curr.left;  
        curr.left = curr.right;  
        curr.right = temp;  
        if (curr.left != null) {  
            queue.offer(curr.left);  
        }  
        if (curr.right != null) {  
            queue.offer(curr.right);  
        }  
    }  
}  
  
public static void main(String[] args) {  
    TreeNode root = new TreeNode( data: 1 );  
    root.left = new TreeNode( data: 2 );  
    root.right = new TreeNode( data: 3 );  
    root.left.left = new TreeNode( data: 4 );  
    root.left.right = new TreeNode( data: 5 );  
    mirror(root);  
    System.out.println(root); // 3 1 5 2 4 |  
}
```

3. 623. Add One Row to Tree

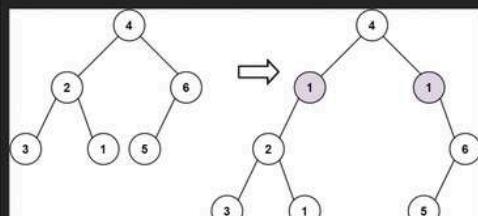
Given the `root` of a binary tree and two integers `val` and `depth`, add a row of nodes with value `val` at the given depth `depth`.

Note that the `root` node is at depth 1.

The adding rule is:

- Given the integer `depth`, for each not null tree node `cur` at the depth `depth - 1`, create two tree nodes with value `val` as `cur`'s left subtree root and right subtree root.
- `cur`'s original left subtree should be the left subtree of the new left subtree root.
- `cur`'s original right subtree should be the right subtree of the new right subtree root.
- If `depth == 1` that means there is no depth `depth - 1` at all, then create a tree node with value `val` as the new root of the whole original tree, and the original tree is the new root's left subtree.

Example 1:



Input: root = [4,2,6,3,1,5], val = 1, depth = 2
Output: [4,1,1,2,null,null,6,3,1,5]

```

class Solution {
    public TreeNode addOneRow(TreeNode root, int val, int depth) {
        if (depth == 1) {
            TreeNode newRoot = new TreeNode(val);
            newRoot.left = root;
            return newRoot;
        }

        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);
        int currentLevel = 1;

        while (!queue.isEmpty()) {
            int size = queue.size();

            for (int i = 0; i < size; i++) {
                TreeNode node = queue.poll();

                if (currentLevel == depth - 1) {
                    TreeNode leftNode = new TreeNode(val);
                    TreeNode rightNode = new TreeNode(val);

                    leftNode.left = node.left;
                    node.left = leftNode;

                    rightNode.right = node.right;
                    node.right = rightNode;
                } else {
                    if (node.left != null) queue.offer(node.left);
                    if (node.right != null) queue.offer(node.right);
                }
            }

            if (currentLevel == depth - 1) break;
            currentLevel++;
        }

        return root;
    }
}

```

4. Height Of A Binary Tree

The height of a binary tree is the number of edges on the longest path from the root node to a leaf node.

```

public class Solution {
    public int height(TreeNode root) {
        if (root == null) return -1; // return 0 if you define height in terms of number
        int leftHeight = height(root.left);
        int rightHeight = height(root.right);
        return 1 + Math.max(leftHeight, rightHeight);
    }
}

```

markdown

Copy Edit

```

1
/
2 3
/
4

```

- Path from root to deepest leaf: 1 → 2 → 4
- Height = 2 (edges), or 3 (nodes), depending on your definition.

```

import java.util.*;

public class Solution {
    public int height(TreeNode root) {
        if (root == null) return -1; // height in terms of edges; return 0 for nodes

        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        int height = -1; // initialize to -1 since root itself is level 0 (0 edges)

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            height++; // we are going down one level

            for (int i = 0; i < levelSize; i++) {
                TreeNode curr = queue.poll();

                if (curr.left != null) queue.offer(curr.left);
                if (curr.right != null) queue.offer(curr.right);
            }
        }

        return height;
    }
}

```

5. Depth Of A Binary Tree

Term	Definition	Measurement	Typically Measured In
Height of a Node	Number of edges on the longest path from that node to a leaf.	Downward	Edges
Height of a Tree	Height of the root node.	Downward	Edges
Depth of a Node	Number of edges from the root to that node.	Upward	Edges
Depth of Tree	Depth of the deepest node (same as height of tree).	Upward	Edges

Key Points

- Height is generally defined in terms of edges.
 - So a leaf node has height 0, and a single-node tree has height 0.
- Depth is also generally counted in edges.
 - Root node has depth 0.

```

public class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) return 0; // depth in terms of nodes
        int left = maxDepth(root.left);
        int right = maxDepth(root.right);
        return 1 + Math.max(left, right);
    }
}

```

```
import java.util.*;

public class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) return 0;

        Queue<TreeNode> queue = new LinkedList<>();
        queue.offer(root);

        int depth = 0;

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            depth++;

            for (int i = 0; i < levelSize; i++) {
                TreeNode current = queue.poll();

                if (current.left != null) queue.offer(current.left);
                if (current.right != null) queue.offer(current.right);
            }
        }

        return depth; // number of levels = depth in nodes
    }
}
```

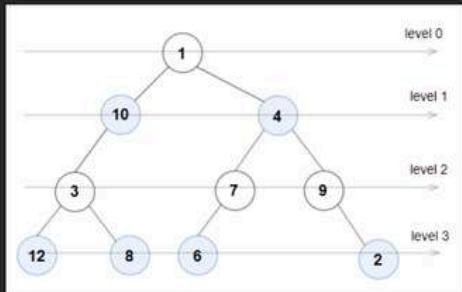
6. Even Odd Tree

A binary tree is named Even-Odd if it meets the following conditions:

- The root of the binary tree is at level index 0, its children are at level index 1, their children are at level index 2, etc.
- For every even-indexed level, all nodes at the level have odd integer values in strictly increasing order (from left to right).
- For every odd-indexed level, all nodes at the level have even integer values in strictly decreasing order (from left to right).

Given the `root` of a binary tree, return `true` if the binary tree is Even-Odd, otherwise return `false`.

Example 1:



Input: `root = [1,10,4,3,null,7,9,12,8,6,null,null,2]`

Output: `true`

Explanation: The node values on each level are:

Level 0: [1]

Level 1: [10,4]

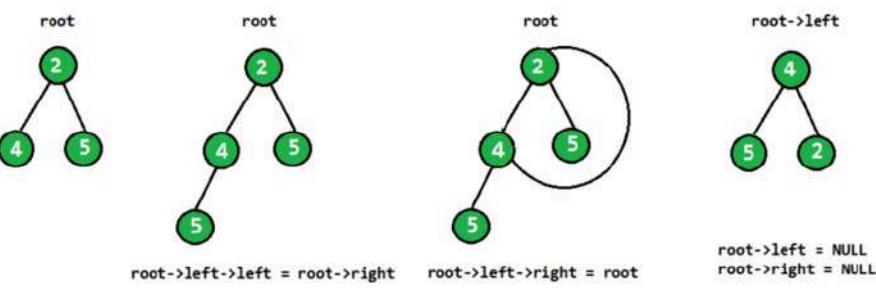
Level 2: [3,7,9]

Level 3: [12,8,6,2]

Since levels 0 and 2 are all odd and increasing and levels 1 and 3 are all even and decreasing, the tree is Even-Odd.

```
class Solution {  
    public boolean isEvenOddTree(TreeNode root) {  
        if (root == null) return true;  
  
        Queue<TreeNode> queue = new LinkedList<>();  
        queue.offer(root);  
        int level = 0;  
  
        while (!queue.isEmpty()) {  
            int size = queue.size();  
            int prevVal = (level % 2 == 0) ? Integer.MIN_VALUE : Integer.MAX_VALUE;  
  
            for (int i = 0; i < size; i++) {  
                TreeNode node = queue.poll();  
                int val = node.val;  
  
                if (level % 2 == 0) {  
                    // Even-indexed level: odd values, strictly increasing  
                    if (val % 2 == 0 || val <= prevVal) return false;  
                } else {  
                    // Odd-indexed level: even values, strictly decreasing  
                    if (val % 2 != 0 || val >= prevVal) return false;  
                }  
  
                prevVal = val;  
  
                if (node.left != null) queue.offer(node.left);  
                if (node.right != null) queue.offer(node.right);  
            }  
  
            level++;  
        }  
  
        return true;  
    }  
}
```

7. Flip Binary Tree



In the flip operation, the leftmost node becomes the root of the flipped tree and its parent becomes its right child and the right sibling becomes its left child and the same should be done for all left most nodes recursively.

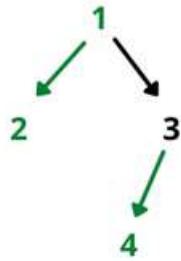
```
public class Solution {  
    public TreeNode upsideDownBinaryTree(TreeNode root) {  
        // base case: if there's no left child, this is the new root  
        if (root == null || root.left == null) return root;  
  
        TreeNode newRoot = upsideDownBinaryTree(root.left);  
  
        root.left.left = root.right; // left child's left becomes current's right  
        root.left.right = root; // left child's right becomes current node  
  
        root.left = null;  
        root.right = null;  
  
        return newRoot;  
    }  
}
```

```
public class Solution {  
    public TreeNode upsideDownBinaryTree(TreeNode root) {  
        TreeNode curr = root;  
        TreeNode prev = null;  
        TreeNode tempRight = null;  
  
        while (curr != null) {  
            TreeNode next = curr.left;  
  
            // flipping  
            curr.left = tempRight;  
            tempRight = curr.right;  
            curr.right = prev;  
  
            // update pointers  
            prev = curr;  
            curr = next;  
        }  
  
        return prev;  
    }  
}
```

Traversal Questions

1. Left View Of Binary Tree

Left View Of A Binary Tree

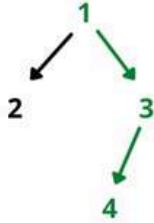


```
class BinaryTree {  
    TreeNode root;  
  
    public void leftView() {  
        if (root == null) return;  
  
        Queue<TreeNode> queue = new LinkedList<>();  
        queue.offer(root);  
  
        while (!queue.isEmpty()) {  
            int n = queue.size();  
  
            for (int i = 0; i < n; i++) {  
                TreeNode curr = queue.poll();  
  
                // Print the first node of each level  
                if (i == 0) {  
                    System.out.println(curr.data);  
                }  
  
                if (curr.left != null)  
                    queue.offer(curr.left);  
                if (curr.right != null)  
                    queue.offer(curr.right);  
            }  
        }  
    }  
}
```

2. Right View Of Binary Tree

Left view of a Binary Tree is set of nodes visible when tree is visited from left side.

Right View Of A Binary Tree



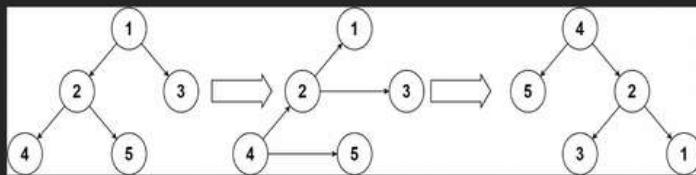
```
class BinaryTree {  
    TreeNode root;  
  
    public void rightView() {  
        if (root == null) return;  
  
        Queue<TreeNode> queue = new LinkedList<>();  
        queue.offer(root);  
  
        while (!queue.isEmpty()) {  
            int n = queue.size();  
  
            for (int i = 0; i < n; i++) {  
                TreeNode curr = queue.poll();  
  
                // Print the last node of each level  
                if (i == n - 1) {  
                    System.out.println(curr.data);  
                }  
  
                if (curr.left != null)  
                    queue.offer(curr.left);  
                if (curr.right != null)  
                    queue.offer(curr.right);  
            }  
        }  
    }  
}
```

3. 156. Binary Tree Upside Down (Flip Binary Tree)

Given a binary tree where all the right nodes are either leaf nodes with a sibling (a left node that shares the same parent node) or empty, flip it upside down and turn it into a tree where the original right nodes turned into left leaf nodes. Return the new root.

Example:

Input: [1,2,3,4,5]



Output: return the root of the binary tree [4,5,2,#,#,3,1]

In the flip operation, the leftmost node becomes the root of the flipped tree and its parent becomes its right child and the right sibling becomes its left child and the same should be done for all left most nodes recursively.

```
public class Solution {  
    public TreeNode upsideDownBinaryTree(TreeNode root) {  
        // base case: if there's no left child, this is the new root  
        if (root == null || root.left == null) return root;  
  
        TreeNode newRoot = upsideDownBinaryTree(root.left);  
  
        root.left.left = root.right; // left child's left becomes current's right  
        root.left.right = root; // left child's right becomes current node  
  
        root.left = null;  
        root.right = null;  
  
        return newRoot;  
    }  
}
```

```
public class Solution {  
    public TreeNode upsideDownBinaryTree(TreeNode root) {  
        TreeNode curr = root;  
        TreeNode prev = null;  
        TreeNode tempRight = null;  
  
        while (curr != null) {  
            TreeNode next = curr.left;  
  
            // flipping  
            curr.left = tempRight;  
            tempRight = curr.right;  
            curr.right = prev;  
  
            // update pointers  
            prev = curr;  
            curr = next;  
        }  
  
        return prev;  
    }  
}
```

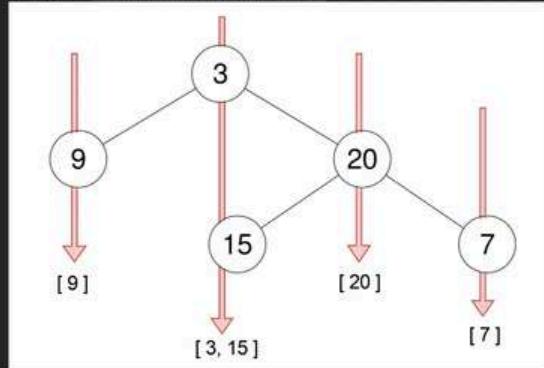
4. 314. Binary Tree Vertical Order Traversal

Given a binary tree, return the *vertical order* traversal of its nodes' values. (ie, from top to bottom, column by column).

If two nodes are in the same row and column, the order should be from **left to right**.

Examples 1:

Input: [3,9,20,null,null,15,7]



Output:

```
[  
    [9],  
    [3,15],  
    [20],  
    [7]  
]
```

```
private static List<List<Integer>> verticalOrder(TreeNode root) {  
    List<List<Integer>> result = new ArrayList<>();  
    if (root == null) return result;  
    // Map: column index -> list of node values  
    Map<Integer, List<Integer>> columnTable = new HashMap<>();  
    // Track min and max column indices  
    int minColumn = 0, maxColumn = 0;  
    // Queue for BFS: each element is [TreeNode, columnIndex]  
    Queue<List<Object>> queue = new LinkedList<>();  
    queue.offer(Arrays.asList(root, 0)); // root at column 0  
    while (!queue.isEmpty()) {  
        List<Object> entry = queue.poll();  
        TreeNode node = (TreeNode) entry.get(0);  
        int column = (int) entry.get(1);  
        columnTable.putIfAbsent(column, new ArrayList<>());  
        columnTable.get(column).add(node.data);  
        minColumn = Math.min(minColumn, column);  
        maxColumn = Math.max(maxColumn, column);  
  
        if (node.left != null) {  
            queue.offer(Arrays.asList(node.left, column - 1));  
        }  
        if (node.right != null) {  
            queue.offer(Arrays.asList(node.right, column + 1));  
        }  
    }  
    for (int col = minColumn; col <= maxColumn; col++) {  
        result.add(columnTable.get(col));  
    }  
    return result;  
}
```

```

public static void main(String[] args) {
    TreeNode root = new TreeNode( data: 1 );
    root.left = new TreeNode( data: 2 );
    root.right = new TreeNode( data: 3 );
    root.left.left = new TreeNode( data: 4 );
    root.left.right = new TreeNode( data: 5 );
    mirror(root);
    System.out.println(root); // 3 1 5 2 4
    System.out.println(verticalOrder(root)); // [[3], [1, 5], [2], [4]]
}
}

```

5. 987. Vertical Order Traversal of a Binary Tree

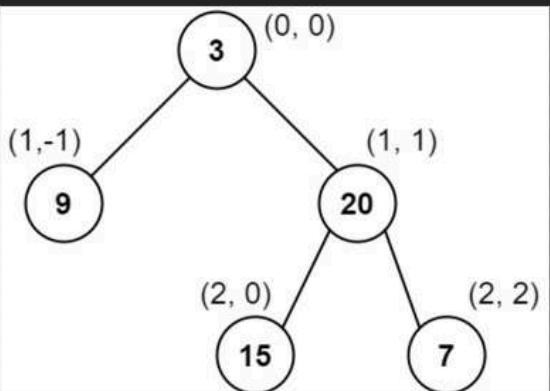
Given the `root` of a binary tree, calculate the vertical order traversal of the binary tree.

For each node at position `(row, col)`, its left and right children will be at positions `(row + 1, col - 1)` and `(row + 1, col + 1)` respectively. The root of the tree is at `(0, 0)`.

The vertical order traversal of a binary tree is a list of top-to-bottom orderings for each column index starting from the leftmost column and ending on the rightmost column. There may be multiple nodes in the same row and same column. In such a case, sort these nodes by their values.

Return the *vertical order traversal* of the binary tree.

Example 1:



`Input: root = [3,9,20,null,null,15,7]`

`Output: [[9],[3,15],[20],[7]]`

`Explanation:`

Column -1: Only node 9 is in this column.

Column 0: Nodes 3 and 15 are in this column in that order from top to bottom.

Column 1: Only node 20 is in this column.

Column 2: Only node 7 is in this column.

```

class Solution {
    static class NodeInfo {
        int row, val;
        NodeInfo(int row, int val) {
            this.row = row;
            this.val = val;
        }
    }

    public List<List<Integer>> verticalTraversal(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) return result;

        // Map: column index -> list of NodeInfo(row, value)
        Map<Integer, List<NodeInfo>> columnMap = new HashMap<>();

        int minCol = 0, maxCol = 0;

        // Queue stores [TreeNode, col, row]
        Queue<List<Object>> queue = new LinkedList<>();
        queue.offer(Arrays.asList(root, 0, 0)); // root at (col=0, row=0)

        while (!queue.isEmpty()) {
            List<Object> curr = queue.poll();
            TreeNode node = (TreeNode) curr.get(0);
            int col = (int) curr.get(1);
            int row = (int) curr.get(2);

            columnMap.putIfAbsent(col, new ArrayList<>());
            columnMap.get(col).add(new NodeInfo(row, node.val));

            minCol = Math.min(minCol, col);
            maxCol = Math.max(maxCol, col);

            if (node.left != null)
                queue.offer(Arrays.asList(node.left, col - 1, row + 1));
            if (node.right != null)
                queue.offer(Arrays.asList(node.right, col + 1, row + 1));
        }

        for (int col = minCol; col <= maxCol; col++) {
            List<NodeInfo> nodeList = columnMap.get(col);
            nodeList.sort((a, b) -> {
                if (a.row == b.row) return a.val - b.val; // same row: sort by val
                return a.row - b.row; // otherwise sort by row
            });
        }

        List<Integer> sortedColumn = new ArrayList<>();
        for (NodeInfo ni : nodeList) {
            sortedColumn.add(ni.val);
        }
        result.add(sortedColumn);
    }

    return result;
}

```



Feature	314. Binary Tree Vertical Order Traversal	987. Vertical Order Traversal of a Binary Tree
Traversal Type	BFS-based vertical level order traversal	BFS with column + row-based vertical traversal
Sorting Within Column	Top to bottom (BFS order)	Top to bottom by row; tie-breaker: node value
Handling Same Position Nodes	Appears in BFS order (left-to-right)	If multiple nodes have same row and column, sort by value
Example Difference	BFS ensures left-to-right for same level	Must sort nodes by row, and by value if tied
Use Case	Simpler vertical grouping	More granular, position-based ordering

6. 1372. Longest ZigZag Path in a Binary Tree

You are given the `root` of a binary tree.

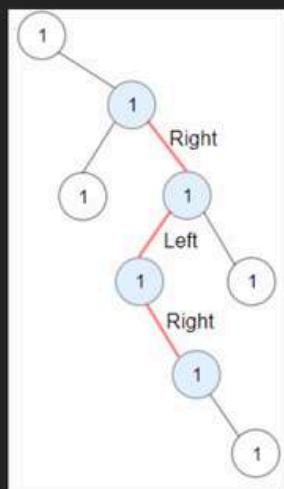
A ZigZag path for a binary tree is defined as follow:

- Choose any node in the binary tree and a direction (right or left).
- If the current direction is right, move to the right child of the current node; otherwise, move to the left child.
- Change the direction from right to left or from left to right.
- Repeat the second and third steps until you can't move in the tree.

Zigzag length is defined as the number of nodes visited - 1. (A single node has a length of 0).

Return the longest ZigZag path contained in that tree.

Example 1:



Input: `root = [1,null,1,1,1,null,null,1,1,null,1,null,null,1]`

Output: 3

Explanation: Longest ZigZag path in blue nodes (right → left → right).

A ZigZag path in a binary tree is defined as a path starting from any node and moving alternately between the left and right child.

```
class Solution {
    private int maxLen = 0;

    public int longestZigZag(TreeNode root) {
        dfs(root, true, 0); // true means move left next
        dfs(root, false, 0); // false means move right next
        return maxLen;
    }

    private void dfs(TreeNode node, boolean isLeft, int length) {
        if (node == null) return;

        maxLen = Math.max(maxLen, length);

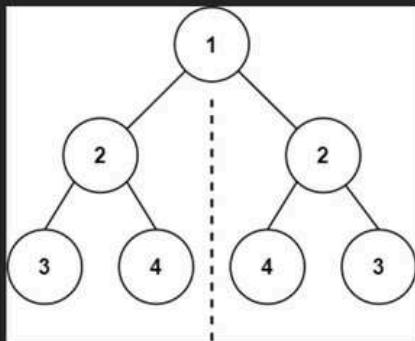
        if (isLeft) {
            dfs(node.left, false, length + 1); // go left, next should go right
            dfs(node.right, true, 1); // restart from right
        } else {
            dfs(node.right, true, length + 1); // go right, next should go left
            dfs(node.left, false, 1); // restart from left
        }
    }
}
```

7.101. Symmetric Tree

Boundary Level Traversal

Given the `root` of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).

Example 1:



Input: `root = [1,2,2,3,4,4,3]`

Output: `true`

```

/*
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if (!root) return true;
        return isMirror(root->left, root->right);
    }

private:
    bool isMirror(TreeNode* t1, TreeNode* t2) {
        if (!t1 && !t2) return true;
        if (!t1 || !t2) return false;
        if (t1->val != t2->val) return false;

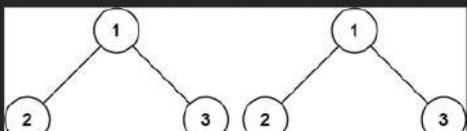
        return isMirror(t1->left, t2->right) && isMirror(t1->right, t2->left);
    }
};
  
```

8.100. Same Tree

Given the roots of two binary trees `p` and `q`, write a function to check if they are the same or not.

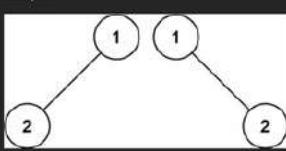
Two binary trees are considered the same if they are structurally identical and the nodes have the same value.

Example 1:



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

Example 2:



Input: `p = [1,2], q = [1,null,2]`
Output: `false`

```

public static List<Integer> getBoundary(TreeNode root) {
    List<Integer> boundary = new ArrayList<>();
    if (root == null) return boundary;
    boundary.add(root.data); // 1. Root
    System.out.println(boundary); // [1]
    // 2. Left boundary (excluding leaf)
    addLeftBoundary(root.left, boundary);
    System.out.println(boundary); // [1, 2]
    // 3. Leaf nodes
    addLeaves(root.left, boundary);
    System.out.println(boundary); // [1, 2, 4, 7, 8]
    addLeaves(root.right, boundary);
    System.out.println(boundary); // [1, 2, 4, 7, 8, 9]
    // 4. Right boundary (excluding leaf, added in reverse)
    addRightBoundary(root.right, boundary);
    System.out.println(boundary); // [1, 2, 4, 7, 8, 9, 6, 3]
    return boundary;
}
  
```

```

public class BoundaryTraversal {
    static class TreeNode {
        int data;
        TreeNode left, right;
        TreeNode(int data) {
            this.data = data;
        }
    }

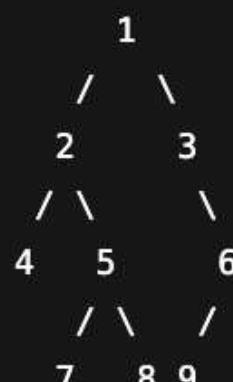
    public static List<Integer> getBoundary(TreeNode root) {
        List<Integer> boundary = new ArrayList<>();
        if (root == null) return boundary;
        boundary.add(root.data); // 1. Root
        // 2. Left boundary (excluding leaf)
        addLeftBoundary(root.left, boundary);
        // 3. Leaf nodes
        addLeaves(root.left, boundary);
        addLeaves(root.right, boundary);
        // 4. Right boundary (excluding leaf, added in reverse)
        addRightBoundary(root.right, boundary);
        return boundary;
    }

    private static void addLeftBoundary(TreeNode node, List<Integer> boundary) {
        while (node != null) {
            if (!(node.left == null && node.right == null)) {
                boundary.add(node.data);
            }
            node = (node.left != null) ? node.left : node.right;
        }
    }

    private static void addRightBoundary(TreeNode node, List<Integer> boundary) {
        List<Integer> temp = new ArrayList<>();
        while (node != null) {
            if (!(node.left == null && node.right == null)) {
                temp.add(node.data);
            }
            node = (node.right != null) ? node.right : node.left;
        }
        // Add in reverse:
        for (int i = temp.size() - 1; i >= 0; i--) {
            boundary.add(temp.get(i));
        }
    }

    private static void addLeaves(TreeNode node, List<Integer> boundary) {
        if (node == null) return;
        if (node.left == null && node.right == null) {
            boundary.add(node.data);
            return;
        }
        addLeaves(node.left, boundary);
        addLeaves(node.right, boundary);
    }

    // Sample usage
    public static void main(String[] args) {
        TreeNode root = new TreeNode(data: 1);
        root.left = new TreeNode(data: 2);
        root.left.left = new TreeNode(data: 4);
        root.left.right = new TreeNode(data: 5);
        root.left.right.left = new TreeNode(data: 7);
        root.left.right.right = new TreeNode(data: 8);
        root.right = new TreeNode(data: 3);
        root.right.right = new TreeNode(data: 6);
        root.right.right.left = new TreeNode(data: 9);
        List<Integer> boundary = getBoundary(root);
        System.out.println(boundary); // Output: [1, 2, 4, 7, 8, 9, 6, 3]
    }
}
  
```



```

class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if (!p && !q) return true;
        if (!p || !q) return false;
        if (p->val != q->val) return false;

        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};

```

Binary Search Tree

A Binary Search Tree (BST) is a type of binary tree with a specific ordering property that makes searching efficient.

A **Binary Search Tree** is a binary tree in which for every node:

- **Left subtree** contains only nodes with values **less than** the node's value.
- **Right subtree** contains only nodes with values **greater than** the node's value.
- No duplicate nodes (in standard BSTs).

⌚ Time Complexities:

Operation	Average Case	Worst Case (Skewed Tree)
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

✓ A balanced BST (like AVL or Red-Black Tree) keeps operations close to $O(\log n)$.

```

class Solution {
    TreeNode root;

    static class TreeNode {
        int val;
        TreeNode left, right;
        TreeNode(int val) {
            this.val = val;
        }
    }

    public TreeNode insert(TreeNode root, int val) {
        if (root == null) return new TreeNode(val);

        if (val < root.val)
            root.left = insert(root.left, val);
        else if (val > root.val)
            root.right = insert(root.right, val);

        return root;
    }

    public boolean search(TreeNode root, int key) {
        if (root == null) return false;
        if (key == root.val) return true;
        else if (key < root.val) return search(root.left, key);
        else return search(root.right, key);
    }

    public void inorder(TreeNode root) {
        if (root != null) {
            inorder(root.left);
            System.out.print(root.val + " ");
            inorder(root.right);
        }
    }
};

```

```

class Solution {
    public TreeNode deleteNode(TreeNode root, int key) {
        if (root == null) {
            return null;
        }

        if (key < root.val) {
            root.left = deleteNode(root.left, key);
        } else if (key > root.val) {
            root.right = deleteNode(root.right, key);
        } else {
            if (root.left == null) {
                return root.right;
            } else if (root.right == null) {
                return root.left;
            }
            root.val = findMin(root.right);
            root.right = deleteNode(root.right, root.val);
        }

        return root;
    }

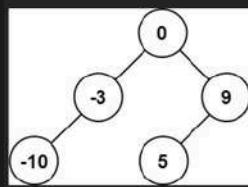
    private int findMin(TreeNode node) {
        while (node.left != null) {
            node = node.left;
        }
        return node.val;
    }
};

```

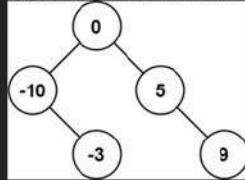
1.108. Convert Sorted Array to Binary Search Tree

Given an integer array `nums` where the elements are sorted in ascending order, convert it to a height-balanced binary search tree.

Example 1:



Input: `nums = [-10,-3,0,5,9]`
Output: `[0,-3,9,-10,null,5]`
Explanation: `[0,-10,5,null,-3,null,9]` is also accepted:



```
@Override
public String toString(){
    return dfs(node);
}

private String dfs(TreeNode node) {
    if (node == null) return "";
    String leftStr = dfs(node.left);
    String rightStr = dfs(node.right);
    return (leftStr.isEmpty() ? "" : leftStr + ", ") + node.val + (rightStr.isEmpty() ? "" : ", " + rightStr);
}

public static TreeNode sortedArrayToBST(int[] nums) {
    return dfs(nums, left: 0, right: nums.length - 1);
}

private static TreeNode dfs(int[] nums, int left, int right) {
    if(left > right){
        return null;
    }

    int mid = left + (right - left) / 2;
    TreeNode root = new TreeNode(nums[mid]);
    root.left = dfs(nums, left, right: mid - 1);
    root.right = dfs(nums, left: mid + 1, right);
    return root;
}

public static void main(String[] args) {
    int[] nums = {-10, -3, 0, 5, 9};
    System.out.println(sortedArrayToBST(nums));
}
```

2.109. Convert Sorted List to Binary Search Tree

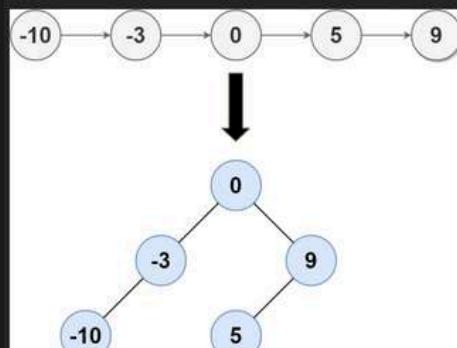
109. Convert Sorted List to Binary Search Tree

Medium

Given the `head` of a singly linked list where elements are sorted in ascending order, convert it to a height-balanced binary search tree.

tree.

Example 1:



Input: `head = [-10,-3,0,5,9]`
Output: `[0,-3,9,-10,null,5]`
Explanation: One possible answer is `[0,-3,9,-10,null,5]`, which represents the shown height balanced BST.

Binary Search Tree ▾ Auto

```
23     */
24     */
25     */
26 class Solution {
27     public TreeNode sortedListToBST(ListNode head) {
28         if (head == null) {
29             return null;
30         }
31         return sortedListToBSTHelper(head, null);
32     }
33
34     private TreeNode sortedListToBSTHelper(ListNode head, ListNode tail) {
35         if (head == tail) {
36             return null;
37         }
38
39         ListNode slow = head;
40         ListNode fast = head;
41
42         while (fast != tail && fast.next != tail) {
43             slow = slow.next;
44             fast = fast.next.next;
45         }
46
47         TreeNode node = new TreeNode(slow.val);
48         node.left = sortedListToBSTHelper(head, slow);
49         node.right = sortedListToBSTHelper(slow.next, tail);
50
51     }
52 }
```

3.776. Split BST

776. Split BST

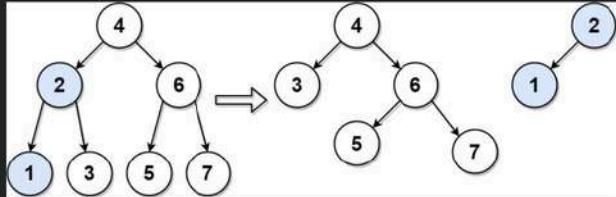
Medium

Given the `root` of a binary search tree (BST) and an integer `target`, split the tree into two subtrees where one subtree has nodes that are all smaller or equal to the target value, while the other subtree has all nodes that are greater than the target value. It is not necessarily the case that the tree contains a node with the value `target`.

Additionally, most of the structure of the original tree should remain. Formally, for any child `c` with parent `p` in the original tree, if they are both in the same subtree after the split, then node `c` should still have the parent `p`.

Return an array of the two roots of the two subtrees.

Example 1:



Input: `root = [4,2,6,1,3,5,7]`, `target = 2`
Output: `[[2,1],[4,3,6,null,null,5,7]]`

Example 2:

Input: `root = [1]`, `target = 1`
Output: `[[1],[]]`

Constraints:

- The number of nodes in the tree is in the range `[1, 50]`.
- $0 \leq \text{Node.val}, \text{target} \leq 1000$

```

class Solution {
    private int target;

    public TreeNode[] splitBST(TreeNode root, int target) {
        this.target = target;
        return split(root);
    }

    private TreeNode[] split(TreeNode node) {
        if (node == null) {
            return new TreeNode[]{null, null};
        }

        if (node.val <= target) {
            TreeNode[] rightSplit = split(node.right);
            node.right = rightSplit[0];
            rightSplit[0] = node;
            return rightSplit;
        } else {
            TreeNode[] leftSplit = split(node.left);
            node.left = leftSplit[1];
            leftSplit[1] = node;
            return leftSplit;
        }
    }
}

def splitBST(root, V):
    if not root:
        return [None, None]

    if root.val <= V:
        # root belongs to left tree
        left_subtree, right_subtree = splitBST(root.right, V)
        root.right = left_subtree
        return [root, right_subtree]
    else:
        # root belongs to right tree
        left_subtree, right_subtree = splitBST(root.left, V)
        root.left = right_subtree
        return [left_subtree, root]

```

InOrder

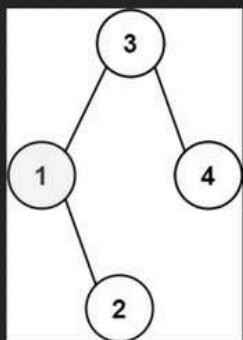
1.230. Kth Smallest Element in a BST

230. Kth Smallest Element in a BST

Medium

Given the `root` of a binary search tree, and an integer `k`, return the k^{th} smallest value (1-indexed) of all the values of the nodes in the tree.

Example 1:



Input: `root = [3,1,4,null,2]`, `k = 1`
Output: `1`

Example 2:



```

1  /**
2  * Definition for a binary tree node.
3  * public class TreeNode {
4  *     int val;
5  *     TreeNode left;
6  *     TreeNode right;
7  *     TreeNode() {}
8  *     TreeNode(int val) { this.val = val; }
9  *     TreeNode(int val, TreeNode left, TreeNode right) {
10 *         this.val = val;
11 *         this.left = left;
12 *         this.right = right;
13 *     }
14 * }
15 */
16 class Solution {
17     private int count = 0;
18     private int result = 0;
19
20     public int kthSmallest(TreeNode root, int k) {
21         inorder(root, k);
22         return result;
23     }
24
25     private void inorder(TreeNode root, int k) {
26         if (root == null) {
27             return;
28         }
29
30         inorder(root.left, k);
31
32         count++;
33         if (count == k) {
34             result = root.val;
35             return;
36         }
37
38         inorder(root.right, k);
39     }
40 }

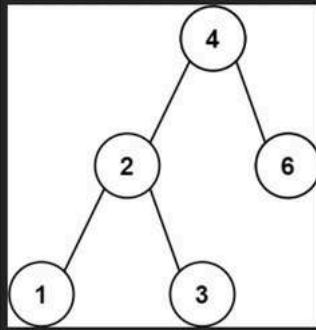
```

530. Minimum Absolute Difference in BST

Easy

Given the `root` of a Binary Search Tree (BST), return *the minimum absolute difference between the values of any two different nodes in the tree*.

Example 1:



Input: `root = [4,2,6,1,3]`
Output: 1

Example 2:



Binary Search Tree ✓ Auto

```
6   *   TreeNode right;
7   *   TreeNode() {}
8   *   TreeNode(int val) { this.val = val; }
9   *   TreeNode(int val, TreeNode left, TreeNode right) {
10      *      this.val = val;
11      *      this.left = left;
12      *      this.right = right;
13   }
14 }
15 */
16 class Solution {
17     private int minDiff;
18     private Integer prev;
19
20     public int getMinimumDifference(TreeNode root) {
21         minDiff = Integer.MAX_VALUE;
22         prev = null;
23         inorder(root);
24         return minDiff;
25     }
26
27     private void inorder(TreeNode node) {
28         if (node == null) {
29             return;
30         }
31         inorder(node.left);
32         if (prev != null) {
33             minDiff = Math.min(minDiff, node.val - prev);
34         }
35         prev = node.val;
36         inorder(node.right);
37     }
38 }
39 }
```

501. Find Mode in Binary Search Tree

Easy

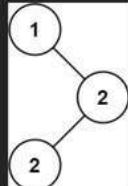
Given the `root` of a binary search tree (BST) with duplicates, return *all the modes* (i.e., the most frequently occurred element) in it.

If the tree has more than one mode, return them in any order.

Assume a BST is defined as follows:

- The left subtree of a node contains only nodes with keys less than or equal to the node's key.
- The right subtree of a node contains only nodes with keys greater than or equal to the node's key.
- Both the left and right subtrees must also be binary search trees.

Example 1:



Input: `root = [1,null,2,2]`
Output: [2]

Example 2:

Input: `root = [0]`
Output: [0]

Code

```
Binary Search Tree ✓ Auto
18 class Solution {
19     private int maxCount;
20     private int currentCount;
21     private int currentValue;
22     private List<Integer> modes;
23
24     public int[] findMode(TreeNode root) {
25         if (root == null) {
26             return new int[0];
27         }
28         maxCount = 0;
29         currentCount = 0;
30         currentValue = Integer.MIN_VALUE;
31         modes = new ArrayList<>();
32         traverse(root);
33
34         int[] result = new int[modes.size()];
35         for (int i = 0; i < modes.size(); i++) {
36             result[i] = modes.get(i);
37         }
38         return result;
39     }
40
41     private void traverse(TreeNode node) {
42         if (node == null) {
43             return;
44         }
45         traverse(node.left);
46         handleValue(node.val);
47         traverse(node.right);
48     }
49
50     private void handleValue(int val) {
51         if (val != currentValue) {
52             currentValue = val;
53             currentCount = 0;
54         }
55         currentCount++;
56         if (currentCount > maxCount) {
57             maxCount = currentCount;
58             modes.clear();
59             modes.add(currentVal);
60         } else if (currentCount == maxCount) {
61             modes.add(currentVal);
62         }
63     }
64 }
```

Testcase

Constraints:

PostOrder

Description Solving

450. Delete Node in a BST

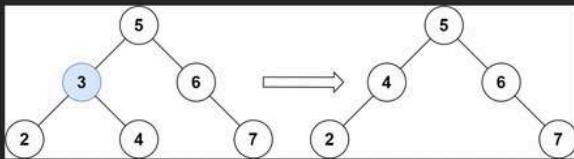
Medium

Given a root node reference of a BST and a key, delete the node with the given key in the BST. Return the root node reference (possibly updated) of the BST.

Basically, the deletion can be divided into two stages:

- Search for a node to remove.
- If the node is found, delete the node.

Example 1:



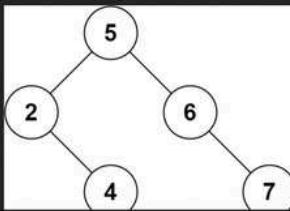
Input: root = [5,3,6,2,4,null,7], key = 3

Output: [5,4,6,2,null,null,7]

Explanation: Given key to delete is 3. So we find the node with value 3 and delete it.

One valid answer is [5,4,6,2,null,null,7], shown in the above BST.

Please notice that another valid answer is [5,2,6,null,4,null,7] and it's also accepted.



Code

```
Binary Search Tree ▾ Auto
```

```
1 //*
2 * Definition for a binary tree node.
3 * public class TreeNode {
4 *     int val;
5 *     TreeNode left;
6 *     TreeNode right;
7 *     TreeNode() {}
8 *     TreeNode(int val) { this.val = val; }
9 *     TreeNode(int val, TreeNode left, TreeNode right) {
10 *         this.val = val;
11 *         this.left = left;
12 *         this.right = right;
13 *     }
14 * }
15 */
16 class Solution {
17     public TreeNode deleteNode(TreeNode root, int key) {
18         if (root == null) {
19             return null;
20         }
21         if (key < root.val) {
22             root.left = deleteNode(root.left, key);
23         } else if (key > root.val) {
24             root.right = deleteNode(root.right, key);
25         } else {
26             if (root.left == null) {
27                 return root.right;
28             } else if (root.right == null) {
29                 return root.left;
30             }
31             root.val = findMin(root.right);
32             root.right = deleteNode(root.right, root.val);
33         }
34         return root;
35     }
36
37
38
39 private int findMin(TreeNode node) {
40     while (node.left != null) {
41         node = node.left;
42     }
43     return node.val;
44 }
```

Testcase

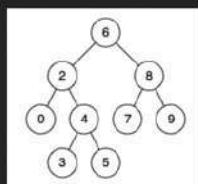
235. Lowest Common Ancestor of a Binary Search Tree

Medium

Given a binary search tree (BST), find the lowest common ancestor (LCA) node of two given nodes in the BST.

According to the [definition of LCA on Wikipedia](#): "The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow a node to be a descendant of itself)."

Example 1:



Input: root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

Output: 6

Explanation: The LCA of nodes 2 and 8 is 6.

Binary Search Tree ▾ Auto

```
1 /**
2 * Definition for a binary tree node.
3 * public class TreeNode {
4 *     int val;
5 *     TreeNode left;
6 *     TreeNode right;
7 *     TreeNode(int x) { val = x; }
8 * }
9 */
10
11 public class Solution {
12
13     public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
14         if (root == null) {
15             return null;
16         }
17
18         if (p.val < root.val && q.val < root.val) {
19             return lowestCommonAncestor(root.left, p, q);
20         } else if (p.val > root.val && q.val > root.val) {
21             return lowestCommonAncestor(root.right, p, q);
22         } else {
23             return root;
24         }
25     }
26
27 }
```

Graph

Graph algorithms are a fundamental part of DSA and come up frequently in interviews and real-world applications

❖ Basic Graph Representations

- Adjacency List – space-efficient for sparse graphs.
- Adjacency Matrix – good for dense graphs.
- Edge List – just a list of edges; useful for Kruskal's algorithm.

Questions:

1. **133. Clone Graph** – Graph traversal + HashMap for visited nodes
2. **785. Is Graph Bipartite?** – BFS/DFS with coloring
3. **802. Find Eventual Safe States** – Reverse graph + cycle detection
4. **834. Sum of Distances in Tree** – Tree DP (postorder + preorder)
5. **1129. Shortest Path with Alternating Colors** – BFS with state (color) tracking
6. **1377. Frog Position After T Seconds** – DFS + probability distribution

```
static class Node {  
    public int val;  
    public List<Node> neighbors;  
  
    public Node(int _val) {  
        val = _val;  
        neighbors = new ArrayList<Node>();  
    }  
}  
  
private static List<Integer> bfs(Node node) {  
    Queue<Node> queue = new ArrayDeque<>();  
    queue.add(node);  
    List<Integer> result = new ArrayList<>();  
    Map<Node, Boolean> visited = new HashMap<>();  
    visited.put(node, true);  
  
    while (!queue.isEmpty()) {  
        Node curr = queue.poll();  
        result.add(curr.val);  
        List<Node> neighbors = curr.neighbors;  
  
        for (Node neighbor : neighbors) {  
            if (!visited.containsKey(neighbor)) {  
                visited.put(neighbor, true);  
                queue.add(neighbor);  
            }  
        }  
    }  
  
    return result;  
}
```

```

public static void main(String[] args) {
    Node node1 = new Node(_val: 1);
    Node node2 = new Node(_val: 2);
    Node node3 = new Node(_val: 3);
    Node node4 = new Node(_val: 4);
    node1.neighbors = Arrays.asList(node2, node4);
    node2.neighbors = Arrays.asList(node1, node3);
    node3.neighbors = Arrays.asList(node2, node4);
    node4.neighbors = Arrays.asList(node3, node1);
    System.out.println(bfs(node1)); // [1, 2, 4, 3]
}
}

```

Adjacency List

```

public class Graph {

    private Map<Integer, List<Integer>> adjList;

    public Graph() {
        adjList = new HashMap<>();
    }

    public void addVertex(int val) {
        adjList.putIfAbsent(val, new ArrayList<>());
    }

    public void addEdge(int from, int to) {
        adjList.putIfAbsent(from, new ArrayList<>());
        adjList.putIfAbsent(to, new ArrayList<>());
        adjList.get(from).add(to);
        adjList.get(to).add(from);
    }

    public void removeVertex(int val) {
        if (!adjList.containsKey(val)) return;
        for (int neighbor : adjList.get(val)) {
            adjList.get(neighbor).remove(Integer.valueOf(val));
        }
        adjList.remove(val);
    }

    public void printGraph() {
        for (Map.Entry<Integer, List<Integer>> entry : adjList.entrySet()) {
            System.out.println(entry.getKey() + " -> " + entry.getValue());
        }
    }
}

```

```

    public static void main(String[] args) {
        Graph g = new Graph();
        g.addVertex( val: 1);
        g.addVertex( val: 2);
        g.addVertex( val: 3);
        g.addVertex( val: 4);

        g.addEdge(1, 2);
        g.addEdge(1, 3);
        g.addEdge(2, 4);
        System.out.println("Graph before removing vertex 2:");
        g.printGraph();
        g.removeVertex( val: 2);

        System.out.println("\nGraph after removing vertex 2:");
        g.printGraph();
    }
}

```

Adjacency Matrix

```

public class GraphMatrix {

    private int[][] adjMatrix;
    private int vertexCount;

    public GraphMatrix() {
        adjMatrix = new int[0][0];
        vertexCount = 0;
    }

    public void addVertex() {
        vertexCount++;
        int[][] newMatrix = new int[vertexCount][vertexCount];
        for (int i = 0; i < vertexCount - 1; i++) {
            for (int j = 0; j < vertexCount - 1; j++) {
                newMatrix[i][j] = adjMatrix[i][j];
            }
        }
        adjMatrix = newMatrix;
    }

    public void addEdge(int from, int to) {
        if (from >= vertexCount || to >= vertexCount) {
            throw new IllegalArgumentException("Vertex index out of bounds");
        }
        adjMatrix[from][to] = 1;
        adjMatrix[to][from] = 1;
    }
}

```

```

public void removeVertex(int index) {
    if (index < 0 || index >= vertexCount) {
        throw new IllegalArgumentException("Invalid vertex index");
    }
    int[][] newMatrix = new int[vertexCount - 1][vertexCount - 1];
    for (int i = 0, r = 0; i < vertexCount; i++) {
        if (i == index) continue;
        for (int j = 0, c = 0; j < vertexCount; j++) {
            if (j == index) continue;
            newMatrix[r][c] = adjMatrix[i][j];
            c++;
        }
        r++;
    }
    vertexCount--;
    adjMatrix = newMatrix;
}

public void printGraph() {
    System.out.println("Adjacency Matrix:");
    for (int i = 0; i < vertexCount; i++) {
        System.out.println(Arrays.toString(adjMatrix[i]));
    }
}

public static void main(String[] args) {
    GraphMatrix graph = new GraphMatrix();
    graph.addVertex();
    graph.addVertex();
    graph.addVertex();
    graph.addVertex();
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 3);
    System.out.println("Graph before removing vertex 1:");
/*
Adjacency List:
[0, 1, 1, 0]
[1, 0, 0, 1]
[1, 0, 0, 0]
[0, 1, 0, 0]
*/
    graph.printGraph();
    graph.removeVertex(index: 1);
    System.out.println("\nGraph after removing vertex 1:");
    graph.printGraph();
/*
Adjacency Matrix:
[0, 1, 0]
[1, 0, 0]
[0, 0, 0]
 */
}
}

```

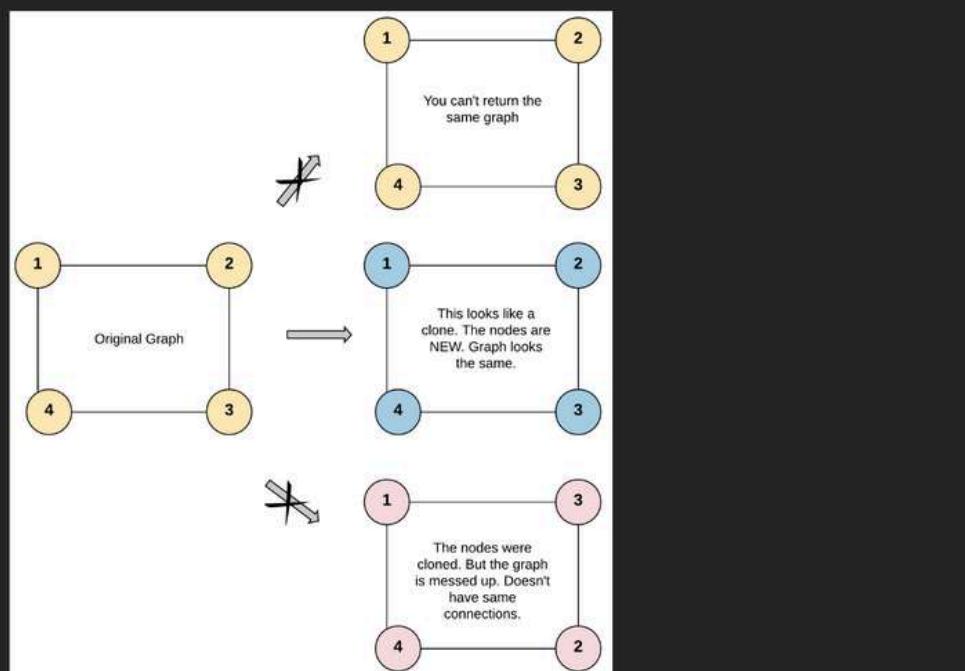
Skip the row increment for new matrix in case of removeVertex method in case the index matches

```
private static List<Integer> dfs(Node node) {
    List<Integer> result = new ArrayList<>();
    Map<Node, Boolean> visited = new HashMap<>();
    visited.put(node, true);
    dfsHelper(node, result, visited);
    return result;
}

private static void dfsHelper(Node node, List<Integer> result, Map<Node, Boolean> visited) {
    visited.put(node, true);
    result.add(node.val);
    List<Node> neighbors = node.neighbors;
    for (Node neighbor : neighbors) {
        if (!visited.containsKey(neighbor)) {
            dfsHelper(neighbor, result, visited);
        }
    }
}

public static void main(String[] args) {
    Node node1 = new Node(_val: 1);
    Node node2 = new Node(_val: 2);
    Node node3 = new Node(_val: 3);
    Node node4 = new Node(_val: 4);
    node1.neighbors = Arrays.asList(node2, node4);
    node2.neighbors = Arrays.asList(node1, node3);
    node3.neighbors = Arrays.asList(node2, node4);
    node4.neighbors = Arrays.asList(node3, node1);
    System.out.println(bfs(node1)); // [1, 2, 4, 3]
    List<Integer> dfsResult = dfs(node1);
    System.out.println(dfsResult); // [1, 2, 3, 4]
}
}
```

133. Clone Graph – Graph traversal + HashMap for visited nodes



Input: adjList = [[2,4],[1,3],[2,4],[1,3]]

Output: [[2,4],[1,3],[2,4],[1,3]]

Explanation: There are 4 nodes in the graph.

1st node (val = 1)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

2nd node (val = 2)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

3rd node (val = 3)'s neighbors are 2nd node (val = 2) and 4th node (val = 4).

4th node (val = 4)'s neighbors are 1st node (val = 1) and 3rd node (val = 3).

```
private static Node cloneGraph(Node node) {
    if (node == null) return null;
    Map<Node, Node> cloneMap = new HashMap<>();
    return dfs(node, cloneMap);
}

private static Node dfs(Node node, Map<Node, Node> cloneMap) {
    List<Node> neighbors = node.neighbors;
    if (cloneMap.containsKey(node)) {
        return cloneMap.get(node);
    }
    Node clone = new Node(node.val);
    cloneMap.put(node, clone);

    for (Node neighbor : neighbors) {
        clone.neighbors.add(dfs(neighbor, cloneMap));
    }

    return clone;
}

public static void main(String[] args) {
    Node node1 = new Node(_val: 1);
    Node node2 = new Node(_val: 2);
    Node node3 = new Node(_val: 3);
    Node node4 = new Node(_val: 4);
    node1.neighbors = Arrays.asList(node2, node4);
    node2.neighbors = Arrays.asList(node1, node3);
    node3.neighbors = Arrays.asList(node2, node4);
    node4.neighbors = Arrays.asList(node3, node1);
    System.out.println(cloneGraph(node1).val); // 1
}
```

2. 785. Is Graph Bipartite? – BFS+ Graph Coloring

Medium

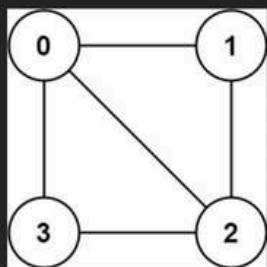
There is an undirected graph with n nodes, where each node is numbered between 0 and $n - 1$. You are given a 2D array `graph`, where `graph[u]` is an array of nodes that node u is adjacent to. More formally, for each v in `graph[u]`, there is an undirected edge between node u and node v . The graph has the following properties:

- There are no self-edges (`graph[u]` does not contain u).
- There are no parallel edges (`graph[u]` does not contain duplicate values).
- If v is in `graph[u]`, then u is in `graph[v]` (the graph is undirected).
- The graph may not be connected, meaning there may be two nodes u and v such that there is no path between them.

A graph is bipartite if the nodes can be partitioned into two independent sets A and B such that every edge in the graph connects a node in set A and a node in set B .

Return `true` if and only if it is bipartite.

Example 1:



Input: `graph = [[1,2,3],[0,2],[0,1,3],[0,2]]`

Output: `false`

Explanation: There is no way to partition the nodes into two independent sets such that every edge connects a node in one and a node in the other.

```
private static boolean isBipartite(int[][] graph) {
    int n = graph.length;
    int[] color = new int[n]; // 0: unvisited, 1: color1, -1: color2

    for (int i = 0; i < n; i++) {
        if (color[i] != 0) continue; // already visited

        Queue<Integer> queue = new LinkedList<>();
        queue.offer(i);
        color[i] = 1;

        while (!queue.isEmpty()) {
            int curr = queue.poll();

            for (int neighbor : graph[curr]) {
                if (color[neighbor] == 0) {
                    color[neighbor] = -color[curr]; // assign opposite color
                    queue.offer(neighbor);
                } else if (color[neighbor] == color[curr]) {
                    return false; // same color neighbor => not bipartite
                }
            }
        }
    }

    return true;
}
```

Mark the node as visited.

Perform the BFS, if it not visited mark it with the opposite color, offer it to the queue or else return false

💡 What is a Bipartite Graph?

A graph is bipartite if you can split its nodes into two groups such that:

- No two nodes within the same group are connected.
- All edges go between the two groups.

This is equivalent to being able to color the graph with two colors (say 1 and -1) such that no two adjacent nodes have the same color.

3. 802. Find Eventual Safe States – Reverse graph + Kahns BFS

There is a directed graph of n nodes with each node labeled from 0 to $n - 1$. The graph is represented by a 0-indexed 2D integer array `graph` where `graph[i]` is an integer array of nodes adjacent to node i , meaning there is an edge from node i to each node in `graph[i]`.

A node is a terminal node if there are no outgoing edges. A node is a safe node if every possible path starting from that node leads to a terminal node (or another safe node).

Return an array containing all the safe nodes of the graph. The answer should be sorted in ascending order.

Example 1:



Input: `graph = [[1,2],[2,3],[5],[0],[5],[],[]]`

Output: `[2,4,5,6]`

Explanation: The given graph is shown above.

Nodes 5 and 6 are terminal nodes as there are no outgoing edges from either of them.

Every path starting at nodes 2, 4, 5, and 6 all lead to either node 5 or 6.

Example 2:

Input: `graph = [[1,2,3,4],[1,2],[3,4],[0,4],[]]`

Output: `[4]`

Explanation:

Only node 4 is a terminal node, and every path starting at node 4 leads to node 4.

A node is a terminal node if there are no outgoing edges. A node is a safe node if every possible path starting from that node leads to a terminal node (or another safe node).

```
class Solution {
    public List<Integer> eventualSafeNodes(int[][] graph) {
        int n = graph.length;
        List<List<Integer>> reversedGraph = new ArrayList<>();
        int[] inDegree = new int[n];

        for (int i = 0; i < n; i++) {
            reversedGraph.add(new ArrayList<>());
        }

        // Reverse the graph
        for (int from = 0; from < n; from++) {
            for (int to : graph[from]) {
                reversedGraph.get(to).add(from);
                inDegree[from]++;
            }
        }

        // Kahn's algorithm
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < n; i++) {
            if (inDegree[i] == 0) {
                queue.offer(i);
            }
        }

        boolean[] safe = new boolean[n];
        while (!queue.isEmpty()) {
            int node = queue.poll();
            safe[node] = true;
            for (int neighbor : reversedGraph.get(node)) {
                inDegree[neighbor]--;
                if (inDegree[neighbor] == 0) {
                    queue.offer(neighbor);
                }
            }
        }

        List<Integer> result = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            if (safe[i]) result.add(i);
        }

        return result;
    }
}
```

Reverse the graph so that for terminal nodes, outdegree becomes indegree and apply kahns BFS

When the indegree is degree mark it as the safe node and return the result

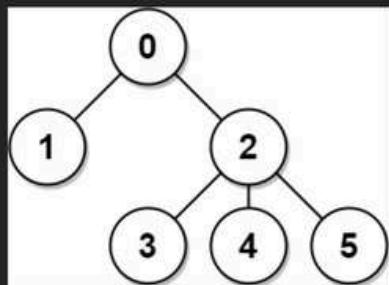
4. 834. Sum of Distances in Tree – Postorder + Preorder)

There is an undirected connected tree with n nodes labeled from 0 to $n - 1$ and $n - 1$ edges.

You are given the integer n and the array `edges` where `edges[i] = [ai, bi]` indicates that there is an edge between nodes a_i and b_i in the tree.

Return an array `answer` of length n where `answer[i]` is the sum of the distances between the i^{th} node in the tree and all other nodes.

Example 1:



Input: $n = 6$, `edges = [[0,1],[0,2],[2,3],[2,4],[2,5]]`

Output: $[8,12,6,10,10,10]$

Explanation: The tree is shown above.

We can see that $\text{dist}(0,1) + \text{dist}(0,2) + \text{dist}(0,3) + \text{dist}(0,4) + \text{dist}(0,5)$ equals $1 + 1 + 2 + 2 + 2 = 8$.

Hence, `answer[0] = 8`, and so on.

Example 2:



Input: $n = 1$, `edges = []`

Output: $[0]$

2. Preorder Traversal (Top-Down):

- Using the previously computed values for node 0 (root), propagate and reroot the tree to calculate the correct `res` for each child:

```
java
res[child] = res[node] - count[child] + (n - count[child]);
```

This formula comes from:

- Removing the subtree of `child`: $- \text{count}[\text{child}]$
- Adding the rest of the tree (excluding that subtree): $+ (n - \text{count}[\text{child}])$

✓ One-line Steps with Intuition:

- Build the tree as an undirected graph using adjacency sets.
- Postorder DFS (bottom-up) from node `0` to:
 - Count how many nodes are in each subtree.
 - Calculate the sum of distances from root `0` to all its subtree nodes.
- Preorder DFS (top-down) to:
 - Reroot the tree at each child and adjust the distance sum accordingly:
 - `res[child] = res[parent] - count[child] + (n - count[child])`
 - Meaning: distance decreases for nodes in child's subtree, and increases for all others.
- Return the final `res` array, which holds the distance sum for each node.

💡 Key Ideas:

- `count[node]` : Number of nodes in the subtree rooted at `node` (including itself).
- `res[node]` : Total distance from `node` to all nodes in its subtree.
- Postorder calculates initial values from leaves to root.
- Preorder reroots and spreads the correct values from root to children.

```

class Solution {
    private List<Set<Integer>> tree;
    private int[] res;
    private int[] count;

    public int[] sumOfDistancesInTree(int n, int[][] edges) {
        tree = new ArrayList<>();
        res = new int[n];
        count = new int[n];
        // Step 1: Build the graph
        for (int i = 0; i < n; i++) {
            tree.add(new HashSet<>());
        }
        for (int[] edge : edges) {
            tree.get(edge[0]).add(edge[1]);
            tree.get(edge[1]).add(edge[0]);
        }
        // Step 2: Postorder traversal to fill count[] and res[] for root = 0
        postorder(0, -1);
        System.out.println(Arrays.toString(count));
        System.out.println(Arrays.toString(res));
        // count [6, 1, 4, 1, 1, 1]
        // res [8, 0, 3, 0, 0, 0]
        // Step 3: Preorder traversal to propagate res[] to all nodes
        preorder(0, -1);
        return res;
    }

    // Postorder traversal: bottom-up
    // Postorder calculates initial values from leaves to root.
    private void postorder(int node, int parent) {
        for (int child : tree.get(node)) {
            if (child == parent) continue;
            postorder(child, node);
            count[node] += count[child];
            res[node] += res[child] + count[child];
        }
        count[node] += 1; // include self
    }

    // Preorder traversal: top-down rerooting
    // Preorder reroots and spreads the correct values from root to children.
    private void preorder(int node, int parent) {
        for (int child : tree.get(node)) {
            if (child == parent) continue;
            res[child] = res[node] - count[child] + (count.length - count[child]);
            preorder(child, node);
        }
    }

    public static void main(String[] args) {
        System.out.println(new Solution().sumOfDistancesInTree(n: 6, new int[][]{
            {0, 1}, {0, 2}, {2, 3}, {2, 4}, {2, 5}
        })); // [8, 12, 6, 10, 10, 10]
    }
}

```

• What does each term represent?

Let's define:

- `res[node]`: the sum of distances from `node` to all nodes in its subtree.
- `res[child]`: the sum of distances from `child` to all nodes in the subtree rooted at `child`.
- `count[child]`: the number of nodes in the subtree rooted at `child`.

- Build initial `res[node]`: total distance from node to all its subtree nodes.
- Build `count[node]`: total number of nodes in the subtree rooted at node, including itself.

```

postorder( node: 0, parent: -1);
System.out.println(Arrays.toString(count));
System.out.println(Arrays.toString(res));
// 
// count [6, 1, 4, 1, 1, 1]
// res [8, 0, 3, 0, 0, 0]
// Step 3: Preorder traversal to propagate results
preorder( node: 0, parent: -1);
System.out.println("#####");
System.out.println(Arrays.toString(count));
System.out.println(Arrays.toString(res));
// 
// [6, 1, 4, 1, 1, 1]
// [8, 12, 6, 10, 10, 10]

```

For count, compute the count of each node from bottom to that root (inclusive) ($1 + 2 + 3 = 6$)
and res computes weights from leaves to that node ($1 + 1 + 2 + 2 + 2 = 8$)

PreOrder computes distance between 2 nodes for 1
 $1 + 3 + 3 + 3 + 2 = 12$

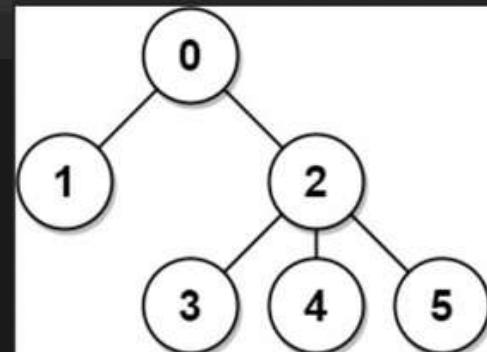
✓ Explanation in terms of distances:

We're computing the sum of distances from node 5 to all other nodes:

From 5:

- to 0 → 5 → 2 → 0 = 2
- to 1 → 5 → 2 → 0 → 1 = 3
- to 2 → 5 → 2 = 1
- to 3 → 5 → 2 → 3 = 2
- to 4 → 5 → 2 → 4 = 2

Total = $2 + 3 + 1 + 2 + 2 = 10$



5. 1129. Shortest Path with Alternating Colors – BFS with state (color) tracking

1129. Shortest Path with Alternating Colors

Medium

You are given an integer n , the number of nodes in a directed graph where the nodes are labeled from 0 to $n - 1$. Each edge is red or blue in this graph, and there could be self-edges and parallel edges.

You are given two arrays `redEdges` and `blueEdges` where:

- `redEdges[i] = [ai, bi]` indicates that there is a directed red edge from node a_i to node b_i in the graph, and
- `blueEdges[j] = [uj, vj]` indicates that there is a directed blue edge from node u_j to node v_j in the graph.

Return an array `answer` of length n , where each `answer[x]` is the length of the shortest path from node 0 to node x such that the edge colors alternate along the path, or -1 if such a path does not exist.

Example 1:

Input: $n = 3$, `redEdges` = $\{[0,1], [1,2]\}$, `blueEdges` = $\{\}$
Output: $[0, 1, -1]$

Example 2:

Input: $n = 3$, `redEdges` = $\{[0,1]\}$, `blueEdges` = $\{[2,1]\}$
Output: $[0, 1, -1]$

Constraints:

- $1 \leq n \leq 100$
- $0 \leq \text{redEdges.length}, \text{blueEdges.length} \leq 400$
- $\text{redEdges}[i].length == \text{blueEdges}[j].length == 2$
- $0 \leq a_i, b_i, u_j, v_j < n$

```

class Solution {
    public int[] shortestAlternatingPaths(int n, int[][] redEdges, int[][] blueEdges) {
        // Adjacency lists for red and blue edges
        List<Integer>[] redGraph = new ArrayList[n];
        List<Integer>[] blueGraph = new ArrayList[n];
        for (int i = 0; i < n; i++) {
            redGraph[i] = new ArrayList<>();
            blueGraph[i] = new ArrayList<>();
        }

        for (int[] edge : redEdges) {
            redGraph[edge[0]].add(edge[1]);
        }
        for (int[] edge : blueEdges) {
            blueGraph[edge[0]].add(edge[1]);
        }

        int[] res = new int[n];
        Arrays.fill(res, val: -1);

        // Queue: node, distance, lastColorUsed (-1 for none, 0 for red, 1 for blue)
        Queue<int[]> queue = new LinkedList<>();
        boolean[][] visited = new boolean[n][2]; // visited[node][color]

        queue.offer(new int[]{0, 0, -1});
        visited[0][0] = true;
        visited[0][1] = true;
        res[0] = 0;

        while (!queue.isEmpty()) {
            int[] current = queue.poll();
            int node = current[0], distance = current[1], lastColor = current[2];

            if (lastColor != 0) { // last was not red → try red edges
                for (int neighbor : redGraph[node]) {
                    if (!visited[neighbor][0]) {
                        visited[neighbor][0] = true;
                        queue.offer(new int[]{neighbor, distance + 1, 0});
                        if (res[neighbor] == -1) res[neighbor] = distance + 1;
                    }
                }
            }

            if (lastColor != 1) { // last was not blue → try blue edges
                for (int neighbor : blueGraph[node]) {
                    if (!visited[neighbor][1]) {
                        visited[neighbor][1] = true;
                        queue.offer(new int[]{neighbor, distance + 1, 1});
                        if (res[neighbor] == -1) res[neighbor] = distance + 1;
                    }
                }
            }
        }

        return res;
    }
}

```

```
public static void main(String[] args) {
    int n = 3;
    int[][] redEdges = {{0, 1}, {1, 2}};
    int[][] blueEdges = {};// no blue edges

    Solution solution = new Solution();
    int[] result = solution.shortestAlternatingPaths(n, redEdges, blueEdges);

    System.out.println("Output: " + Arrays.toString(result));
}
}
```

```
private static List<String> getNodes(Node node) {
    Map<Node, Boolean> visited = new HashMap<>();
    visited.put(node, true);
    List<String> result = new ArrayList<>();
    dfs(node, visited, result);
    return result;
}

private static void dfs(Node node, Map<Node, Boolean> visited, List<String> result) {
    visited.put(node, true);
    List<Node> neighbors = node.neighbors;
    result.add(node.val);
    for (Node neighbor : neighbors) {
        if (!visited.containsKey(neighbor)) {
            visited.put(neighbor, true);
            dfs(neighbor, visited, result);
        }
    }
}

public static void main(String[] args) {
    Node node1 = new Node(val: "MUM");
    Node node2 = new Node(val: "DEL");
    Node node3 = new Node(val: "HYD");
    Node node4 = new Node(val: "KOL");
    Node node5 = new Node(val: "BAN");
    node1.neighbors = Arrays.asList(node2, node5);
    node2.neighbors = Arrays.asList(node3, node1);
    node3.neighbors = Arrays.asList(node2, node4);
    node4.neighbors = Arrays.asList(node3, node5);
    node5.neighbors = Arrays.asList(node4, node1);
    System.out.println(getNodes(node1)); // [MUM, DEL, HYD, KOL, BAN]
}
}
```

6. 1377. Frog Position After T Seconds – DFS + probability distribution

1377. Frog Position After T Seconds

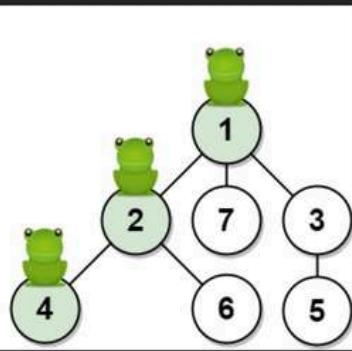
Hard

Given an undirected tree consisting of n vertices numbered from 1 to n . A frog starts jumping from vertex 1. In one second, the frog jumps from its current vertex to another unvisited vertex if they are directly connected. The frog can not jump back to a visited vertex. In case the frog can jump to several vertices, it jumps randomly to one of them with the same probability. Otherwise, when the frog can not jump to any unvisited vertex, it jumps forever on the same vertex.

The edges of the undirected tree are given in the array `edges`, where `edges[i] = [ai, bi]` means that exists an edge connecting the vertices a_i and b_i .

Return the probability that after t seconds the frog is on the vertex `target`. Answers within 10^{-5} of the actual answer will be accepted.

Example 1:



Input: $n = 7$, `edges` = $[[1,2], [1,3], [1,7], [2,4], [2,6], [3,5]]$, $t = 2$, `target` = 4

Output: 0.1666666666666666

Explanation: The figure above shows the given graph. The frog starts at vertex 1, jumping with 1/3 probability to the vertex 2 after second 1 and then jumping with 1/2 probability to vertex 4 after second 2. Thus the probability for the frog is on the vertex 4 after 2 seconds is $1/3 * 1/2 = 1/6 = 0.1666666666666666$.

```
class Solution {
    public double frogPosition(int totalNodes, int[][] edges, int maxSeconds, int targetNode) {
        List<List<Integer>> adjacencyList = new ArrayList<>();

        // Build adjacency list for the graph
        for (int i = 0; i <= totalNodes; i++) {
            adjacencyList.add(new ArrayList<>());
        }
        for (int[] edge : edges) {
            adjacencyList.get(edge[0]).add(edge[1]);
            adjacencyList.get(edge[1]).add(edge[0]);
        }

        boolean[] visited = new boolean[totalNodes + 1];
        Queue<int[]> nodeQueue = new LinkedList<>();
        Queue<Double> probabilityQueue = new LinkedList<>();

        nodeQueue.offer(new int[]{1, 0}); // Start from node 1 at time 0
        probabilityQueue.offer(1.0); // Initial probability is 1
        visited[1] = true;
```

```

while (!nodeQueue.isEmpty()) {
    int[] current = nodeQueue.poll();
    int currentNode = current[0];
    int currentTime = current[1];
    double currentProbability = probabilityQueue.poll();
    // Count number of unvisited neighbors
    int unvisitedChildren = 0;
    for (int neighbor : adjacencyList.get(currentNode)) {
        if (!visited[neighbor]) {
            unvisitedChildren++;
        }
    }
    // If reached the target node
    if (currentNode == targetNode) {
        if (currentTime == maxSeconds || unvisitedChildren == 0) {
            return currentProbability;
        } else {
            return 0.0;
        }
    }
    // Continue to next level if time allows
    if (currentTime < maxSeconds) {
        for (int neighbor : adjacencyList.get(currentNode)) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                nodeQueue.offer(new int[]{neighbor, currentTime + 1});
                probabilityQueue.offer(e: currentProbability / unvisitedChildren);
            }
        }
    }
}
return 0.0;
}

public static void main(String[] args) {
    System.out.println(new Solution().frogPosition( totalNodes: 7, new int[][]{
        {1, 2}, {1, 3}, {1, 7}, {2, 4}, {2, 6}, {3, 5}
    }, maxSeconds: 2, targetNode: 4)); // 0.1666666666666666
}
}

```

- Mark root as visited
- And probability as 1 for the root
- Take the level order traversal of a tree
- Count number of unvisited neighbors
- If target node is reached and if currentTime is maxTime and if unvisitedNode count is 0, return currentProbability
- Push neighbor and currentTime +1 to the nodeQueue
- If currentTime is lesser than maxTime and for unvisited neighbors, if visited is false, mark it as true and push it to nodeQueue and probabilityQueue using currentProb / number of leaf nodes

All Paths From Source To Destination

```
private static List<List<Node>> allPathsDfs(Node source, Node destination) {
    List<List<Node>> allPaths = new ArrayList<>();
    Set<Node> visited = new HashSet<>();
    List<Node> currentPath = new ArrayList<>();

    dfs(source, destination, visited, currentPath, allPaths);
    return allPaths;
}

private static void dfs(Node current, Node destination, Set<Node> visited,
                      List<Node> currentPath, List<List<Node>> allPaths) {
    visited.add(current);
    currentPath.add(current);

    if (current == destination) {
        allPaths.add(new ArrayList<>(currentPath));
    } else {
        for (Node neighbor : current.neighbors) {
            if (!visited.contains(neighbor)) {
                dfs(neighbor, destination, visited, currentPath, allPaths);
            }
        }
    }

    currentPath.remove(index: currentPath.size() - 1);
    visited.remove(current);
}

public static void main(String[] args) {
    Node node0 = new Node( val: 0 );
    Node node1 = new Node( val: 1 );
    Node node2 = new Node( val: 2 );
    Node node3 = new Node( val: 3 );
    node0.neighbors = Arrays.asList(node1, node3);
    node1.neighbors = Arrays.asList(node0, node2);
    node2.neighbors = Arrays.asList(node1, node3);
    node3.neighbors = Arrays.asList(node2, node0);
    List<List<Node>> paths = allPathsDfs(node1, node3);
    for (List<Node> path : paths) {
        System.out.println(path);
    }
}
// [1, 0, 3]
// [1, 2, 3]
}

function dfs(root, result, curr = [], pathLen = 0) {
    if (root) {
        curr[pathLen] = root.data;
        pathLen++;
        if (!root.left && !root.right) {
            result.push(curr.slice());
            return;
        }
        dfs(root.left, result, curr, pathLen);
        dfs(root.right, result, curr, pathLen);
    }
}

public static List<List<Integer>> allRootToLeafPaths(TreeNode root) {
    List<List<Integer>> result = new ArrayList<>();
    dfs(root, new ArrayList<>(), result);
    return result;
}

private static void dfs(TreeNode node, List<Integer> currentPath, List<List<Integer>>
    if (node == null) return;

    currentPath.add(node.data); // Add current node

    if (node.left == null && node.right == null) {
        result.add(new ArrayList<>(currentPath)); // Copy the path
    } else {
        dfs(node.left, currentPath, result);
        dfs(node.right, currentPath, result);
    }

    currentPath.remove(currentPath.size() - 1); // Backtrack
}
```

If we use `List<Integer>` instead of an array like `int[]`, you'll need to manually backtrack

Graph Coloring

Graph coloring is assigning colors to each vertex of a graph such that no two adjacent vertices share the same color.

For 2-coloring (bipartite check), we only use 2 colors, say: 0 and 1.

```
public class GraphColoringGreedy {

    public static boolean colorGraph(int[][] graph, int maxColors) {
        int n = graph.length;
        int[] color = new int[n]; // 0 means uncolored
        Arrays.fill(color, val: 0);

        for (int start = 0; start < n; start++) {
            if (color[start] != 0) continue;

            Queue<Integer> queue = new LinkedList<>();
            queue.offer(start);

            while (!queue.isEmpty()) {
                int node = queue.poll();

                // Track used colors for neighbors
                boolean[] usedColors = new boolean[maxColors + 1]; // 1 to 5

                for (int neighbor : graph[node]) {
                    if (color[neighbor] != 0) {
                        usedColors[color[neighbor]] = true;
                    } else {
                        queue.offer(neighbor);
                    }
                }

                // Assign the first available color
                for (int c = 1; c <= maxColors; c++) {
                    if (!usedColors[c]) {
                        color[node] = c;
                        break;
                    }
                }
            }
        }
    }
}
```

Offer Uncolored Nodes To The Graph

```

        if (color[node] == 0) {
            return false; // No color could be assigned
        }
    }

    System.out.println("Color assignment:");
    for (int i = 0; i < n; i++) {
        System.out.println("Node " + i + ": Color " + color[i]);
    }

    return true;
}

public static void main(String[] args) {
    int[][] graph = {
        {1, 2}, // Node 0
        {0, 3}, // Node 1
        {0, 3, 4}, // Node 2
        {1, 2}, // Node 3
        {2} // Node 4
    };

    int maxColors = 5;

    boolean result = colorGraph(graph, maxColors);
    System.out.println("Graph is 5-colorable: " + result);
}

```

// Color assignment:
// Node 0: Color 1
// Node 1: Color 2
// Node 2: Color 2
// Node 3: Color 1
// Node 4: Color 1
// Graph is 5-colorable: true

In the `colorGraph` function, we have this loop:

```
java
Copy Edit
for (int start = 0; start < n; start++) {
    if (color[start] != 0) continue;
    ...
}
```

💡 Why do we loop over all nodes (`start = 0` to `n-1`)?

Because the graph could be disconnected — meaning it might have more than one component.

Real Example

Take this graph:

```
yaml
Copy Edit
Component 1: 0 - 1 - 2
Component 2: 3 - 4
```

Topological Sorting

Topological Sorting

A linear ordering of vertices such that for every directed edge $u \rightarrow v$, node u appears before v in the ordering. Only possible if the directed acyclic graph has no directed cycles

```
public class KahnBFS {

    List<Integer> process(List<Node> nodes) {
        int n = nodes.size();
        int[] in_degree = new int[n];
        Queue<Integer> queue = new LinkedList<>();

        for (int u = 0; u < n; u++) {
            for (Node x : nodes.get(u).neighbors)
                in_degree[x.val]++;
        }

        for (int i = 0; i < n; i++)
            if (in_degree[i] == 0)
                queue.add(i);

        List<Integer> result = new ArrayList<>();

        while (!queue.isEmpty()) {
            int val = queue.poll();
            result.add(val);
            for (Node x : nodes.get(val).neighbors) {
                if (--in_degree[x.val] == 0)
                    queue.add(x.val);
            }
        }

        return result;
    }

    public static void main(String[] args) {
        Node node0 = new Node(_val: 0);
        Node node1 = new Node(_val: 1);
        Node node2 = new Node(_val: 2);
        Node node3 = new Node(_val: 3);
        Node node4 = new Node(_val: 4);
        Node node5 = new Node(_val: 5);
        node2.neighbors.addAll(Arrays.asList(node3));
        node3.neighbors.addAll(Arrays.asList(node1));
        node4.neighbors.addAll(Arrays.asList(node0, node1));
        node5.neighbors.addAll(Arrays.asList(node2, node0));
        List<Node> nodes = new ArrayList<>();
        nodes.addAll(Arrays.asList(node0, node1, node2, node3, node4, node5));
        List<Integer> result = new KahnBFS().process(nodes);
        System.out.println(result);
        // [4, 5, 2, 0, 3, 1]
    }
}
```

```

public class TopologicalSortingDFS {

    void dfs(Node node, boolean[] visited, Stack<Integer> stack) {
        visited[node.val] = true;
        for (Node x : node.neighbors) {
            if (!visited[x.val]) {
                dfs(x, visited, stack);
            }
        }
        stack.push(node.val);
    }

    List<Integer> process(List<Node> nodes) {
        Stack<Integer> stack = new Stack<>();
        int n = nodes.size();
        boolean[] visited = new boolean[n];
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {
                dfs(nodes.get(i), visited, stack);
            }
        }
        Collections.reverse(stack);
        return stack;
    }

    public static void main(String[] args) {
        Node node0 = new Node(_val: 0);
        Node node1 = new Node(_val: 1);
        Node node2 = new Node(_val: 2);
        Node node3 = new Node(_val: 3);
        Node node4 = new Node(_val: 4);
        Node node5 = new Node(_val: 5);
        node2.neighbors.addAll(Arrays.asList(node3));
        node3.neighbors.addAll(Arrays.asList(node1));
        node4.neighbors.addAll(Arrays.asList(node0, node1));
        node5.neighbors.addAll(Arrays.asList(node2, node0));
        List<Node> nodes = new ArrayList<>();
        nodes.addAll(Arrays.asList(node0, node1, node2, node3, node4, node5));
        List<Integer> result = new TopologicalSortingDFS().process(nodes);
        System.out.println(result);
        // [4, 5, 2, 0, 3, 1]
    }
}

public boolean hasCycle(int n, List<List<Integer>> graph) {
    int[] inDegree = new int[n];
    for (List<Integer> neighbors : graph) {
        for (int v : neighbors) {
            inDegree[v]++;
        }
    }

    Queue<Integer> q = new LinkedList<>();
    for (int s = 0; s < n; s++) {
        if (inDegree[s] == 0) q.offer(s);
    }

    int count = 0;
    while (!q.isEmpty()) {
        int node = q.poll();
        count++;
        for (int neighbor : graph.get(node)) {
            if (--inDegree[neighbor] == 0) q.offer(neighbor);
        }
    }

    return count != n; // If true, cycle exists
}

```

1.207. CourseSchedule

207. Course Schedule

Medium

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [ai, bi]` indicates that you must take course `bi` first if you want to take course `ai`.

- For example, the pair `[0, 1]`, indicates that to take course `0` you have to first take course `1`.

Return `true` if you can finish all courses. Otherwise, return `false`.

Example 1:

Input: `numCourses = 2, prerequisites = [[1,0]]`

Output: `true`

Explanation: There are a total of 2 courses to take.

To take course 1 you should have finished course 0. So it is possible.

Example 2:

Input: `numCourses = 2, prerequisites = [[1,0],[0,1]]`

Output: `false`

Explanation: There are a total of 2 courses to take.

To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

```
public boolean canFinish(int numCourses, int[][] prerequisites) {  
    List<Node> nodes = new ArrayList<>();  
    int visitedCount = 0;  
    int[] in_degree = new int[numCourses];  
    Queue<Integer> queue = new ArrayDeque<>();  
    for (int i = 0; i < numCourses; i++) {  
        nodes.add(new Node(i));  
    }  
    for (int i = 0; i < numCourses; i++) {  
        int course = prerequisites[i][0];  
        int prerequisite = prerequisites[i][1];  
        nodes.get(prerequisite).neighbors.add(nodes.get(course));  
        in_degree[course]++;  
    }  
    for (int i = 0; i < numCourses; i++) {  
        if(in_degree[i] == 0){  
            queue.add(nodes.get(i).val);  
        }  
    }  
    while (!queue.isEmpty()){  
        int val = queue.poll();  
        visitedCount++;  
        for(Node node: nodes.get(val).neighbors){  
            if(--in_degree[node.val]==0){  
                queue.add(node.val);  
            }  
        }  
    }  
    return numCourses == visitedCount;  
}  
  
public static void main(String[] args) {  
    System.out.println(new CourseSchedule_1_207().canFinish( numCourses: 2, new int[][]{{1, 0}}));  
}
```

2. 210. Course Schedule II

210. Course Schedule II

Medium

There are a total of `numCourses` courses you have to take, labeled from `0` to `numCourses - 1`. You are given an array `prerequisites` where `prerequisites[i] = [ai, bi]` indicates that you must take course `bi` first if you want to take course `ai`.

- For example, the pair `[0, 1]`, indicates that to take course `0` you have to first take course `1`.

Return the ordering of courses you should take to finish all courses. If there are many valid answers, return any of them. If it is impossible to finish all courses, return an empty array.

Example 1:

```
Input: numCourses = 2, prerequisites = [[1,0]]
Output: [0,1]
Explanation: There are a total of 2 courses to take. To take course 1 you
should have finished course 0. So the correct course order is [0,1].
```

Example 2:

```
Input: numCourses = 4, prerequisites = [[1,0],[2,0],[3,1],[3,2]]
Output: [0,2,1,3]
Explanation: There are a total of 4 courses to take. To take course 3 you
should have finished both courses 1 and 2. Both courses 1 and 2 should be
taken after you finished course 0.
So one correct course order is [0,1,2,3]. Another correct ordering is
[0,2,1,3].
```

Example 3:

```
Input: numCourses = 1, prerequisites = []
Output: [0]
```

```
public class CourseSchedule_2_210 {
    public int[] findOrder(int numCourses, int[][] prerequisites) {
        List<Node> nodes = new ArrayList<>();
        int visitedCount = 0;
        int[] in_degree = new int[numCourses];
        Queue<Integer> queue = new ArrayDeque<>();
        List<Integer> result = new ArrayList<>();
        for (int i = 0; i < numCourses; i++) {
            nodes.add(new Node(i));
        }
        for (int i = 0; i < numCourses; i++) {
            int course = prerequisites[i][0];
            int prerequisite = prerequisites[i][1];
            nodes.get(prerequisite).neighbors.add(new Node(course));
            in_degree[course]++;
        }
        for (int i = 0; i < numCourses; i++) {
            if(in_degree[i] == 0){
                queue.add(nodes.get(i).val);
            }
        }
        while (!queue.isEmpty()){
            int val = queue.poll();
            visitedCount++;
            result.add(val);
            for(Node node: nodes.get(val).neighbors){
                if(--in_degree[node.val]==0){
                    queue.add(node.val);
                }
            }
        }
        if (visitedCount == numCourses) {
            return result.stream().mapToInt(Integer::intValue).toArray();
        }
        return new int[0];
    }
}
```

Trie

- **Insertion:**
 - **Time Complexity:** $O(m)$, where (m) is the length of the word being inserted.
 - **Space Complexity:** $O(m)$, in the worst case.
- **Search:**
 - **Time Complexity:** $O(m)$, where (m) is the length of the word being searched.
 - **Space Complexity:** $O(1)$.
- **Prefix Search (StartsWith):**
 - **Time Complexity:** $O(m)$, where (m) is the length of the prefix being searched.
 - **Space Complexity:** $O(1)$.

1.208. Implement Trie (Prefix Tree)

```
public class Trie {  
    private final TrieNode root;  
  
    static class TrieNode {  
        TrieNode[] children;  
        boolean isEndOfWord;  
  
        TrieNode() {  
            children = new TrieNode[26]; // For lowercase 'a' to 'z'  
            isEndOfWord = false;  
        }  
    }  
  
    public Trie() {  
        root = new TrieNode();  
    }  
  
    public void insert(String word) {  
        TrieNode curr = root;  
        for (char c : word.toCharArray()) {  
            int index = c - 'a';  
            if (curr.children[index] == null) {  
                curr.children[index] = new TrieNode();  
            }  
            curr = curr.children[index];  
        }  
        curr.isEndOfWord = true;  
    }  
  
    public boolean search(String word) {  
        TrieNode node = searchPrefix(word);  
        return node != null && node.isEndOfWord;  
    }  
}
```

```

public boolean startsWith(String prefix) {
    return searchPrefix(prefix) != null;
}

private TrieNode searchPrefix(String prefix) {
    TrieNode curr = root;
    for (char c : prefix.toCharArray()) {
        int index = c - 'a';
        if (curr.children[index] == null) {
            return null;
        }
        curr = curr.children[index];
    }
    return curr;
}

public static void main(String[] args) {
    Trie trie = new Trie();

    // Sample words to insert
    List<String> words = Arrays.asList("apple", "app", "ape", "bat", "ball");

    // Insert words into Trie
    for (String word : words) {
        trie.insert(word);
        System.out.println("Inserted: " + word);
    }

    // Test search
    System.out.println("\n--- Search Tests ---");
    System.out.println("Search 'app': " + trie.search(word: "app")); // true
    System.out.println("Search 'apples': " + trie.search(word: "apples")); // false

    // Test startsWith
    System.out.println("\n--- Prefix Tests ---");
    System.out.println("StartsWith 'cat': " + trie.startsWith("cat")); // false
}

```

A Trie (pronounced *try*), also known as a **Prefix Tree**, is a tree-like data structure that is used for efficient retrieval of a key in a large dataset of strings. It's commonly used in autocomplete, spell checkers, and IP routing.

Key Characteristics:

- Each node represents a character of a word.
- Paths from root to leaf form words.
- Efficient for prefix matching and word searching.

Basic Operations:

Operation	Time Complexity
Insert(word)	O(L)
Search(word)	O(L)
StartsWith(prefix)	O(L)

Where L is the length of the word/prefix.

Initialize the Trie with a root node where each node contains an array of 26 child TrieNodes (for each lowercase alphabet character) and a boolean isEndOfWord.

For search, iterate through the characters of the input word. For each character, compute its index and check if the corresponding child node exists. If it doesn't, return false. Otherwise, move to the next child node. Continue this process until the end of the word. If the final node marks the end of a word (isEndOfWord == true), return true.

For insert, traverse through the characters of the word. If the child node at the given index doesn't exist, create a new TrieNode. After processing all characters, mark the final node's isEndOfWord as true

Here's a visual representation of the **Trie (Prefix Tree)** after inserting the words: "apple", "app",

"ape", "bat", and "ball" :



2. 211. Design Add and Search Words Data Structure

211. Design Add and Search Words Data Structure

Solved

[Medium](#) [Topics](#) [Companies](#) [Hint](#)

Design a data structure that supports adding new words and finding if a string matches any previously added string.

Implement the `WordDictionary` class:

- `WordDictionary()` Initializes the object.
- `void addWord(word)` Adds `word` to the data structure, it can be matched later.
- `bool search(word)` Returns `true` if there is any string in the data structure that matches `word` or `false` otherwise. `word` may contain dots `'.'` where dots can be matched with any letter.

Example:

Input
["WordDictionary","addWord","addWord","addWord","search","search","search","search"]
[[],["bad"],["dad"],["mad"],["pad"],["bad"],[".ad"],["b.."]]
Output
[null,null,null,null,false,true,true,true]

Explanation
WordDictionary wordDictionary = new WordDictionary();
wordDictionary.addWord("bad");
wordDictionary.addWord("dad");
wordDictionary.addWord("mad");
wordDictionary.search("pad"); // return False
wordDictionary.search("bad"); // return True
wordDictionary.search(".ad"); // return True
wordDictionary.search("b.."); // return True

```

public class WordDictionary {

    private static class TrieNode {
        TrieNode[] children = new TrieNode[26];
        boolean isEndOfWord = false;
    }

    private final TrieNode root;

    public WordDictionary() {
        root = new TrieNode();
    }

    public void addWord(String word) {
        TrieNode currentNode = root;
        for (char character : word.toCharArray()) {
            int index = character - 'a';
            if (currentNode.children[index] == null) {
                currentNode.children[index] = new TrieNode();
            }
            currentNode = currentNode.children[index];
        }
        currentNode.isEndOfWord = true;
    }

    public boolean search(String word) {
        return searchFromNode(word, wordIndex: 0, root);
    }

    private boolean searchFromNode(String word, int wordIndex, TrieNode currentNode) {
        if (currentNode == null) return false;

        if (wordIndex == word.length()) {
            return currentNode.isEndOfWord;
        }

        char currentChar = word.charAt(wordIndex);

        if (currentChar == '.') {
            // Try all possible children for wildcard
            for (TrieNode childNode : currentNode.children) {
                if (childNode != null & searchFromNode(word, wordIndex: wordIndex + 1, childNode)) {
                    return true;
                }
            }
            return false;
        } else {
            int index = currentChar - 'a';
            return searchFromNode(word, wordIndex: wordIndex + 1, currentNode.children[index]);
        }
    }
}

```

Initialize the Trie with a root node where each node contains an array of 26 child TrieNodes (for each lowercase alphabet character) and a boolean isEndOfWord.

For insert, traverse through the characters of the word. If the child node at the given index doesn't exist, create a new TrieNode. After processing all characters, mark the final node's isEndOfWord as true

For search, if its currentLength is wordLength return true, else increment currentLength with 1, it the word is “.” traverse through the neighbors and childNode is not null, if the search of next word is true return true else return search of index + 1 and next char in the children

```

// Driver code
public static void main(String[] args) {
    WordDictionary dictionary = new WordDictionary();

    dictionary.addWord("bad");
    dictionary.addWord("dad");
    dictionary.addWord("mad");

    System.out.println(dictionary.search("pad")); // false
    System.out.println(dictionary.search("bad")); // true
    System.out.println(dictionary.search(".ad")); // true
    System.out.println(dictionary.search("b..")); // true
    System.out.println(dictionary.search("b.d")); // true
    System.out.println(dictionary.search("..d")); // true
    System.out.println(dictionary.search("...")); // true
    System.out.println(dictionary.search("....")); // false
}
}

/***
 * Your WordDictionary object will be instantiated and called as such:
 * WordDictionary obj = new WordDictionary();
 * obj.addWord(word);
 * boolean param_2 = obj.search(word);
 */

```

3. 212. Word Search II

Given an $m \times n$ board of characters and a list of strings words, return all words on the board.

Each word must be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once in a word.

Example 1:

o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v

Input: board = [["o","a","a","n"],["e","t","a","e"],["i","h","k","r"],["i","f","l","v"]],
words = ["oath","pea","eat","rain"]
Output: ["eat","oath"]

```

class Solution {
    private static final int[][] DIRECTIONS = { { 0, 1 }, { 1, 0 }, { 0, -1 }, { -1, 0 } };

    public List<String> findWords(char[][] board, String[] words) {
        List<String> result = new ArrayList<>();
        if (board == null || board.length == 0 || words == null || words.length == 0) {
            return result;
        }

        TrieNode root = buildTrie(words);
        int rows = board.length;
        int cols = board[0].length;

        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                dfs(board, i, j, root, result);
            }
        }

        return result;
    }

    private void dfs(char[][] board, int row, int col, TrieNode node, List<String> result) {
        char c = board[row][col];
        if (c == '#' || node.children[c - 'a'] == null) {
            return;
        }

        node = node.children[c - 'a'];
        if (node.word != null) { // Found a word
            result.add(node.word);
            node.word = null; // To avoid duplicate results
        }

        board[row][col] = '#';
        for (int[] dir : DIRECTIONS) {
            int newRow = row + dir[0];
            int newCol = col + dir[1];
            if (newRow >= 0 && newRow < board.length && newCol >= 0 && newCol < board[0].length) {
                dfs(board, newRow, newCol, node, result);
            }
        }
        board[row][col] = c;
    }

    private TrieNode buildTrie(String[] words) {
        TrieNode root = new TrieNode();
        for (String word : words) {
            TrieNode node = root;
            for (char c : word.toCharArray()) {
                int index = c - 'a';
                if (node.children[index] == null) {
                    node.children[index] = new TrieNode();
                }
                node = node.children[index];
            }
            node.word = word;
        }
        return root;
    }

    static class TrieNode {
        TrieNode[] children = new TrieNode[26];
        String word;
    }
}

```

Rather than using a boolean flag like `isEndOfWord`, check for the complete string `word` during traversal. Explore all directions in the matrix, marking visited cells with `#` and reverting them after backtracking.

4. 1858. Longest Word With All Prefixes

Feature	Red-Black Tree	AVL Tree
✓ Balancing	Less strictly balanced	More strictly balanced
🔴 Balance Factor	Uses color + structural rules	Uses balance factor (-1, 0, 1)
🟡 Rotations (Insert)	Fewer rotations (at most 2)	More rotations (can be $O(\log n)$)
🟡 Rotations (Delete)	Easier to handle, fewer fix-ups	More complex, might require multiple rotations
● Search Performance	Slower due to less strict balance (deeper tree)	Faster due to tighter balancing
🟠 Insertion & Deletion	Faster due to fewer adjustments	Slower, especially for deletion
🟢 Use Cases	Good for write-heavy systems (OS, DB indexes)	Good for read-heavy systems (memory lookup)
💡 Height Guarantee	$O(2 * \log n) \rightarrow$ looser bound	$O(\log n) \rightarrow$ tighter bound
🔄 Rebalancing Strategy	Recoloring + single/double rotations	Purely single/double rotations

Red Black Tree

A Red-Black Tree is a type of self-balancing binary search tree (BST). It guarantees $O(\log n)$ time complexity for search, insert, and delete operations by maintaining certain properties through coloring and rotations.

🌈 Red-Black Tree Properties

- Each node is either red or black.
 - Root is always black.
 - Red nodes cannot have red children (no two reds in a row).
 - Every path from a node to its descendant null leaves has the same number of black nodes.
 - New nodes are inserted as red.
-
- Why Use Red-Black Tree?
 -
 - Maintains logarithmic height even in worst-case.
 - More flexible and faster than AVL trees in scenarios with frequent inserts/deletes.
 - Used in Java's TreeMap, C++ STL's map/set, Linux kernel, etc.

🔧 Insertion (High-Level Steps)

1. Insert node using standard BST insert.
2. Color new node **red**.
3. Fix any property violations using:
 - Recoloring
 - Rotations (Left and Right)

⌚ There are 5 major cases during fixing after insertion.

⟳ Rotations

- **Left Rotation:** shifts node down to the left.
- **Right Rotation:** shifts node down to the right.

These maintain BST structure and help fix red-black violations.

👉 Terms Breakdown

📍 **node**

The newly inserted red node that might cause a Red-Black Tree violation (like two consecutive red nodes).

👩‍最关键的 parent

The direct parent of the inserted node.

大大小的 grandparent

The parent of the parent. Basically, the node two levels above the inserted node.

👬 uncle

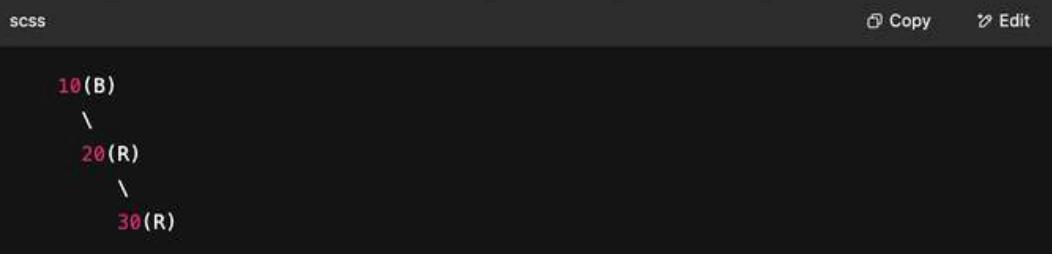
The sibling of the parent, i.e., the other child of the grandparent.

Example

Saved memory full ⓘ

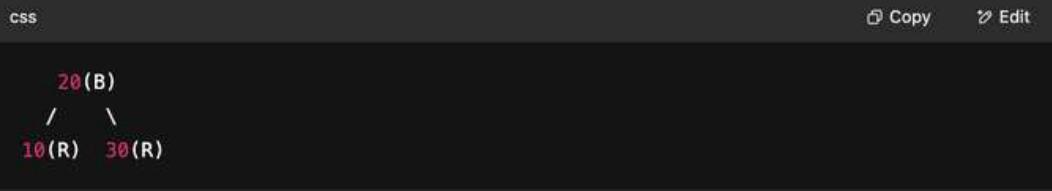
After inserting: 10, 20, 30

Initial BST:



Violates rule 3 (two consecutive reds). Fix with **left rotate at 10**, recolor.

Fixed tree:



● Time Complexities

Operation	Time
Search	$O(\log n)$
Insert	$O(\log n)$
Delete	$O(\log n)$

```
public class RedBlackTree {

    static class Node {
        int val;
        Node left, right, parent;
        boolean isRed;

        Node(int val) {
            this.val = val;
            this.isRed = true; // new nodes are red
        }
    }

    private Node root;

    public void insert(int val) {
        Node node = new Node(val);
        root = bstInsert(root, node);
        fixViolation(node);
    }

    private Node bstInsert(Node root, Node node) {
        if (root == null) return node;

        if (node.val < root.val) {
            root.left = bstInsert(root.left, node);
            root.left.parent = root;
        } else {
            root.right = bstInsert(root.right, node);
            root.right.parent = root;
        }

        return root;
    }
}
```

```

private void fixViolation(Node node) {
    Node parent, grandparent;

    while (node != root && node.isRed && node.parent.isRed) {
        parent = node.parent;
        grandparent = parent.parent;

        if (parent == grandparent.left) {
            Node uncle = grandparent.right;

            if (uncle != null && uncle.isRed) {
                parent.isRed = false;
                uncle.isRed = false;
                grandparent.isRed = true;
                node = grandparent;
            } else {
                if (node == parent.right) {
                    node = parent;
                    leftRotate(node);
                }
                parent.isRed = false;
                grandparent.isRed = true;
                rightRotate(grandparent);
            }
        } else {
            Node uncle = grandparent.left;

            if (uncle != null && uncle.isRed) {
                parent.isRed = false;
                uncle.isRed = false;
                grandparent.isRed = true;
                node = grandparent;
            } else {
                if (node == parent.left) {
                    node = parent;
                    rightRotate(node);
                }
                parent.isRed = false;
                grandparent.isRed = true;
                leftRotate(grandparent);
            }
        }
    }

    root.isRed = false;
}

```

```

private void leftRotate(Node x) {
    Node y = x.right;
    x.right = y.left;
    if (y.left != null) y.left.parent = x;

    y.parent = x.parent;

    if (x.parent == null) root = y;
    else if (x == x.parent.left) x.parent.left = y;
    else x.parent.right = y;

    y.left = x;
    x.parent = y;
}

```

```
private void rightRotate(Node x) {
    Node y = x.left;
    x.left = y.right;
    if (y.right != null) y.right.parent = x;

    y.parent = x.parent;
    if (x.parent == null) root = y;
    else if (x == x.parent.left) x.parent.left = y;
    else x.parent.right = y;

    y.right = x;
    x.parent = y;
}

// ● Search method
public boolean search(int val) {
    return searchNode(root, val) != null;
}

private Node searchNode(Node node, int val) {
    if (node == null || node.val == val) return node;
    if (val < node.val) return searchNode(node.left, val);
    return searchNode(node.right, val);
}

// ✗ Delete method
public void delete(int val) {
    Node node = searchNode(root, val);
    if (node == null) return;

    deleteNode(node);
}
```

```

private void deleteNode(Node node) {
    Node y = node;
    Node x;
    boolean yOriginalColor = y.isRed;

    if (node.left == null) {
        x = node.right;
        transplant(node, node.right);
    } else if (node.right == null) {
        x = node.left;
        transplant(node, node.left);
    } else {
        y = minimum(node.right);
        yOriginalColor = y.isRed;
        x = y.right;

        if (y.parent == node) {
            if (x != null) x.parent = y;
        } else {
            transplant(y, y.right);
            y.right = node.right;
            y.right.parent = y;
        }

        transplant(node, y);
        y.left = node.left;
        y.left.parent = y;
        y.isRed = node.isRed;
    }

    if (!yOriginalColor) fixDelete(x);
}

private void fixDelete(Node x) {
    while (x != root && (x == null || !x.isRed)) {
        if (x == x.parent.left) {
            Node w = x.parent.right;

            if (w != null && w.isRed) {
                w.isRed = false;
                x.parent.isRed = true;
                leftRotate(x.parent);
                w = x.parent.right;
            }

            if ((w.left == null || !w.left.isRed) && (w.right == null || !w.right.isRed)) {
                w.isRed = true;
                x = x.parent;
            } else {
                if (w.right == null || !w.right.isRed) {
                    if (w.left != null) w.left.isRed = false;
                    w.isRed = true;
                    rightRotate(w);
                    w = x.parent.right;
                }

                w.isRed = x.parent.isRed;
                x.parent.isRed = false;
                if (w.right != null) w.right.isRed = false;
                leftRotate(x.parent);
                x = root;
            }
        }
    }
}

```

```

    } else {
        Node w = x.parent.left;

        if (w != null && w.isRed) {
            w.isRed = false;
            x.parent.isRed = true;
            rightRotate(x.parent);
            w = x.parent.left;
        }

        if ((w.right == null || !w.right.isRed) && (w.left == null || !w.left.isRed)) {
            w.isRed = true;
            x = x.parent;
        } else {
            if (w.left == null || !w.left.isRed) {
                if (w.right != null) w.right.isRed = false;
                w.isRed = true;
                leftRotate(w);
                w = x.parent.left;
            }

            w.isRed = x.parent.isRed;
            x.parent.isRed = false;
            if (w.left != null) w.left.isRed = false;
            rightRotate(x.parent);
            x = root;
        }
    }
}

if (x != null) x.isRed = false;
}

private void transplant(Node u, Node v) {
    if (u.parent == null) root = v;
    else if (u == u.parent.left) u.parent.left = v;
    else u.parent.right = v;

    if (v != null) v.parent = u.parent;
}

private Node minimum(Node node) {
    while (node.left != null) node = node.left;
    return node;
}
}

```

Node Structure

```

java                                         ⌂ Copy   ⌂ Edit

class Node {
    int val;
    Node left, right, parent;
    boolean isRed;
}

```

- Every node stores:
 - `val`: data
 - `left`, `right`: child nodes
 - `parent`: pointer to parent node
 - `isRed`: red/black color flag (`true` = red, `false` = black)

Insertion

Method: `insert(int val)`

Purpose:

Add a node while preserving BST properties, then fix Red-Black Tree violations (like double red).

Steps:

1. Create a new red node.
2. Insert it like a regular BST node (`bstInsert`).
3. Rebalance the tree using `fixViolation`.

BST Insertion Helper

Method: `bstInsert(Node root, Node node)`

- Classic BST insert: go left or right.
- Also assigns the `parent` pointer.
- This step doesn't handle color rules.

Fixing Red-Black Violations

Method: `fixViolation(Node node)`

Goal:

Fix cases where:

- A red node has a red parent (violates RBT rules).
- Tree becomes unbalanced.

Handles 3 Cases:

Case 1: Uncle is red

- Recolor parent and uncle to black, grandparent to red.
- Move `node` pointer up to `grandparent`.

Case 2: Node is inner grandchild

- Perform a `rotation` to turn into Case 3.

Case 3: Node is outer grandchild

- Recolor parent black, grandparent red.
- Rotate grandparent.

If parent is a **left child**, apply **right rotation**.

If parent is a **right child**, apply **left rotation**.

Deleting a Node

Method: `deleteNode(Node node)`

Handles 3 Cases:

1. Node has 0 or 1 child: directly remove and connect child.
2. Node has 2 children:
 - Replace with in-order successor (minimum from right subtree).
 - Delete the successor node (which will have ≤ 1 child).
- Use `transplant` to connect parent to child/subtree.
- After deletion, if a **black node** is removed, it may violate RBT balance — so we fix it with `fixDelete`.

Fixing After Deletion

Method: `fixDelete(Node x)`

Goal:

Restore red-black properties if a **black node** was removed.

Uses 4 main cases depending on the **sibling's** color and its children:

- If sibling is **red** → rotate and recolor.
- If sibling is black with **black children** → recolor sibling and move **x** up.
- If sibling is black with **at least one red child** → rotate and recolor to balance tree.

Transplant

Method: `transplant(Node u, Node v)`

Replaces the subtree rooted at **u** with subtree rooted at **v**.

Used during deletion to rearrange links.

Minimum

Method: `minimum(Node node)`

Finds the node with the **smallest value** in a subtree (leftmost node).

Used to find the in-order successor during deletion.

Summary

Operation	Time Complexity	Purpose
<code>insert</code>	$O(\log n)$	Add node while keeping tree balanced
<code>delete</code>	$O(\log n)$	Remove node & restore RBT properties
<code>search</code>	$O(\log n)$	Find if value exists
<code>rotate</code>	$O(1)$	Rebalance locally
<code>fix*</code>	$O(\log n)$	Restore color and structure rules

AVL Tree

An AVL Tree is a self-balancing Binary Search Tree (BST) where the height difference (balance factor) between the left and right subtrees of any node is at most 1.

It was named after its inventors: Adelson-Velsky and Landis.

A regular BST can become unbalanced (skewed), degenerating into a linked list ($O(n)$ time for search, insert, delete).

👉 AVL Tree ensures $O(\log n)$ time for search, insert, delete by maintaining balance.

⚖️ Balance Factor

For each node:

```
sql
```

Balance Factor = Height(left subtree) - Height(right subtree)

It must be in `{ -1, 0, 1 }` for the tree to be AVL.

🔄 Rotations in AVL

When the balance is broken (factor becomes < -1 or > 1), we rebalance using rotations:

1. Left-Left (LL) → Right Rotation

```
ruby
```

```
30
/
20
/
10

=> Rotate right around 30
```

2. Right-Right (RR) → Left Rotation

```
ruby
```

```
10
 \
 20
 \
 30

=> Rotate left around 10
```

3. Left-Right (LR) → Left then Right

```
ruby
```

```
30
/
10
 \
 20

=> Rotate left around 10, then right around 30
```

4. Right-Left (RL) → Right then Left

ruby

Copy Edit

```
10
 \
 30
 /
20
```

=> Rotate right around 30, then left around 10

💡 Insertion Example (with balance checks):

Insert: 10, 20, 30

- Insert 10 → Fine
- Insert 20 → Fine
- Insert 30 → Skewed (Right-Right) → Rotate left around 10

Final Tree:

markdown

Copy Edit

```
20
/ \
10  30
```

⌚ Time Complexity (AVL)

Operation	Time
Search	$O(\log n)$
Insert	$O(\log n)$
Delete	$O(\log n)$

Because tree height is always $\log n$.

✓ Summary

- AVL Trees maintain **balance** automatically.
- Ensures **fast lookup, insert, delete**.
- Uses **rotations** to fix imbalances.
- More rigidly balanced than Red-Black Trees → slightly more overhead but better balance.


```

// Delete key from AVL Tree
public static TreeNode delete(TreeNode root, int key) {
    if (root == null) return null;
    if (key < root.val) root.left = delete(root.left, key);
    else if (key > root.val) root.right = delete(root.right, key);
    else {
        if (root.left == null || root.right == null) {
            root = (root.left != null) ? root.left : root.right;
        } else {
            int successor = findMin(root.right);
            root.val = successor;
            root.right = delete(root.right, successor);
        }
    }
    if (root == null) return null;
    root.height = 1 + Math.max(height(root.left), height(root.right));
    int balance = getBalance(root);

    // But in deletion, the node has already been removed - so we don't
    // have a meaningful key to guide direction anymore.

    // Rebalance
    if (balance > 1 && getBalance(root.left) >= 0) return rightRotate(root);
    if (balance > 1 && getBalance(root.left) < 0) {
        root.left = leftRotate(root.left);
        return rightRotate(root);
    }

    if (balance < -1 && getBalance(root.right) <= 0) return leftRotate(root);
    if (balance < -1 && getBalance(root.right) > 0) {
        root.right = rightRotate(root.right);
        return leftRotate(root);
    }

    return root;
}

// Inorder Traversal (sorted)
public static void inorder(TreeNode root) {
    if (root != null) {
        inorder(root.left);
        System.out.print(root.val + " ");
        inorder(root.right);
    }
}

// Demo
public static void main(String[] args) {
    TreeNode root = null;
    int[] values = {10, 20, 30, 40, 50, 25};
    for (int val : values) {
        root = insert(root, val);
    }
    System.out.print("Inorder after insertions: ");
    inorder(root);
    System.out.println();
    System.out.println("Search 30? " + search(root, key: 30)); // true
    System.out.println("Search 100? " + search(root, key: 100)); // false
    root = delete(root, key: 20); // delete a node
    System.out.print("Inorder after deleting 20: ");
    inorder(root);
    System.out.println();
}

Inorder after insertions: 10 20 25 30 40 50
Search 30? true
Search 100? false
Inorder after deleting 20: 10 25 30 40 50

```

Backtracking

Backtracking builds a solution incrementally, and whenever it determines that the current path cannot possibly lead to a valid solution, it backtracks by removing the last step and tries another path.

1. **17. Letter Combinations of a Phone Number**
2. **37. Sudoku Solver – Fill cells recursively with valid digits using backtracking.**
3. **51. N-Queens – Place queens row by row avoiding column and diagonal conflicts.**
4. **79. Word Search – DFS with backtracking to match word on a 2D grid.**
5. **126. Word Ladder II – BFS to build graph, then backtrack all shortest paths.**
6. **140. Word Break II – Recursively partition string using dictionary with memoization.**
7. **494. Target Sum – Backtrack to add/subtract numbers to reach target.**
8. **784. Letter Case Permutation – Explore all case changes using backtracking.**
9. **797. All Paths From Source to Target – DFS all paths from node 0 to n - 1.**
10. **980. Unique Paths III – DFS all valid paths visiting every cell exactly once.**
11. **2664. The Knight's Tour – Try all knight moves with visited tracking and backtrack.**

1. 17. Letter Combinations of a Phone Number

Given a string containing digits from 2–9 inclusive, return all possible letter combinations that the number could represent. Return the answer in any order.

A mapping of digits to letters (just like on the telephone buttons) is given below. Note that 1 does not map to any letters.



Example 1:

Input: digits = "23"
Output: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]

Example 2:

Input: digits = ""
Output: []

Example 3:

Input: digits = "2"
Output: ["a", "b", "c"]

```
public class TimeConverterDFS {  
  
    static class TimeUnit {  
        String label;  
        int valueInMinutes;  
  
        TimeUnit(String label, int valueInMinutes) {  
            this.label = label;  
            this.valueInMinutes = valueInMinutes;  
        }  
  
    }  
  
    static List<TimeUnit> units = Arrays.asList(  
        new TimeUnit("Years", 525600), // 365 days  
        new TimeUnit("Months", 43200), // 30 days  
        new TimeUnit("Days", 1440),  
        new TimeUnit("Hours", 60),  
        new TimeUnit("Minutes", 1),  
        new TimeUnit("Seconds", 1 / 60) // Just for fun – handled differently  
    );  
  
    static Map<String, Integer> result = new LinkedHashMap<>();  
  
    public static void convert(int minutesLeft, int unitIndex) {  
        if (unitIndex >= units.size() || minutesLeft <= 0) return;  
  
        TimeUnit current = units.get(unitIndex);  
        if (current.label.equals("Seconds")) {  
            result.put("Seconds", minutesLeft * 60); // final leftover minutes to seconds  
        } else {  
            int count = minutesLeft / current.valueInMinutes;  
            result.put(current.label, count);  
            convert(minutesLeft % current.valueInMinutes, unitIndex + 1);  
        }  
    }  
  
    public static void main(String[] args) {  
        int totalMinutes = 1000000;  
  
        convert(totalMinutes, 0);  
  
        for (Map.Entry<String, Integer> entry : result.entrySet()) {  
            System.out.println(entry.getKey() + ": " + entry.getValue());  
        }  
    }  
}
```

```

private static final HashMap<Integer, String> map = new HashMap<Integer, String>() {{
    put(0, "");
    put(1, "");
    put(2, "abc");
    put(3, "def");
    put(4, "ghi");
    put(5, "jkl");
    put(6, "mno");
    put(7, "pqrs");
    put(8, "tuv");
    put(9, "wxyz");
}};

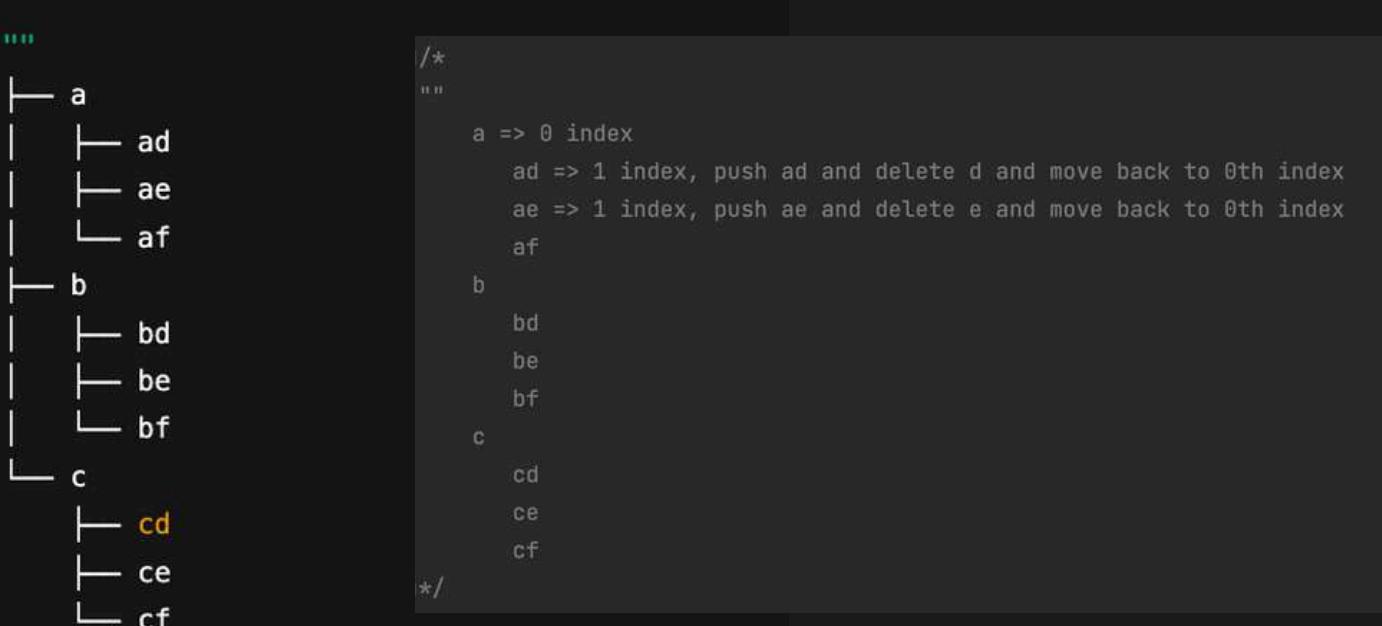
private static void dfs(List<String> result, StringBuilder current, String digits, int index){
    if(digits.length() == index){
        result.add(current.toString());
        return;
    }

    String letters = map.get(Integer.parseInt(String.valueOf(digits.charAt(index))));

    for(char letter: letters.toCharArray()){
        current.append(letter);
        dfs(result, current, digits, index: index + 1);
        current.deleteCharAt(current.length() - 1);
    }
}

public static void main(String[] args) {
    String s = "23"; // => abc * def => ad, ae, af, bd, be, bf, cd, ce, cf
    List<String> result = new ArrayList<>();
    StringBuilder curr = new StringBuilder();
    dfs(result, curr, s, index: 0);
    System.out.println(result);
//    [ad, ae, af, bd, be, bf, cd, ce, cf]
}

```



2. 37. Sudoku Solver – Fill cells recursively with valid digits using backtracking.

```
public class Solution {

    public boolean isSafe(int[][] board, int row, int col, int num, int n) {
        for (int j = 0; j < n; j++) {
            if (board[row][j] == num)
                return false;

        for (int i = 0; i < n; i++) {
            if (board[i][col] == num)
                return false;

        int startRow = row - row % 3,
            startCol = col - col % 3;

        // check within the section (3 * 3)(n = 3)
        for (int i = 0; i < 3; i++) {
            for (int j = 0; j < 3; j++) {
                if (board[i + startRow][j + startCol] == num)
                    return false;
            }
        }

        return true;
    }

    boolean solve(int[][] board, int row, int col, int n) {
        // avoid further tracking if all cells are explored
        if (row == n - 1 && col == n)
            return true;

        // move to next row
        if (col == n) {
            row++;
            col = 0;
        }

        // check for next element horizontally if element is
        // already present
        if (board[row][col] != 0)
            return solve(board, row, col + 1, n);

        // if present value is 0, replace it with values from 0-9
        for (int num = 1; num < n + 1; num++) {
            if (isSafe(board, row, col, num, n)) {
                board[row][col] = num;

                // check for next column
                if (solve(board, row, col + 1, n))
                    return true;
            }
        }
        board[row][col] = 0;
    }

    return false;
}
}
```

- If we've reached the bottom right cell return true
- If we've reached the last column, move to next row
- If value is already present move to next col
- Start from num 1 to n+1
- mark cell as num
- check for next col using recursion, if its true return the same
- Backtrack to check for other value

```

public void solveSudoku(char[][] board) {
    int n = board[0].length;
    int grid[][] = new int[n][n];
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (board[i][j] == '.') {
                grid[i][j] = 0;
            } else {
                grid[i][j] = Integer.parseInt(String.valueOf(board[i][j]));
            }
        }
    }
    solve(grid, 0, 0, n);
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 0) {
                board[i][j] = '.';
            } else {
                board[i][j] = (char) (grid[i][j] + '0');
            }
        }
    }
}

```

```

solve(0,0)
└─ Try 1 ✓
  └─ solve(0,1)
    └─ Already filled → solve(0,2)
    └─ Try 5 ✓
      └─ solve(0,3)
        └─ Try 2 ✓
          └─ solve(0,4)
          └─ ...
          └─ Backtrack (none worked)
    └─ Backtrack (other numbers fail)
└─ Try 2 ✗ → not safe
└─ Try 3 ✓
  └─ solve(0,1) ...

```

3.22. Generate Parentheses

Given n pairs of parentheses, write a function to generate all combinations of well-formed parentheses.

Example 1:

```

Input: n = 3
Output: ["((()))","(()())","(())()","(())()","(()()())"]

```

Example 2:

```

Input: n = 1
Output: ["()"]

```

Constraints:

- $1 \leq n \leq 8$

```

import java.util.ArrayList;
import java.util.List;

class Solution {
    public List<String> generateParenthesis(int n) {
        List<String> result = new ArrayList<>();
        dfs("", n, n, result);
        return result;
    }

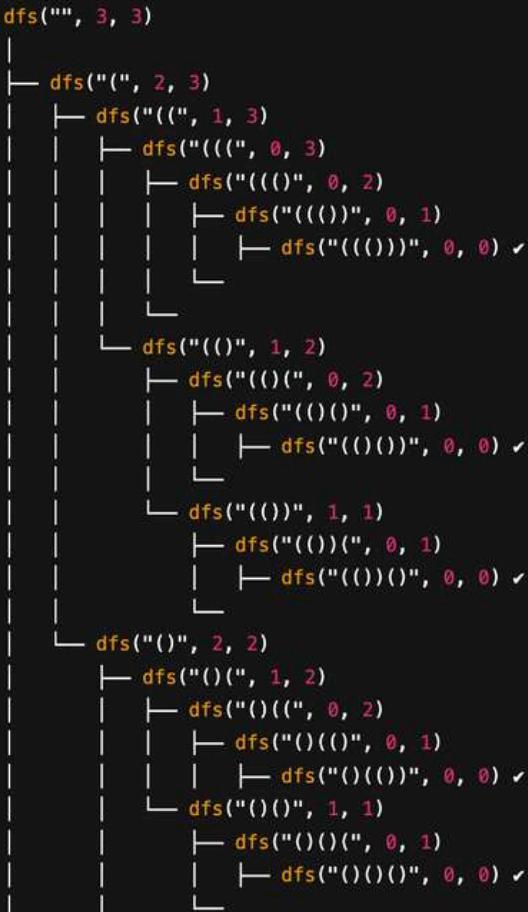
    private void dfs(String current, int left, int right, List<String> result) {
        if (left == 0 && right == 0) {
            result.add(current);
            return;
        }

        if (left > 0) {
            dfs(current + "(", left - 1, right, result);
        }

        if (left < right) {
            dfs(current + ")", left, right - 1, result);
        }
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        List<String> combinations = solution.generateParenthesis(3);
        System.out.println(combinations); // Output: [((())), ((())), ((())()), ((())()), ((())(), ((())))
    }
}

```



4. 39. Combination Sum

Given an array of distinct integers `candidates` and a target integer `target`, return a list of all unique combinations of `candidates` where the chosen numbers sum to `target`. You may return the combinations in any order.

The same number may be chosen from `candidates` an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.

The test cases are generated such that the number of unique combinations that sum up to `target` is less than 150 combinations for the given input.

Example 1:

```
Input: candidates = [2,3,6,7], target = 7
Output: [[2,2,3],[7]]
Explanation:
2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.
7 is a candidate, and 7 = 7.
These are the only two combinations.
```

Example 2:

```
Input: candidates = [2,3,5], target = 8
Output: [[2,2,2,2],[2,3,3],[3,5]]
```

Example 3:

```
Input: candidates = [2], target = 1
Output: []
```

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class Solution {

    public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>> res = new ArrayList<()>;
        Arrays.sort(candidates);
        dfs(0, candidates, target, res, new ArrayList<()>());
        return res;
    }

    private void dfs(int s, int[] candidates, int target, List<List<Integer>> res, List<Integer> curr) {
        if (target < 0)
            return;
        if (target == 0) {
            res.add(new ArrayList<()>(curr));
            return;
        }
        for (int i = s; i < candidates.length; ++i) {
            // if(int i=0... , ans = [[2, 2, 3], [2, 3, 2], [3, 2, 2], [7]])
            curr.add(candidates[i]);
            dfs(i, candidates, target - candidates[i], res, curr);
            curr.remove(curr.size() - 1);
        }
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] candidates = { 2, 3, 6, 7 };
        int target = 7;
        List<List<Integer>> combinations = solution.combinationSum(candidates, target);
        System.out.println(combinations); // Output: [[2, 2, 3], [7]]
    }
}
```



5. 40. Combination Sum II

Medium

Given a collection of candidate numbers (`candidates`) and a target number (`target`), find all unique combinations in `candidates` where the candidate numbers sum to `target`.

Each number in `candidates` may only be used once in the combination.

Note: The solution set must not contain duplicate combinations.

Example 1:

Input: candidates = [10,1,2,7,6,1,5], target = 8

Output:

```
[  
    [1,1,6],  
    [1,2,5],  
    [1,7],  
    [2,6]  
]
```

Example 2:

Input: candidates = [2,5,2,1,2], target = 5

Output:

```
[  
    [1,2,2],  
    [5]  
]
```

```
import java.util.*;
```

```
class Solution {  
    public List<List<Integer>> combinationSum2(int[] candidates, int target) {  
        List<List<Integer>> result = new ArrayList<List<Integer>>();  
        if (candidates == null || candidates.length == 0)  
            return result;  
        ArrayList<Integer> current = new ArrayList<Integer>();  
        Arrays.sort(candidates);  
        dfs(candidates, target, 0, current, result);  
        return result;  
    }  
  
    public void dfs(int[] candidates, int target, int j, ArrayList<Integer> curr,  
                    List<List<Integer>> result) {  
        if (target == 0) {  
            ArrayList<Integer> temp = new ArrayList<Integer>(curr);  
            result.add(temp);  
            return;  
        }  
        for (int i = j; i < candidates.length; i++) {  
            if (target < candidates[i]) {  
                return;  
            }  
            if (i == j || candidates[i] != candidates[i - 1]) {  
                curr.add(candidates[i]);  
                dfs(candidates, target - candidates[i], i + 1, curr, result);  
                curr.remove(curr.size() - 1);  
            }  
        }  
    }  
}
```

First sort the candidates, if the consecutive elements are not the same. Perform dfs

```
dfs(7, 0, [])  
|   Add 2 → dfs(5, 1, [2])  
|   |   Add 3 → dfs(2, 2, [2, 3])  
|   |   |   Add 6 → dfs(-4, 3, [2, 3, 6]) ✗  
|   |   |   Add 7 → dfs(-5, 4, [2, 3, 7]) ✗  
|   |   Add 6 → dfs(-1, 3, [2, 6]) ✗  
|   |   Add 7 → dfs(-2, 4, [2, 7]) ✗  
|   Add 3 → dfs(4, 2, [3])  
|   |   Add 6 → dfs(-2, 3, [3, 6]) ✗  
|   |   Add 7 → dfs(-3, 4, [3, 7]) ✗  
|   Add 6 → dfs(1, 3, [6])  
|       Add 7 → dfs(-6, 4, [6, 7]) ✗  
|       Add 7 → dfs(0, 4, [7]) ✓ (found)
```

6. 46. Permutations

Given an array `nums` of distinct integers, return all the possible permutations. You can return the answer in any order.

Example 1:

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

Example 2:

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

Example 3:

```
Input: nums = [1]
Output: [[1]]
```

Constraints:

- $1 \leq \text{nums.length} \leq 6$
- $-10 \leq \text{nums}[i] \leq 10$
- All the integers of `nums` are unique.

```
import java.util.ArrayList;
import java.util.List;

class Solution {

    public List<List<Integer>> permute(int[] nums) {
        List<List<Integer>> res = new ArrayList<()>();
        boolean[] visited = new boolean[nums.length];
        dfs(nums, res, new ArrayList<()>, visited);
        return res;
    }

    private void dfs(int[] nums, List<List<Integer>> res, List<Integer> curr, boolean[] visited) {
        if (curr.size() == nums.length) {
            res.add(new ArrayList<()>(curr));
            return;
        }
        for (int i = 0; i < nums.length; i++) {
            if (!visited[i]) {
                visited[i] = true;
                curr.add(nums[i]);
                dfs(nums, res, curr, visited);
                curr.remove(curr.size() - 1);
                visited[i] = false;
            }
        }
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {1, 2, 3};
        List<List<Integer>> permutations = solution.permute(nums);
        System.out.println(permutations); // Output: [[1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1]]
    }
}
```

```
/*
1
  1 visited true
  2 visited false
    1 visited true
    2 visited true
      3 visited false [1,2,3]
2
  1 visited false
    1 visited true
    2 visited true
      3 visited false [2,1,3]
3
  ...
*/

```

7.47. Permutations 2

Given a collection of numbers, `nums`, that might contain duplicates, return *all possible unique permutations in any order*.

Example 1:

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

```
[[1,1,2],  
[1,2,1],  
[2,1,1]]
```

Example 2:

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

Constraints:

- `1 <= nums.length <= 8`
- `-10 <= nums[i] <= 10`

```
1 import java.util.ArrayList;
2 import java.util.Arrays;
3 import java.util.List;
4
5 class Solution {
6
7     public List<List<Integer>> permuteUnique(int[] nums) {
8         List<List<Integer>> res = new ArrayList<>();
9         boolean[] visited = new boolean[nums.length];
10        if (nums == null || nums.length == 0) {
11            return res;
12        }
13        Arrays.sort(nums); // Sort the array to handle duplicates.
14        dfs(nums, res, new ArrayList<>(), visited);
15        return res;
16    }
17
18    private void dfs(int[] nums, List<List<Integer>> res, List<Integer> curr, boolean[] visited) {
19        if (curr.size() == nums.length) {
20            res.add(new ArrayList<>(curr));
21            return;
22        }
23        for (int i = 0; i < nums.length; i++) {
24            if (visited[i]) {
25                continue;
26            }
27            if (i == 0 || nums[i] != nums[i - 1] || (nums[i] == nums[i - 1] && visited[i - 1])) {
28                visited[i] = true;
29                curr.add(nums[i]);
30                dfs(nums, res, curr, visited);
31                curr.remove(curr.size() - 1); // Backtrack
32                visited[i] = false;
33            }
34        }
35    }
36
37    public static void main(String[] args) {
38        Solution solution = new Solution();
39        int[] nums = {1, 1, 2};
40        List<List<Integer>> permutations = solution.permuteUnique(nums);
41        System.out.println(permutations); // Output: [[1, 1, 2], [1, 2, 1], [2, 1, 1]]
42    }
43 }
44
```

```
dfs([], visited=[F,F,F])
| Add 1 (i=0) → dfs([1], [T,F,F])
| Skip i=0 (visited)
| | Add 1 (i=1) → dfs([1,1], [T,T,F])
| | | Skip i=0, i=1 (visited)
| | | | Add 2 (i=2) → dfs([1,1,2], [T,T,T]) ✓
| | | | Backtrack → [1,1]
| | | Backtrack → [1]
| | | Add 2 (i=2) → dfs([1,2], [T,F,T])
| | | | Add 1 (i=1) → dfs([1,2,1], [T,T,T]) ✓
| | | | Backtrack → [1,2]
| | | Backtrack → []
|
| Skip 1 (i=1) (duplicate not allowed because visited[i-1] = false)
| Add 2 (i=2) → dfs([2], [F,F,T])
| | Add 1 (i=0) → dfs([2,1], [T,F,T])
| | | Add 1 (i=1) → dfs([2,1,1], [T,T,T]) ✓
| | | Backtrack → [2,1]
| | Skip 1 (i=1) (duplicate not allowed because visited[i-1] = false)
| | Backtrack → [2]
```

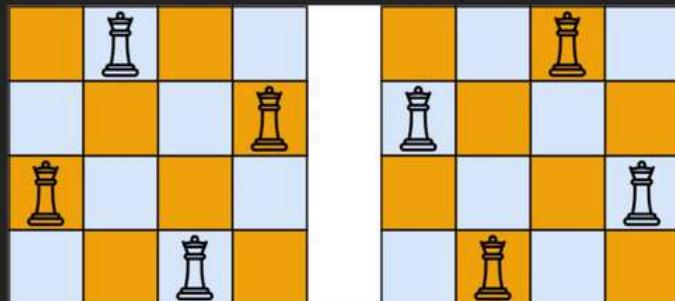
8. 51. N-Queens

The **n-queens** puzzle is the problem of placing n queens on an $n \times n$ chessboard such that no two queens attack each other.

Given an integer n , return all distinct solutions to the **n-queens puzzle**. You may return the answer in any order.

Each solution contains a distinct board configuration of the n -queens' placement, where '`'Q'`' and '`'.'`' both indicate a queen and an empty space, respectively.

Example 1:



Input: $n = 4$

Output: `[["Q...","...Q...","...Q...","...Q..."], ["...Q...","Q...Q...","...Q...","...Q..."], [...]`

Explanation: There exist two distinct solutions to the 4-queens puzzle as shown above.

Example 2:

Input: $n = 1$

Output: `[["Q"]]`

```
class Solution {
    public List<List<String>> solveNQueens(int n) {
        List<List<String>> res = new ArrayList<>();
        char[][] board = new char[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                board[i][j] = '.';
            }
        }
        dfs(board, 0, res);
        return res;
    }

    private void dfs(char[][] board, int row, List<List<String>> res) {
        if (row == board.length) {
            res.add(construct(board));
            return;
        }
        for (int col = 0; col < board.length; col++) {
            if (isValid(board, row, col)) {
                board[row][col] = 'Q';
                dfs(board, row + 1, res);
                board[row][col] = '.';
            }
        }
    }

    private boolean isValid(char[][] board, int row, int col) {
        for (int i = 0; i < row; i++) {
            if (board[i][col] == 'Q') {
                return false;
            }
        }
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 'Q') {
                return false;
            }
        }
        for (int i = row - 1, j = col + 1; i >= 0 && j < board.length; i--, j++) {
            if (board[i][j] == 'Q') {
                return false;
            }
        }
        return true;
    }
}
```

For each row, check if placing a queen in a column is safe using `isValid`. If valid, place the queen, recurse to the next row, and backtrack after exploring.

```

private List<String> construct(char[][] board) {
    List<String> path = new ArrayList<>();
    for (int i = 0; i < board.length; i++) {
        path.add(new String(board[i]));
    }
    return path;
}

public static void main(String[] args) {
    Solution solution = new Solution();
    int n = 4;
    List<List<String>> result = solution.solveNQueens(n);
    for (List<String> solutionBoard : result) {
        for (String row : solutionBoard) {
            System.out.println(row);
        }
        System.out.println();
    }
}

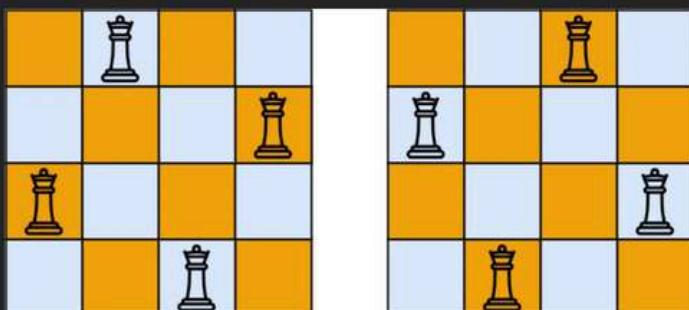
```

9. 52. N-Queens II

The **n-queens** puzzle is the problem of placing n queens on an $n \times n$ chessboard such that no two queens attack each other.

Given an integer n , return the number of distinct solutions to the **n-queens puzzle**.

Example 1:



Input: $n = 4$

Output: 2

Explanation: There are two distinct solutions to the 4-queens puzzle as shown.

Example 2:

Input: $n = 1$

Output: 1

```

class Solution {
    private int count = 0;
    public int totalNQueens(int n) {
        char[][] board = new char[n][n];
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n; j++) {
                board[i][j] = '.';
            }
        }
        dfs(board, 0);
        return count;
    }

    private void dfs(char[][] board, int row) {
        if (row == board.length) {
            count++;
            return;
        }
        for (int col = 0; col < board.length; col++) {
            if (isValid(board, row, col)) {
                board[row][col] = 'Q';
                dfs(board, row + 1);
                board[row][col] = '.';
            }
        }
    }

    private boolean isValid(char[][] board, int row, int col) {
        for (int i = 0; i < row; i++) {
            if (board[i][col] == 'Q') return false;
        }
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] == 'Q') return false;
        }
        for (int i = row - 1, j = col + 1; i >= 0 && j < board.length; i--, j++) {
            if (board[i][j] == 'Q') return false;
        }
        return true;
    }
}

```

10.131. Palindrome Partitioning

Given a string s , partition s such that every substring of the partition is a palindrome. Return all possible palindrome partitioning

of s .

Example 1:

Input: $s = "aab"$
Output: $[["a", "a", "b"], ["aa", "b"]]$

Example 2:

Input: $s = "a"$
Output: $[["a"]]$

Constraints:

- $1 \leq s.length \leq 16$
- s contains only lowercase English letters.

```
class Solution {
    public List<List<String>> partition(String s) {
        List<List<String>> result = new ArrayList<>();
        dfs(s, 0, new ArrayList<>(), result);
        return result;
    }

    private void dfs(String s, int start, List<String> current, List<List<String>> result) {
        if (start == s.length()) {
            result.add(new ArrayList<>(current));
            return;
        }
        for (int end = start; end < s.length(); end++) {
            String substring = s.substring(start, end + 1);
            if (isPalindrome(substring)) {
                current.add(substring);
                dfs(s, end + 1, current, result);
                current.remove(current.size() - 1);
            }
        }
    }

    private boolean isPalindrome(String s) {
        int left = 0;
        int right = s.length() - 1;
        while (left < right) {
            if (s.charAt(left) != s.charAt(right)) {
                return false;
            }
            left++;
            right--;
        }
        return true;
    }
}
```

11. 78. Subsets

Given an integer array `nums` of unique elements, return all possible `subsets` (the power set).

The solution set must not contain duplicate subsets. Return the solution in any order.

Example 1:

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

Example 2:

Input: `nums = [0]`
Output: `[[],[0]]`

Constraints:

- `1 <= nums.length <= 10`
- `-10 <= nums[i] <= 10`
- All the numbers of `nums` are unique.

```
import java.util.ArrayList;
import java.util.List;

class Solution {
    public List<List<Integer>> subsets(int[] nums) {
        List<List<Integer>> result = new ArrayList<>();
        dfs(result, new ArrayList<>(), nums, 0);
        return result;
    }

    private void dfs(List<List<Integer>> result, List<Integer> current, int[] nums, int start) {
        result.add(new ArrayList<>(current));
        for (int i = start; i < nums.length; i++) {
            current.add(nums[i]);
            dfs(result, current, nums, i + 1);
            current.remove(current.size() - 1);
        }
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {1, 2, 3};
        List<List<Integer>> result = solution.subsets(nums);
        for (List<Integer> subset : result) {
            System.out.println(subset);
        }
    }
}
```

12. 77. Combinations

Given two integers n and k , return all possible combinations of k numbers chosen from the range $[1, n]$.

You may return the answer in **any order**.

Example 1:

Input: $n = 4$, $k = 2$

Output: $\{[1,2], [1,3], [1,4], [2,3], [2,4], [3,4]\}$

Explanation: There are $4 \text{ choose } 2 = 6$ total combinations.

Note that combinations are unordered, i.e., $[1,2]$ and $[2,1]$ are considered to be the same combination.

Example 2:

Input: $n = 1$, $k = 1$

Output: $\{[1]\}$

Explanation: There is $1 \text{ choose } 1 = 1$ total combination.

Constraints:

- $1 \leq n \leq 20$
- $1 \leq k \leq n$

```
import java.util.ArrayList;
import java.util.List;

class Solution {
    public List<List<Integer>> combine(int n, int k) {
        List<List<Integer>> result = new ArrayList<List<Integer>>();
        dfs(result, new ArrayList<Integer>(), n, k, 1);
        return result;
    }

    private void dfs(List<List<Integer>> result, List<Integer> combination, int n,
                    int k, int start) {
        if (combination.size() == k) {
            result.add(new ArrayList<Integer>(combination));
            return;
        }

        for (int i = start; i <= n; i++) {
            combination.add(i);
            dfs(result, combination, n, k, i + 1);
            combination.remove(combination.size() - 1);
        }
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int n = 4;
        int k = 2;
        List<List<Integer>> result = solution.combine(n, k);
        for (List<Integer> combo : result) {
            System.out.println(combo);
        }
    }
}
```

13. 2664. The Knight's Tour

Given two positive integers m and n which are the height and width of a 0-indexed 2D-array `board`, a pair of positive integers (r, c) which is the starting position of the knight on the board.

Your task is to find an order of movements for the knight, in a manner that every cell of the `board` gets visited exactly once (the starting cell is considered visited and you shouldn't visit it again).

Return the array `board` in which the cells' values show the order of visiting the cell starting from 0 (the initial place of the knight).

Note that a knight can move from cell (r_1, c_1) to cell (r_2, c_2) if $0 \leq r_2 \leq m - 1$ and $0 \leq c_2 \leq n - 1$ and $\min(\text{abs}(r_1 - r_2), \text{abs}(c_1 - c_2)) = 1$ and $\max(\text{abs}(r_1 - r_2), \text{abs}(c_1 - c_2)) = 2$.

Example 1:

Input: $m = 1, n = 1, r = 0, c = 0$
Output: `[[0]]`
Explanation: There is only 1 cell and the knight is initially on it so there is only a 0 inside the 1x1 grid.

Example 2:

Input: $m = 3, n = 4, r = 0, c = 0$
Output: `[[0,3,6,9],[11,8,1,4],[2,5,10,7]]`
Explanation: By the following order of movements we can visit the entire board.
 $(0,0) \rightarrow (1,2) \rightarrow (2,0) \rightarrow (0,1) \rightarrow (1,3) \rightarrow (2,1) \rightarrow (0,2) \rightarrow (2,3) \rightarrow (1,1) \rightarrow (0,3) \rightarrow (2,2) \rightarrow (1,0)$

```
import java.util.*;

class Solution {
    private int[][] board;
    private int rows, cols;
    private boolean foundSolution = false;
    private final int[][] knightMoves = {
        {-2, -1}, {-1, -2}, {1, -2}, {2, -1},
        {2, 1}, {1, 2}, {-1, 2}, {-2, 1}
    };

    public List<List<Integer>> tourOfKnight(int rows, int cols, int startRow, int startCol) {
        this.rows = rows;
        this.cols = cols;
        board = new int[rows][cols];
        for (int[] row : board) {
            Arrays.fill(row, -1);
        }

        board[startRow][startCol] = 0;
        dfs(startRow, startCol, 0);

        // Convert int[][] to List<List<Integer>> to match C++ vector<vector<int>>
        List<List<Integer>> result = new ArrayList<>();
        for (int[] row : board) {
            List<Integer> list = new ArrayList<>();
            for (int val : row) {
                list.add(val);
            }
            result.add(list);
        }
        return result;
    }

    private void dfs(int row, int col, int step) {
        if (step == rows * cols) {
            foundSolution = true;
            return;
        }

        for (int[] move : knightMoves) {
            int newRow = row + move[0];
            int newCol = col + move[1];
            if (newRow < 0 || newRow >= rows || newCol < 0 || newCol >= cols || board[newRow][newCol] != -1) {
                continue;
            }

            board[newRow][newCol] = step;
            dfs(newRow, newCol, step + 1);
            board[newRow][newCol] = -1;
        }
    }
}
```

```
private void dfs(int row, int col, int moveCount) {
    if (moveCount == rows * cols - 1) {
        foundSolution = true;
        return;
    }

    for (int[] move : knightMoves) {
        int newRow = row + move[0];
        int newCol = col + move[1];
        if (isSafe(newRow, newCol)) {
            board[newRow][newCol] = moveCount + 1;
            dfs(newRow, newCol, moveCount + 1);
            if (foundSolution) return;
            board[newRow][newCol] = -1; // backtrack
        }
    }
}

private boolean isSafe(int r, int c) {
    return r >= 0 && r < rows && c >= 0 && c < cols && board[r][c] == -1;
}

// Optional main for testing
public static void main(String[] args) {
    Solution sol = new Solution();
    List<List<Integer>> res = sol.tourOfKnight(5, 5, 0, 0);
    for (List<Integer> row : res) {
        System.out.println(row);
    }
}
```

14. 79. Word Search

79. Word Search

Given an $m \times n$ grid of characters `board` and a string `word`, return `true` if `word` exists in the grid.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example 1:

A	B	C	E
S	F	C	S
A	D	E	E

```
Input: board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]],  
       word = "ABCED"
```

```
Output: true
```

```
class Solution {  
    private static final int[][] DIRECTIONS = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};  
  
    public boolean exist(char[][] board, String word) {  
        int rows = board.length;  
        int cols = board[0].length;  
  
        for (int row = 0; row < rows; row++) {  
            for (int col = 0; col < cols; col++) {  
                if (dfs(board, word, row, col, 0)) {  
                    return true;  
                }  
            }  
        }  
  
        return false;  
    }  
  
    private boolean dfs(char[][] board, String word, int row, int col, int index)  
    {  
        if (index == word.length()) {  
            return true;  
        }  
        if (row < 0 || row >= board.length || col < 0 || col >= board[0].length ||  
            board[row][col] != word.charAt(index)) {  
            return false;  
        }  
        char temp = board[row][col];  
        board[row][col] = '#';  
        for (int[] direction : DIRECTIONS) {  
            int newRow = row + direction[0];  
            int newCol = col + direction[1];  
            if (dfs(board, word, newRow, newCol, index + 1)) {  
                return true;  
            }  
        }  
        board[row][col] = temp;  
        return false;  
    }  
}
```

15. Word Break II

Given a string `s` and a dictionary of strings `wordDict`, add spaces in `s` to construct a sentence where each word is a valid dictionary word. Return all such possible sentences in any order.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

Input: `s = "catsanddog"`, `wordDict = ["cat", "cats", "and", "sand", "dog"]`
Output: `["cats and dog", "cat sand dog"]`

Example 2:

Input: `s = "pineapplepenapple"`, `wordDict = ["apple", "pen", "applepen", "pine", "pineapple"]`
Output: `["pine apple pen apple", "pineapple pen apple", "pine applepen apple"]`
Explanation: Note that you are allowed to reuse a dictionary word.

Example 3:

Input: `s = "catsandog"`, `wordDict = ["cats", "dog", "sand", "and", "cat"]`
Output: `[]`

```
class Solution {

    public List<String> wordBreak(String s, List<String> wordDict) {
        return dfs(s, wordDict, new HashMap<>());
    }

    private List<String> dfs(String s, List<String> wordDict, Map<String, List<String>> memo) {
        if (memo.containsKey(s)) {
            return memo.get(s);
        }
        if (s.length() == 0) {
            List<String> baseResult = new ArrayList<>();
            baseResult.add("");
            return baseResult;
        }

        List<String> res = new ArrayList<>();
        for (String word : wordDict) {
            if (s.startsWith(word)) {
                List<String> sublist = dfs(s.substring(word.length()), wordDict, memo);
                for (String sub : sublist) {
                    res.add(word + (sub.isEmpty() ? "" : " ") + sub);
                }
            }
        }
        memo.put(s, res);
        return res;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        String s = "catsanddog";
        List<String> wordDict = new ArrayList<>();
        wordDict.add("cat");
        wordDict.add("cats");
        wordDict.add("and");
        wordDict.add("sand");
        wordDict.add("dog");

        List<String> result = solution.wordBreak(s, wordDict);
        for (String sentence : result) {
            System.out.println(sentence);
        }
    }
}
```

15. Target Sum

494. Target Sum

Medium

You are given an integer array `nums` and an integer `target`.

You want to build an expression out of `nums` by adding one of the symbols `'+'` and `'-'` before each integer in `nums` and then concatenate all the integers.

- For example, if `nums = [2, 1]`, you can add a `'+'` before `2` and a `'-'` before `1` and concatenate them to build the expression `"+2-1"`.

Return the number of different expressions that you can build, which evaluates to `target`.

Example 1:

```
Input: nums = [1,1,1,1,1], target = 3
Output: 5
Explanation: There are 5 ways to assign symbols to make the sum of nums be
target 3.
-1 + 1 + 1 + 1 + 1 = 3
+1 - 1 + 1 + 1 + 1 = 3
+1 + 1 - 1 + 1 + 1 = 3
+1 + 1 + 1 - 1 + 1 = 3
+1 + 1 + 1 + 1 - 1 = 3
```

Example 2:

```
Input: nums = [1], target = 1
Output: 1
```

Constraints:

- `1 <= nums.length <= 20`

```
import java.util.HashMap;
import java.util.Map;

class Solution {
    public int findTargetSumWays(int[] nums, int target) {
        return dfs(nums, 0, 0, target, new HashMap<>());
    }

    private int dfs(int[] nums, int index, int currentSum, int target, Map<String, Integer> memo) {
        if (index == nums.length) {
            return currentSum == target ? 1 : 0;
        }
        String key = index + "," + currentSum;
        if (memo.containsKey(key)) {
            return memo.get(key);
        }
        int add = dfs(nums, index + 1, currentSum + nums[index], target, memo);
        int subtract = dfs(nums, index + 1, currentSum - nums[index], target, memo);
        memo.put(key, add + subtract);
        return add + subtract;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums = {1, 1, 1, 1, 1};
        int target = 3;
        System.out.println(solution.findTargetSumWays(nums, target)); // Output: 5
    }
}
```

16. Letter Case Permutation

Given a string `s`, you can transform every letter individually to be lowercase or uppercase to create another string.

Return a list of all possible strings we could create. Return the output in any order.

Example 1:

```
Input: s = "a1b2"
Output: ["a1b2","a1B2","A1b2","A1B2"]
```

Example 2:

```
Input: s = "3z4"
Output: ["3z4","3Z4"]
```

Constraints:

- `1 <= s.length <= 12`
- `s` consists of lowercase English letters, uppercase English letters, and digits.

```
class Solution {
    public List<String> letterCasePermutation(String S) {
        List<String> ans = new ArrayList<>();
        dfs(new StringBuilder(S), 0, ans);
        return ans;
    }

    private void dfs(StringBuilder sb, int i, List<String> ans) {
        if (i == sb.length()) {
            ans.add(sb.toString());
            return;
        }
        if (Character.isDigit(sb.charAt(i))) {
            dfs(sb, i + 1, ans);
        } else {
            sb.setCharAt(i, Character.toLowerCase(sb.charAt(i)));
            dfs(sb, i + 1, ans);
            sb.setCharAt(i, Character.toUpperCase(sb.charAt(i)));
            dfs(sb, i + 1, ans);
        }
    }
}
```

17. All Paths From Source to Target

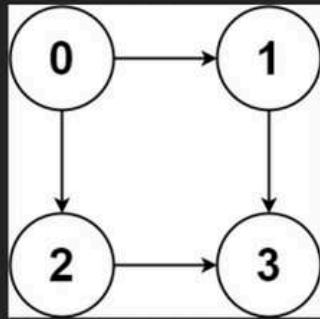
797. All Paths From Source to Target

Medium

Given a directed acyclic graph (DAG) of n nodes labeled from 0 to $n - 1$, find all possible paths from node 0 to node $n - 1$ and return them in any order.

The graph is given as follows: `graph[i]` is a list of all nodes you can visit from node i (i.e., there is a directed edge from node i to node `graph[i][j]`).

Example 1:



Input: `graph = [[1,2],[3],[3],[]]`

Output: `[[0,1,3],[0,2,3]]`

Explanation: There are two paths: $0 \rightarrow 1 \rightarrow 3$ and $0 \rightarrow 2 \rightarrow 3$.

```
class Solution {
public:
    vector<vector<int>> allPathsSourceTarget(vector<vector<int>>& graph) {
        vector<vector<int>> result;
        vector<int> path;
        dfs(graph, 0, path, result);
        return result;
    }

    void dfs(vector<vector<int>>& graph, int node, vector<int>& path, vector<vector<int>>& result) {
        path.push_back(node);

        if (node == graph.size() - 1) {
            result.push_back(path);
        } else {
            for (int neighbor : graph[node]) {
                dfs(graph, neighbor, path, result);
            }
        }

        path.pop_back();
    }
};
```

18. 93. Restore IP Addresses

93. Restore IP Addresses

Medium

A valid IP address consists of exactly four integers separated by single dots. Each integer is between 0 and 255 (inclusive) and cannot have leading zeros.

- For example, "0.1.2.201" and "192.168.1.1" are valid IP addresses, but "0.011.255.245", "192.168.1.312" and "192.168@1.1" are invalid IP addresses.

Given a string s containing only digits, return all possible valid IP addresses that can be formed by inserting dots into s. You are not allowed to reorder or remove any digits in s. You may return the valid IP addresses in any order.

Example 1:

```
Input: s = "25525511135"
Output: ["255.255.11.135","255.255.111.35"]
```

Example 2:

```
Input: s = "0000"
Output: ["0.0.0.0"]
```

Example 3:

```
Input: s = "101023"
Output: ["1.0.10.23","1.0.102.3","10.1.0.23","10.10.2.3","101.0.2.3"]
```

Constraints:

- 1 <= s.length <= 20
- s consists of digits only.

```
public class Solution {
    public List<String> restoreIpAddresses(String s) {
        List<String> result = new ArrayList<>();
        List<String> segments = new ArrayList<>();
        dfs(s, 0, segments, result);
        return result;
    }

    private void dfs(String s, int start, List<String> segments, List<String> result) {
        if (segments.size() == 4) {
            if (start == s.length()) {
                result.add(String.join(".", segments));
            }
            return;
        }

        for (int len = 1; len <= 3; len++) {
            if (start + len > s.length()) break;

            String segment = s.substring(start, start + len);

            if (isValid(segment)) {
                segments.add(segment);
                dfs(s, start + len, segments, result);
                segments.remove(segments.size() - 1); // backtrack
            }
        }
    }

    private boolean isValid(String segment) {
        if (segment.length() == 0 || segment.length() > 3) return false;
        if (segment.startsWith("0") && segment.length() > 1) return false;

        int value = Integer.parseInt(segment);
        return value >= 0 && value <= 255;
    }
}
```

19. 988. Smallest String Starting From Leaf

You are given the `root` of a binary tree where each node has a value in the range `[0, 25]` representing the letters `'a'` to `'z'`.

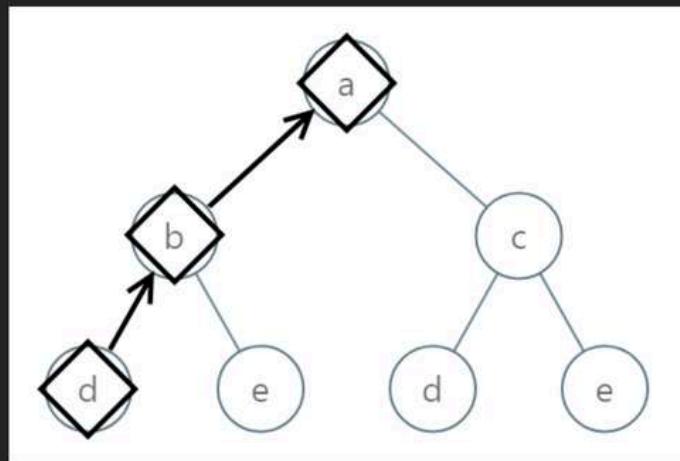
Return the lexicographically smallest string that starts at a leaf of this tree and ends at the root.

As a reminder, any shorter prefix of a string is lexicographically smaller.

- For example, `"ab"` is lexicographically smaller than `"aba"`.

A leaf of a node is a node that has no children.

Example 1:



Input: `root = [0,1,2,3,4,3,4]`

Output: `"dba"`

```
class Solution {
public:
    string smallestFromLeaf(TreeNode* root) {
        string result = "";
        dfs(root, "", result);
        return result;
    }

    void dfs(TreeNode* node, string path, string& result) {
        if (!node) return;
        path = char('a' + node->val) + path;

        if (!node->left && !node->right) {
            if (result == "" || path < result) {
                result = path;
            }
        }

        dfs(node->left, path, result);
        dfs(node->right, path, result);
    }
};
```

Dynamic Programming

1.10. Regular Expression Matching

Given an input string s and a pattern p , implement regular expression matching with support for $'.'$ and $'*'$ where:

- $'.'$ Matches any single character.
- $'*'$ Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

Example 1:

```
Input: s = "aa", p = "a"
Output: false
Explanation: "a" does not match the entire string "aa".
```

Example 2:

```
Input: s = "aa", p = "a*"
Output: true
Explanation: '*' means zero or more of the preceding element, 'a'. Therefore, by
repeating 'a' once, it becomes "aa".
```

Example 3:

```
Input: s = "ab", p = ".*"
Output: true
Explanation: ".*" means "zero or more (*) of any character (.)".
```

```
class Solution {
    public boolean isMatch(String s, String p) {
        int m = s.length();
        int n = p.length();
        boolean dp[][] = new boolean[m + 1][n + 1];
        dp[0][0] = true;
        for (int j = 1; j <= n; j++) {
            if (p.charAt(j - 1) == '*') {
                dp[0][j] = dp[0][j - 2];
            }
        }

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (p.charAt(j - 1) == '.' || p.charAt(j - 1) == s.charAt(i - 1)) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else if (p.charAt(j - 1) == '*') {
                    dp[i][j] = dp[i][j - 2] ||
                               (dp[i - 1][j] && (s.charAt(i - 1) == p.charAt(j - 2) || p.charAt(j - 2) == '.'));
                }
            }
        }

        System.out.println(Arrays.toString(dp));
        return dp[m][n];
    }
}
```

Example 1:

Input: $s = "aa"$, $p = "a"$
Output: false

graphql

ϵ	a
ϵ	True False
a	False True
a	False False

$dp[i - 1][j]$

This checks if the substring $s[0\dots i-2]$ matches with $p[0\dots j-1]$.

We're saying: "We already had a match up to the previous character in s , so if the character at $i-1$ still matches the repeating character before $*$, we can carry forward the match."

$(s.charAt(i - 1) == p.charAt(j - 2) \mid p.charAt(j - 2) == '.')$

This checks if:

- The current character in the string s matches the character before $*$ in pattern p , OR
- The character before the $*$ is a $.$ which can match any character.

java

Copy Edit

```
(dp[i - 1][j] && (s.charAt(i - 1) == p.charAt(j - 2) || p.charAt(j - 2) == '.'))
```

represents handling the case where the current pattern character is $*$ and we want to match one or more occurrences of the character before $*$.

Example 2:

Input: s = "aa", p = "a*"

Output: true

ε	a	*
ε	True	False
a	False	True
a	False	True

Example 3:

Input: s = "ab", p = ".*"

Output: true

ε	.	*
ε	True	False
a	False	True
b	False	True

If the pattern character is *, set $dp[0][j] = dp[0][j-2]$ for the first row to account for zero occurrences of the preceding element.

Then, iterate through the string (i from 1 to length of string) and the pattern (j from 1 to length of pattern):

- If the current pattern character is ., set $dp[i][j] = dp[i-1][j-1]$, as . matches any single character.
- If the current pattern character is *, then:
 - First, check for zero occurrences: if $dp[i][j-2]$ is true, set $dp[i][j] = \text{true}$.
 - Then, if the preceding character matches the current string character or is ., and $dp[i-1][j]$ is true, set $dp[i][j] = \text{true}$.

2. 44. Wildcard Matching

Given an input string (s) and a pattern (p), implement wildcard pattern matching with support for '?' and '*' where:

- '?' Matches any single character.
- '*' Matches any sequence of characters (including the empty sequence).

The matching should cover the **entire** input string (not partial).

Example 1:

Input: s = "aa", p = "a"

Output: false

Explanation: "a" does not match the entire string "aa".

Example 2:

Input: s = "aa", p = "*"

Output: true

Explanation: '*' matches any sequence.

Example 3:

Input: s = "cb", p = "?a"

Output: false

Explanation: '?' matches 'c', but the second letter is 'a', which does not match 'b'.

```

class Solution {
    public static boolean isMatch(String s, String p) {
        int m = s.length();
        int n = p.length();
        boolean dp[][] = new boolean[m + 1][n + 1];
        dp[0][0] = true;
        for (int j = 1; j <= n; j++) {
            if (p.charAt(j - 1) == '*') {
                dp[0][j] = dp[0][j - 1];
            }
        }
        for (int i = 1; i < m + 1; i++) {
            for (int j = 1; j < n + 1; j++) {
                if (p.charAt(j - 1) == '?' || p.charAt(j - 1) == s.charAt(i - 1)) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else if (p.charAt(j - 1) == '*') {
                    dp[i][j] = dp[i - 1][j] || dp[i][j - 1];
                }
            }
        }
        System.out.println(Arrays.toString(dp));
        return dp[m][n];
    }
}

```

3. 55. Jump Game

You are given an integer array `nums`. You are initially positioned at the array's first index, and each element in the array represents your maximum jump length at that position.

Return `true` if you can reach the last index, or `false` otherwise.

Example 1:

`Input: nums = [2,3,1,1,4]`
`Output: true`
`Explanation: Jump 1 step from index 0 to 1, then 3 steps to the last index.`

Example 2:

`Input: nums = [3,2,1,0,4]`
`Output: false`
`Explanation: You will always arrive at index 3 no matter what. Its maximum jump length is 0, which makes it impossible to reach the last index.`

Constraints:

- $1 \leq \text{nums.length} \leq 10^4$
- $0 \leq \text{nums}[i] \leq 10^5$

```

class Solution {
    public boolean canJump(int[] nums) {
        int n = nums.length;
        boolean[] dp = new boolean[n];
        dp[0] = true; // You are always at the start

        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                // If j is reachable and we can jump from j to i
                if (dp[j] && j + nums[j] >= i) {
                    dp[i] = true;
                    break; // No need to check further
                }
            }
        }

        return dp[n - 1]; // Can we reach the last index?
    }
}

```

4.45. Jump Game II

You are given a 0-indexed array of integers `nums` of length `n`. You are initially positioned at `nums[0]`.

Each element `nums[i]` represents the maximum length of a forward jump from index `i`. In other words, if you are at `nums[i]`, you can jump to any `nums[i + j]` where:

- $0 \leq j \leq \text{nums}[i]$ and
- $i + j < n$

Return the minimum number of jumps to reach `nums[n - 1]`. The test cases are generated such that you can reach `nums[n - 1]`.

Example 1:

Input: `nums = [2,3,1,1,4]`
Output: 2
Explanation: The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

Example 2:

Input: `nums = [2,3,0,1,4]`
Output: 2

```

class Solution {
    public static int jump(int[] nums) {
        int n = nums.length;
        int[] dp = new int[n + 1];
        Arrays.fill(dp, Integer.MAX_VALUE);
        dp[0] = 0;
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (j + nums[j] >= i) {
                    dp[i] = Math.min(dp[i], dp[j] + 1);
                }
            }
        }
        System.out.println(Arrays.toString(dp));
        return dp[n - 1];
    }
}

```

`dp[i] = Math.min(dp[i], dp[j] + 1);`

is saying:

- To reach index `i`, if you can jump from index `j` to `i` (i.e., $j + \text{nums}[j] \geq i$), then the number of jumps needed to reach `i` could be `dp[j] + 1`.
- The `+1` represents the jump from index `j` to index `i`.

So why `+1`?

Because:

- `dp[j]` is the minimum number of jumps required to reach index `j`.
- If from `j` you can jump to `i`, then reaching `i` would take one more jump — hence `dp[j] + 1`.

The `Math.min(...)` part ensures we keep the minimum number of jumps among all possible previous `j` indices that can jump to `i`.

Jump Game 3

Given an array of non-negative integers `arr`, you are initially positioned at `start` index of the array. When you are at index `i`, you can jump to `i + arr[i]` or `i - arr[i]`, check if you can reach **any** index with value 0.

Notice that you can not jump outside of the array at any time.

Example 1:

```
Input: arr = [4,2,3,0,3,1,2], start = 5
Output: true
Explanation:
All possible ways to reach at index 3 with value 0 are:
index 5 -> index 4 -> index 1 -> index 3
index 5 -> index 6 -> index 4 -> index 1 -> index 3
```

Example 2:

```
Input: arr = [4,2,3,0,3,1,2], start = 0
Output: true
Explanation:
One possible way to reach at index 3 with value 0 is:
index 0 -> index 4 -> index 1 -> index 3
```

Example 3:

```
Input: arr = [3,0,2,1,2], start = 2
Output: false
Explanation: There is no way to reach at index 1 with value 0.
```

```
class Solution {
    public boolean canReach(int[] arr, int start) {
        return dfs(arr, start, new boolean[arr.length]);
    }

    private boolean dfs(int[] arr, int i, boolean[] visited) {
        if (i < 0 || i >= arr.length || visited[i]) {
            return false;
        }
        if (arr[i] == 0) {
            return true;
        }
        visited[i] = true;

        return dfs(arr, i + arr[i], visited) || dfs(arr, i - arr[i], visited);
    }
}
```

Jump Game 4

Given an array of integers `arr`, you are initially positioned at the first index of the array.

In one step you can jump from index `i` to index:

- `i + 1` where: `i + 1 < arr.length`.
- `i - 1` where: `i - 1 >= 0`.
- `j` where: `arr[i] == arr[j]` and `i != j`.

Return the minimum number of steps to reach the last index of the array.

Notice that you can not jump outside of the array at any time.

Example 1:

```
Input: arr = [100,-23,-23,404,100,23,23,23,3,404]
Output: 3
Explanation: You need three jumps from index 0 --> 4 --> 3 --> 9. Note that index 9 is the
last index of the array.
```

Example 2:

```
Input: arr = [7]
Output: 0
Explanation: Start index is the last index. You do not need to jump.
```

Example 3:

```
Input: arr = [7,6,9,6,9,6,9,7]
Output: 1
Explanation: You can jump directly from index 0 to index 7 which is last index of the array.
```

```
class Solution {
    public int minJumps(int[] arr) {
        int n = arr.length;
        if (n == 1) return 0;

        Map<Integer, List<Integer>> graph = new HashMap<>();
        for (int i = 0; i < n; i++) {
            graph.computeIfAbsent(arr[i], x -> new ArrayList<>()).add(i);
        }
        boolean[] visited = new boolean[n];
        Queue<Integer> queue = new LinkedList<>();
        queue.offer(0);
        visited[0] = true;
        int steps = 0;

        while (!queue.isEmpty()) {
            int size = queue.size();
            while (size-- > 0) {
                int i = queue.poll();
                if (i == n - 1) return steps;
                // Jump to i - 1
                if (i - 1 >= 0 && !visited[i - 1]) {
                    visited[i - 1] = true;
                    queue.offer(i - 1);
                }
                // Jump to i + 1
                if (i + 1 < n && !visited[i + 1]) {
                    visited[i + 1] = true;
                    queue.offer(i + 1);
                }
                // Jump to all indices with same value
                if (graph.containsKey(arr[i])) {
                    for (int j : graph.get(arr[i])) {
                        if (!visited[j]) {
                            visited[j] = true;
                            queue.offer(j);
                        }
                    }
                }
                // Remove the list to avoid unnecessary future lookups
                graph.remove(arr[i]);
            }
            steps++;
        }
        return -1;
    }
}
```

Jump Game 5

Given an array of integers arr and an integer d . In one step you can jump from index i to index:

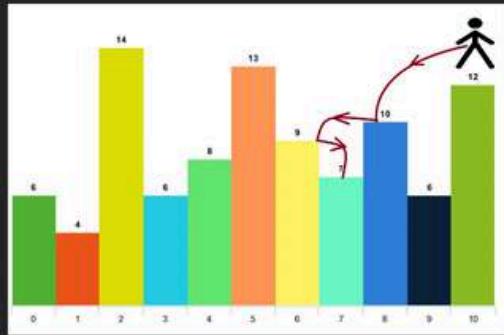
- $i + x$ where: $i + x < \text{arr.length}$ and $0 < x \leq d$.
- $i - x$ where: $i - x \geq 0$ and $0 < x \leq d$.

In addition, you can only jump from index i to index j if $\text{arr}[i] > \text{arr}[j]$ and $\text{arr}[i] > \text{arr}[k]$ for all indices k between i and j (More formally $\min(i, j) < k < \max(i, j)$).

You can choose any index of the array and start jumping. Return the maximum number of indices you can visit.

Notice that you can not jump outside of the array at any time.

Example 1:



Input: $\text{arr} = [6, 4, 14, 6, 8, 13, 9, 7, 10, 6, 12]$, $d = 2$

Output: 4

Explanation: You can start at index 10. You can jump $10 \rightarrow 8 \rightarrow 6 \rightarrow 7$ as shown. Note that if you start at index 6 you can only jump to index 7. You cannot jump to index 5 because $13 > 9$. You cannot jump to index 4 because index 9 is between index 4 and 6 and $13 > 9$.

Similarly You cannot jump from index 3 to index 2 or index 1.

Example 2:

Input: $\text{arr} = [3, 3, 3, 3, 3]$, $d = 3$

Output: 1

Explanation: You can start at any index. You always cannot jump to any index.

Example 3:

Input: $\text{arr} = [7, 6, 5, 4, 3, 2, 1]$, $d = 1$

Output: 7

Explanation: Start at index 0. You can visit all the indicies.

Constraints:

- $1 \leq \text{arr.length} \leq 1000$
- $1 \leq \text{arr}[i] \leq 10^5$
- $1 \leq d \leq \text{arr.length}$

```
public class Solution {
    public int maxJumps(int[] arr, int d) {
        int n = arr.length;
        int maxJumpDistance = d;
        int[] memo = new int[n];
        int maxJumps = 0;

        for (int i = 0; i < n; i++) {
            maxJumps = Math.max(maxJumps, dfs(i, arr, maxJumpDistance, memo));
        }

        return maxJumps;
    }

    private int dfs(int i, int[] arr, int maxJumpDistance, int[] memo) {
        if (memo[i] != 0) return memo[i];

        int max = 1; // count self

        // Explore to the left
        for (int j = i - 1; j >= Math.max(0, i - maxJumpDistance); j--) {
            if (arr[j] >= arr[i]) break;
            max = Math.max(max, 1 + dfs(j, arr, maxJumpDistance, memo));
        }

        // Explore to the right
        for (int j = i + 1; j <= Math.min(arr.length - 1, i + maxJumpDistance); j++) {
            if (arr[j] >= arr[i]) break;
            max = Math.max(max, 1 + dfs(j, arr, maxJumpDistance, memo));
        }

        return memo[i] = max;
    }
}
```

Jump Game 6

You are given a **0-indexed** integer array `nums` and an integer `k`.

You are initially standing at index `0`. In one move, you can jump at most `k` steps forward without going outside the boundaries of the array. That is, you can jump from index `i` to any index in the range `[i + 1, min(n - 1, i + k)]` **inclusive**.

You want to reach the last index of the array (index `n - 1`). Your **score** is the **sum** of all `nums[j]` for each index `j` you visited in the array.

Return *the maximum score* you can get.

Example 1:

Input: `nums = [1,-1,-2,4,-7,3], k = 2`

Output: `7`

Explanation: You can choose your jumps forming the subsequence `[1,-1,4,3]` (underlined above). The sum is `7`.

Example 2:

Input: `nums = [10,-5,-2,4,0,3], k = 3`

Output: `17`

Explanation: You can choose your jumps forming the subsequence `[10,4,3]` (underlined above). The sum is `17`.

Example 3:

Input: `nums = [1,-5,-20,4,-1,3,-6,-3], k = 2`

Output: `0`

```
import java.util.*;

public class Solution {
    public int maxResult(int[] scoreArray, int maxJumpDistance) {
        int n = scoreArray.length;
        Deque<Integer> deque = new ArrayDeque<>();
        int[] dp = new int[n]; // maxScoreAtIndex

        dp[0] = scoreArray[0];
        deque.offer(0);

        for (int i = 1; i < n; i++) {
            // Remove indices out of the current window
            while (!deque.isEmpty() && deque.peek() < i - maxJumpDistance) {
                deque.poll();
            }

            // Get max score from deque's front
            dp[i] = scoreArray[i] + dp[deque.peek()];

            // Maintain decreasing order in deque
            while (!deque.isEmpty() && dp[i] >= dp[deque.peekLast()]) {
                deque.pollLast();
            }

            deque.offer(i);
        }

        return dp[n - 1];
    }
}
```

Jump Game 7

You are given a **0-indexed** binary string `s` and two integers `minJump` and `maxJump`. In the beginning, you are standing at index `0`, which is equal to `'0'`. You can move from index `i` to index `j` if the following conditions are fulfilled:

- `i + minJump <= j <= min(i + maxJump, s.length - 1)`, and
- `s[j] == '0'`.

Return `true` if you can reach index `s.length - 1` in `s`, or `false` otherwise.

Example 1:

Input: `s = "011010"`, `minJump = 2`, `maxJump = 3`

Output: `true`

Explanation:

In the first step, move from index `0` to index `3`.

In the second step, move from index `3` to index `5`.

Example 2:

Input: `s = "01101110"`, `minJump = 2`, `maxJump = 3`

Output: `false`

Constraints:

- `2 <= s.length <= 10^5`
- `s[i]` is either `'0'` or `'1'`.
- `s[0] == '0'`
- `1 <= minJump <= maxJump < s.length`

```
public class Solution {  
    public boolean canReach(String binaryString, int minJump, int maxJump) {  
        int stringLength = binaryString.length();  
        boolean[] isReachable = new boolean[stringLength];  
        isReachable[0] = true;  
  
        int reachableCountInWindow = 0;  
        int left = 1; // Start of sliding window  
        int right = 1; // End of current index  
  
        while (right < stringLength) {  
            // Expand left bound of the window (adds reachable count)  
            if (right - minJump >= 0 && isReachable[right - minJump]) {  
                reachableCountInWindow++;  
            }  
  
            // Shrink right bound of the window (removes outdated reachable count)  
            if (right - maxJump - 1 >= 0 && isReachable[right - maxJump - 1]) {  
                reachableCountInWindow--;  
            }  
  
            // Determine if current index is reachable  
            isReachable[right] = binaryString.charAt(right) == '0' && reachableCountInWindow > 0;  
  
            right++;  
        }  
  
        return isReachable[stringLength - 1];  
    }  
}
```

Jump Game 8

You are given a 0-indexed integer array `nums` of length `n`. You are initially standing at index `0`. You can jump from index `i` to index `j` where `i < j` if:

- `nums[i] <= nums[j]` and `nums[k] < nums[i]` for all indexes `k` in the range `i < k < j`, or
- `nums[i] >= nums[j]` and `nums[k] >= nums[i]` for all indexes `k` in the range `i < k < j`.

You are also given an integer array `costs` of length `n` where `costs[i]` denotes the cost of jumping to index `i`.

Return the **minimum cost** to jump to the index `n - 1`.

Example 1:

```
Input: nums = [3,2,4,4,1], costs = [3,7,6,4,2]
Output: 8
Explanation: You start at index 0.
- Jump to index 2 with a cost of costs[2] = 6.
- Jump to index 4 with a cost of costs[4] = 2.
The total cost is 8. It can be proven that 8 is the minimum cost needed.
Two other possible paths are from index 0 -> 1 -> 4 and index 0 -> 2 -> 3 -> 4.
These have a total cost of 9 and 12, respectively.
```

Example 2:

```
Input: nums = [0,1,2], costs = [1,1,1]
Output: 2
Explanation: Start at index 0.
- Jump to index 1 with a cost of costs[1] = 1.
- Jump to index 2 with a cost of costs[2] = 1.
The total cost is 2. Note that you cannot jump directly from index 0 to index 2 because nums[0] <= nums[1].
```

Constraints:

- `n == nums.length == costs.length`
- `1 <= n <= 105`
- `0 <= nums[i], costs[i] <= 105`

```
public long minCost(int[] nums, int[] costs) {
    int n = nums.length;
    List<Integer>[] graph = new List[n];
    for (int i = 0; i < n; i++) {
        graph[i] = new ArrayList<>();
    }
    // Monotonic decreasing stack for condition 1
    Deque<Integer> decreasingStack = new ArrayDeque<>();
    for (int i = n - 1; i >= 0; i--) {
        while (!decreasingStack.isEmpty() && nums[decreasingStack.peek()] < nums[i]) {
            decreasingStack.pop();
        }
        if (!decreasingStack.isEmpty()) {
            graph[i].add(decreasingStack.peek());
        }
        decreasingStack.push(i);
    }
    // Monotonic increasing stack for condition 2
    Deque<Integer> increasingStack = new ArrayDeque<>();
    for (int i = n - 1; i >= 0; i--) {
        while (!increasingStack.isEmpty() && nums[increasingStack.peek()] >= nums[i]) {
            increasingStack.pop();
        }
        if (!increasingStack.isEmpty()) {
            graph[i].add(increasingStack.peek());
        }
        increasingStack.push(i);
    }
    // Dynamic programming to find minimum cost
    long[] dp = new long[n];
    Arrays.fill(dp, Long.MAX_VALUE);
    dp[0] = 0;
    for (int i = 0; i < n; i++) {
        for (int j : graph[i]) {
            dp[j] = Math.min(dp[j], dp[i] + costs[j]);
        }
    }
    return dp[n - 1];
}

public static void main(String[] args) {
    System.out.println(new Solution().minCost(new int[]{0, 1, 2}, new int[]{1, 1, 1}));
}
```

💡 Approach

To efficiently solve this problem, we can use a combination of **monotonic stacks** and **dynamic programming**:

1. Monotonic Stacks:

- Use two stacks to identify valid jumps:
 - A decreasing stack to find the next greater or equal elements for condition 1.
 - An increasing stack to find the next smaller elements for condition 2.

2. Dynamic Programming:

- Initialize a DP array `dp`, where `dp[i]` represents the minimum cost to reach index `i`.
- Set `dp[0] = 0` since we start at index 0 with no cost.
- For each index, update the `dp` values of reachable indices based on the identified valid jumps and their associated costs.

5. 53. Maximum Subarray

Given an integer array `nums`, find the subarray with the largest sum, and return *its sum*.

Example 1:

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

Example 2:

```
Input: nums = [1]
Output: 1
Explanation: The subarray [1] has the largest sum 1.
```

Example 3:

```
Input: nums = [5,4,-1,7,8]
Output: 23
Explanation: The subarray [5,4,-1,7,8] has the largest sum 23.
```

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

Follow up: If you have figured out the $O(n)$ solution, try coding another solution using the divide and conquer approach, which is more subtle.

```
public class Solution {
    // dp[i]: Maximum subarray sum ending at index i
    public static int maxSubArray(int[] a) {
        int dp[] = new int[a.length];
        dp[0] = a[0];
        for (int i = 1; i < a.length; i++) {
            dp[i] = Math.max(dp[i - 1] + a[i], a[i]);
        }
        System.out.println(Arrays.toString(dp));
        return Arrays.stream(dp).max().getAsInt();
    }
}
```

5. 300. Longest Increasing Subsequence

300. Longest Increasing Subsequence

Medium

Given an integer array `nums`, return *the length of the longest strictly increasing subsequence*.

Example 1:

```
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.
```

Example 2:

```
Input: nums = [0,1,0,3,2,3]
Output: 4
```

Example 3:

```
Input: nums = [7,7,7,7,7,7]
Output: 1
```

Constraints:

- $1 \leq \text{nums.length} \leq 2500$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

Follow up: Can you come up with an algorithm that runs in $O(n \log(n))$ time complexity?

```

public class Solution {

    private static int lengthOfLIS(int[] nums) {
        int n = nums.length;
        int[] dp = new int[n];
        Arrays.fill(dp, val: 1);
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (nums[i] > nums[j] && dp[i] < dp[j] + 1) {
                    dp[i] = dp[j] + 1;
                }
            }
        }
        return Arrays.stream(dp).max().getAsInt();
    }

    public static void main(String[] args) {
        System.out.println(lengthOfLIS(new int[]{10, 9, 2, 5, 3, 7, 101, 18})); // 4
    }
}

```

Iterate through the array with two loops: i from 1 to n, and j from 0 to i - 1.

If $\text{nums}[i] > \text{nums}[j]$ (i.e., the sequence is increasing) and the current longest increasing subsequence length ending at i is less than $\text{dp}[j] + 1$, then update $\text{dp}[i]$ to $\text{dp}[j] + 1$.

After processing all elements, the maximum value in the dp array will be the length of the longest increasing subsequence.

n.log n complexity

```

public class Solution {

    public static int lcs(int[] nums) {
        List<Integer> tail = new ArrayList<>();
        // tail[i] represents the smallest possible tail of all increasing subsequences of length i + 1.

        for (int num : nums) {
            int index = Collections.binarySearch(tail, num);
            if (index < 0) {
                index = -(index + 1);
            }

            if (index == tail.size()) {
                tail.add(num); // New longest subsequence
            } else {
                tail.set(index, num); // Replace to maintain smallest possible tail
            }
        }

        return tail.size(); // Length of LIS: 2, 3, 7, 18
    }

    public static void main(String[] args) {
        System.out.println(lcs(new int[]{
            10, 9, 2, 5, 3, 7, 101, 18
        })); // 4
    }
}

```

If next greater element is found, add it to the list or else replace the element in the array.

```

Start: sub = []
Step 1: num = 10
  • Binary search → pos = 0
  • pos == sub.size() → sub.add(10)
  • sub = [10]

Step 2: num = 9
  • Binary search → find first >= 9 → index 0 (10)
  • Replace: sub[0] = 9
  • sub = [9]

Step 3: num = 2
  • Binary search → first >= 2 → index 0 (9)
  • Replace: sub[0] = 2
  • sub = [2]

Step 4: num = 5
  • Binary search → first >= 5 → index 1 (none)
  • sub.add(5)
  • sub = [2, 5]

Step 5: num = 3
  • Binary search → first >= 3 → index 1 (5)
  • Replace: sub[1] = 3
  • sub = [2, 3]

Step 6: num = 7
  • Binary search → first >= 7 → index 2 (none)
  • sub.add(7)
  • sub = [2, 3, 7]

Step 7: num = 101
  • Binary search → first >= 101 → index 3 (none)
  • sub.add(101)
  • sub = [2, 3, 7, 101]

Step 8: num = 18
  • Binary search → first >= 18 → index 3 (101)
  • Replace: sub[3] = 18
  • sub = [2, 3, 7, 18]

```

The Behavior of `Collections.binarySearch()`

`Collections.binarySearch(list, key)` returns:

- $\geq 0 \rightarrow$ index of the key if it exists
- $< 0 \rightarrow$ insertion point if it doesn't exist, as $-(\text{insertion point}) - 1$

So, if the element is not found, the return value is:

```

java                                     ⌂ Copy ⌂ Edit
return -(insertion_index) - 1;

```

Example: `Collections.binarySearch([2], 5)`

- The list is [2]
- We're searching for 5
- 5 is greater than 2, so the insertion point is index 1
- According to the formula:

```

java                                     ⌂ Copy ⌂ Edit
return -(1) - 1 = -2;

```

So it returns -2.

6. Maximum Chain Length

```
public class Solution {

    private static int maximumChainLength(int[][] nums) {
        int n = nums.length;
        int[] dp = new int[n];
        Arrays.fill(dp, val: 1);
        Arrays.sort(nums, Comparator.comparingInt(a -> a[0]));
        System.out.println(Arrays.deepToString(nums)); // [[5, 24], [27, 40], [39, 60], [50, 90]]
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (nums[i][0] > nums[j][1] && dp[i] < dp[j] + 1) {
                    dp[i] = dp[j] + 1;
                }
            }
        }
        return Arrays.stream(dp).max().getAsInt(); // [5, 24], [27, 40], [50, 90]
    }

    public static void main(String[] args) {
        System.out.println(maximumChainLength(new int[][]{
            {5, 24},
            {39, 60},
            {27, 40},
            {50, 90}
        })); // 4
    }
}
```

7. Russian Doll Envelopes

You are given a 2D array of integers `envelopes` where `envelopes[i] = [wi, hi]` represents the width and the height of an envelope.

One envelope can fit into another if and only if both the width and height of one envelope are greater than the other envelope's width and height.

Return the maximum number of envelopes you can Russian doll (i.e., put one inside the other).

Note: You cannot rotate an envelope.

Example 1:

Input: envelopes = [[5,4],[6,4],[6,7],[2,3]]
Output: 3
Explanation: The maximum number of envelopes you can Russian doll is 3 ([2,3] => [5,4] => [6,7]).

Example 2:

Input: envelopes = [[1,1],[1,1],[1,1]]
Output: 1

Constraints:

- $1 \leqslant \text{envelopes.length} \leqslant 10^5$
- $\text{envelopes}[i].length == 2$
- $1 \leqslant w_i, h_i \leqslant 10^5$

```

public class Solution {

    public static int maxEnvelopes(int[][] envelopes) {
        int n = envelopes.length;
        int[] dp = new int[n];
        Arrays.fill(dp, val: 1);
        Arrays.sort(envelopes, (a, b) -> {
            if (a[0] != b[0]) {
                return a[0] - b[0];
            } else {
                return a[1] - b[1];
            }
        });

        System.out.println(Arrays.deepToString(envelopes)); // [[2, 3], [5, 4], [6, 4], [6, 7]]
        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (envelopes[i][0] > envelopes[j][0] && envelopes[i][1] > envelopes[j][1]
                    && dp[i] < dp[j] + 1) {
                    dp[i] = dp[j] + 1;
                }
            }
        }
        return Arrays.stream(dp).max().getAsInt(); // [2, 3] < [5, 4] < [6, 7]
    }

    public static void main(String[] args) {
        System.out.println(maxEnvelopes(new int[][]{
            {5, 4},
            {6, 4},
            {6, 7},
            {2, 3}
        })); // 3
    }
}

```

```

public class Solution {
    public int maxEnvelopes(int[][] envelopes) {
        // Step 1: Sort by width asc, then height desc
        Arrays.sort(envelopes, (a, b) -> {
            if (a[0] != b[0]) return a[0] - b[0];
            return b[1] - a[1]; // reverse height sort
        });

        // Step 2: Extract heights
        int[] heights = new int[envelopes.length];
        for (int i = 0; i < envelopes.length; i++) {
            heights[i] = envelopes[i][1];
        }

        // Step 3: Perform LIS on heights using manual binary search
        List<Integer> sub = new ArrayList<>();
        for (int height : heights) {
            int left = 0, right = sub.size() - 1;
            int pos = sub.size();

            while (left <= right) {
                int mid = left + (right - left) / 2;
                if (sub.get(mid) >= height) {
                    pos = mid;
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            }

            if (pos == sub.size()) {
                sub.add(height);
            } else {
                sub.set(pos, height);
            }
        }

        return sub.size();
    }
}

```

8. 72. Edit Distance

Given two strings `word1` and `word2`, return the minimum number of operations required to convert `word1` to `word2`.

You have the following three operations permitted on a word:

- Insert a character
- Delete a character
- Replace a character

Example 1:

```
Input: word1 = "horse", word2 = "ros"
Output: 3
Explanation:
horse -> rorse (replace 'h' with 'r')
rorse -> rose (remove 'r')
rose -> ros (remove 'e')
```

Example 2:

```
Input: word1 = "intention", word2 = "execution"
Output: 5
Explanation:
intention -> inention (remove 't')
inention -> enention (replace 'i' with 'e')
enention -> exention (replace 'n' with 'x')
exention -> exection (replace 'n' with 'c')
exection -> execution (insert 'u')
```

```
public class Solution {

    // dp[i][j] represents the minimum number of operations (edits) required to
    // transform the substring word1[0...i-1] to the substring word2[0...j-1].
    public static int minDistance(String word1, String word2) {
        int m = word1.length();
        int n = word2.length();
        int[][] dp = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++) {
            dp[i][0] = i;
        }

        for (int j = 1; j <= n; j++) {
            dp[0][j] = j;
        }

        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (word1.charAt(i - 1) == word2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1];
                } else {
                    dp[i][j] = Math.min(dp[i - 1][j - 1], Math.min(dp[i - 1][j], dp[i][j - 1])) + 1;
                }
            }
        }
        return dp[m][n];
    }

    public static void main(String[] args) {
        String word1 = "horse";
        String word2 = "ros";
        System.out.println(minDistance(word1, word2)); // 3
    }
}
```

9.85. Maximal Rectangle

Given a `rows x cols` binary matrix filled with `0`'s and `1`'s, find the largest rectangle containing only `1`'s and return its area.

Example 1:

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

Input: `matrix = [[1,"0","1","0","0"],[1,"0","1","1","1"],[1,"1","1","1","1"],[1,"0","0","1","0"]]`

Output: 6

Explanation: The maximal rectangle is shown in the above picture.

Example 2:

Input: `matrix = [[0]]`

Output: 0

Example 3:

Input: `matrix = [[1]]`

Output: 1

```
class Solution {
    // dp[i][j] represents the height of consecutive '1's ending at the cell (i, j) in the binary matrix
    public int maximalRectangle(char[][] matrix) {
        if (matrix.length == 0) return 0;
        int n = matrix.length;
        int m = matrix[0].length;
        int[][] dp = new int[n][m];
        int maxArea = 0;
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < m; j++) {
                if (i == 0)
                    dp[i][j] = matrix[i][j] == '1' ? 1 : 0;
                else
                    dp[i][j] = matrix[i][j] == '1' ? (dp[i - 1][j] + 1) : 0;
                int min = dp[i][j];
                for (int k = j; k >= 0; k--) {
                    if (min == 0) break;
                    if (dp[i][k] < min) min = dp[i][k];
                    maxArea = Math.max(maxArea, min * (j - k + 1));
                }
            }
        }
        System.out.println(Arrays.deepToString(dp).replaceAll("[,]", "],\n"));
        return maxArea;
    }
}
```

10. 1143. Longest Common Subsequence

Given two strings `text1` and `text2`, return *the length of their longest common subsequence*. If there is no common subsequence, return `0`.

A subsequence of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

- For example, "ace" is a subsequence of "abcde".

A common subsequence of two strings is a subsequence that is common to both strings.

Example 1:

```
Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.
```

Example 2:

```
Input: text1 = "abc", text2 = "abc"
Output: 3
Explanation: The longest common subsequence is "abc" and its length is 3.
```

Example 3:

```
Input: text1 = "abc", text2 = "def"
Output: 0
Explanation: There is no such common subsequence, so the result is 0.
```

Constraints:

- $1 \leq \text{text1.length}, \text{text2.length} \leq 1000$
- `text1` and `text2` consist of only lowercase English characters.

```
public class Solution {
    public int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length();
        int n = text2.length();

        // dp[i][j] = LCS of text1[0..i-1] and text2[0..j-1]
        int[][] dp = new int[m + 1][n + 1];

        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (text1.charAt(i) == text2.charAt(j)) {
                    dp[i + 1][j + 1] = 1 + dp[i][j];
                } else {
                    dp[i + 1][j + 1] = Math.max(dp[i][j + 1], dp[i + 1][j]);
                }
            }
        }

        return dp[m][n];
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println(sol.longestCommonSubsequence("abcde", "ace")); // Output: 3
    }
}
```

11. 139. Word Break

139. Word Break

Medium

Given a string `s` and a dictionary of strings `wordDict`, return `true` if `s` can be segmented into a space-separated sequence of one or more dictionary words.

Note that the same word in the dictionary may be reused multiple times in the segmentation.

Example 1:

```
Input: s = "leetcode", wordDict = ["leet","code"]
Output: true
Explanation: Return true because "leetcode" can be segmented as "leet code".
```

Example 2:

```
Input: s = "applepenapple", wordDict = ["apple","pen"]
Output: true
Explanation: Return true because "applepenapple" can be segmented as "apple pen apple".
Note that you are allowed to reuse a dictionary word.
```

Example 3:

```
Input: s = "catsandog", wordDict = ["cats","dog","sand","and","cat"]
Output: false
```

Constraints:

- `1 <= s.length <= 300`
- `1 <= wordDict.length <= 1000`
- `1 <= wordDict[i].length <= 20`
- `s` and `wordDict[i]` consist of only lowercase English letters.
- All the strings of `wordDict` are unique.

```
import java.util.Arrays;
import java.util.HashSet;
import java.util.Set;

class Solution {
    // dp[i] represents whether the substring of s from index 0 to index i-1 can be
    // segmented into words from the given wordDict.
    static boolean wordBreak(String s, List<String> wordDict) {
        int n = s.length();
        Set<String> wordSet = new HashSet<>(wordDict);
        boolean[] dp = new boolean[n + 1];
        dp[0] = true;
        for (int i = 1; i <= n; ++i)
            for (int j = 0; j < i; ++j)
                if (dp[j] && wordSet.contains(s.substring(j, i)))
                    dp[i] = true;
                    break;
        System.out.println(Arrays.toString(dp));
        // [true, false, false, false, true, false, false, false, true]
        return dp[n];
    }
}
```

12. 84. Minimum Path Sum

Given a $m \times n$ grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

Example 1:

1	3	1
1	5	1
4	2	1

Input: grid = [[1,3,1],[1,5,1],[4,2,1]]

Output: 7

Explanation: Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.

Example 2:

Input: grid = [[1,2,3],[4,5,6]]

Output: 12

```
class Solution {
    // dp[i][j] represents the minimum path sum to reach the cell at row i and column j in a 2D grid
    public int minPathSum(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        int[][] dp = grid;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (i > 0 && j > 0) {
                    dp[i][j] += Math.min(dp[i - 1][j], dp[i][j - 1]);
                } else if (i > 0) {
                    dp[i][0] += dp[i - 1][0];
                } else if (j > 0) {
                    dp[0][j] += dp[0][j - 1];
                }
            }
        }

        System.out.println(Arrays.deepToString(dp));
        return dp[m - 1][n - 1];
    }
}
```

13. Longest Palindromic Subsequence

```
public class LongestPalindromicSubsequence {  
  
    public static int longestPalindromeSubseq(String s) {  
        int n = s.length();  
        int[][] dp = new int[n][n];  
  
        // Base case: Single letters are palindromes of length 1  
        for (int i = 0; i < n; i++) {  
            dp[i][i] = 1;  
        }  
  
        // Fill dp array from bottom to top, right to left  
        for (int len = 2; len <= n; len++) {  
            for (int i = 0; i <= n - len; i++) {  
                int j = i + len - 1;  
  
                if (s.charAt(i) == s.charAt(j)) {  
                    dp[i][j] = 2 + dp[i + 1][j - 1];  
                } else {  
                    dp[i][j] = Math.max(dp[i + 1][j], dp[i][j - 1]);  
                }  
            }  
        }  
  
        return dp[0][n - 1]; // LPS for the entire string  
    }  
  
    public static void main(String[] args) {  
        String s = "bbbab";  
        System.out.println("Longest Palindromic Subsequence Length: " + longestPalindromeSubseq(s)); // Output: 4  
    }  
}
```

Final dp Table:

i\j	0	1	2	3	4
0	1	2	3	3	4
1		1	2	2	3
2			1	1	3
3				1	1
4					1

if string is same, track for the characters around i and j and include them using $i+1$ and $j-1$ or else

When characters don't match, the longest palindromic subsequence must lie in either $s[i+1\dots j]$ or $s[i\dots j-1]$. So, we take the maximum of those two.

Explanation of a few cells:

- $dp[0][4] = 4$: Longest palindromic subsequence is "bbbb"
- $dp[1][3] = 2$: Either "bb" or "ba" → "b"
- $dp[2][4] = 3$: "bab" or "bbb"

✓ Final Output:

mathematica

Longest Palindromic Subsequence Length: 4

14. Best time to buy and sell stock

You are given an array `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock.

Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

Example 1:

```
Input: prices = [7,1,5,3,6,4]
Output: 5
Explanation: Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.
Note that buying on day 2 and selling on day 1 is not allowed because you must buy before
you sell.
```

Example 2:

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: In this case, no transactions are done and the max profit = 0.
```

Constraints:

- $1 \leq \text{prices.length} \leq 10^5$
- $0 \leq \text{prices}[i] \leq 10^4$

```
class Solution {
    // dp[i] represents the minimum price of the stock among the prices encountered up to index i in the prices array
    // You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock

    public int maxProfit(int[] prices) {
        int res = 0;
        int[] dp = new int[prices.length];
        dp[0] = prices[0];
        for (int i = 1; i < prices.length; i++) {
            dp[i] = Math.min(dp[i - 1], prices[i]);
        }
        for (int i = 0; i < prices.length; i++) {
            res = Math.max(res, prices[i] - dp[i]);
        }
        System.out.println(Arrays.toString(dp));
        return res;
    }
}
```

15. Best time to buy and sell stock II

You are given an integer array `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

On each day, you may decide to buy and/or sell the stock. You can only hold at most one share of the stock at any time. However, you can buy it then immediately sell it on the same day.

Find and return *the maximum profit you can achieve*.

Example 1:

```
Input: prices = [7,1,5,3,6,4]
Output: 7
Explanation: Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.
Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.
Total profit is 4 + 3 = 7.
```

Example 2:

```
Input: prices = [1,2,3,4,5]
Output: 4
Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = 5-1 = 4.
Total profit is 4.
```

Example 3:

```
Input: prices = [7,6,4,3,1]
Output: 0
Explanation: There is no way to make a positive profit, so we never buy the stock to
achieve the maximum profit of 0.
```

Constraints:

- $1 \leq \text{prices.length} \leq 3 * 10^4$
- $0 \leq \text{prices}[i] \leq 10^4$

```
class Solution {

    // dp[i]: Maximum profit attainable on or before day i.

    public static int maxProfit(int[] prices) {
        if (prices == null || prices.length == 0) {
            return 0;
        }

        int n = prices.length;
        int[] dp = new int[n];

        dp[0] = 0;
        for (int i = 1; i < n; i++) {
            int diff = prices[i] - prices[i - 1];
            dp[i] = Math.max(dp[i - 1] + diff, dp[i - 1]);
        }

        return dp[n - 1];
    }
}
```

15. Best time to buy and sell stock III

You are given an array `prices` where `prices[i]` is the price of a given stock on the i^{th} day.

Find the maximum profit you can achieve. You may complete at most two transactions.

Note: You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

Example 1:

Input: `prices = [3,3,5,0,0,3,1,4]`

Output: 6

Explanation: Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = $3 - 0 = 3$. Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = $4 - 1 = 3$.

Example 2:

Input: `prices = [1,2,3,4,5]`

Output: 4

Explanation: Buy on day 1 (price = 1) and sell on day 5 (price = 5), profit = $5 - 1 = 4$.

Note that you cannot buy on day 1, buy on day 2 and sell them later, as you are engaging multiple transactions at the same time. You must sell before buying again.

Example 3:

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

Output: 0

Explanation: In this case, no transaction is done, i.e. max profit = 0.

Constraints:

- $1 \leq \text{prices.length} \leq 10^5$
- $0 \leq \text{prices}[i] \leq 10^5$

```
class Solution {  
    public static int maxProfit(int[] prices) {  
        if (prices == null || prices.length == 0) {  
            return 0;  
        }  
  
        int n = prices.length;  
        int[] leftProfit = new int[n];  
        int[] rightProfit = new int[n];  
  
        // Calculate maximum profit from left to right  
        int minPrice = prices[0];  
        leftProfit[0] = 0;  
        for (int i = 1; i < n; i++) {  
            minPrice = Math.min(minPrice, prices[i]);  
            leftProfit[i] = Math.max(leftProfit[i - 1], prices[i] - minPrice);  
        }  
  
        // Calculate maximum profit from right to left  
        int maxPrice = prices[n - 1];  
        rightProfit[n - 1] = 0;  
        for (int i = n - 2; i >= 0; i--) {  
            maxPrice = Math.max(maxPrice, prices[i]);  
            rightProfit[i] = Math.max(rightProfit[i + 1], maxPrice - prices[i]);  
        }  
  
        int maxProfit = 0;  
        // Find the maximum profit by combining left and right profits  
        for (int i = 0; i < n; i++) {  
            maxProfit = Math.max(maxProfit, leftProfit[i] + rightProfit[i]);  
        }  
  
        return maxProfit;  
    }  
  
    public static void main(String[] args) {  
        int[] prices = { 3, 3, 5, 0, 0, 3, 1, 4 };  
        System.out.println("Maximum profit: " + maxProfit(prices));  
    }  
}
```

Description: Solution

1062. Longest Repeating Substring

Medium

Given a string S , find out the length of the longest repeating substring(s). Return 0 if no repeating substring exists.

Example 1:
 Input: "abba"
 Output: 2
 Explanation: There is no repeating substring.

Example 2:
 Input: "abbaba"
 Output: 3
 Explanation: The longest repeating substrings are "ab" and "ba", each of which occurs twice.

Example 3:
 Input: "abcbabcbabc"
 Output: 3
 Explanation: The longest repeating substring is "abc", which occurs 3 times.

Example 4:
 Input: "aaaa"
 Output: 4
 Explanation: The longest repeating substring is "aaaa", which occurs twice.

Note:

- The string S consists of only lowercase English letters from $'a'$ - $'z'$.
- $0 \leq S.length \leq 1000$

```

class Solution {
    public int longestRepeatingSubstring(String S) {
        int n = S.length();
        int[] dp = new int[n];
        for (int i = 0; i + 1 < n; i++) {
            for (int j = i + 1; j + 1 < n; j++) {
                if (S.charAt(i) == S.charAt(j)) {
                    if (dp[i] > 0) {
                        dp[j] = dp[i] + 1;
                    } else {
                        dp[j] = 1;
                    }
                }
            }
        }
        return dp[n - 1];
    }
}

```

Frog Jump

A frog is crossing a river. The river is divided into some number of units, and at each unit, there may or may not exist a stone. The frog can jump on a stone, but it must not jump into the water.

Given a list of `stones` positions (in units) in sorted ascending order, determine if the frog can cross the river by landing on the last stone. Initially, the frog is on the first stone and assumes the first jump must be 1 unit.

If the frog's last jump was k units, its next jump must be either $k - 1$, k , or $k + 1$ units. The frog can only jump in the forward direction.

Example 1:

Input: stones = [0,1,3,5,6,8,12,17]
 Output: true
 Explanation: The frog can jump to the last stone by jumping 1 unit to the 2nd stone, then 2 units to the 3rd stone, then 2 units to the 4th stone, then 3 units to the 6th stone, 4 units to the 7th stone, and 5 units to the 8th stone.

Example 2:

Input: stones = [0,1,2,3,4,8,9,11]
 Output: false
 Explanation: There is no way to jump to the last stone as the gap between the 5th and 6th stone is too large.

Constraints:

- $2 \leq \text{stones.length} \leq 2000$
- $0 \leq \text{stones}[i] \leq 2^{31} - 1$
- $\text{stones}[0] = 0$
- stones is sorted in a strictly increasing order.

import java.util.*;

```

public class Solution {
    public boolean canCross(int[] stones) {
        int n = stones.length;
        // Early check: second stone must be 1 unit away
        if (stones[1] != 1) return false;

        // Mapping stone position to its index
        Map<Integer, Integer> stoneIndexMap = new HashMap<>();
        for (int i = 0; i < n; i++) {
            stoneIndexMap.put(stones[i], i);
        }

        // dp[i][k] = whether we can reach stone i with a jump of size k
        boolean[][] dp = new boolean[n][n + 1];
        dp[0][0] = true;

        for (int current = 0; current < n; current++) {
            for (int jump = 0; jump <= n; jump++) {
                if (!dp[current][jump]) continue;

                for (int step = jump - 1; step <= jump + 1; step++) {
                    if (step > 0) {
                        int nextPosition = stones[current] + step;
                        if (stoneIndexMap.containsKey(nextPosition)) {
                            int nextIndex = stoneIndexMap.get(nextPosition);
                            dp[nextIndex][step] = true;
                        }
                    }
                }
            }
        }

        // Check if any jump can reach the last stone
        for (int jump = 0; jump <= n; jump++) {
            if (dp[n - 1][jump]) return true;
        }

        return false;
    }
}

```

Matrixes

1.200. Number of Islands

200. Number of Islands

Medium

Given an $m \times n$ 2D binary grid grid which represents a map of '1's (land) and '0's (water), return the number of islands.

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

Example 1:

```
Input:  
grid = [  
  ["1","1","1","1","0"],  
  ["1","1","0","1","0"],  
  ["1","1","0","0","0"],  
  ["0","0","0","0","0"]  
]  
Output: 1
```

Example 2:

```
Input:  
grid = [  
  ["1","1","0","0","0"],  
  ["1","1","0","0","0"],  
  ["0","0","1","0","0"],  
  ["0","0","0","1","1"]  
]  
Output: 3
```

For all rows and columns, perform recursive DFS and mark the cell as visited.

```
private static final int[][] DIRECTIONS = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};
```

```
public int numIslands(char[][] grid) {  
    if (grid == null || grid.length == 0) {  
        return 0;  
    }  
  
    int m = grid.length;  
    int n = grid[0].length;  
    int numIslands = 0;  
  
    for (int i = 0; i < m; i++) {  
        for (int j = 0; j < n; j++) {  
            if (grid[i][j] == '1') {  
                numIslands++;  
                dfs(grid, i, j);  
            }  
        }  
    }  
  
    return numIslands;  
}
```

```
private void dfs(char[][] grid, int i, int j) {  
    int m = grid.length;  
    int n = grid[0].length;  
  
    if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] != '1') {  
        return;  
    }  
    grid[i][j] = '0';  
    for (int[] direction : DIRECTIONS) {  
        int newRow = i + direction[0];  
        int newCol = j + direction[1];  
        dfs(grid, newRow, newCol);  
    }  
}
```

```
public class NumberOfIslandsBFS {  
    public static int numIslands(char[][] grid) {  
        if (grid == null || grid.length == 0) return 0;  
  
        int rows = grid.length, cols = grid[0].length;  
        int islandCount = 0;  
  
        // Directions: right, down, left, up  
        int[][] directions = {{0,1}, {1,0}, {0,-1}, {-1,0}};  
  
        for (int r = 0; r < rows; r++) {  
            for (int c = 0; c < cols; c++) {  
  
                if (grid[r][c] == '1') {  
                    islandCount++;  
                    Queue<int[]> queue = new LinkedList<int[]>();  
                    queue.offer(new int[]{r, c});  
                    grid[r][c] = '0'; // mark as visited  
  
                    while (!queue.isEmpty()) {  
                        int[] curr = queue.poll();  
                        int x = curr[0], y = curr[1];  
  
                        for (int[] dir : directions) {  
                            int nx = x + dir[0];  
                            int ny = y + dir[1];  
  
                            if (nx >= 0 && ny >= 0 && nx < rows && ny < cols && grid[nx][ny] == '1') {  
                                queue.offer(new int[]{nx, ny});  
                                grid[nx][ny] = '0'; // mark as visited  
                            }  
                        }  
                    }  
                }  
            }  
        }  
        return islandCount;  
    }  
  
    public static void main(String[] args) {  
        char[][] grid = {  
            {'1','1','0','0','0'},  
            {'1','1','0','0','0'},  
            {'0','0','1','0','0'},  
            {'0','0','0','1','1'}  
        };  
  
        System.out.println("Number of Islands: " + numIslands(grid)); // Output: 3  
    }  
}
```

If its 1, mark the islands and increment count

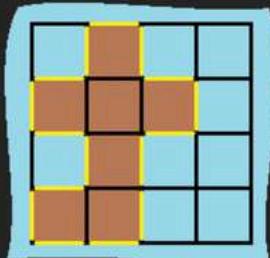
2. 463. Island Perimeter

You are given `row x col grid` representing a map where `grid[i][j] = 1` represents land and `grid[i][j] = 0` represents water.

Grid cells are connected horizontally/vertically (not diagonally). The `grid` is completely surrounded by water, and there is exactly one island (i.e., one or more connected land cells).

The island doesn't have "lakes", meaning the water inside isn't connected to the water around the island. One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.

Example 1:



If its not safe, increment perimeter

Input: `grid = [[0,1,0,0],[1,1,1,0],[0,1,0,0],[1,1,0,0]]`

Output: 16

Explanation: The perimeter is the 16 yellow stripes in the image above.

Example 2:

Input: `grid = [[1]]`

Output: 4

Example 3:

Input: `grid = [[1,0]]`

Output: 4

```
class Solution {
    private final int[][] DIRECTIONS = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    public int islandPerimeter(int[][] grid) {
        if (grid == null || grid.length == 0 || grid[0].length == 0) {
            return 0;
        }
        int m = grid.length;
        int n = grid[0].length;
        int perimeter = 0;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 1) {
                    for (int[] direction : DIRECTIONS) {
                        int x = i + direction[0];
                        int y = j + direction[1];

                        if (x < 0 || x >= m || y < 0 || y >= n || grid[x][y] == 0) {
                            perimeter++;
                        }
                    }
                }
            }
        }

        return perimeter;
    }
}
```

```

public class IslandPerimeterBFS {

    public static int islandPerimeter(int[][] grid) {
        int rows = grid.length, cols = grid[0].length;
        boolean[][] visited = new boolean[rows][cols];
        Queue<int[]> queue = new LinkedList<>();

        int[][] directions = {{0,1}, {1,0}, {0,-1}, {-1,0}};
        boolean found = false;

        // Find the first land cell without using outer loop
        for (int r = 0; r < rows && !found; r++) {
            for (int c = 0; c < cols; c++) {
                if (grid[r][c] == 1) {
                    queue.offer(new int[]{r, c});
                    visited[r][c] = true;
                    found = true;
                    break;
                }
            }
        }

        int perimeter = 0;

        while (!queue.isEmpty()) {
            int[] current = queue.poll();
            int row = current[0], col = current[1];

            for (int[] dir : directions) {
                int newRow = row + dir[0];
                int newCol = col + dir[1];

                // Water or boundary increases perimeter
                if (newRow < 0 || newCol < 0 || newRow >= rows || newCol >= cols || grid[newRow][newCol] == 0) {
                    perimeter++;
                } else if (!visited[newRow][newCol]) {
                    visited[newRow][newCol] = true;
                    queue.offer(new int[]{newRow, newCol});
                }
            }
        }

        return perimeter;
    }

    public static void main(String[] args) {
        int[][] grid = {
            {0,1,0,0},
            {1,1,1,0},
            {0,1,1,0},
            {1,1,0,0}
        };

        System.out.println("Island Perimeter: " + islandPerimeter(grid)); // Output: 16
    }
}

```

If its not safe, increment perimeter.

First the first cell which is 1, mark it as visited. Push it to the queue.

Perform BFS, if the new cell is unsafe increase perimeter else add it to the queue and mark it as visited.

3. 695. Max Area of Island

You are given an $m \times n$ binary matrix `grid`. An island is a group of 1's (representing land) connected 4-directionally (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The area of an island is the number of cells with a value 1 in the island.

Return the maximum area of an island in `grid`. If there is no island, return 0.

Example 1:

0	0	1	0	0	0	0	1	0	0	0	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0
0	1	1	0	1	0	0	0	0	0	0	0	0
0	1	0	0	1	1	0	0	1	0	1	0	0
0	1	0	0	1	1	0	0	1	1	1	0	0
0	0	0	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	0	0	1	1	1	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0

Input: `grid = [[0,0,1,0,0,0,0,1,0,0,0,0,0], [0,0,0,0,0,0,0,1,1,1,0,0,0], [0,1,1,0,1,0,0,0,0,0,0,0,0], [0,1,0,0,1,1,0,0,1,0,1,0,0], [0,1,0,0,1,1,0,0,1,0,0,1,0,0], [0,0,0,0,0,0,0,0,0,1,0,0,0], [0,0,0,0,0,0,0,0,1,1,1,0,0,0], [0,0,0,0,0,0,0,1,1,1,0,0,0,0]]`

Output: 6

Explanation: The answer is not 11, because the island must be connected 4-directionally.

```
public int maxAreaOfIsland(int[][] grid) {
    if (grid == null || grid.length == 0 || grid[0].length == 0) {
        return 0;
    }

    int m = grid.length;
    int n = grid[0].length;
    int maxArea = 0;

    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 1) {
                int area = dfs(grid, i, j);
                maxArea = Math.max(maxArea, area);
            }
        }
    }

    return maxArea;
}

private int dfs(int[][] grid, int i, int j) {
    if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length || grid[i][j] == 0) {
        return 0;
    }

    grid[i][j] = 0;
    int area = 1;

    for (int[] direction : DIRECTIONS) {
        int newI = i + direction[0];
        int newJ = j + direction[1];
        area += dfs(grid, newI, newJ);
    }

    return area;
}
```

Red Colored will be the maxArea,

Add area to the answer if its one and take the max of maxArea and curr

4. 1254. Number of Closed Islands

Given a 2D grid consists of 0s (land) and 1s (water). An island is a maximal 4-directionally connected group of 0s and a closed island is an island totally (all left, top, right, bottom) surrounded by 1s.

Return the number of closed islands.

Example 1:

1	1	1	1	1	1	1	0
1	0	0	0	0	1	1	0
1	0	1	0	1	1	1	0
1	0	0	0	0	1	0	1
1	1	1	1	1	1	1	0

Input: grid = [[1,1,1,1,1,1,0],[1,0,0,0,0,1,1,0],[1,0,1,0,1,1,1,0],[1,0,0,0,1,0,1],[1,1,1,1,1,1,0]]

Output: 2

Explanation:

Islands in gray are closed because they are completely surrounded by water
(group of 1s).

Example 2:

0	0	1	0	0
0	1	0	1	0
0	1	1	1	0

Input: grid = [[0,0,1,0,0],[0,1,0,1,0],[0,1,1,1,0]]

Output: 1

Mark all boundary 0s as 1s, if the cell is 0 increment count and perform dfs, keep on incrementing count if its safe

```
class Solution {
    public int closedIsland(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        int closedIslands = 0;

        // Mark all '0's connected to the boundary as '1's
        for (int i = 0; i < m; i++) {
            if (grid[i][0] == 0) dfs(grid, i, 0);
            if (grid[i][n - 1] == 0) dfs(grid, i, n - 1);
        }
        for (int j = 0; j < n; j++) {
            if (grid[0][j] == 0) dfs(grid, 0, j);
            if (grid[m - 1][j] == 0) dfs(grid, m - 1, j);
        }

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 0) {
                    dfs(grid, i, j);
                    closedIslands++;
                }
            }
        }

        return closedIslands;
    }

    private void dfs(int[][] grid, int i, int j) {
        if (i < 0 || i >= grid.length || j < 0 || j >= grid[0].length || grid[i][j] == 1) {
            return;
        }

        grid[i][j] = 1;
        dfs(grid, i + 1, j);
        dfs(grid, i - 1, j);
        dfs(grid, i, j + 1);
        dfs(grid, i, j - 1);
    }
}
```

5. 1905. Count Sub Islands

You are given two $m \times n$ binary matrices `grid1` and `grid2` containing only 0's (representing water) and 1's (representing land). An island is a group of 1's connected 4-directionally (horizontal or vertical). Any cells outside of the grid are considered water cells.

An island in `grid2` is considered a sub-island if there is an island in `grid1` that contains all the cells that make up this island in `grid2`.

Return the *number of islands in `grid2` that are considered sub-islands*.

Example 1:

1	1	1	0	0
0	1	1	1	1
0	0	0	0	0
1	0	0	0	0
1	1	0	1	1

1	1	1	0	0
0	0	1	1	1
0	1	0	0	0
1	0	1	1	0
0	1	0	1	0

Input: `grid1 = [[1,1,1,0,0],[0,1,1,1,1],[0,0,0,0,0],[1,0,0,0,0],[1,1,0,1,1]]`,
`grid2 = [[1,1,1,0,0],[0,0,1,1,1],[0,1,0,0,0],[1,0,1,1,0],[0,1,0,1,0]]`

Output: 3

Explanation: In the picture above, the grid on the left is `grid1` and the grid on the right is `grid2`.

The 1s colored red in `grid2` are those considered to be part of a sub-island.
There are three sub-islands.

```
class Solution {
    public int countSubIslands(int[][] grid1, int[][] grid2) {
        int m = grid1.length;
        int n = grid1[0].length;
        int count = 0;

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid2[i][j] == 1) {
                    if (isSubIsland(grid1, grid2, i, j)) {
                        count++;
                    }
                }
            }
        }

        return count;
    }

    private boolean isSubIsland(int[][] grid1, int[][] grid2, int i, int j) {
        boolean isSubIsland = true;
        isSubIsland = dfs(grid1, grid2, i, j, isSubIsland);
        return isSubIsland;
    }

    private boolean dfs(int[][] grid1, int[][] grid2, int i, int j, boolean isSubIsland) {
        if (i < 0 || i >= grid1.length || j < 0 || j >= grid1[0].length || grid2[i][j] == 0) {
            return isSubIsland;
        }
        if (grid1[i][j] != 1) {
            isSubIsland = false;
        }
        grid2[i][j] = 0;
        isSubIsland = dfs(grid1, grid2, i + 1, j, isSubIsland);
        isSubIsland = dfs(grid1, grid2, i - 1, j, isSubIsland);
        isSubIsland = dfs(grid1, grid2, i, j + 1, isSubIsland);
        isSubIsland = dfs(grid1, grid2, i, j - 1, isSubIsland);
        return isSubIsland;
    }
}
```

If the cell is 1, check for neighbors by marking them as 0

6. 827. Making A Large Island

6. 994. Rotting Oranges

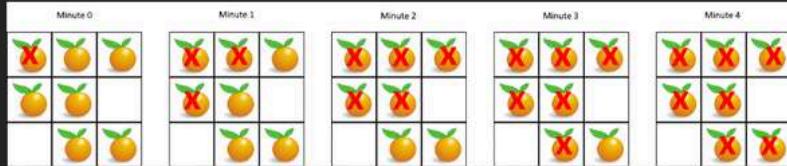
You are given an $m \times n$ grid where each cell can have one of three values:

- 0 representing an empty cell,
- 1 representing a fresh orange, or
- 2 representing a rotten orange.

Every minute, any fresh orange that is **4-directionally adjacent** to a rotten orange becomes rotten.

Return the *minimum number of minutes* that must elapse until no cell has a fresh orange. If this is impossible, return -1.

Example 1:



Input: grid = [[2,1,1],[1,1,0],[0,1,1]]
Output: 4

Example 2:

Input: grid = [[2,1,1],[0,1,1],[1,0,1]]
Output: -1
Explanation: The orange in the bottom left corner (row 2, column 0) is never rotten, because rotting only happens 4-directionally.

Example 3:

Input: grid = [[0,2]]
Output: 0
Explanation: Since there are already no fresh oranges at minute 0, the answer is just 0.

```
public class Solution {

    private static final int[][] DIRECTIONS = {
        {0, 1}, {1, 0}, {0, -1}, {-1, 0}
    };

    public static int orangesRotting(int[][] grid) {
        int m = grid.length;
        int n = grid[0].length;
        Queue<int[]> queue = new LinkedList<>();
        int freshOranges = 0;

        // Step 1: Initialize the queue with all initially rotten oranges
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == 2) {
                    queue.offer(new int[]{i, j});
                } else if (grid[i][j] == 1) {
                    freshOranges++;
                }
            }
        }

        if (freshOranges == 0) return 0;

        int minutes = -1;
```

```

// Step 2: BFS traversal
while (!queue.isEmpty()) {
    int size = queue.size();
    minutes++;
    for (int k = 0; k < size; k++) {
        int[] cell = queue.poll();
        int row = cell[0];
        int col = cell[1];

        for (int[] dir : DIRECTIONS) {
            int newRow = row + dir[0];
            int newCol = col + dir[1];

            if (isValid(newRow, newCol, m, n, grid)) {
                grid[newRow][newCol] = 2;
                freshOranges--;
                queue.offer(new int[]{newRow, newCol});
            }
        }
    }
}

return freshOranges == 0 ? minutes : -1;
}

private static boolean isValid(int i, int j, int m, int n, int[][] grid) {
    return i >= 0 && i < m && j >= 0 && j < n && grid[i][newCol] == 1;
}

public static void main(String[] args) {
    int[][] grid = {
        {2, 1, 1},
        {1, 1, 0},
        {0, 1, 1}
    };
    System.out.println(orangesRotting(grid)); // Output: 4
}
}

```

- Count Fresh Oranges
- Track Rotten Oranges using queue
- For each node, traverse to neighbor nodes using directions and if the neighbor is valid offer to the queue

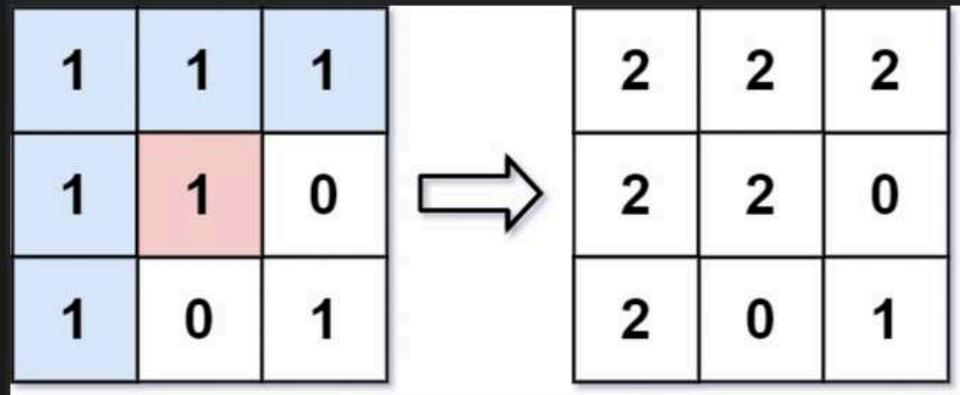
7. Flood Fill

Example 1:

Input: image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, color = 2

Output: [[2,2,2],[2,2,0],[2,0,1]]

Explanation:



From the center of the image with position $(sr, sc) = (1, 1)$ (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.

Note the bottom corner is not colored 2, because it is not horizontally or vertically connected to the starting pixel.

```
class Solution {
    // Directions: right, down, left, up
    private static final int[][] DIRECTIONS = {{0, 1}, {1, 0}, {0, -1},
    {-1, 0}};
    // O(1)

    public int[][] floodFill(int[][] image, int sr, int sc, int newColor) {
        int originalColor = image[sr][sc];

        // If the original color is the same as the new color, no need to p
        if (originalColor != newColor) {
            dfs(image, sr, sc, newColor, originalColor);
        }

        return image;
    }

    private void dfs(int[][] image, int i, int j, int newColor,
    int originalColor) {
        if (i < 0 || i >= image.length || j < 0 || j >= image[0].length
        || image[i][j] != originalColor) {
            return;
        }

        image[i][j] = newColor;

        for (int[] direction : DIRECTIONS) {
            int newRow = i + direction[0];
            int newCol = j + direction[1];
            dfs(image, newRow, newCol, newColor, originalColor);
        }
    }
}
```

Matrix Patterns

Made with Notion

Matrix Patterns | Notion

Count number of subMatrices with all 1

weak-lunary-372 on Notion

1. Spiral Matrix
2. Word Search
3. Rotate Matrix Clockwise by 1 cell
4. Rotate Matrix AntiClockwise by 1 cell
5. Spiral Matrix II
6. Diagonal Traverse
7. Search a 2D Matrix
8. Search a 2D Matrix II
9. Set Matrix Zeroes
10. Word Search
11. Surrounded Regions
12. Dungeon Game
13. Rotate Image
14. Valid Sudoku

1275. Find Winner on a Tic Tac Toe Game

Solved ✓

Easy Topics Companies Hint

Tic-tac-toe is played by two players A and B on a 3×3 grid. The rules of Tic-Tac-Toe are:

- Players take turns placing characters into empty squares.
- The first player A always places 'X' characters, while the second player B always places 'O' characters.
- 'X' and 'O' characters are always placed into empty squares, never on filled ones.
- The game ends when there are three of the same (non-empty) character filling any row, column, or diagonal.
- The game also ends if all squares are non-empty.
- No more moves can be played if the game is over.

Given a 2D integer array moves where moves[i] = [row_i, col_i] indicates that the i^{th} move will be played on grid[row_i][col_i]. return the winner of the game if it exists (A or B). In case the game ends in a draw return "Draw". If there are still movements to play return "Pending".

You can assume that moves is valid (i.e., it follows the rules of Tic-Tac-Toe), the grid is initially empty, and A will play first.

Example 1:

X		
	X	
O	O	X

Input: moves = [[0,0],[2,0],[1,1],[2,1],[2,2]]

Output: "A"

Explanation: A wins, they always play first.

```
class Solution {
    public String tictactoe(int[][] moves) {
        boolean playerAWins = false;
        boolean playerBWins = false;
        boolean draw = true;

        int[][] matrix = new int[3][3];
        for (int[] row : matrix) {
            Arrays.fill(row, -1);
        }
        int n = moves.length;

        for (int i = 0; i < n; i++) {
            int row = moves[i][0];
            int col = moves[i][1];
            if (i % 2 == 0) {
                matrix[row][col] = 1;
            } else {
                matrix[row][col] = 0;
            }
        }
        for (int i = 0; i < 3; i++) {
            if (matrix[i][0] == 1 && matrix[i][1] == 1 && matrix[i][2] == 1) {
                playerAWins = true;
            } else if (matrix[i][0] == 0 && matrix[i][1] == 0 && matrix[i][2] == 0) {
                playerBWins = true;
            }
        }
        for (int j = 0; j < 3; j++) {
            if (matrix[0][j] == 1 && matrix[1][j] == 1 && matrix[2][j] == 1) {
                playerAWins = true;
            } else if (matrix[0][j] == 0 && matrix[1][j] == 0 && matrix[2][j] == 0) {
                playerBWins = true;
            }
        }
        if (matrix[0][0] == 1 && matrix[1][1] == 1 && matrix[2][2] == 1) {
            playerAWins = true;
        } else if (matrix[0][0] == 0 && matrix[1][1] == 0 && matrix[2][2] == 0) {
            playerBWins = true;
        }

        if (playerAWins) {
            return "A";
        } else if (playerBWins) {
            return "B";
        } else if (draw) {
            return "Draw";
        } else {
            return "Pending";
        }
    }
}
```

Design

1. LRU Cache

146. LRU Cache

Medium

Design a data structure that follows the constraints of a [Least Recently Used \(LRU\) cache](#).

Implement the `LRUCache` class:

- `LRUCache(int capacity)` Initialize the LRU cache with positive size `capacity`.
- `int get(int key)` Return the value of the `key` if the key exists, otherwise return `-1`.
- `void put(int key, int value)` Update the value of the `key` if the `key` exists. Otherwise, add the `key-value` pair to the cache. If the number of keys exceeds the `capacity` from this operation, evict the least recently used key.

The functions `get` and `put` must each run in $O(1)$ average time complexity.

Example 1:

Input
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, null, -1, 3, 4]
Explanation
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // cache is {1=1}
lRUCache.put(2, 2); // cache is {1=1, 2=2}
lRUCache.get(1); // return 1
lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3}
lRUCache.get(2); // returns -1 (not found)
lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3}
lRUCache.get(1); // return -1 (not found)
lRUCache.get(3); // return 3
lRUCache.get(4); // return 4

```
class LRUCache {  
    class Node {  
        int key;  
        int value;  
        Node prev;  
        Node next;  
  
        public Node(int key, int value) {  
            this.key = key;  
            this.value = value;  
        }  
    }  
  
    private int capacity;  
    private Map<Integer, Node> map;  
    private Node head;  
    private Node tail;  
  
    public LRUCache(int capacity) {  
        this.capacity = capacity;  
        map = new HashMap<>();  
        head = new Node(-1, -1);  
        tail = new Node(-1, -1);  
        head.next = tail;  
        tail.prev = head;  
    }  
  
    public int get(int key) {  
        if (!map.containsKey(key)) {  
            return -1;  
        }  
  
        Node node = map.get(key);  
        moveToHead(node);  
  
        return node.value;  
    }  
}
```

```

public void put(int key, int value) {
    if (map.containsKey(key)) {
        Node node = map.get(key);
        node.value = value;
        moveToHead(node);
    } else {
        Node newNode = new Node(key, value);
        map.put(key, newNode);
        addToHead(newNode);

        if (map.size() > capacity) {
            Node removed = removeTail();
            map.remove(removed.key);
        }
    }
}

private void moveToHead(Node node) {
    removeNode(node);
    addToHead(node);
}

private void removeNode(Node node) {
    node.prev.next = node.next;
    node.next.prev = node.prev;
}

private void addToHead(Node node) {
    node.next = head.next;
    node.next.prev = node;
    head.next = node;
    node.prev = head;
}

private Node removeTail() {
    Node nodeToRemove = tail.prev;
    removeNode(nodeToRemove);
    return nodeToRemove;
}
}

```

```

public class Main {
    public static void main(String[] args) {
        LRUcache cache = new LRUcache(2); // Capacity is 2
        cache.put(1, 1);
        cache.put(2, 2);
        System.out.println(cache.get(1)); // Output: 1
        cache.put(3, 3);
        System.out.println(cache.get(2)); // Output: -1 (not found)
        cache.put(4, 4);
        System.out.println(cache.get(1)); // Output: -1 (not found)
        System.out.println(cache.get(3)); // Output: 3
        System.out.println(cache.get(4)); // Output: 4
    }
}

```

Add least recently used node to the front of linked list and maintain a map to put the key and node there.

3. Least Frequently Used Cache (LFU Cache)

Design and implement a data structure for a [Least Frequently Used \(LFU\) cache](#).

Implement the `LFUCache` class:

- `LFUCache(int capacity)` Initializes the object with the `capacity` of the data structure.
- `int get(int key)` Gets the value of the `key` if the `key` exists in the cache. Otherwise, returns `-1`.
- `void put(int key, int value)` Update the value of the `key` if present, or inserts the `key` if not already present. When the cache reaches its `capacity`, it should invalidate and remove the least frequently used key before inserting a new item. For this problem, when there is a tie (i.e., two or more keys with the same frequency), the least recently used `key` would be invalidated.

To determine the least frequently used key, a use counter is maintained for each key in the cache. The key with the smallest use counter is the least frequently used key.

When a key is first inserted into the cache, its use counter is set to `1` (due to the `put` operation). The use counter for a key in the cache is incremented either a `get` or `put` operation is called on it.

The functions `get` and `put` must each run in $O(1)$ average time complexity.

Example 1:

Input
["LFUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [3], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, 3, null, -1, 3, 4]
Explanation
// cnt(x) = the use counter for key x
// cache=[] will show the last used order for tiebreakers (leftmost element is most recent)
LFUCache lfu = new LFUCache(2);
lfu.put(1, 1); // cache=[1,1], cnt(1)=1
lfu.put(2, 2); // cache=[2,1], cnt(2)=1, cnt(1)=1
lfu.get(1); // return 1
// cache=[1,2], cnt(2)=1, cnt(1)=2
lfu.put(3, 3); // 2 is the LFU key because cnt(2)=1 is the smallest, invalidate 2.

```
class LFUCache {  
    class Node {  
        int key, value, freq;  
        Node prev, next;  
        Node(int k, int v) {  
            key = k;  
            value = v;  
            freq = 1;  
        }  
    }  
  
    class DoublyLinkedList {  
        Node head, tail;  
        DoublyLinkedList() {  
            head = new Node(0, 0);  
            tail = new Node(0, 0);  
            head.next = tail;  
            tail.prev = head;  
        }  
  
        void addNode(Node node) {  
            Node nextNode = head.next;  
            head.next = node;  
            node.prev = head;  
            node.next = nextNode;  
            nextNode.prev = node;  
        }  
  
        void removeNode(Node node) {  
            Node prevNode = node.prev;  
            Node nextNode = node.next;  
            prevNode.next = nextNode;  
            nextNode.prev = prevNode;  
        }  
  
        boolean isEmpty() {  
            return head.next == tail;  
        }  
    }  
  
    private int capacity, size, minFreq;  
    private Map<Integer, Node> keyToNode;  
    private Map<Integer, DoublyLinkedList> freqToDLL;
```

```
public LFUCache(int capacity) {
    this.capacity = capacity;
    this.size = 0;
    this.minFreq = 0;
    this.keyToNode = new HashMap<>();
    this.freqToDLL = new HashMap<>();
}

public int get(int key) {
    if (!keyToNode.containsKey(key)) return -1;
    Node node = keyToNode.get(key);
    updateFrequency(node);
    return node.value;
}

public void put(int key, int value) {
    if (capacity == 0) return;
    if (keyToNode.containsKey(key)) {
        Node node = keyToNode.get(key);
        node.value = value;
        updateFrequency(node);
    } else {
        if (size == capacity) {
            DoublyLinkedList minFreqList = freqToDLL.get(minFreq);
            keyToNode.remove(minFreqList.tail.prev.key);
            minFreqList.removeNode(minFreqList.tail.prev);
            size--;
        }
        Node newNode = new Node(key, value);
        keyToNode.put(key, newNode);
        minFreq = 1;
        freqToDLL.computeIfAbsent(1, k -> new DoublyLinkedList()).addNode(newNode);
        size++;
    }
}

private void updateFrequency(Node node) {
    int oldFreq = node.freq;
    DoublyLinkedList oldList = freqToDLL.get(oldFreq);
    oldList.removeNode(node);
    if (oldFreq == minFreq && oldList.isEmpty()) minFreq++;
    node.freq++;
    freqToDLL.computeIfAbsent(node.freq, k -> new DoublyLinkedList()).addNode(node);
}
```

2. Max Stack

Design a max stack that supports push, pop, top, peekMax and popMax.

- `push(x)` -- Push element `x` onto stack.
- `pop()` -- Remove the element on top of the stack and return it.
- `top()` -- Get the element on the top.
- `peekMax()` -- Retrieve the maximum element in the stack.
- `popMax()` -- Retrieve the maximum element in the stack, and remove it. If you find more than one maximum elements, only remove the top-most one.

Example 1:

```
MaxStack stack = new MaxStack();
stack.push(5);
stack.push(1);
stack.push(5);
stack.top(); -> 5
stack.popMax(); -> 5
stack.top(); -> 1
stack.peekMax(); -> 5
stack.pop(); -> 1
stack.top(); -> 5
```

Note:

- $-1e7 \leq x \leq 1e7$
- Number of operations won't exceed 10000.
- The last four operations won't be called when stack is empty.

```
class Node {
    public int val;
    public Node prev, next;

    public Node() {}

    public Node(int val) {
        this.val = val;
    }
}

class DoubleLinkedList {
    private final Node head = new Node();
    private final Node tail = new Node();

    public DoubleLinkedList() {
        head.next = tail;
        tail.prev = head;
    }

    public Node append(int val) {
        Node node = new Node(val);
        node.next = tail;
        node.prev = tail.prev;
        tail.prev = node;
        node.prev.next = node;
        return node;
    }

    public static Node remove(Node node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
        return node;
    }

    public Node pop() {
        return remove(tail.prev);
    }
}
```

```
public Node pop() {
    return remove(tail.prev);
}

public int peek() {
    return tail.prev.val;
}
}

class MaxStack {
    private DoubleLinkedList stack = new DoubleLinkedList();
    private TreeMap<Integer, List<Node>> map = new TreeMap<>();

    public MaxStack() {
    }

    public void push(int x) {
        Node node = stack.append(x);
        map.computeIfAbsent(x, k -> new ArrayList<>()).add(node);
    }

    public int pop() {
        Node node = stack.pop();
        List<Node> nodes = map.get(node.val);
        int x = nodes.remove(nodes.size() - 1).val;
        if (nodes.isEmpty()) {
            map.remove(node.val);
        }
        return x;
    }

    public int top() {
        return stack.peek();
    }

    public int peekMax() {
        return map.lastKey();
    }

    public int popMax() {
        int x = peekMax();
        List<Node> nodes = map.get(x);
        Node node = nodes.remove(nodes.size() - 1);
        if (nodes.isEmpty()) {
            map.remove(x);
        }
        DoubleLinkedList.remove(node);
        return x;
    }
}
```

4. Most Recently Used Cache (MRU Cache)

```
import java.util.*;

public class MRUCache {
    private class Node {
        int key, value;
        Node prev, next;

        Node(int k, int v) {
            key = k;
            value = v;
        }
    }

    private final int capacity;
    private final Map<Integer, Node> map;
    private final Node head, tail;

    public MRUCache(int capacity) {
        this.capacity = capacity;
        map = new HashMap<>();
        head = new Node(0, 0); // dummy head
        tail = new Node(0, 0); // dummy tail
        head.next = tail;
        tail.prev = head;
    }

    public int get(int key) {
        if (!map.containsKey(key)) return -1;
        Node node = map.get(key);
        remove(node);
        insertToTail(node); // most recently used - move to tail
        return node.value;
    }

    public void put(int key, int value) {
        if (map.containsKey(key)) {
            Node node = map.get(key);
            node.value = value;
            remove(node);
            insertToTail(node);
        } else {
            if (map.size() == capacity) {
                // Remove the most recently used node (tail.prev)
                Node mru = tail.prev;
                remove(mru);
                map.remove(mru.key);
            }
            Node newNode = new Node(key, value);
            insertToTail(newNode);
            map.put(key, newNode);
        }
    }

    private void remove(Node node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }

    private void insertToTail(Node node) {
        Node last = tail.prev;
        last.next = node;
        node.prev = last;
        node.next = tail;
        tail.prev = node;
    }
}
```

The Most Recently Used (MRU) Cache is a type of cache eviction policy where, when the cache is full, the item that was most recently used is removed first to make space for a new item. This is the opposite of the Least Recently Used (LRU) cache, where the least recently accessed item is removed.

⌚ Use Case

The MRU cache is useful in scenarios where the least used items are more likely to be accessed again, and the most recently accessed items are less likely to be needed in the future.

✓ Operations

Just like other caches, MRU should support:

get(key): Retrieve the value (will mark it as most recently used).

put(key, value): Insert or update a value (may evict the most recently used item if at capacity).

4. Max Stack

Design a max stack that supports push, pop, top, peekMax and popMax.

- `push(x)` -- Push element `x` onto stack.
- `pop()` -- Remove the element on top of the stack and return it.
- `top()` -- Get the element on the top.
- `peekMax()` -- Retrieve the maximum element in the stack.
- `popMax()` -- Retrieve the maximum element in the stack, and remove it. If you find more than one maximum elements, only remove the top-most one.

Example 1:

```
MaxStack stack = new MaxStack();
stack.push(5);
stack.push(1);
stack.push(5);
stack.top(); -> 5
stack.popMax(); -> 5
stack.top(); -> 1
stack.peekMax(); -> 5
stack.pop(); -> 1
stack.top(); -> 5
```

Note:

- $-1e7 \leq x \leq 1e7$
- Number of operations won't exceed 10000.
- The last four operations won't be called when stack is empty.

```
class Node {
    public int val;
    public Node prev, next;

    public Node() {}

    public Node(int val) {
        this.val = val;
    }
}

class DoubleLinkedList {
    private final Node head = new Node();
    private final Node tail = new Node();

    public DoubleLinkedList() {
        head.next = tail;
        tail.prev = head;
    }

    public Node append(int val) {
        Node node = new Node(val);
        node.next = tail;
        node.prev = tail.prev;
        tail.prev = node;
        node.prev.next = node;
        return node;
    }

    public static Node remove(Node node) {
        node.prev.next = node.next;
        node.next.prev = node.prev;
        return node;
    }

    public Node pop() {
        return remove(tail.prev);
    }

    public int peek() {
        return tail.prev.val;
    }
}
```

```

class MaxStack {
    private DoubleLinkedList stack = new DoubleLinkedList();
    private TreeMap<Integer, List<Node>> map = new TreeMap<>();

    public MaxStack() {}

    public void push(int x) {
        Node node = stack.append(x);
        map.computeIfAbsent(x, k -> new ArrayList<>()).add(node);
    }

    public int pop() {
        Node node = stack.pop();
        List<Node> nodes = map.get(node.val);
        int x = nodes.remove(nodes.size() - 1).val;
        if (nodes.isEmpty()) {
            map.remove(node.val);
        }
        return x;
    }

    public int top() {
        return stack.peek();
    }

    public int peekMax() {
        return map.lastKey();
    }

    public int popMax() {
        int x = peekMax();
        List<Node> nodes = map.get(x);
        Node node = nodes.remove(nodes.size() - 1);
        if (nodes.isEmpty()) {
            map.remove(x);
        }
        DoubleLinkedList.remove(node);
        return x;
    }
}

```

5. Snake And Ladders

You are given an $n \times n$ integer matrix `board` where the cells are labeled from 1 to n^2 in a Boustrophedon style starting from the bottom left of the board (i.e. `board[n - 1][0]`) and alternating direction each row.

You start on square 1 of the board. In each move, starting from square `curr`, do the following:

- Choose a destination square `next` with a label in the range $[curr + 1, \min(curr + 6, n^2)]$.
 - This choice simulates the result of a standard 6-sided die roll: i.e., there are always at most 6 destinations, regardless of the size of the board.
- If `next` has a snake or ladder, you must move to the destination of that snake or ladder. Otherwise, you move to `next`.
- The game ends when you reach the square n^2 .

A board square on row r and column c has a snake or ladder if `board[r][c] != -1`. The destination of that snake or ladder is `board[r][c]`. Squares 1 and n^2 are not the starting points of any snake or ladder.

Note that you only take a snake or ladder at most once per dice roll. If the destination to a snake or ladder is the start of another snake or ladder, you do not follow the subsequent snake or ladder.

- For example, suppose the board is $\begin{bmatrix} [-1, 4], [-1, 3] \end{bmatrix}$, and on the first move, your destination square is 2 . You follow the ladder to square 3 , but do not follow the subsequent ladder to 4 .

Return the least number of dice rolls required to reach the square n^2 . If it is not possible to reach the square, return -1 .

Example 1:

36	35	34	33	32	31	
25	26	27	28	29	30	
24	25	22	21	20	19	
13	14	15	16	17	18	
12	11	10	9	8	7	
1	2	3	4	5	6	

Input: board = [[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1], [-1,35,-1,-1,13,-1],[-1,-1,-1,-1,-1,-1],[-1,15,-1,-1,-1,-1]]

Output: 4

Explanation:

In the beginning, you start at square 1 (at row 5, column 0).

You decide to move to square 2 and must take the ladder to square 35.

You then decide to move to square 17 and must take the snake to square 13.

You then decide to move to square 14 and must take the ladder to square 35.

You then decide to move to square 36, ending the game.

This is the lowest possible number of moves to reach the last square, so return 4.

Example 2:

Input: board = [[-1,-1],[-1,3]]

Output: 1

```
public class Solution {
    public int snakesAndLadders(int[][] board) {
        int n = board.length;
        int[] flattenedBoard = new int[n * n];
        int index = 0;
        boolean leftToRight = true;
        for (int i = n - 1; i >= 0; i--) {
            if (leftToRight) {
                for (int j = 0; j < n; j++) {
                    flattenedBoard[index++] = board[i][j];
                }
            } else {
                for (int j = n - 1; j >= 0; j--) {
                    flattenedBoard[index++] = board[i][j];
                }
            }
            leftToRight = !leftToRight;
        }
        Queue<Integer> queue = new ArrayDeque<>();
        boolean[] visited = new boolean[n * n];
        queue.offer(0);
        visited[0] = true;
        int moves = 0;

        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                int current = queue.poll();
                if (current == n * n - 1) {
                    return moves;
                }
                for (int dice = 1; dice <= 6; dice++) {
                    int next = current + dice;
                    if (next >= n * n) break;
                    int destination = flattenedBoard[next] == -1 ? next : flattenedBoard[next] - 1;
                    if (!visited[destination]) {
                        visited[destination] = true;
                        queue.offer(destination);
                    }
                }
            }
            moves++;
        }
        return -1;
    }
}
```

Example 1:

36	35	34	33	32	31	
25	26	27	28	29	30	
24	25	22	21	20	19	
13	14	15	16	17	18	
12	11	10	9	8	7	
1	2	3	4	5	6	

Input: board = [[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1],[-1,-1,-1,-1,-1,-1], [-1,35,-1,-1,13,-1],[-1,-1,-1,-1,-1,-1],[-1,15,-1,-1,-1,-1]]

Output: 4

Explanation:

In the beginning, you start at square 1 (at row 5, column 0).

You decide to move to square 2 and must take the ladder to square 35.

You then decide to move to square 17 and must take the snake to square 13.

You then decide to move to square 14 and must take the ladder to square 35.

You then decide to move to square 36, ending the game.

This is the lowest possible number of moves to reach the last square, so return 4.

Example 2:

Input: board = [[-1,-1],[-1,3]]

Output: 1

```
public class Solution {
    public int snakesAndLadders(int[][] board) {
        int n = board.length;
        int[] flattenedBoard = new int[n * n];
        int index = 0;
        boolean leftToRight = true;
        for (int i = n - 1; i >= 0; i--) {
            if (leftToRight) {
                for (int j = 0; j < n; j++) {
                    flattenedBoard[index++] = board[i][j];
                }
            } else {
                for (int j = n - 1; j >= 0; j--) {
                    flattenedBoard[index++] = board[i][j];
                }
            }
            leftToRight = !leftToRight;
        }
        Queue<Integer> queue = new ArrayDeque<>();
        boolean[] visited = new boolean[n * n];
        queue.offer(0);
        visited[0] = true;
        int moves = 0;

        while (!queue.isEmpty()) {
            int size = queue.size();
            for (int i = 0; i < size; i++) {
                int current = queue.poll();
                if (current == n * n - 1) {
                    return moves;
                }
                for (int dice = 1; dice <= 6; dice++) {
                    int next = current + dice;
                    if (next >= n * n) break;
                    int destination = flattenedBoard[next] == -1 ? next : flattenedBoard[next] - 1;
                    if (!visited[destination]) {
                        visited[destination] = true;
                        queue.offer(destination);
                    }
                }
            }
            moves++;
        }
        return -1;
    }
}
```

Minimum Spanning Tree

A Minimum Spanning Tree (MST) of a weighted, connected, undirected graph is a spanning tree whose sum of edge weights is as small as possible. The spanning tree includes all the vertices of the graph, and the total weight of all the edges in the tree is minimized.

Kruskal's algorithm and Prim's algorithm are both popular algorithms used to find the minimum spanning tree (MST) of a weighted undirected graph, but they use different approaches to achieve this.

Kruskal's Algorithm

- Approach: Kruskal's algorithm grows the minimum spanning tree edge by edge by selecting the smallest edge that does not form a cycle.
- Data Structure: It typically uses a disjoint-set data structure (Union-Find) to efficiently check for cycles and maintain connectivity.
- Edge Selection: The algorithm sorts all the edges by weight and then iterates over them, adding edges to the MST if they do not create a cycle.
- Complexity: With efficient data structures like Union-Find, Kruskal's algorithm runs in $O(E \log E)$ time, where E is the number of edges.

Prim's Algorithm:

- Approach: Prim's algorithm grows the minimum spanning tree vertex by vertex by selecting the smallest edge that connects a vertex in the MST to a vertex outside the MST.
- Data Structure: It typically uses a priority queue (or a min-heap) to efficiently select the next edge to consider.
- Edge Selection: The algorithm starts with an arbitrary vertex and repeatedly grows the MST by adding the smallest edge that connects a vertex in the MST to a vertex outside the MST.
- Complexity: With a priority queue implementation, Prim's algorithm runs in $O(E + V \log V)$ time, where V is the number of vertices and E is the number of edges.

Kruskal Algorithm

$$E=V-1$$

```
public class Solution {
    int[] parent;
    int[] rank;

    static class Edge {
        int from, to, weight;

        Edge(int from, int to, int weight) {
            this.from = from;
            this.to = to;
            this.weight = weight;
        }
    }

    public int kruskal(int n, List<Edge> edges) {
        Collections.sort(edges, Comparator.comparingInt(e -> e.weight));
        parent = new int[n];
        rank = new int[n];

        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
        int mstWeight = 0;
        for (Edge edge : edges) {
            if (union(edge.from, edge.to)) {
                mstWeight += edge.weight;
                System.out.println("Added edge: " + edge.from + " - " + edge.to + " (weight=" + edge.weight + ")");
            }
        }
        System.out.println(edges.stream().map(x -> x.weight).collect(Collectors.toList())); // [4, 5, 6, 10, 15]
        System.out.println(Arrays.toString(parent)); // [2, 2, 2, 2]
        System.out.println(Arrays.toString(rank)); // [0, 0, 1, 0]
        return mstWeight;
    }
}
```

// Edges:
// 2 - 3 (4)
// 0 - 3 (5)
// 0 - 2 (6)
// 0 - 1 (10)
// 1 - 3 (15)

```

private int find(int u) {
    if (parent[u] != u) {
        parent[u] = find(parent[u]); // path compression
    }
    return parent[u];
}

private boolean union(int u, int v) {
    int rootU = find(u);
    int rootV = find(v);

    if (rootU == rootV) return false;

    if (rank[rootU] < rank[rootV]) {
        parent[rootU] = rootV;
    } else if (rank[rootU] > rank[rootV]) {
        parent[rootV] = rootU;
    } else {
        parent[rootV] = rootU;
        rank[rootU]++;
    }

    return true;
}

public static void main(String[] args) {
    List<Edge> edges = Arrays.asList(
        new Edge( from: 0, to: 1, weight: 10),
        new Edge( from: 0, to: 2, weight: 6),
        new Edge( from: 0, to: 3, weight: 5),
        new Edge( from: 1, to: 3, weight: 15),
        new Edge( from: 2, to: 3, weight: 4)
    );
    int n = 4; // number of nodes

    Solution kruskal = new Solution();
    int totalWeight = kruskal.kruskal(n, edges);

    System.out.println("Total weight of MST: " + totalWeight);

    // Graph structure:
    //
    //      (10)
    //      0 ----- 1
    //      | \     |
    //      |  \   |
    // (6) |  \ (5)  | (15)
    //      |    \  |
    //      |    \  |
    // 2 ---- 3
    //      (4)
    //
    // Edges:          /*
    // 0 - 1 (10)      Added edge: 2 - 3 (weight=4)
    // 0 - 2 (6)       Added edge: 0 - 3 (weight=5)
    // 0 - 3 (5)       Added edge: 0 - 1 (weight=10)
    // 1 - 3 (15)      Total weight of MST: 19
    // 2 - 3 (4)       */
}

```

Let's compute `parent[]` and `rank[]` step by step for the MST built using Kruskal's Algorithm on this graph:

✓ Initial State:

Before any union:

java

Copy Edit

```
parent = [0, 1, 2, 3]
rank   = [0, 0, 0, 0]
```

✓ Sorted Edges (by weight):

csharp

Copy Edit

```
[2-3 (4), 0-3 (5), 0-2 (6), 0-1 (10), 1-3 (15)]
```

∅ Step 1: `union(2, 3)`

- Roots: 2 and 3
- Equal rank → attach 3 under 2
- `rank[2]++`

java

Copy Edit

```
parent = [0, 1, 2, 2]
rank   = [0, 0, 1, 0]
```

∅ Step 2: `union(0, 3)`

- Find(3) → `parent[3] = 2` → root is 2
- Roots: 0 and 2
- `rank[0] < rank[2]` → attach 0 under 2

java

Copy Edit

```
parent = [2, 1, 2, 2]
rank   = [0, 0, 1, 0]
```

∅ Step 3: `union(0, 1)`

- Find(0) → `parent[0] = 2` → root is 2
- Roots: 2 and 1
- `rank[2] > rank[1]` → attach 1 under 2

java

Copy Edit

```
parent = [2, 2, 2, 2]
rank   = [0, 0, 1, 0]
```

∅ Step 4: `union(0, 2)`

- Find(0) → 2, Find(2) → 2
- Already in the same set → skip

∅ Step 5: `union(1, 3)`

- Find(1) → 2, Find(3) → 2
- Already in the same set → skip

Final State:

java

Copy Edit

```
parent = [2, 2, 2, 2]
rank   = [0, 0, 1, 0]
```

This shows all nodes are connected in a single component rooted at 2, and rank[2] = 1 means the tree height rooted at 2 is 1.

```
//      0    1    2        3
//      {}    {}  {0, 1, 3}  {}
|
// Edges:
```

1489. Find Critical and Pseudo-Critical Edges in Minimum Spanning Tree

1489. Find Critical and Pseudo-Critical Edges in Minimum Spanning Tree

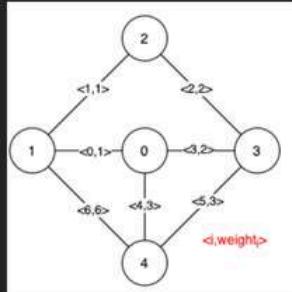
Hard

Given a weighted undirected connected graph with n vertices numbered from 0 to $n - 1$, and an array `edges` where `edges[i] = [ai, bi, weighti]` represents a bidirectional and weighted edge between nodes a_i and b_i . A minimum spanning tree (MST) is a subset of the graph's edges that connects all vertices without cycles and with the minimum possible total edge weight.

Find all the critical and pseudo-critical edges in the given graph's minimum spanning tree (MST). An MST edge whose deletion from the graph would cause the MST weight to increase is called a *critical edge*. On the other hand, a pseudo-critical edge is that which can appear in some MSTs but not all.

Note that you can return the indices of the edges in any order.

Example 1:

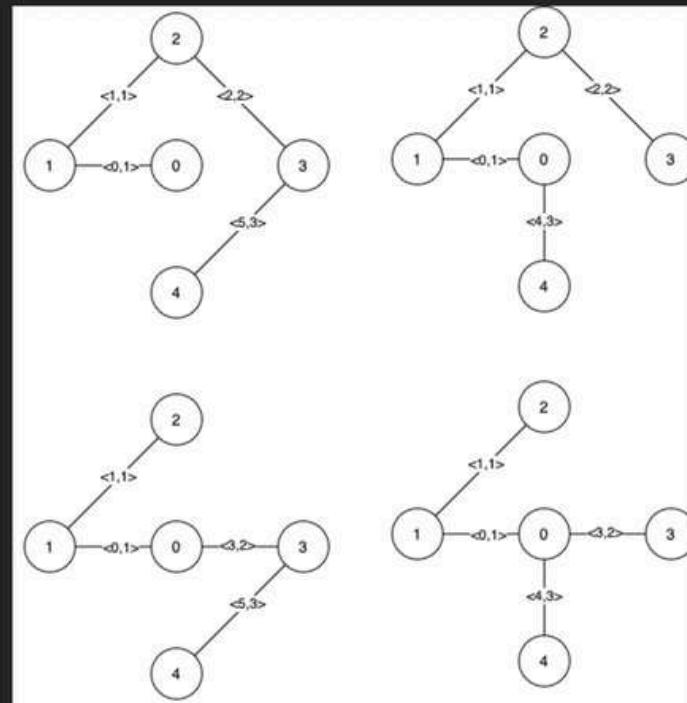


Input: $n = 5$, edges = [[0,1,1],[1,2,1],[2,3,2],[0,3,2],[0,4,3],[3,4,3],[1,4,6]]

Output: [[0,1],[2,3,4,5]]

Explanation: The figure above describes the graph.

The following figure shows all the possible MSTs:



Notice that the two edges 0 and 1 appear in all MSTs, therefore they are critical edges, so we return them in the first list of the output. The edges 2, 3, 4, and 5 are only part of some MSTs, therefore they are considered pseudo-critical edges. We add them to the second list of the output.


```

private int getMSTWeight(int n, int[][] edges, int[] firstEdge, int deletedEdgeIndex) {
    int mstWeight = 0;
    UnionFind uf = new UnionFind(n);

    if (firstEdge.length == 4) {
        uf.unionByRank(firstEdge[0], firstEdge[1]);
        mstWeight += firstEdge[2];
    }

    for (int[] edge : edges) {
        final int u = edge[0];
        final int v = edge[1];
        final int weight = edge[2];
        final int index = edge[3];
        if (index == deletedEdgeIndex)
            continue;
        if (uf.find(u) == uf.find(v))
            continue;
        uf.unionByRank(u, v);
        mstWeight += weight;
    }

    final int root = uf.find(0);
    for (int i = 0; i < n; ++i)
        if (uf.find(i) != root)
            return Integer.MAX_VALUE;

    return mstWeight;
}

public static void main(String[] args) {
    Solution sol = new Solution();
    int n = 14;
    int[][] edges = {
        {0, 1, 13}, {0, 2, 6}, {2, 3, 13}, {3, 4, 4}, {0, 5, 11}, {4, 6, 14}, {4, 7, 8},
        {2, 8, 6}, {4, 9, 6}, {7, 10, 4}, {5, 11, 3}, {6, 12, 7}, {12, 13, 9}, {7, 13, 2},
        {5, 13, 10}, {0, 6, 4}, {2, 7, 3}, {0, 7, 8}, {1, 12, 9}, {10, 12, 11}, {1, 2, 7},
        {1, 3, 10}, {3, 10, 6}, {6, 10, 4}, {4, 8, 5}, {1, 13, 4}, {11, 13, 8}, {2, 12, 10},
        {5, 8, 1}, {3, 7, 6}, {7, 12, 12}, {1, 7, 9}, {5, 9, 1}, {2, 13, 10}, {10, 11, 4},
        {3, 5, 10}, {6, 11, 14}, {5, 12, 3}, {0, 8, 13}, {8, 9, 1}, {3, 6, 8}, {0, 3, 4},
        {2, 9, 6}, {0, 11, 4}, {2, 5, 14}, {4, 11, 2}, {7, 11, 11}, {1, 11, 6}, {2, 10, 12},
        {0, 13, 4}, {3, 9, 9}, {4, 12, 3}, {6, 7, 10}, {6, 8, 13}, {9, 11, 3}, {1, 6, 2},
        {2, 4, 12}, {0, 10, 3}, {3, 12, 1}, {3, 8, 12}, {1, 8, 6}, {8, 13, 2}, {10, 13, 12},
        {9, 13, 11}, {2, 11, 14}, {5, 10, 9}, {5, 6, 10}, {2, 6, 9}, {8, 11, 3}
    };
    List<List<Integer>> result = sol.findCriticalAndPseudoCriticalEdges(n, edges);
    System.out.println(result);
}
}

```

- UnionFind Class:** Implements the union-find data structure with path compression and union by rank for efficient merging of sets.
- findCriticalAndPseudoCriticalEdges Method:**
 - Sort edges by their weights and add an index to each edge for tracking.
 - Find MST weight using Kruskal's algorithm without any edge removal or forced addition.
 - Check each edge:
 - If removing an edge increases MST weight, it's **critical**.
 - If adding an edge first doesn't change the MST weight, it's **pseudo-critical**.
- getMSTWeight Method:** Runs Kruskal's algorithm, optionally adding or removing specific edges, and calculates the total MST weight.

Prim's Algorithm

Prim's Algorithm: It does indeed select adjacent edges in the sense that it starts from an initial vertex and greedily adds the smallest edge that connects a vertex in the MST to a vertex outside the MST. It grows the tree vertex by vertex, always selecting the smallest available edge.

```
public class Solution {

    static class Edge {
        int from, to, weight;

        public Edge(int from, int to, int weight) {
            this.from = from;
            this.to = to;
            this.weight = weight;
        }
    }

    public static int primMST(int n, List<Edge> edges) {
        // Build adjacency list from edge list
        List<Edge>[] adj = new ArrayList[n];
        for (int i = 0; i < n; i++) adj[i] = new ArrayList<>();
        for (Edge e : edges) {
            adj[e.from].add(new Edge(e.from, e.to, e.weight));
            adj[e.to].add(new Edge(e.to, e.from, e.weight));
        }

        boolean[] visited = new boolean[n];
        PriorityQueue<Edge> pq = new PriorityQueue<>(Comparator.comparingInt(e -> e.weight));
        visited[0] = true;
        pq.addAll(adj[0]);

        int mstWeight = 0;

        while (!pq.isEmpty()) {
            Edge curr = pq.poll();
            if (visited[curr.to]) continue;
            visited[curr.to] = true;
            mstWeight += curr.weight;
            for (Edge next : adj[curr.to]) {
                if (!visited[next.to]) {
                    pq.offer(next);
                    /*
                     * [0 -> 1 : 10, 0 -> 2 : 6, 0 -> 3 : 5]
                     * [1 -> 0 : 10, 1 -> 3 : 15]
                     * [2 -> 0 : 6, 2 -> 3 : 4]
                     * [3 -> 0 : 5, 3 -> 1 : 15, 3 -> 2 : 4]
                     */
                }
            }
        }
        return mstWeight;
    }

    public static void main(String[] args) {
        List<Edge> edges = Arrays.asList(
            new Edge( from: 0, to: 1, weight: 10),
            new Edge( from: 0, to: 2, weight: 6),
            new Edge( from: 0, to: 3, weight: 5),
            new Edge( from: 1, to: 3, weight: 15),
            new Edge( from: 2, to: 3, weight: 4)
        );

        int n = 4; // number of nodes
        int result = primMST(n, edges);
        System.out.println("Minimum Spanning Tree Weight: " + result);
        /*
        For this input, MST edges will be:
        2-3 (4)
        0-3 (5)
        0-1 (10)
        Total Weight: 19
        */
    }
}
```

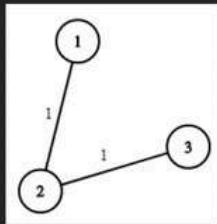
1168. Optimize Water Distribution in a Village

There are n houses in a village. We want to supply water for all the houses by building wells and laying pipes.

For each house i , we can either build a well inside it directly with cost $\text{wells}[i]$, or pipe in water from another well to it. The costs to lay pipes between houses are given by the array pipes , where each $\text{pipes}[i] = [\text{house}_1, \text{house}_2, \text{cost}]$ represents the cost to connect house_1 and house_2 together using a pipe. Connections are bidirectional.

Find the minimum total cost to supply water to all houses.

Example 1:



Input: $n = 3$, $\text{wells} = [1, 2, 2]$, $\text{pipes} = [[1, 2, 1], [2, 3, 1]]$

Output: 3

Explanation:

The image shows the costs of connecting houses using pipes.

The best strategy is to build a well in the first house with cost 1 and connect the other houses to it with cost 2 so the total cost is 3.

```
public class Solution {

    static class Edge {
        int from, to, cost;

        Edge(int from, int to, int cost) {
            this.from = from;
            this.to = to;
            this.cost = cost;
        }
    }

    public int minCostToSupplyWater(int n, int[] wells, int[][] pipes) {
        List<Edge>[] graph = new ArrayList[n + 1]; // 0 is virtual well node
        for (int i = 0; i <= n; i++) graph[i] = new ArrayList<>();

        // Connect virtual node 0 to each house with well cost
        for (int i = 0; i < n; i++) {
            graph[0].add(new Edge(0, i + 1, wells[i]));
            graph[i + 1].add(new Edge(i + 1, 0, wells[i]));
        }

        // Add all pipe connections
        for (int[] pipe : pipes) {
            int u = pipe[0], v = pipe[1], cost = pipe[2];
            graph[u].add(new Edge(u, v, cost));
            graph[v].add(new Edge(v, u, cost));
        }

        // Prim's Algorithm
        boolean[] visited = new boolean[n + 1];
        PriorityQueue<Edge> pq = new PriorityQueue<>(Comparator.comparingInt(e -> e.cost));
        visited[0] = true;
        pq.addAll(graph[0]);
        ...
```


Union Find

- Union-Find (or Disjoint Set Union) is a data structure that keeps track of a partition of a set into disjoint subsets.
- It supports two primary operations:
 - **Find:** Determine which subset a particular element is in. This can be used to determine if two elements are in the same subset.
 - **Union:** Join two subsets into a single subset.

Typical Uses in LeetCode Problems:

- **Connected Components:**
 - **Example:** Determine the number of connected components in a graph.
 - **Approach:** Use Union-Find to group connected nodes.
- **Cycle Detection:**
 - **Example:** Detect cycles in an undirected graph.
 - **Approach:** Use Union-Find to detect if adding an edge creates a cycle.
- **Kruskal's Algorithm:**
 - **Example:** Find the Minimum Spanning Tree (MST) of a graph.
 - **Approach:** Use Union-Find to efficiently check for cycles while adding edges in Kruskal's algorithm.

Advantages of Union-Find:

- **Efficiency:** Both the Find and Union operations can be performed in nearly constant time, $O(\alpha(N))$, where α is the inverse Ackermann function, which grows very slowly.
- **Simplicity:** The Union-Find data structure is simple to implement and use.

The Union-Find (also known as Disjoint Set Union or DSU) algorithm is a data structure that keeps track of elements partitioned into disjoint (non-overlapping) sets. It supports two primary operations efficiently:

Operations

1. **Find:** Determine which subset a particular element is in.
 - Used to check if two elements are in the same set.
 - Uses **path compression** to flatten the structure for fast future queries.
2. **Union:** Join two subsets into a single subset.
 - Uses **union by rank/size** to attach the smaller tree under the larger one, minimizing tree height.

In the Union-Find (Disjoint Set Union) data structure, the `parent` array is used to keep track of the representative (or root) of each set.

💡 Think of it like this:

Every element belongs to a set. Each set is organized like a **tree**, where:

- Every node points to its **parent**.
- The **root node** is the representative of the set, and it points to **itself** (i.e., `parent[x] == x`).

💡 Example:

Let's say we have `n = 5` elements: `{0, 1, 2, 3, 4}`

Initially:

```
java                                     ⌂ Copy ⌂ Edit  
parent = [0, 1, 2, 3, 4]; // each node is its own parent
```

Now suppose we do:

```
java
```

Copy Edit

```
union(0, 1);
```

We make one the parent of the other (e.g., 0 becomes the parent of 1):

```
java
```

Copy Edit

```
parent = [0, 0, 2, 3, 4]; // now, 1's parent is 0
```

This means:

- `find(1)` will go to `0` (the root/representative of the set that 1 belongs to).
- Elements `0` and `1` now belong to the same set.

In `find(x)`:

```
java
```

Copy Edit

```
while (parent[x] != x) {  
    x = parent[x];  
}
```

You climb up the parent chain until you reach the **root**.

If using **path compression**, you also **flatten** the tree by making each node directly point to the root:

```
java
```

Copy Edit

```
if (parent[x] != x) {  
    parent[x] = find(parent[x]); // recursively go to root and set parent  
}
```

Summary:

Term	Meaning
<code>parent[x]</code>	The immediate ancestor of node <code>x</code> . If <code>x</code> is the root of its set, <code>parent[x] == x</code> .
<code>find(x)</code>	Finds the root (representative) of the set containing <code>x</code> .
<code>union(x, y)</code>	Merges the sets containing <code>x</code> and <code>y</code> by updating parent pointers.

Operation	Description	Time Complexity
MakeSet	Initialize each node as its own root	$O(n)$
Find	Find set representative (with path compression)	$O(\alpha(n))$ amortized
Union	Merge two sets using rank/size	$O(\alpha(n))$ amortized

Union by Size

Instead of tracking the height (rank) of trees, we track the number of elements (size) in each set and always attach the smaller set under the larger one.

📦 Why use Union by Size?

- Prevents the tree from becoming too deep.
- Keeps find operations fast.
- Simpler to understand compared to rank in some cases.

```
public int find(int x) {  
    if (parent[x] != x) {  
        parent[x] = find(parent[x]); // Path compression  
    }  
    return parent[x];  
}  
  
public boolean union(int a, int b) {  
    int rootA = find(a);  
    int rootB = find(b);  
  
    if (rootA == rootB) return false; // already connected  
  
    // attach smaller tree under larger  
    if (size[rootA] < size[rootB]) {  
        parent[rootA] = rootB;  
        size[rootB] += size[rootA];  
    } else {  
        parent[rootB] = rootA;  
        size[rootA] += size[rootB];  
    }  
  
    return true;  
}
```

Initial:

```
makefile  
  
Sets: {0}, {1}, {2}, {3}, {4}  
parent: [0, 1, 2, 3, 4]  
size: [1, 1, 1, 1, 1]
```

After union(0, 1)

```
arduino  
  
root0 = 0, root1 = 1  
size[root0] = size[root1] = 1 → attach root1 to root0  
  
parent: [0, 0, 2, 3, 4]  
size: [2, 1, 1, 1, 1]
```

After union(0, 2)

```
matlab  
  
find(0) ~ 0, find(2) ~ 2  
size[root0] = 2, size[root2] = 1 → attach root2 to root0  
  
parent: [0, 0, 0, 3, 4]  
size: [3, 1, 1, 1, 1]
```

⌚ Time Complexity:

With both **path compression + union by size**, all operations are **nearly constant time**:

- **Amortized Time:** $O(\alpha(n))$ — inverse Ackermann function (very slow-growing)

Union by Rank

Union by Rank is an optimization used in the Disjoint Set Union (DSU) or Union-Find data structure to keep the trees shallow, which makes the find operation faster.

Each set is represented as a tree. When you do a union, you want the **shorter tree to be merged under the taller one**. This avoids making the resulting tree too deep.

Instead of measuring exact height (which is costly), we use a heuristic called **rank**:

- `rank[x]` is an upper bound on the height of the tree rooted at `x`.

⚡ How `unionByRank(x, y)` works:

1. Find roots of `x` and `y`:

```
java Copy Edit
int rootX = find(x);
int rootY = find(y);
```

2. If both elements are in the same set (i.e., same root), do nothing.

3. Otherwise:

- Attach the root with **smaller rank** to the root with **larger rank**.
- If ranks are equal, pick one as the new root and increment its rank.

```
boolean union(int x, int y) {
    int rootX = find(x);
    int rootY = find(y);

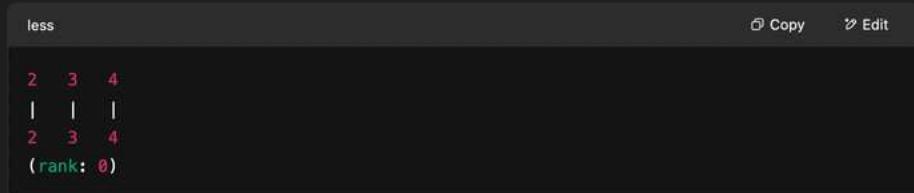
    if (rootX == rootY) return false; // already in the same set

    // Union by rank
    if (rank[rootX] < rank[rootY]) {
        parent[rootX] = rootY;
    } else if (rank[rootX] > rank[rootY]) {
        parent[rootY] = rootX;
    } else {
        parent[rootY] = rootX;
        rank[rootX]++;
    }

    return true;
}
```

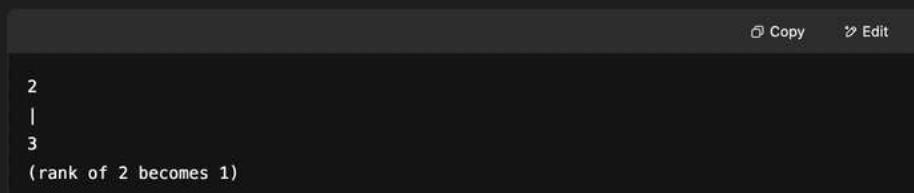
💡 Visual Example:

Before union(2, 3):



After union(2, 3):

- Let's say 2 becomes the parent:

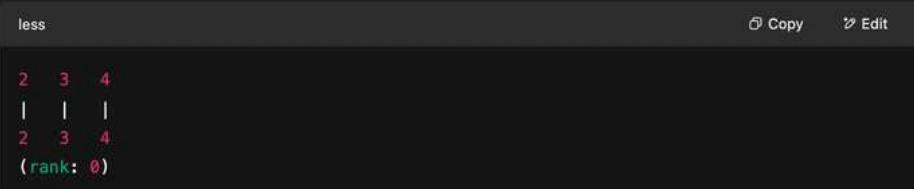


📌 Summary:

Term	Description
rank[x]	Approximate height of the tree rooted at <code>x</code>
Union by rank	Always attach the smaller tree under the larger one
Benefit	Keeps trees shallow → faster <code>find</code> calls

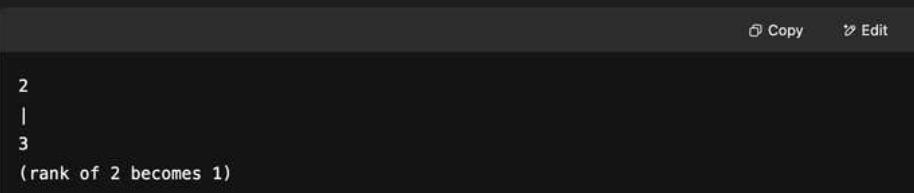
💡 Visual Example:

Before union(2, 3):



After union(2, 3):

- Let's say 2 becomes the parent:



Term	Description
rank[x]	Approximate height of the tree rooted at <code>x</code>
Union by rank	Always attach the smaller tree under the larger one
Benefit	Keeps trees shallow → faster <code>find</code> calls

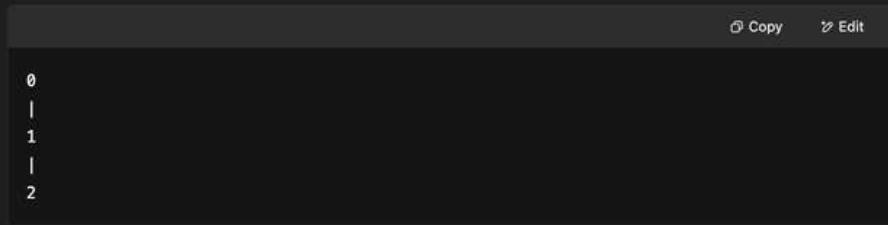
Height (used in Union by Rank):

- Refers to the **maximum depth** of the tree (i.e., the longest path from root to any leaf).
- We use `rank[x]` as a proxy for height.
- When we do `union`, we attach the **shorter tree under the taller one** to avoid increasing the height too much.

Size (used in Union by Size):

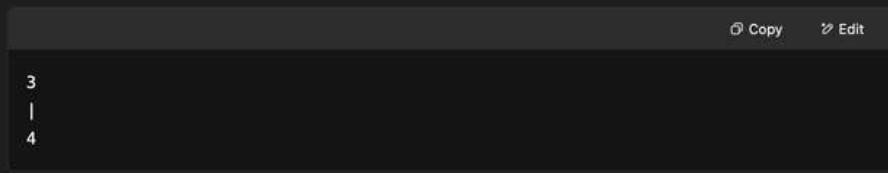
- Refers to the **total number of elements** in the set (i.e., size of the tree rooted at `x`).
- When doing `union`, we attach the **smaller set under the larger one**.

Tree A (rootA):



- Height = 2
- Size = 3

Tree B (rootB):



- Height = 1
- Size = 2

Now if you:

- Use Union by Height (Rank) → attach rootB under rootA (because height 2 > height 1)
- Use Union by Size → attach rootB under rootA (because size 3 > size 2)

In this case, both give the same result — but not always!

Cycle Detection

```
public boolean hasCycle(int n, List<List<Integer>> graph) {  
    int[] inDegree = new int[n];  
    for (List<Integer> neighbors : graph) {  
        for (int v : neighbors) {  
            inDegree[v]++;  
        }  
    }  
  
    Queue<Integer> q = new LinkedList<>();  
    for (int i = 0; i < n; i++) {  
        if (inDegree[i] == 0) q.offer(i);  
    }  
  
    int count = 0;  
    while (!q.isEmpty()) {  
        int node = q.poll();  
        count++;  
        for (int neighbor : graph.get(node)) {  
            if (--inDegree[neighbor] == 0) q.offer(neighbor);  
        }  
    }  
  
    return count != n; // If true, cycle exists  
}
```

Used in Kruskal's Algorithm:

```
java  
class UnionFind {  
    int[] parent;  
  
    public UnionFind(int n) {  
        parent = new int[n];  
        for (int i = 0; i < n; i++) parent[i] = i;  
    }  
  
    public int find(int u) {  
        if (parent[u] != u) parent[u] = find(parent[u]);  
        return parent[u];  
    }  
  
    public boolean union(int u, int v) {  
        int pu = find(u), pv = find(v);  
        if (pu == pv) return false; // cycle detected  
        parent[pu] = pv;  
        return true;  
    }  
}
```

Usage:

```
java  
for (int[] edge : edges) {  
    if (!uf.union(edge[0], edge[1])) {  
        System.out.println("Cycle Detected!");  
    }  
}
```

Cycle Detection

```
import java.util.*;

class UnionFind {
    int[] parent;
    int[] rank;

    public UnionFind(int n) {
        parent = new int[n];
        rank = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }

    public int find(int u) {
        if (parent[u] != u) {
            parent[u] = find(parent[u]);
        }
        return parent[u];
    }

    public boolean union(int u, int v) {
        int pu = find(u);
        int pv = find(v);

        if (pu == pv) {
            return false;
        }

        if (rank[pu] > rank[pv]) {
            parent[pv] = pu;
        } else if (rank[pu] < rank[pv]) {
            parent[pu] = pv;
        } else {
            parent[pu] = pv;
            rank[pv]++;
        }
        return true;
    }
}

public class CycleDetection {
    public static void main(String[] args) {
        int n = 4;
        List<int[]> edges = Arrays.asList(
            new int[]{0, 1},
            new int[]{1, 2},
            new int[]{2, 3},
            new int[]{3, 0}
        );
        UnionFind uf = new UnionFind(n);

        for (int[] edge : edges) {
            int u = edge[0];
            int v = edge[1];
            if (!uf.union(u, v)) {
                System.out.println("Cycle Detected!");
                return;
            }
        }
        System.out.println("No Cycle Detected.");
    }
}
```

261. Graph Valid Tree

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 make up a valid tree.

Example 1:

```
Input: n = 5, and edges = [[0,1], [0,2], [0,3], [1,4]]  
Output: true
```

Example 2:

```
Input: n = 5, and edges = [[0,1], [1,2], [2,3], [1,3], [1,4]]  
Output: false
```

Note: you can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, `[0,1]` is the same as `[1,0]` and thus will not appear together in `edges`.

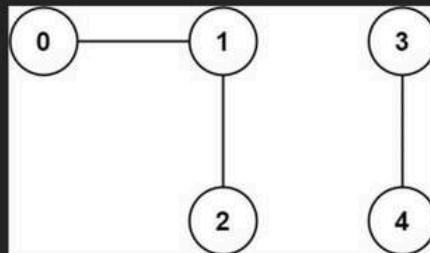
```
public class Solution {  
  
    static class UnionFind {  
        int[] parent;  
        int[] rank;  
  
        public UnionFind(int n) {  
            parent = new int[n];  
            rank = new int[n];  
            for (int i = 0; i < n; i++) {  
                parent[i] = i;  
                rank[i] = 0;  
            }  
        }  
  
        public int find(int x) {  
            if (parent[x] != x) {  
                parent[x] = find(parent[x]);  
            }  
            return parent[x];  
        }  
  
        public boolean union(int x, int y) {  
            int rootX = find(x);  
            int rootY = find(y);  
  
            if (rootX == rootY) {  
                return false; // Cycle detected  
            }  
  
            if (rank[rootX] > rank[rootY]) {  
                parent[rootY] = rootX;  
            } else if (rank[rootX] < rank[rootY]) {  
                parent[rootX] = rootY;  
            } else {  
                parent[rootX] = rootY;  
            }  
            return true;  
        }  
    }  
}
```


323. Number of Connected Components in an Undirected Graph

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 find the number of connected components in an undirected graph.

Example 1:

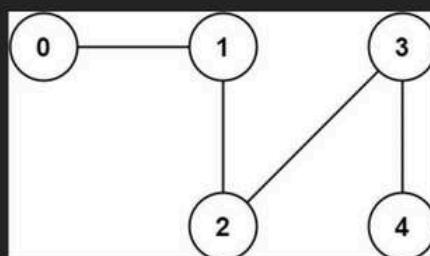
Input: $n = 5$ and $\text{edges} = [[0, 1], [1, 2], [3, 4]]$



Output: 2

Example 2:

Input: $n = 5$ and $\text{edges} = [[0, 1], [1, 2], [2, 3], [3, 4]]$



Output: 1

Note:

You can assume that no duplicate edges will appear in `edges`. Since all edges are undirected, $[0, 1]$ is the same as $[1, 0]$ and thus will not appear together in `edges`.

```
public class Solution {
    static class UnionFind {
        int[] parent;
        int[] rank;

        public UnionFind(int n) {
            parent = new int[n];
            rank = new int[n];
            for (int i = 0; i < n; i++) {
                parent[i] = i;
                rank[i] = 0;
            }
        }

        public int find(int x) {
            if (parent[x] != x) {
                parent[x] = find(parent[x]); // Path compression
            }
            return parent[x];
        }

        public void union(int x, int y) {
            int rootX = find(x);
            int rootY = find(y);

            if (rootX != rootY) {
                if (rank[rootX] > rank[rootY]) {
                    parent[rootY] = rootX;
                } else if (rank[rootX] < rank[rootY]) {
                    parent[rootX] = rootY;
                } else {
                    parent[rootX] = rootY;
                    rank[rootY]++;
                }
            }
        }
    }
}
```

```
public int countComponents(int n, int[][] edges) {
    UnionFind uf = new UnionFind(n);

    for (int[] edge : edges) {
        int u = edge[0];
        int v = edge[1];
        uf.union(u, v);
    }

    // Count the number of unique root nodes (i.e., the number of connected components)
    int count = 0;
    for (int i = 0; i < n; i++) {
        if (uf.find(i) == i) {
            count++;
        }
    }

    return count;
}

public static void main(String[] args) {
    Solution solution = new Solution();

    // Test Case 1
    int n = 5;
    int[][] edges = {{0, 1}, {1, 2}, {3, 4}};
    System.out.println(solution.countComponents(n, edges)); // Output: 2

    // Test Case 2
    n = 4;
    edges = new int[][]{{0, 1}, {2, 3}};
    System.out.println(solution.countComponents(n, edges)); // Output: 2
```

different roots

Shortest Path Algorithms

The "Shortest Path" is a common algorithmic problem that aims to find the shortest distance or minimum number of steps required to travel between two points, usually in a graph or grid. This concept is widely used in many LeetCode problems.

Key Concepts:

• **Graph Representation:**

- Graphs can be represented using adjacency lists, adjacency matrices, or edge lists.
- Nodes (vertices) and edges (connections) form the basis of the graph.

• **Common Algorithms:**

• **Breadth-First Search (BFS):**

- Used for unweighted graphs or grids.
- Explores all neighbors at the present depth before moving on to nodes at the next depth level.
- Guarantees finding the shortest path in terms of the number of edges (or steps).

• **Dijkstra's Algorithm:**

- Used for weighted graphs.
- Finds the shortest path from a source node to all other nodes.
- Maintains a priority queue to explore the next closest node.

• **Floyd-Warshall Algorithm:**

- Used for finding shortest paths in graphs with weighted edges (positive or negative).
- Computes shortest paths between all pairs of nodes.
- Dynamic programming approach that iteratively updates shortest path estimates.

• **A*:**

- Used for weighted graphs, similar to Dijkstra's but with a heuristic to improve efficiency.
- Combines the cost to reach a node and the estimated cost to the goal.

Dijkstra (Single Source Shortest Path)

Time Complexity

$$O((V + E) \log V)$$

Where:

- V = number of vertices
- E = number of edges

Why?

- Each node is inserted into the priority queue **once**, and each edge is considered at most once.
- Operations on the priority queue (`offer`, `poll`) take $O(\log V)$ time.
- So:
 - Visiting all vertices: $O(V \log V)$
 - Relaxing all edges: $O(E \log V)$ (since each edge might cause a new insert)

→ Together: $O((V + E) \log V)$

Space Complexity

$$O(V + E)$$

Because:

- `dist[]` array takes $O(V)$
- The graph is stored as an **adjacency list**: $O(E)$ for storing edges
- Priority queue may contain up to $O(V)$ elements at any time

```

public class Solution {

    static class Pair {
        int node, weight;
        Pair(int node, int weight) {
            this.node = node;
            this.weight = weight;
        }
    }

    public static int[] dijkstra(int V, List<List<Pair>> graph, int source) {
        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[source] = 0;

        PriorityQueue<Pair> pq = new PriorityQueue<>(Comparator.comparingInt(p -> p.weight));
        pq.offer(new Pair(source, weight: 0));

        while (!pq.isEmpty()) {
            Pair current = pq.poll();
            int u = current.node;
            int d = current.weight;

            if (d > dist[u]) continue; // Skip if we already found a better path

            for (Pair neighbor : graph.get(u)) {
                int v = neighbor.node;
                int weight = neighbor.weight;

                if (dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    pq.offer(new Pair(v, dist[v]));
                }
            }
        }

        return dist;
    }

    public static void main(String[] args) {
        int V = 5; // Number of vertices
        List<List<Pair>> graph = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            graph.add(new ArrayList<>());
        }

        // Add edges: graph.get(from).add(new Pair(to, weight));
        graph.get(0).add(new Pair(node: 1, weight: 4));
        graph.get(0).add(new Pair(node: 2, weight: 1));
        graph.get(2).add(new Pair(node: 1, weight: 2));
        graph.get(1).add(new Pair(node: 3, weight: 1));
        graph.get(2).add(new Pair(node: 3, weight: 5));
        graph.get(3).add(new Pair(node: 4, weight: 3));

        int[] dist = dijkstra(V, graph, source: 0); // source = 0

        System.out.println("Shortest distances from source 0:");
        for (int i = 0; i < V; i++) {
            System.out.println("To node " + i + ": " + dist[i]);
        }
    }
}

```

Shortest distances from source 0:

```

To node 0: 0
To node 1: 3
To node 2: 1
To node 3: 4
To node 4: 7
*/

```

Aspect	Complexity
Time	$O((V + E) \log V)$
Space	$O(V + E)$

Bellman-Ford

The Bellman-Ford algorithm is a single-source shortest path algorithm used to compute the shortest paths from a given source vertex to all other vertices in a weighted directed graph. Unlike Dijkstra's algorithm, Bellman-Ford can handle graphs with negative weight edges.

The Bellman-Ford Algorithm is used for single-source shortest paths, just like Dijkstra's – but with the added advantage of handling negative edge weights.

✓ When to Use Bellman-Ford

- Graph has negative edge weights
- Need to detect negative weight cycles
-

The Bellman-Ford algorithm relaxes all the edges $V - 1$ times, where V is the number of vertices. During each pass, it updates the shortest known distance to each vertex by considering all edges. If after $V - 1$ passes, a further relaxation is possible, it indicates the presence of a negative weight cycle.

🧠 Key Idea

1. Initialize distances from the source to all vertices as infinity (∞) except the source itself (set to 0).
2. Relax all edges $V-1$ times: For each edge $(u \rightarrow v)$ with weight w , update $dist[v]$ if $dist[u] + w < dist[v]$.
3. After $V-1$ passes, run one more pass to check for negative cycles — if a distance still improves, there's a cycle.

⌚ Relaxation Rule

For each edge $u \rightarrow v$ with weight w :

```
java
if (dist[u] + w < dist[v]) {
    dist[v] = dist[u] + w;
}
```

📌 Summary Table

Feature	Bellman-Ford
Handles negative weights	✓
Detects negative cycles	✓
Time Complexity	$O(V * E)$
Space Complexity	$O(V)$
Greedy or DP?	Dynamic Programming (relaxation approach)

```

public class Solution {

    static class Edge {
        int from, to, weight;
        Edge(int from, int to, int weight) {
            this.from = from;
            this.to = to;
            this.weight = weight;
        }
    }

    public static int[] bellmanFord(int V, List<Edge> edges, int source) {
        int[] dist = new int[V];
        Arrays.fill(dist, Integer.MAX_VALUE);
        dist[source] = 0;

        // Relax edges V - 1 times
        for (int i = 0; i < V - 1; i++) {
            for (Edge e : edges) {
                if (dist[e.from] != Integer.MAX_VALUE && dist[e.from] + e.weight < dist[e.to]) {
                    dist[e.to] = dist[e.from] + e.weight;
                }
            }
        }

        // Check for negative weight cycles
        for (Edge e : edges) {
            if (dist[e.from] != Integer.MAX_VALUE && dist[e.from] + e.weight < dist[e.to]) {
                System.out.println("Negative weight cycle detected!");
                return null;
            }
        }
    }

    return dist;
}

public static void main(String[] args) {
    int V = 5;
    List<Edge> edges = new ArrayList<>();

    // Add edges: new Edge(from, to, weight)
    edges.add(new Edge( from: 0, to: 1, weight: -1));
    edges.add(new Edge( from: 0, to: 2, weight: 4));
    edges.add(new Edge( from: 1, to: 2, weight: 3));
    edges.add(new Edge( from: 1, to: 3, weight: 2));
    edges.add(new Edge( from: 1, to: 4, weight: 2));
    edges.add(new Edge( from: 3, to: 2, weight: 5));
    edges.add(new Edge( from: 3, to: 1, weight: 1));
    edges.add(new Edge( from: 4, to: 3, weight: -3));

    int[] dist = bellmanFord(V, edges, source: 0);

    if (dist != null) {
        System.out.println("Shortest distances from source 0:");
        for (int i = 0; i < V; i++) {
            System.out.println("To node " + i + ": " + dist[i]);
        }
    }
}
}

```

Imagine you're trying to track the shortest time to complete a set of tasks (represented as nodes). If you relax the time estimates (by considering the edges), after you've done this for $V-1$ times, all the shortest paths should be found. But if after V times you can still reduce the time for some task, it means you've entered a situation where tasks keep taking negative time, representing a cycle that allows you to keep going back in time.

A negative weight cycle exists in the graph if you can still relax an edge after the $V-1$ iterations. This is because, in a graph with no negative cycles, the shortest path between any two nodes is always found in at most $V-1$ relaxations.

Output Example

```

vbnet
Shortest distances from source 0:
To node 0: 0
To node 1: -1
To node 2: 2
To node 3: -2
To node 4: 1

```

Complexity

Aspect	Value
Time	$O(V * E)$
Space	$O(V)$
Handles Neg Weights	✓
Detects Neg Cycles	✓

A*

Find the shortest path from a start node to a target node, using both:

- **Actual cost from the start** → current node ($g(n)$)
- **Heuristic estimate from current node** → target ($h(n)$)

It chooses the path that minimizes $f(n) = g(n) + h(n)$.

12 Components

$g(n)$ = Actual cost from the start to node n

$h(n)$ = Estimated cost from n to goal (heuristic)

$f(n) = g(n) + h(n)$ → total estimated cost through n

The heuristic $h(n)$ should be:

- Admissible (never overestimates)
- Common choices: Manhattan distance, Euclidean, etc.

Pseudocode

```
text
Copy Edit
1. Add start node to open set with  $f(\text{start}) = g(\text{start}) + h(\text{start})$ 
2. While open set is not empty:
   a. Pick node with lowest  $f(n)$ 
   b. If node is goal, reconstruct path
   c. Else, for each neighbor:
      - Calculate tentative  $g(n)$ 
      - If better path, update  $g(n)$ ,  $f(n)$ , and parent
3. If goal not found, return failure
```

```
package com.sai.designPatterns.shortestPath;

import java.util.*;

class Node {
    int row, col;
    int costFromStart; // g(n)
    int estimatedToGoal; // h(n)
    int totalEstimatedCost; // f(n) = g + h
    Node parent;
}

Node(int row, int col, int costFromStart, int estimatedToGoal, Node parent) {
    this.row = row;
    this.col = col;
    this.costFromStart = costFromStart;
    this.estimatedToGoal = estimatedToGoal;
    this.totalEstimatedCost = costFromStart + estimatedToGoal;
    this.parent = parent;
}

public class AStarPathfinder {
    // Directions: right, down, left, up.
    static final int[][] DIRECTIONS = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    public static List<int[]> findPath(int[][] grid, int[] start, int[] goal) {
        int rows = grid.length, cols = grid[0].length;
        boolean[][] visited = new boolean[rows][cols];

        PriorityQueue<Node> openSet = new PriorityQueue<>(Comparator.comparingInt(node -> node.totalEstimatedCost));
        Node startNode = new Node(start[0], start[1], costFromStart: 0, calculateHeuristic(start, goal), parent: null);
        openSet.offer(startNode);
```

```

while (!openSet.isEmpty()) {
    Node current = openSet.poll();

    if (current.row == goal[0] && current.col == goal[1]) {
        return reconstructPath(current);
    }

    if (visited[current.row][current.col]) continue;
    visited[current.row][current.col] = true;

    for (int[] direction : DIRECTIONS) {
        int newRow = current.row + direction[0];
        int newCol = current.col + direction[1];

        if (isValidCell(newRow, newCol, grid, visited)) {
            int newCostFromStart = current.costFromStart + 1;
            int estimatedToGoal = calculateHeuristic(new int[]{newRow, newCol}, goal);
            Node neighbor = new Node(newRow, newCol, newCostFromStart, estimatedToGoal, current);
            openSet.offer(neighbor);
        }
    }
}

return Collections.emptyList(); // No path found
}

private static boolean isValidCell(int row, int col, int[][] grid, boolean[][] visited) {
    return row >= 0 && col >= 0 && row < grid.length && col < grid[0].length
        && grid[row][col] == 0 && !visited[row][col];
}

private static int calculateHeuristic(int[] from, int[] to) {
    // Manhattan distance heuristic
    return Math.abs(from[0] - to[0]) + Math.abs(from[1] - to[1]);
}

private static List<int[]> reconstructPath(Node node) {
    List<int[]> path = new ArrayList<>();
    while (node != null) {
        path.add(new int[]{node.row, node.col});
        node = node.parent;
    }
    Collections.reverse(path);
    return path;
}

public static void main(String[] args) {
    int[][] grid = {
        {0, 0, 0, 0},
        {1, 1, 0, 1},
        {0, 0, 0, 0},
        {0, 1, 1, 0},
        {0, 0, 0, 0}
    };

    int[] start = {0, 0};
    int[] goal = {4, 3};

    List<int[]> shortestPath = findPath(grid, start, goal);
    for (int[] cell : shortestPath) {
        System.out.println(Arrays.toString(cell));
    }
}
}

```

✓ Ideal Use Cases:

- Grid maps
- Game AI
- Pathfinding with weighted heuristics
- Real-world maps (Google Maps-style shortest path)

Floyd Warshall (All pairs shortest path)

The **Floyd-Warshall algorithm** is a dynamic programming-based algorithm used to find the **shortest paths** between all pairs of vertices in a weighted graph. It is particularly useful when you need to compute the shortest paths between **every pair of vertices** in a graph, including handling negative weights (but not negative weight cycles).

Time Complexity:

- **Time Complexity:** $O(V^3)$ where V is the number of vertices.
- **Space Complexity:** $O(V^2)$ for storing the distance matrix.

Algorithm Explanation:

The algorithm uses a **dynamic programming** approach to update a distance matrix. The key idea is that the shortest path between two nodes i and j can either go directly from i to j , or it can go through an intermediate node k . This allows us to iteratively refine the shortest path between every pair of nodes.

1. Initialization:

- Create a distance matrix `dist[][]` where `dist[i][j]` is initialized to:
 - 0 if $i = j$,
 - The weight of the edge from i to j if there is an edge,
 - ∞ (or a large number) if there is no edge between i and j .

2. Iterative Relaxation:

- For each possible intermediate vertex k , update the shortest paths between all pairs of vertices i and j .
- For each pair of vertices (i, j) , check if the path from i to j through vertex k is shorter than the direct path from i to j :

$$\text{dist}[i][j] = \min(\text{dist}[i][j], \text{dist}[i][k] + \text{dist}[k][j])$$

```
public class Solution {

    static class Edge {
        int from, to, weight;

        Edge(int from, int to, int weight) {
            this.from = from;
            this.to = to;
            this.weight = weight;
        }
    }

    public static void floydWarshall(int V, List<Edge> edges) {
        // Distance matrix initialized to infinity
        int[][] dist = new int[V][V];

        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                if (i == j) {
                    dist[i][j] = 0; // Distance to itself is 0
                } else {
                    dist[i][j] = Integer.MAX_VALUE / 2; // Use a large number for infinity
                }
            }
        }

        for (Edge edge : edges) {
            dist[edge.from][edge.to] = edge.weight;
        }
    }
}
```


743. Network Delay Time

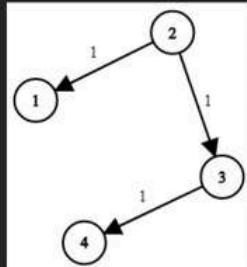
743. Network Delay Time

Medium

You are given a network of n nodes, labeled from 1 to n . You are also given times , a list of travel times as directed edges $\text{times}[i] = (u_i, v_i, w_i)$, where u_i is the source node, v_i is the target node, and w_i is the time it takes for a signal to travel from source to target.

We will send a signal from a given node K . Return the minimum time it takes for all the n nodes to receive the signal. If it is impossible for all the n nodes to receive the signal, return -1 .

Example 1:



Input: $\text{times} = [[2,1,1], [2,3,1], [3,4,1]]$, $n = 4$, $K = 2$
Output: 2

Example 2:

Input: $\text{times} = [[1,2,1]]$, $n = 2$, $K = 1$
Output: 1

Example 3:

Input: $\text{times} = [[1,2,1]]$, $n = 2$, $K = 2$
Output: -1

```
public int networkDelayTime(int[][] times, int totalNodes, int sourceNode) {
    // Build adjacency list: node -> list of [neighbor, time]
    Map<Integer, List<int[]>> adjacencyMap = new HashMap<>();
    for (int[] edge : times) {
        int from = edge[0], to = edge[1], weight = edge[2];
        adjacencyMap.computeIfAbsent(from, k -> new ArrayList<>()).add(new int[]{to, weight});
    }

    // Min-heap priority queue: [currentTime, currentNode]
    PriorityQueue<int[]> minHeap = new PriorityQueue<>(Comparator.comparingInt(a -> a[0]));
    minHeap.offer(new int[]{0, sourceNode});

    Map<Integer, Integer> shortestTimeToNode = new HashMap<>();

    while (!minHeap.isEmpty()) {
        int[] current = minHeap.poll();
        int currentTime = current[0];
        int currentNode = current[1];

        if (shortestTimeToNode.containsKey(currentNode)) continue;

        shortestTimeToNode.put(currentNode, currentTime);

        if (adjacencyMap.containsKey(currentNode)) {
            for (int[] neighbor : adjacencyMap.get(currentNode)) {
                int neighborNode = neighbor[0];
                int travelTime = neighbor[1];

                if (!shortestTimeToNode.containsKey(neighborNode)) {
                    minHeap.offer(new int[]{currentTime + travelTime, neighborNode});
                }
            }
        }
    }
}
```

```

        // If we couldn't reach all nodes
        if (shortestTimeToNode.size() != totalNodes) {
            return -1;
        }

        int maxTime = 0;
        for (int time : shortestTimeToNode.values()) {
            maxTime = Math.max(maxTime, time);
        }

        return maxTime;
    }

    public static void main(String[] args) {
        Solution solver = new Solution();
        int[][] times = {
            {2, 1, 1},
            {2, 3, 1},
            {3, 4, 1}
        };
        int totalNodes = 4;
        int sourceNode = 2;

        int result = solver.networkDelayTime(times, totalNodes, sourceNode);
        System.out.println("Minimum time for all nodes to receive signal: " + result);
    }
}

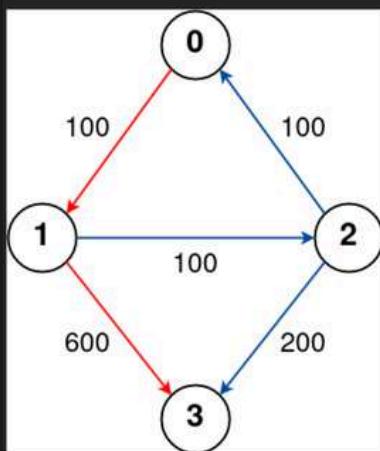
```

787. Cheapest Flights Within K Stops

There are n cities connected by some number of flights. You are given an array `flights` where `flights[i] = [fromi, toi, pricei]` indicates that there is a flight from city `fromi` to city `toi` with cost `pricei`.

You are also given three integers `src`, `dst`, and `k`, return *the cheapest price from `src` to `dst` with at most `k` stops*. If there is no such route, return `-1`.

Example 1:



Input: $n = 4$, `flights = [[0,1,100],[1,2,100],[2,0,100],[1,3,600],[2,3,200]], src = 0, dst = 3, k = 1`

Output: 700

Explanation:

The graph is shown above.

The optimal path with at most 1 stop from city 0 to 3 is marked in red and has cost $100 + 600 = 700$.

Note that the path through cities $[0,1,2,3]$ is cheaper but is invalid because it uses 2 stops.

```

public class Solution {

    static class Route {
        int city, cost, stops;
        Route(int city, int cost, int stops) {
            this.city = city;
            this.cost = cost;
            this.stops = stops;
        }
    }

    public int findCheapestPrice(int n, int[][] flights, int src, int dst, int k) {
        // Build graph: from > list of Route(to, cost, stops=0 not used in graph)
        Map<Integer, List<Route>> graph = new HashMap<>();
        for (int[] flight : flights) {
            int from = flight[0], to = flight[1], price = flight[2];
            graph.computeIfAbsent(from, x -> new ArrayList<>()).add(new Route(to, price, stops: 0));
        }

        // Min-heap sorted by cost
        PriorityQueue<Route> pq = new PriorityQueue<>(Comparator.comparingInt(r -> r.cost));
        pq.offer(new Route(src, cost: 0, stops: 0));

        // Track best known cost to reach (city, stops)
        Map<String, Integer> visited = new HashMap<>();

        while (!pq.isEmpty()) {
            Route current = pq.poll();

            if (current.city == dst) return current.cost;

            String key = current.city + "," + current.stops;
            if (visited.containsKey(key) && visited.get(key) <= current.cost) continue;
            visited.put(key, current.cost);

            if (current.stops > k) continue;

            if (graph.containsKey(current.city)) {
                for (Route neighbor : graph.get(current.city)) {
                    pq.offer(new Route(neighbor.city, cost: current.cost + neighbor.cost, stops: current.stops + 1));
                }
            }
        }

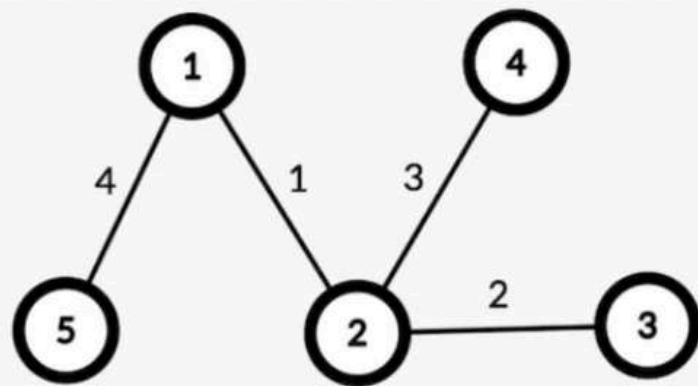
        return -1;
    }

    public static void main(String[] args) {
        Solution solver = new Solution();
        int[][] flights = {
            {0, 1, 100},
            {1, 2, 100},
            {2, 3, 100},
            {0, 2, 500}
        };
        int n = 4, src = 0, dst = 3, k = 1;

        int result = solver.findCheapestPrice(n, flights, src, dst, k);
        System.out.println("Cheapest price: " + result); // Output: 600
    }
}

```

viii. A travel agent sends couple's from one city to another via the shortest path possible. If there are N cities and N - 1 highways connecting them. Help him plan the trip for all the couples such that he maximizes the total distance travelled by all couples. You must return to the same node from where you've originated.



Using a PriorityQueue for this problem is a bit unusual since we're working with a tree and we're trying to maximize the total round-trip distance of couples – not finding a shortest path from a single source like in Dijkstra's.

```

1 class Graph {
2     constructor(n, v) {
3         this.n = n
4         this.g = {}
5     }
6     addVertex(i) {
7         this.g[i] = {}
8     }
9     addEdge(u, v, w) {
10        this.g[u][v] = w
11        this.g[v][u] = w
12    }
13 }
14 paths(source, destination) {
15     let result = []
16     const curr = []
17     curr.push(source)
18     this.pathsUsingDfs(source, destination, result, curr)
19     return Math.min(...result)
20 }

```

```

1 {
2 '1': { '2': 3 },
3 '2': { '1': 3, '3': 2 },
4 '3': { '2': 2, '4': 2 },
5 '4': { '3': 2 }
6 }
7 18
8 {
9 '1': { '2': 1, '5': 4 },
10 '2': { '1': 1, '3': 2, '4': 3 },
11 '3': { '2': 2 },
12 '4': { '2': 3 },
13 '5': { '1': 4 }
14 }
15 22

```

```

pathsUsingDfs(s, d, result, curr = [], visited = {}) {
    if (s === d) {
        let path = curr.slice()
        let max = 0
        for (let i = 0; i < path.length; i++) {
            if (this.g[path[i + 1]]) {
                max += this.g[path[i + 1]][path[i]]
            }
        }
        result.push(max)
        return
    }
    visited[s] = true
    const list = Object.keys(this.g[s]).map(x => +x)
    for (let i of list) {
        if (!visited[i]) {
            curr.push(i)
            this.pathsUsingDfs(i, d, result, curr, visited)
            curr.splice(curr.indexOf(i), 1)
        }
    }
}

```

```

visited[s] = false
}

solve() {
    let entriesL = Object.entries(this.g).map(x => [+x[0], Object.entries(x[1]).length])
    const maxN = Math.max(...entriesL.map(x => x[1]))
    entriesL = entriesL.filter(e => e[1] === maxN)[0]
    const source = entriesL[0]
    let max = 0
    const neighbors = Object.entries(this.g[source])
    const keys = Object.keys(this.g[source]).map(x => +x)
    for (let n of neighbors) {
        max += n[1]
    }
    const others = [...Array(this.n).keys()].map(x => +x + 1).filter(x => !keys.includes(x))
    for (let i of others) {
        max += this.paths(source, i)
    }
    return max * 2
}

function solution(n, distances) {
    const g = new Graph(n)
    for (let i = 1; i <= n; i++)
        g.addVertex(i)
    for (let d of distances) {
        g.addEdge(d[0], d[1], d[2])
    }
    console.log(g.g)
    return g.solve()
}

console.log(solution(4, [
    [1, 2, 3],
    [2, 3, 2],
    [4, 3, 2],
    [1, 2, 1]
]));
32
33 console.log(solution(5, [
    [1, 2, 1],
    [2, 3, 1],
    [2, 4, 3],
    [1, 5, 4]
]));

```

```

public class MaximumCoupleTravelDistance {

    static int numberOfCities = 5;
    static List<List<int[]>> adjacencyList = new ArrayList<>();
    static int[][] shortestDistances = new int[numberOfCities + 1][numberOfCities + 1];

    public static void main(String[] args) {
        // Initialize graph
        for (int i = 0; i <= numberOfCities; i++) {
            adjacencyList.add(new ArrayList<>());
        }

        // Build the undirected tree
        addRoad( cityA: 1, cityB: 2, distance: 1);
        addRoad( cityA: 2, cityB: 3, distance: 2);
        addRoad( cityA: 2, cityB: 4, distance: 3);
        addRoad( cityA: 1, cityB: 5, distance: 4);

        // Step 1: Compute shortest distances from every city to every other city
        for (int city = 1; city <= numberOfCities; city++) {
            computeShortestPathsFrom(city, shortestDistances[city]);
        }

        // Step 2: Store all possible city pairs with distances in a max heap
        PriorityQueue<int[]> maxDistancePairs = new PriorityQueue<>(
            (a, b) -> b[0] - a[0] // Sort by descending distance
        );

        for (int cityA = 1; cityA <= numberOfCities; cityA++) {
            for (int cityB = cityA + 1; cityB <= numberOfCities; cityB++) {
                int distance = shortestDistances[cityA][cityB];
                maxDistancePairs.offer(new int[]{distance, cityA, cityB});
            }
        }

        // Step 3: Greedily pair cities for maximum round-trip distance
        boolean[] isCityUsed = new boolean[numberOfCities + 1];
        int totalRoundTripDistance = 0;
        int pairsFormed = 0;

        while (!maxDistancePairs.isEmpty() && pairsFormed < numberOfCities / 2) {
            int[] topPair = maxDistancePairs.poll();
            int distance = topPair[0];
            int cityU = topPair[1];
            int cityV = topPair[2];

            if (!isCityUsed[cityU] && !isCityUsed[cityV]) {
                isCityUsed[cityU] = true;
                isCityUsed[cityV] = true;
                totalRoundTripDistance += 2 * distance; // Round-trip
                pairsFormed++;
            }
        }

        System.out.println("Maximum Total Round-Trip Distance: " + totalRoundTripDistance);
    }

    static void addRoad(int cityA, int cityB, int distance) {
        adjacencyList.get(cityA).add(new int[]{cityB, distance});
        adjacencyList.get(cityB).add(new int[]{cityA, distance});
    }
}

```

```
static void computeShortestPathsFrom(int startCity, int[] distanceArray) {
    Arrays.fill(distanceArray, Integer.MAX_VALUE);
    Queue<int[]> queue = new LinkedList<>() ;
    distanceArray[startCity] = 0;
    queue.offer(new int[]{startCity, 0});

    while (!queue.isEmpty()) {
        int[] current = queue.poll();
        int currentCity = current[0];
        int currentDistance = current[1];

        for (int[] neighbor : adjacencyList.get(currentCity)) {
            int neighborCity = neighbor[0];
            int roadLength = neighbor[1];

            if (distanceArray[neighborCity] > currentDistance + roadLength) {
                distanceArray[neighborCity] = currentDistance + roadLength;
                queue.offer(new int[]{neighborCity, distanceArray[neighborCity]});
            }
        }
    }
}
```

```
[[0, 0, 0, 0, 0, 0], [2147483647, 0, 1, 3, 4, 4], [2147483647, 1, 0, 2, 3, 5], [2147483647, 3, 2, 0, 5, 7], [2147483647, 4, 3, 5, 0, 8],
 [2147483647, 4, 5, 7, 8, 0]]
[[I@3e3abc88, [I@6ce253f1, [I@53d8d10a, [I@e9e54c2, [I@65ab7765, [I@1b28cd8a, [I@eed1f14, [I@7229724f, [I@4c873330, [I@119d7047]
Maximum Total Round-Trip Distance: 22
```

Strongly Connected Components

In graph theory, a strongly connected component (SCC) of a directed graph is a maximal strongly connected subgraph. That is, a subgraph where there is a path from each vertex to every other vertex within the subgraph. SCCs are significant in various applications like understanding the structure of networks, optimizing compilers, and analyzing social networks.

Kosaraju's Algorithm:

Simpler to understand and implement.

Requires two passes of the graph (two DFS traversals).

Uses an auxiliary stack to record finishing order.

Tarjan's Algorithm:

More efficient with a single pass of the graph.

Uses low-link values to identify SCCs.

Slightly more complex due to the management of low-link values and the stack.

Both algorithms have a time complexity of $O(V+E)$ and are efficient for large graphs. The choice depends on the specific problem constraints and personal preference for implementation complexity.

A **strongly connected component (SCC)** in a directed graph is a maximal subgraph (a subset of the graph's vertices and edges) where every vertex is reachable from every other vertex in that subgraph.

Key Characteristics of SCC:

- **Reachability:** In a strongly connected component, for every pair of vertices u and v , there is a directed path from u to v and a directed path from v to u .
- **Maximal:** A strongly connected component is as large as possible, meaning that if you add any vertex or edge, it will no longer maintain strong connectivity.
- **Subgraph:** An SCC is a subgraph where all the vertices are mutually reachable.

Kosaraju Algorithm:

The Kosaraju algorithm is a well-known method for finding strongly connected components (SCCs) in a directed graph. It uses two depth-first search (DFS) traversals, and it operates in linear time, i.e., $O(V + E)$, where V is the number of vertices and E is the number of edges.

Here's a basic implementation of Kosaraju's SCC algorithm in Java:

Steps of Kosaraju's Algorithm:

1. **First DFS:** Perform a DFS on the original graph and store the vertices in a stack according to the finish time (i.e., when a vertex is completely processed, push it to the stack).
2. **Transpose the Graph:** Reverse the direction of all edges in the graph.
3. **Second DFS:** Pop vertices from the stack one by one and perform DFS on the transposed graph, starting from the vertex popped, marking all reachable nodes in that DFS as part of the same SCC.

DfS => reverse order => DFS on reversed graph using stack

```

// Define the Node class with a value and a list of neighbors
static class Node {
    int val;
    List<Node> neighbors;

    Node(int val) {
        this.val = val;
        this.neighbors = new ArrayList<>();
    }
}

static class Graph {
    int V;
    Node[] nodes;

    // Constructor
    Graph(int V) {
        this.V = V;
        nodes = new Node[V];
        for (int i = 0; i < V; i++) {
            nodes[i] = new Node(i);
        }
    }

    // Add edge to the graph
    void addEdge(int u, int v) {
        nodes[u].neighbors.add(nodes[v]);
    }

    // DFS function to perform DFS traversal
    void dfs(Node node, boolean[] visited, Stack<Node> stack) {
        visited[node.val] = true;
        for (Node neighbor : node.neighbors) {
            if (!visited[neighbor.val]) {
                dfs(neighbor, visited, stack);
            }
        }
        stack.push(node); // Push the node after processing all its neighbors
    }

    // Transpose the graph
    Graph getTranspose() {
        Graph transpose = new Graph(V);
        for (int u = 0; u < V; u++) {
            for (Node neighbor : nodes[u].neighbors) {
                transpose.addEdge(neighbor.val, u);
            }
        }
        return transpose;
    }

    // DFS on the transposed graph to find SCCs
    void dfsOnTranspose(Node node, boolean[] visited) {
        visited[node.val] = true;
        System.out.print(node.val + " ");
        for (Node neighbor : node.neighbors) {
            if (!visited[neighbor.val]) {
                dfsOnTranspose(neighbor, visited);
            }
        }
    }
}

```

```

// Find all SCCs using Kosaraju's algorithm
void kosarajuSCC() {
    Stack<Node> stack = new Stack<>();
    boolean[] visited = new boolean[V];

    // Step 1: DFS on the original graph to get the nodes in stack
    for (int i = 0; i < V; i++) {
        if (!visited[i]) {
            dfs(nodes[i], visited, stack);
        }
    }

    // Step 2: Transpose the graph
    Graph transpose = getTranspose();

    // Step 3: Perform DFS on the transposed graph
    Arrays.fill(visited, val: false);
    System.out.println("Strongly Connected Components (SCCs):");
    while (!stack.isEmpty()) {
        Node node = stack.pop();
        if (!visited[node.val]) {
            transpose.dfsOnTranspose(node, visited);
            System.out.println(); // New SCC
        }
    }
}

public static void main(String[] args) {
    // Create a graph with 5 vertices
    Graph g = new Graph(V: 5);

    System.out.print("Stack after first DFS (finish time order): ");
    for (int i = stack.size() - 1; i >= 0; i--) {
        System.out.print(stack.get(i).val + " ");
    }
    System.out.println("\n");

    Stack after first DFS (finish time order): 0 3 4 2 1

    Strongly Connected Components (SCCs):
    0 2 1 3 4
}

```

```

// Initial graph representation:
// 0 -> 2
// 2 -> 1
// 1 -> 0
// 0 -> 3
// 3 -> 4

/*
4 <- 3 <- 0 -> 2 -> 1
      |           |
      -----  

*/

```

```

g.addEdge( u: 0, v: 2);
g.addEdge( u: 2, v: 1);
g.addEdge( u: 1, v: 0);
g.addEdge( u: 0, v: 3);
g.addEdge( u: 3, v: 4);

// Find SCCs
g.kosarajuSCC();

/*
Strongly Connected Components (SCCs): 0 2 1 3 4
4
3
0 2 1
*/
}

```

Tarjan's Algorithm:

● What is Tarjan's Algorithm?

Tarjan's Algorithm is used to find all **Strongly Connected Components (SCCs)** in a **directed graph** in a single **DFS traversal**.

It assigns:

- A **discovery time** (`disc`) to each node when it's first visited.
- A **low-link value** (`low`) — the smallest discovery time reachable from that node (including itself and back edges).
- A **stack** to track the current DFS path.
- When `disc[u] == low[u]`, it means `u` is the **root of an SCC**, and nodes are popped from the stack to form the SCC.

● Core Idea:

While doing DFS, we track the earliest reachable node (`low[u]`) from each node. If a node can reach itself through a cycle (including back edges), it is part of an SCC.

✓ Steps:

1. Do DFS from each unvisited node.
2. Assign `disc` and `low` values.
3. Use a stack to track current DFS path.
4. Update `low` values during traversal.
5. If `disc[u] == low[u]`, pop stack until `u` → this is one SCC.

● Time Complexity:

- $O(V + E)$

Each node and edge is visited **exactly once** during DFS.

■ Space Complexity:

- $O(V)$ for:
 - `disc[]` and `low[]`
 - `stack[]` and `onStack[]` boolean array
- So total: $O(V)$ auxiliary space (not including input graph)


```

// If u is the head of an SCC
if (low[u.val] == disc[u.val]) {
    Node v;
    do {
        v = stack.pop();
        onStack[v.val] = false;
        System.out.print(v.val + " ");
    } while (v != u);
    System.out.println();
}
}

public static void main(String[] args) {
    Graph g = new Graph( V: 5);

    // Initial graph representation:
    // 0 → 2
    // 2 → 1
    // 1 → 0
    // 0 → 3
    // 3 → 4

    g.addEdge( u: 0, v: 2);
    g.addEdge( u: 2, v: 1);
    g.addEdge( u: 1, v: 0);
    g.addEdge( u: 0, v: 3);
    g.addEdge( u: 3, v: 4);

    g.tarjansSCC();
}

```

Strongly Connected Components (SCCs):

```

4
3
1 2 0

```

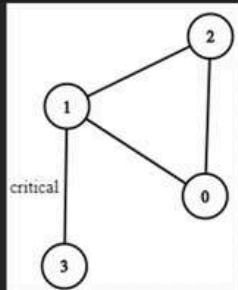
Critical Components in a Network

There are n servers numbered from 0 to $n - 1$ connected by undirected server-to-server connections forming a network where $\text{connections}[i] = [a_i, b_i]$ represents a connection between servers a_i and b_i . Any server can reach other servers directly or indirectly through the network.

A *critical connection* is a connection that, if removed, will make some servers unable to reach some other server.

Return all critical connections in the network in any order.

Example 1:



In the context of finding critical connections (bridges), if the low link value of a vertex v is greater than the discovery time of its parent vertex u (in the DFS tree), then the edge (u, v) is a bridge. This is because there is no alternative way to reach vertex v or its descendants from vertex u or any of its ancestors, other than through the edge (u, v) .

Input: $n = 4$, connections = [[0,1],[1,2],[2,0],[1,3]]
Output: [[1,3]]
Explanation: [[3,1]] is also accepted.

Example 2:

Input: $n = 2$, connections = [[0,1]]
Output: [[0,1]]

```
class Solution {
    List<List<Integer>> graph;
    int[] low;
    int[] disc;
    int time;
    List<List<Integer>> result;

    public List<List<Integer>> criticalConnections(int n, List<List<Integer>> connections) {
        graph = new ArrayList<>();
        low = new int[n];
        disc = new int[n];
        result = new ArrayList<>();
        time = 0;

        for (int i = 0; i < n; i++) {
            graph.add(new ArrayList<>());
        }

        for (List<Integer> conn : connections) {
            int u = conn.get(0);
            int v = conn.get(1);
            graph.get(u).add(v);
            graph.get(v).add(u);
        }

        dfs(0, -1);

        return result;
    }

    private void dfs(int u, int parent) {
        low[u] = disc[u] = ++time;

        for (int v : graph.get(u)) {
            if (v == parent) continue;

            if (disc[v] == 0) { // If v is not visited
                dfs(v, u);
                low[u] = Math.min(low[u], low[v]);
                if (low[v] > disc[u]) {
                    // (u, v) is a critical connection
                    result.add(Arrays.asList(u, v));
                }
            } else {
                low[u] = Math.min(low[u], disc[v]);
            }
        }
    }
}
```

Bridge Condition

If `low[v] > disc[u]` → then (u, v) is a **bridge**
→ Why? Because:

- There's no back edge from v or its subtree to u or any ancestor of u .
- Hence, removing (u, v) would break the only connection between these parts of the graph.

DFS Traversal Example:

plaintext

Copy Edit

```
0 - 1 - 2  
|  
3
```

DFS Tree

- `disc[1] = 1`
- `disc[3] = 2`, and since there's no back edge from $3 \rightarrow 0$ or 2 , we get:
- `low[3] = 2 > disc[1] = 1` → so edge $(1, 3)$ is a **bridge**

Visual Intuition:

Think of it like this:

- `low[v]` represents how far back you can go from v or its children.
- If you can't go back to u or above, then $u - v$ is a **critical connection**.

Let me know if you'd like:

- A dry-run of the algorithm on a sample input with `disc` & `low` values
- An annotated DFS tree
- Or visualization in code with prints/logging

Biconnected Component

💡 What is a Biconnected Component?

A **Biconnected Component** in an undirected graph is a maximal subgraph where removing any single vertex does **NOT** disconnect the subgraph.

In other words:

- There are **no articulation points** (a.k.a. cut vertices) within a BCC.
- The graph can be decomposed into multiple BCCs.

🧠 Tarjan's Algorithm for BCCs (Using DFS)

We use:

- `disc[u]` : Discovery time
- `low[u]` : Lowest time reachable
- A **stack of edges** to track edges in the current DFS path
- When `low[v] >= disc[u]`, we found a BCC and pop edges from the stack

💻 Sample Output:

```
less
Copy Edit
BCC #1: [(4, 5), (3, 5), (1, 4), (1, 3)]
BCC #2: [(1, 6)]
BCC #3: [(2, 0), (1, 2), (0, 1)]
```

⌚ Time Complexity:

- $O(V + E)$

💻 Space Complexity:

- $O(V + E)$ for storing `disc`, `low`, graph, and stack

```
public class Solution {

    static class Edge {
        int u, v;

        Edge(int u, int v) {
            this.u = u;
            this.v = v;
        }

        public String toString() {
            return "(" + u + ", " + v + ")";
        }
    }

    private int time = 0;
    private int[] disc, low;
    private List<List<Edge>> bccList = new ArrayList<>();
    private Map<Integer, List<Integer>> graph = new HashMap<>();
    private Deque<Edge> stack = new ArrayDeque<>();
```



```
public static void main(String[] args) {
    Solution finder = new Solution();
    int n = 7;

    List<List<Integer>> edges = Arrays.asList(
        Arrays.asList(0, 1),
        Arrays.asList(1, 2),
        Arrays.asList(2, 0),
        Arrays.asList(1, 3),
        Arrays.asList(1, 4),
        Arrays.asList(1, 6),
        Arrays.asList(3, 5),
        Arrays.asList(4, 5)
    );

    finder.findBCCs(n, edges);
    /*
    BCC #1: [(4, 1), (5, 4), (3, 5), (1, 3)]
    BCC #2: [(1, 6)]
    BCC #3: [(2, 0), (1, 2), (0, 1)]
    */
}
```

Articulation Points

💡 What is an Articulation Point?

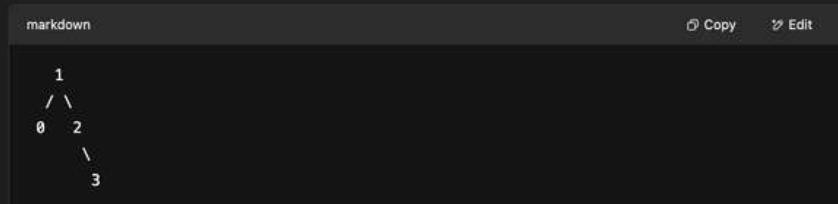
An articulation point in an undirected graph is a vertex that, when removed along with its incident edges, increases the number of connected components (i.e., disconnects the graph).

● Why Is It Important?

- These points are crucial for connectivity.
- In network terms: if an articulation point fails, parts of the network get isolated.

✓ Use Case Example

Graph:



Here, vertex `1` is an articulation point.

If you remove it, the graph splits into two parts: `{0}` and `{2, 3}`.

✓ Tarjan's Algorithm for Articulation Points

We use DFS and track:

- `disc[u]` : Discovery time of node `u`
- `low[u]` : Lowest discovery time reachable from subtree rooted at `u`
- `parent[u]` : Parent of node `u` in DFS tree

⌚ Conditions to Mark a Node as Articulation Point

During DFS traversal:

1. Root node (starting node of DFS):
 - If it has 2 or more children, it's an articulation point.
2. Non-root node `u`:
 - If there's a child `v` such that `low[v] >= disc[u]`
 - That means `v` and its subtree cannot reach `u` or any of its ancestors without passing through `u`.

```

public class Solution {
    private int time = 0;
    private int[] disc, low, parent;
    private boolean[] visited, isArticulation;
    private Map<Integer, List<Integer>> graph = new HashMap<>();

    public List<Integer> findArticulationPoints(int n, List<List<Integer>> edges) {
        disc = new int[n];
        low = new int[n];
        parent = new int[n];
        visited = new boolean[n];
        isArticulation = new boolean[n];

        Arrays.fill(parent, val: -1);
        for (int i = 0; i < n; i++) graph.put(i, new ArrayList<>());
        for (List<Integer> edge : edges) {
            int u = edge.get(0), v = edge.get(1);
            graph.get(u).add(v);
            graph.get(v).add(u);
        }

        for (int i = 0; i < n; i++) {
            if (!visited[i]) dfs(i);
        }

        List<Integer> result = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            if (isArticulation[i]) result.add(i);
        }
        return result;
    }

    private void dfs(int u) {
        visited[u] = true;
        disc[u] = low[u] = ++time;
        int children = 0;

        for (int v : graph.get(u)) {
            if (!visited[v]) {
                children++;
                parent[v] = u;
                dfs(v);

                low[u] = Math.min(low[u], low[v]);
            }
        }

        // Articulation point conditions
        if (parent[u] == -1 && children > 1) {
            isArticulation[u] = true;
        }
        if (parent[u] != -1 && low[v] >= disc[u]) {
            isArticulation[u] = true;
        }
        if (v != parent[u]) {
            low[u] = Math.min(low[u], disc[v]);
        }
    }
}

```

✓ Articulation Point Conditions:

Root Node Condition:

```
js
if (parent[u] == -1 && children > 1) {
    isArticulation[u] = true;
}
```

Explanation:

- If u is the root of the DFS tree ($\text{parent}[u] = -1$) and has more than one child, then it's an articulation point.
 - Why? Because removing the root will disconnect its children from each other — no back edges can connect them otherwise.
- Think of this like: if the root has 2+ subtrees, removing it will split the graph.

Non-root Node Condition:

```
js
if (parent[u] != -1 && low[v] >= disc[u]) {
    isArticulation[u] = true;
}
```

Explanation:

- If u is not the root, and any child v in the DFS tree cannot reach an ancestor of u (i.e., $\text{low}[v] \geq \text{disc}[u]$), then u is a critical bridge for that subtree.
 - So, removing u will disconnect v (and all of its descendants) from the rest of the graph.
- In essence, u is a necessary connection between its subtree and the upper part of the DFS tree.

```
public static void main(String[] args) {
    Solution solver = new Solution();

    int n = 5;
    List<List<Integer>> edges = Arrays.asList(
        Arrays.asList(0, 1),
        Arrays.asList(1, 2),
        Arrays.asList(2, 0),
        Arrays.asList(1, 3),
        Arrays.asList(3, 4)
    );

    List<Integer> points = solver.findArticulationPoints(n, edges);
    System.out.println("Articulation Points: " + points); // Output: [1, 3]
}
}
```

Output:

less

Copy Edit

Articulation Points: [1, 3]

Time and Space Complexity

- Time: $O(V + E)$ (DFS traversal of the graph)
- Space: $O(V)$ for arrays like `disc`, `low`, `visited`, etc.

Eulerian Circuit

An Eulerian Circuit in a graph is a circuit that visits every edge exactly once and returns to the starting vertex. The necessary conditions for an Eulerian Circuit are:

1. For undirected graphs: Every vertex must have an even degree, and all vertices with non-zero degree must be connected.
2. For directed graphs: Every vertex must have equal in-degree and out-degree, and all vertices with non-zero degree must be strongly connected.

Hierholzer's algorithm is used to find an Eulerian path or circuit in a graph. In this problem, we adapt the algorithm to find an Eulerian path in a directed graph, where we must visit every edge (flight) exactly once. The algorithm involves traversing the graph and backtracking as necessary to cover all edges.

A De Bruijn graph is a directed graph that represents overlapping subsequences of symbols from a larger sequence. It has applications in various fields, including computer science, bioinformatics, and cryptography.

Consider a De Bruijn graph $B(2, 3)$ over the binary alphabet $\{(0, 1)\}$. It consists of vertices representing all possible 3-bit binary sequences: 000, 001, 010, 011, 100, 101, 110, and 111. The edges are determined by the overlapping subsequences of length $n - 1 = 2$.

For example, there is an edge from 001 to 010 because the last 2 bits of 001 match the first 2 bits of 010. Similarly, there is an edge from 110 to 101 because the last 2 bits of 110 match the first 2 bits of 101.

1. Eulerian Circuit

Definition:

An **Eulerian Circuit** is a cycle in a graph that:

- Starts and ends at the **same vertex**
- Visits every edge **exactly once**

Conditions:

For a **directed graph**:

- Every vertex has **equal in-degree and out-degree**
- All vertices with nonzero degree are in the **same strongly connected component**

For an **undirected graph**:

- All vertices with **even degree**
- The graph is **connected**

Example:

Graph:

```
css
A → B
↑ ↓
D ← C
```

Copy Edit

This forms a cycle using all edges exactly once:

$A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$

2. Hierholzer's Algorithm

Purpose:

Used to find an Eulerian path or circuit in a graph.

Steps:

1. Start from any vertex with remaining edges.
2. Follow edges one at a time (removing them as you go), forming a cycle.
3. If you reach a vertex with unused edges, recurse from there, then merge the cycles.
4. Repeat until all edges are used.

Time Complexity:

- $O(E)$ → Efficient, since each edge is visited exactly once.

Java Pseudocode:

```
java
void findEulerianCircuit(Map<Integer, List<Integer>> graph, int start) {
    Stack<Integer> stack = new Stack<>();
    List<Integer> circuit = new ArrayList<>();
    stack.push(start);

    while (!stack.isEmpty()) {
        int u = stack.peek();
        if (graph.get(u).size() == 0) {
            circuit.add(stack.pop());
        } else {
            int v = graph.get(u).remove(0);
            stack.push(v);
        }
    }

    Collections.reverse(circuit);
    System.out.println("Eulerian Circuit: " + circuit);
}
```

Copy Edit

3. De Bruijn Graph

Definition:

A De Bruijn graph is a special kind of directed graph used to model overlaps in sequences, especially in bioinformatics and data compression.

- Nodes represent sequences of length $k-1$
- Edges represent sequences of length $k \rightarrow$ edge from `prefix` to `suffix`

Example (binary, $k=2$):

Strings: "00", "01", "10", "11"

Graph:

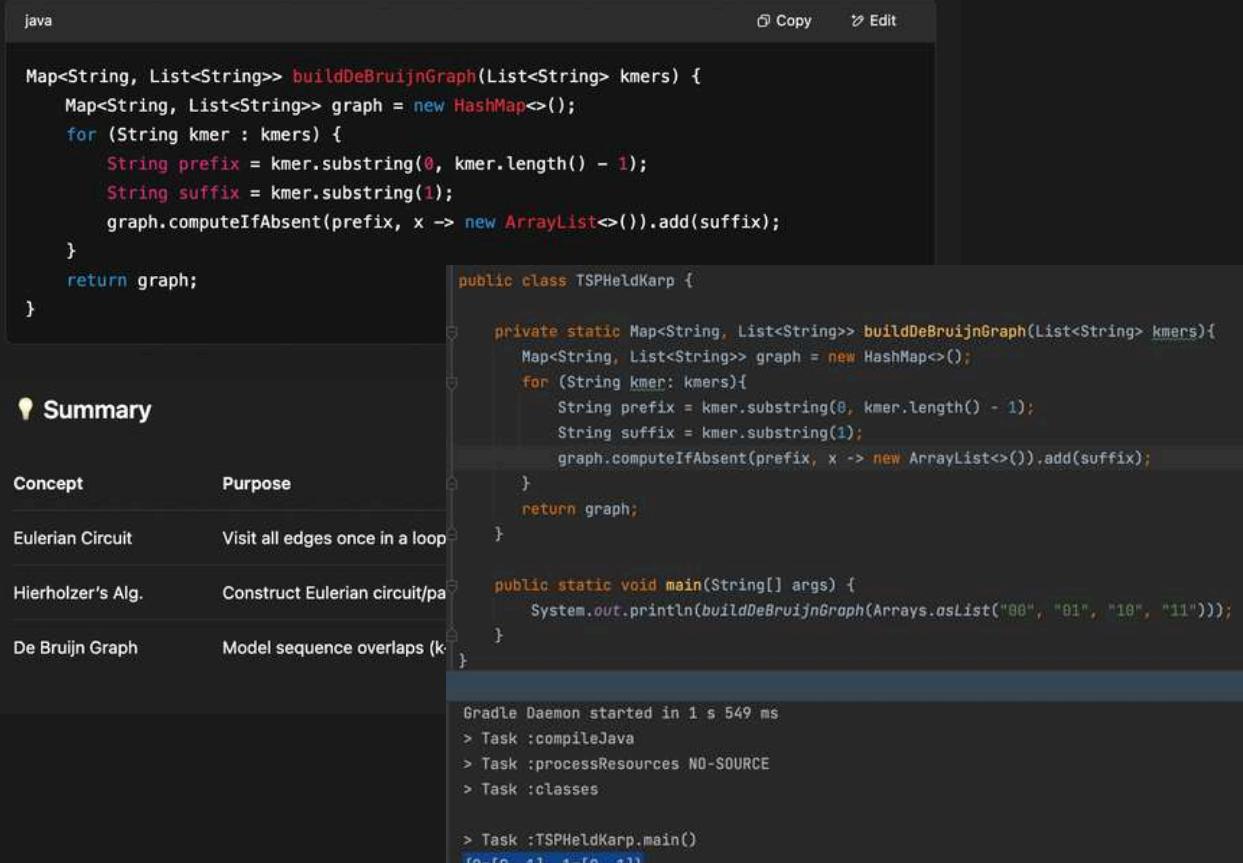
- Nodes: "0", "1"
- Edges:
 - "0" → 0" (for "00")
 - "0" → 1" (for "01")
 - "1" → 0" (for "10")
 - "1" → 1" (for "11")

This graph has an Eulerian circuit!

Application:

- DNA assembly (overlap of DNA fragments)
- Network routing
- Constructing shortest common superstring
- Password/permutation generation

Bonus: Building De Bruijn Graph in Code



The screenshot shows an IDE interface with a Java code editor and a terminal window.

Java Code:

```
java
Map<String, List<String>> buildDeBruijnGraph(List<String> kmers) {
    Map<String, List<String>> graph = new HashMap<>();
    for (String kmer : kmers) {
        String prefix = kmer.substring(0, kmer.length() - 1);
        String suffix = kmer.substring(1);
        graph.computeIfAbsent(prefix, x -> new ArrayList<>()).add(suffix);
    }
    return graph;
}

public class TSPHeldKarp {
    private static Map<String, List<String>> buildDeBruijnGraph(List<String> kmers) {
        Map<String, List<String>> graph = new HashMap<>();
        for (String kmer: kmers){
            String prefix = kmer.substring(0, kmer.length() - 1);
            String suffix = kmer.substring(1);
            graph.computeIfAbsent(prefix, x -> new ArrayList<>()).add(suffix);
        }
        return graph;
    }

    public static void main(String[] args) {
        System.out.println(buildDeBruijnGraph(Arrays.asList("00", "01", "10", "11")));
    }
}
```

Summary

Concept	Purpose
Eulerian Circuit	Visit all edges once in a loop
Hierholzer's Alg.	Construct Eulerian circuit/path
De Bruijn Graph	Model sequence overlaps (k)

Terminal Output:

```
Gradle Daemon started in 1 s 549 ms
> Task :compileJava
> Task :processResources NO-SOURCE
> Task :classes
> Task :TSPHeldKarp.main()
{0=[0, 1], 1=[0, 1]}
```

⌚ What is a Hamiltonian Cycle?

A **Hamiltonian cycle** (or circuit) is a **cycle** in a graph that:

- Visits each vertex exactly once
- Returns to the starting vertex

It differs from an **Eulerian cycle**, which visits **every edge** exactly once.

✓ Example:

For a graph with vertices A-B-C-D-A :

- This is a **Hamiltonian cycle** if every vertex is visited once (no repeats) and we return to the start.

💡 Conditions:

- Unlike Eulerian cycles, there's **no simple necessary and sufficient condition** to check if a Hamiltonian cycle exists.
- It is an **NP-complete problem**.

🧠 Time Complexity:

- Brute-force solution (backtracking): **O(N!)**, where N = number of vertices.

Hierholzer's algorithm is used to find an Eulerian path or circuit in a graph. In this problem, we adapt the algorithm to find an Eulerian path in a directed graph, where we must visit every edge (flight) exactly once. The algorithm involves traversing the graph and backtracking as necessary to cover all edges.

A De Bruijn graph is a directed graph that represents overlapping subsequences of symbols from a larger sequence. It has applications in various fields, including computer science, bioinformatics, and cryptography.

Consider a De Bruijn graph $B(2, 3)$ over the binary alphabet $\{(0, 1)\}$. It consists of vertices representing all possible 3-bit binary sequences: 000, 001, 010, 011, 100, 101, 110, and 111. The edges are determined by the overlapping subsequences of length $n - 1 = 2$.

For example, there is an edge from 001 to 010 because the last 2 bits of 001 match the first 2 bits of 010. Similarly, there is an edge from 110 to 101 because the last 2 bits of 110 match the first 2 bits of 101.

```
public class Solution {
    static class Node {
        int val;
        List<Node> neighbors = new ArrayList<>();

        Node(int val) {
            this.val = val;
        }
    }

    private final List<Node> nodes;
    private final Set<Integer> visited = new HashSet<>();
    private final List<Node> path = new ArrayList<>();
    private Node startNode;

    public Solution(List<Node> nodes) {
        this.nodes = nodes;
    }

    public boolean findHamiltonianCycle() {
        if (nodes.isEmpty()) return false;

        startNode = nodes.get(0);
        path.add(startNode);
        visited.add(startNode.val);

        return dfs(startNode);
    }

    private boolean dfs(Node current) {
        if (path.size() == nodes.size()) {
            // Check if last node connects back to start
            return current.neighbors.contains(startNode);
        }

        for (Node neighbor : current.neighbors) {
            if (!visited.contains(neighbor.val)) {
                visited.add(neighbor.val);
                path.add(neighbor);

                if (dfs(neighbor)) return true;

                visited.remove(neighbor.val);
                path.remove(index: path.size() - 1);
            }
        }
    }

    return false;
}

public void printCycle() {
    if (findHamiltonianCycle()) {
        System.out.print("Hamiltonian Cycle: ");
        for (Node n : path) System.out.print(n.val + " ");
        System.out.println(path.get(0).val); // To complete the cycle
    } else {
        System.out.println("No Hamiltonian Cycle found");
    }
}
```

```

public static void main(String[] args) {
    // Build graph
    Node n0 = new Node( val: 0);
    Node n1 = new Node( val: 1);
    Node n2 = new Node( val: 2);
    Node n3 = new Node( val: 3);

    n0.neighbors.addAll(Arrays.asList(n1, n2, n3));
    n1.neighbors.addAll(Arrays.asList(n0, n2));
    n2.neighbors.addAll(Arrays.asList(n0, n1, n3));
    n3.neighbors.addAll(Arrays.asList(n0, n2));

    List<Node> graph = Arrays.asList(n0, n1, n2, n3);

    Solution solver = new Solution(graph);
    solver.printCycle();
    // Hamiltonian Cycle: 0 1 2 3 0
}
}

```

```

public class Solution {

    // Build the De Bruijn graph from k-mers
    private static Map<String, List<String>> buildDeBruijnGraph(List<String> kmers) {
        Map<String, List<String>> graph = new HashMap<>();
        for (String kmer : kmers) {
            String prefix = kmer.substring(0, kmer.length() - 1);
            String suffix = kmer.substring(1);
            graph.computeIfAbsent(prefix, k -> new ArrayList<>()).add(suffix);
        }
        return graph;
    }

    // Find Eulerian Circuit using Hierholzer's Algorithm
    private static List<String> findEulerianCircuit(Map<String, List<String>> graph, String start) {
        Stack<String> stack = new Stack<>();
        List<String> circuit = new ArrayList<>();
        stack.push(start);

        while (!stack.isEmpty()) {
            String u = stack.peek();
            List<String> neighbors = graph.getOrDefault(u, new ArrayList<>());

            if (neighbors.isEmpty()) {
                circuit.add(stack.pop());
            } else {
                String v = neighbors.remove(0);
                stack.push(v);
            }
        }

        Collections.reverse(circuit);
        return circuit;
    }
}

```

```
public static void main(String[] args) {
    // Input list of k-mers (example: binary strings of length 3)
    List<String> kmers = Arrays.asList("000", "001", "010", "011", "100", "101", "110", "111");

    // Build graph and find Eulerian circuit
    Map<String, List<String>> graph = buildDeBruijnGraph(kmers);
    String startNode = kmers.get(0).substring(0, kmers.get(0).length() - 1);
    List<String> circuit = findEulerianCircuit(graph, startNode);

    // Output circuit as reconstructed string (optional)
    StringBuilder reconstructed = new StringBuilder(circuit.get(0));
    for (int i = 1; i < circuit.size(); i++) {
        reconstructed.append(circuit.get(i).charAt(circuit.get(i).length() - 1));
    }

    // Print results
    System.out.println("Eulerian Circuit (nodes): " + circuit);
    System.out.println("Reconstructed String from Eulerian Circuit: " + reconstructed.toString());
}
}
```

For kmers = [000, 001, 010, 011, 100, 101, 110, 111], a sample output would be:

javascript Copy Edit

```
Eulerian Circuit (nodes): [00, 01, 11, 10, 01, 10, 00, 00]
Reconstructed String from Eulerian Circuit: 00110100
```

Math

1. Rearrange an array so that arr[i] becomes arr[arr[i]] with O(1) extra space

C++ Java Python C# JavaScript

```
import java.util.*;

class GFG{
    static void rearrange(int[] arr) {
        int n = arr.length;

        // First step: Increase all values
        // by (arr[arr[i]]%n)*n
        for (int i = 0; i < n; i++)
            arr[i] += (arr[arr[i]] % n) * n;

        // Second Step: Divide all values by n
        for (int i = 0; i < n; i++)
            arr[i] /= n;
    }

    static void printArr(int[] arr) {
        int n = arr.length;
        for (int i = 0; i < n; i++)
            System.out.print(arr[i] + " ");
        System.out.println();
    }

    public static void main(String[] args) {
        int[] arr = { 3, 2, 0, 1 };
        rearrange(arr);
        printArr(arr);
    }
}
```



Output

```
3 2 0 1
```

Related Articles

Since all elements are in range $[0, n-1]$, and we know n , we can cleverly store two numbers (a and b) in one:

Copy Edit

```
arr[i] = a + b * n
```

Where:

- a is the original value ($arr[i]$)
- b is the new value ($arr[arr[i]]$)

Why does this work?

- $\% n$ gives the remainder → recovers a
- $/ n$ gives the quotient → gives b

We want to store both the old value and the new value at the same index in the array. Since array values are less than n , we can cleverly encode both using modulo and division operations.

Swift and check which range it falls within.

Similar, to reverse integer

Step-by-Step Breakdown:

java

Copy Edit

```
arr[i] += (arr[arr[i]] % n) * n;
```

Let's walk through what each part does:

1. $arr[i]$ → This still holds the original value a .
2. $arr[arr[i]] \% n$ →
 - $arr[i]$ is used as index.
 - $arr[arr[i]]$ may have already been encoded with another number, so we take $\% n$ to extract only the original value.

2. Reverse Integer

Given a signed 32-bit integer x , return x with its digits reversed. If reversing x causes the value to go outside the signed 32-bit integer range $[-2^{31}, 2^{31} - 1]$, then return 0.

Assume the environment does not allow you to store 64-bit integers (signed or unsigned).

Example 1:

```
Input: x = 123
Output: 321
```

Example 2:

```
Input: x = -123
Output: -321
```

Example 3:

```
Input: x = 120
Output: 21
```

Constraints:

- $-2^{31} \leq x \leq 2^{31} - 1$

```
public class Solution {
    public int reverse(int x) {
        int result = 0;

        while (x != 0) {
            int digit = x % 10;
            x = x / 10;
            if (result > Integer.MAX_VALUE / 10 || (result == Integer.MAX_VALUE / 10 && digit > 7)) {
                return 0;
            }
            if (result < Integer.MIN_VALUE / 10 || (result == Integer.MIN_VALUE / 10 && digit < -8)) {
                return 0;
            }
            result = result * 10 + digit;
        }

        return result;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        int input = 123;
        System.out.println("Reversed: " + sol.reverse(input)); // Output: 321
        input = -456;
        System.out.println("Reversed: " + sol.reverse(input)); // Output: -654
        input = 1534236469;
        System.out.println("Reversed: " + sol.reverse(input)); // Output: 0 (overflow)
    }
}
```

- **Palindrome Number**

9. Palindrome Number

Easy

Given an integer x , return `true` if x is a palindrome, and `false` otherwise.

Example 1:

Input: $x = 121$
Output: true
Explanation: 121 reads as 121 from left to right and from right to left.

Example 2:

Input: $x = -121$
Output: false
Explanation: From left to right, it reads -121. From right to left, it becomes 121-. Therefore it is not a palindrome.

Example 3:

Input: $x = 10$
Output: false
Explanation: Reads 01 from right to left. Therefore it is not a palindrome.

Constraints:

- $-2^{31} \leq x \leq 2^{31} - 1$

Follow up: Could you solve it without converting the integer to a string?

```
public class Solution {

    public static boolean isPalindrome(int x) {
        // Negative numbers are not palindromes
        if (x < 0 || (x % 10 == 0 && x != 0)) return false;

        int reversed = 0;
        while (x > reversed) {
            reversed = reversed * 10 + x % 10;
            x = x / 10;
        }

        // For even digits, x == reversed
        // For odd digits, x == reversed / 10
        return x == reversed || x == reversed / 10;
    }

    public static void main(String[] args) {
        int[] testCases = {121, -121, 10, 0, 12321};

        for (int num : testCases) {
            System.out.println("Is " + num + " a palindrome? " + isPalindrome(num));
        }
    }
}
```

• Factorial Trailing Zeroes

```
public class Solution {

    public static int trailingZeroes(int n) {
        int count = 0;
        while (n > 0) {
            n /= 5;
            count += n;
        }
        return count;
    }

    // Driver code
    public static void main(String[] args) {
        int[] testCases = {3, 5, 10, 25, 100, 125};

        for (int n : testCases) {
            System.out.println("Trailing zeroes in " + n + "!" + trailingZeroes(n));
        }
    }
}
```

the number of trailing zeroes = number of 5s in the prime factorization of $n!$.

💡 Trailing Zero Means a Factor of 10

- A number ends in a trailing zero if it is divisible by 10.
- $10 = 2 \times 5$
- So, to form a 10, we need both a 2 and a 5 as factors.

💡 What contributes to $n!$?

- $n! = n \times (n-1) \times (n-2) \times \dots \times 1$
- So, all numbers from 1 to n are multiplied together.

These numbers contain multiple 2s and 5s as part of their prime factorization.

💡 But 2s are more frequent than 5s

Let's observe:

- Every even number contributes a factor of 2. So 2, 4, 6, 8, etc.
 - But only every multiple of 5 contributes a factor of 5 (like 5, 10, 15...).
- 👉 So, we'll always have more 2s than 5s in the prime factorization of $n!$.

✓ Therefore:

- Each pair of (2, 5) makes a 10 → one trailing zero.
- Since 2s are abundant, the number of 5s becomes the limiting factor.

⌚ Hence:

Trailing zeroes in $n!$ = Number of 5s in its prime factorization

• Pow(x,n)

```

public class Solution {
    public double myPow(double x, int n) {
        long N = n;
        if (N < 0) {
            x = 1 / x;
            N = -N;
        }
        return powHelper(x, N);
    }

    private double powHelper(double x, long n) {
        if (n == 0) return 1.0;

        double half = powHelper(x, n: n / 2);
        if (n % 2 == 0) {
            return half * half;
        } else {
            return half * half * x;
        }
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println("3^5 = " + sol.myPow( x: 3.0, n: 5));      // 243.0
        System.out.println("2^-3 = " + sol.myPow( x: 2.0, n: -3));   // 0.125
    }
}

```

This is a fast algorithm that reduces the time complexity from $O(n)$ to $O(\log n)$ by:

- Splitting the exponent:
 - $x^n = (x^2)^{n/2}$ if n is even
 - $x^n = x \times (x^2)^{n-1/2}$ if n is odd
- Repeating until exponent becomes 0.

Sqrt(x)

```

public class Solution {
    public int mySqrt(int x) {
        if (x < 2) return x;

        int left = 1, right = x / 2;
        int ans = 0;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            long square = (long) mid * mid;

            if (square == x) {
                return mid;
            } else if (square < x) {
                ans = mid;           // store best candidate
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return ans;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println("Sqrt of 4: " + sol.mySqrt(4));    // 2
        System.out.println("Sqrt of 8: " + sol.mySqrt(8));    // 2
        System.out.println("Sqrt of 0: " + sol.mySqrt(0));    // 0
        System.out.println("Sqrt of 2147395600: " + sol.mySqrt(2147395600)); // 46340
    }
}

```

✓ Problem Statement

Implement `int mySqrt(int x)`

Return $\lfloor \sqrt{x} \rfloor$, i.e., the square root rounded down to the nearest integer.

● Example

- `mySqrt(4) → 2`
- `mySqrt(8) → 2` (because $\sqrt{8} \approx 2.828\dots$)

💡 Optimal Approach: Binary Search

We binary search for the square root in the range $[0, x]$:

- If $mid * mid == x$, return `mid`
- If $mid * mid < x$, move `low` up and store `mid` as the last valid value
- If $mid * mid > x$, move `high` down

✓ Greatest lower bound

- Power of 2

Problem Statement:

Given an integer `n`, return `true` if it is a power of two, otherwise return `false`.

A number is a power of two if:

`n == 2^x` for some integer `x >= 0`.

Examples:

- Input: 1 → true (2^0)
- Input: 16 → true (2^4)
- Input: 3 → false
- Input: 0 → false

Optimal Java Solution (Bit Manipulation)

```
java                                     ⌂ Copy ⌂ Edit

public class Solution {
    public boolean isPowerOfTwo(int n) {
        // Edge case: 0 or negative numbers are not powers of 2
        if (n <= 0) return false;

        // A power of two has exactly one bit set
        return (n & (n - 1)) == 0;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println(sol.isPowerOfTwo(1));    // true
        System.out.println(sol.isPowerOfTwo(16));   // true
        System.out.println(sol.isPowerOfTwo(3));    // false
        System.out.println(sol.isPowerOfTwo(0));    // false
    }
}
```

Explanation:

- A power of 2 in binary:
`1 = 0001, 2 = 0010, 4 = 0100, 8 = 1000` — only one bit is set.
- For such numbers: `n & (n - 1)` will always be `0`
E.g., `8 & 7 = 1000 & 0111 = 0000`

- Power of 3

Problem Statement:

Given an integer `n`, return `true` if it is a power of three, otherwise return `false`.

That means:

Check if `n == 3^x` for some integer `x >= 0`.

Examples:

- Input: 27 → true (3^3)
- Input: 9 → true (3^2)
- Input: 45 → false
- Input: 0 or -3 → false

```
public class Solution {  
    public boolean isPowerOfThree(int n) {  
        if (n <= 0) return false;  
  
        while (n % 3 == 0) {  
            n /= 3;  
        }  
  
        return n == 1;  
    }  
  
    public static void main(String[] args) {  
        Solution sol = new Solution();  
        System.out.println(sol.isPowerOfThree(27)); // true  
        System.out.println(sol.isPowerOfThree(9)); // true  
        System.out.println(sol.isPowerOfThree(45)); // false  
        System.out.println(sol.isPowerOfThree(1)); // true ( $3^0$ )  
    }  
}
```

Method 3: Integer Limit Trick (No Loop / Recursion)

- Highest power of 3 within 32-bit signed int = $3^{19} = 1162261467$
- So if `n > 0` and divides 1162261467 exactly, it is a power of 3.

```
java  
Copy Edit  
public boolean isPowerOfThree(int n) {  
    return n > 0 && 1162261467 % n == 0;  
}
```

- Power of 4

Problem Statement:

Given an integer `n`, return `true` if it is a power of four, otherwise return `false`.

This means checking if `n == 4^x` for some integer `x >= 0`.

Examples:

- Input: 16 → true (4^2)
- Input: 5 → false
- Input: 1 → true (4^0)

Method 1: Bit Manipulation + Position Check

Key Observations:

- A power of 4 is also a power of 2 → it has only one bit set.
- That 1-bit must be at an even index (0-based) — positions 0, 2, 4, 6, ...

```
java                                     ⌂ Copy ⌂ Edit

public class Solution {
    public boolean isPowerOfFour(int n) {
        // Check positive, power of 2, and bit position
        return n > 0 && (n & (n - 1)) == 0 && (n & 0x55555555) != 0;
    }

    public static void main(String[] args) {
        Solution sol = new Solution();
        System.out.println(sol.isPowerOfFour(1));      // true
        System.out.println(sol.isPowerOfFour(4));      // true
        System.out.println(sol.isPowerOfFour(16));     // true
        System.out.println(sol.isPowerOfFour(8));       // false
    }
}
```

● Why `0x55555555`?

- It's the binary number `0101...` (32 bits), which masks all even positions.
- So `n & 0x55555555 != 0` ensures the 1-bit is at an even position.

Method 2: Loop (Divide by 4)

```
java                                     ⌂ Copy ⌂ Edit

public boolean isPowerOfFour(int n) {
    if (n <= 0) return false;

    while (n % 4 == 0) {
        n /= 4;
    }

    return n == 1;
}
```

- **Base 7**

Problem Statement:

Given an integer `num`, return its base-7 representation as a string.

Approach:

1. If the number is `0`, return `"0"`.
2. For negative numbers, remember the sign.
3. Repeatedly divide the number by 7 and record the remainders.
4. Reverse the result to get the correct base-7 string.

Java Code:

```
java                                     ⌂ Copy ⌂ Edit

public class Base7Converter {

    public static String convertToBase7(int num) {
        if (num == 0) return "0";

        boolean isNegative = num < 0;
        num = Math.abs(num);
        StringBuilder sb = new StringBuilder();

        while (num > 0) {
            sb.append(num % 7);
            num /= 7;
        }

        if (isNegative) {
            sb.append("-");
        }

        return sb.reverse().toString();
    }

    // Driver code
    public static void main(String[] args) {
        int[] testCases = {100, -7, 0, 49, 343};

        for (int num : testCases) {
            System.out.println("Base 7 of " + num + " = " + convertToBase7(num));
        }
    }
}
```

Output:

```
yaml                                     ⌂ Copy ⌂ Edit

Base 7 of 100 = 202
Base 7 of -7 = -10
Base 7 of 0 = 0
Base 7 of 49 = 100
Base 7 of 343 = 1000
```

- Convert to base of 2

```
java                                     ⌂ Copy   ⌂ Edit

public class Base2Converter {

    public static String toBinary(int n) {
        if (n == 0) return "0";

        StringBuilder sb = new StringBuilder();

        while (n > 0) {
            sb.append(n % 2);
            n /= 2;
        }

        return sb.reverse().toString();
    }

    // Driver code
    public static void main(String[] args) {
        int[] testCases = {0, 1, 2, 5, 10, 15, 32, 100};

        for (int num : testCases) {
            System.out.println("Binary of " + num + " = " + toBinary(num));
        }
    }
}
```

Output:

```
sql                                     ⌂ Copy   ⌂ Edit

Binary of 0 = 0
Binary of 1 = 1
Binary of 2 = 10
Binary of 5 = 101
Binary of 10 = 1010
Binary of 15 = 1111
Binary of 32 = 100000
Binary of 100 = 1100100
```

Alternatively, you can use:

```
java                                     ⌂ Copy   ⌂ Edit

Integer.toBinaryString(n);
```

Let me know if you want conversion of negative numbers, or want to do the reverse (Base 2 to Decimal)!

- Convert to base of -2

✓ Java Code:

```
java Copy Edit  
  
public class BaseNegative2Converter {  
  
    public static String baseNeg2(int n) {  
        if (n == 0) return "0";  
  
        StringBuilder sb = new StringBuilder();  
  
        while (n != 0) {  
            int remainder = n % -2;  
            n /= -2;  
  
            // Adjust remainder to be non-negative  
            if (remainder < 0) {  
                remainder += 2;  
                n += 1;  
            }  
  
            sb.append(remainder);  
        }  
  
        return sb.reverse().toString();  
    }  
  
    // Driver code  
    public static void main(String[] args) {  
        int[] testCases = {2, 3, 4, -5, 0};  
  
        for (int num : testCases) {  
            System.out.println("Base -2 of " + num + " = " + baseNeg2(num));  
        }  
    }  
}
```

✓ Output:

```
nginx Copy Edit  
  
Base -2 of 2 = 110  
Base -2 of 3 = 111  
Base -2 of 4 = 100  
Base -2 of -5 = 11011  
Base -2 of 0 = 0
```

- Base 2 To Base 6

Java Code:

```
java
public class Base2ToBase6Converter {

    public static String base2ToBase6(String binaryStr) {
        // Step 1: Convert base 2 to decimal
        int decimal = Integer.parseInt(binaryStr, 2);

        // Step 2: Convert decimal to base 6
        if (decimal == 0) return "0";

        StringBuilder sb = new StringBuilder();
        while (decimal > 0) {
            sb.append(decimal % 6);
            decimal /= 6;
        }

        return sb.reverse().toString();
    }

    // Driver code
    public static void main(String[] args) {
        String[] binaryInputs = {"0", "1", "10", "11", "100", "1101", "100100"};
        for (String binary : binaryInputs) {
            System.out.println("Base 2: " + binary + " -> Base 6: " + base2ToBase6(binary));
        }
    }
}
```

Output:

```
yaml
Base 2: 0 -> Base 6: 0
Base 2: 1 -> Base 6: 1
Base 2: 10 -> Base 6: 2
Base 2: 11 -> Base 6: 3
Base 2: 100 -> Base 6: 4
Base 2: 1101 -> Base 6: 15
Base 2: 100100 -> Base 6: 44
```

• Armstrong Number

```
public class ArmstrongNumber {  
  
    public static boolean isArmstrong(int num) {  
        int original = num;  
        int digits = String.valueOf(num).length();  
        int sum = 0;  
  
        while (num > 0) {  
            int digit = num % 10;  
            sum += Math.pow(digit, digits);  
            num /= 10;  
        }  
  
        return sum == original;  
    }  
  
    // Driver code  
    public static void main(String[] args) {  
        int[] testCases = {153, 9474, 370, 407, 123, 9475};  
  
        for (int num : testCases) {  
            System.out.println(num + " is Armstrong? " + isArmstrong(num));  
        }  
    }  
}
```

Output:

csharp ⌂ Copy ⌓ Edit

```
153 is Armstrong? true  
9474 is Armstrong? true
```

● What is an Armstrong Number?

An Armstrong number (also known as a narcissistic number) of n digits is a number such that the sum of its digits each raised to the power n is equal to the number itself.

■ Formula:

For a 3-digit number like 153 :

$$1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153 \checkmark$$

For 9474 (4 digits):

$$9^4 + 4^4 + 7^4 + 4^4 = 9474 \checkmark$$

- Count anagrams

✓ Count Anagrams in Java

● Problem Statement:

Given a word, count the total number of unique anagrams that can be formed using all letters of the word.

If the word has **repeated letters**, the formula is:

$$\text{Anagrams} = \frac{n!}{f_1! \cdot f_2! \cdots \cdot f_k!}$$

Where:

- n = total number of characters
- f_1, f_2, \dots = frequencies of repeated characters

```
public class Solution {  
  
    // Helper: factorial  
    public static long factorial(int n) {  
        long fact = 1;  
        for (int i = 2; i <= n; i++) {  
            fact *= i;  
        }  
        return fact;  
    }  
  
    // Main function  
    public static long countAnagrams(String word) {  
        int[] freq = new int[26];  
        for (char c : word.toCharArray()) {  
            freq[c - 'a']++;  
        }  
        long numerator = factorial(word.length());  
        long denominator = 1;  
        for (int count : freq) {  
            if (count > 1) {  
                denominator *= factorial(count);  
            }  
        }  
        return numerator / denominator;  
    }  
  
    // Driver code  
    public static void main(String[] args) {  
        String[] testCases = {"abc", "aab", "aabb", "banana", "mississippi"};  
  
        for (String word : testCases) {  
            System.out.println("Unique anagrams of \"" + word + "\": " + countAnagrams(word));  
        }  
    }  
}
```

Output:

```
sql  
Copy Edit  
Unique anagrams of "abc": 6  
Unique anagrams of "aab": 3  
Unique anagrams of "aabb": 6  
Unique anagrams of "banana": 60  
Unique anagrams of "mississippi": 34650
```

• Find Minimum Operations to Make All Elements Divisible by Three

● Problem Statement:

Given an integer array `nums`, find the minimum number of operations required to make every element divisible by 3.

In one operation, you can:

- Increment or decrement any element by 1.

💡 Key Insight:

For each number:

- If it's already divisible by 3 → 0 operations
- If `num % 3 == 1` or `2` → Minimum 1 operation needed (either `+1` or `-1`)

So, for each number `n`, operations needed are:

```
java Copy Edit
Math.min(n % 3, 3 - (n % 3))
```

✓ Java Code:

```
java Copy Edit
public class MakeDivisibleBy3 {

    public static int minOperations(int[] nums) {
        int operations = 0;

        for (int num : nums) {
            int remainder = num % 3;
            operations += Math.min(remainder, 3 - remainder);
        }

        return operations;
    }

    // Driver code
    public static void main(String[] args) {
        int[][] testCases = {
            {3, 6, 9},           // already divisible → 0
            {4, 7, 10},          // all % 3 == 1 → 1 op each → 3
            {5, 8, 11},          // all % 3 == 2 → 1 op each → 3
            {1, 2, 3, 4, 5},     // mixed
            {0, 1, 2, 3, 4, 5, 6}
        };

        for (int[] test : testCases) {
            System.out.print("Array: ");
            for (int n : test) System.out.print(n + " ");
            System.out.println("→ Min Ops: " + minOperations(test));
        }
    }
}
```

Output:

```
yaml Copy Edit
Array: 3 6 9 → Min Ops: 0
Array: 4 7 10 → Min Ops: 3
Array: 5 8 11 → Min Ops: 3
Array: 1 2 3 4 5 → Min Ops: 4
Array: 0 1 2 3 4 5 6 → Min Ops: 6
```

• Permutation Sequence

● Problem Statement:

Given two integers n and k , return the k -th permutation sequence of numbers from 1 to n in lexicographic order.

■ Example:

- $n = 3, k = 3$
- Permutations: ["123", "132", "213", "231", "312", "321"]
- Output: "213"

◆ Key Idea:

This is a math-based backtracking alternative using factorials to determine the correct number at each position.

```
public class PermutationSequence {  
  
    public static String getPermutation(int n, int k) {  
        List<Integer> numbers = new ArrayList<>();  
        int[] fact = new int[n];  
        fact[0] = 1;  
  
        // Fill numbers list and factorial array  
        for (int i = 1; i < n; i++) {  
            fact[i] = fact[i - 1] * i;  
        }  
        for (int i = 1; i <= n; i++) {  
            numbers.add(i);  
        }  
  
        k--; // Convert to 0-based index  
        StringBuilder result = new StringBuilder();  
  
        for (int i = n; i >= 1; i--) {  
            int idx = k / fact[i - 1];  
            result.append(numbers.get(idx));  
            numbers.remove(idx);  
            k %= fact[i - 1];  
        }  
  
        return result.toString();  
    }  
  
    // Driver code  
    public static void main(String[] args) {  
        System.out.println(getPermutation(3, 3)); // Output: 213  
        System.out.println(getPermutation(4, 9)); // Output: 2314  
        System.out.println(getPermutation(1, 1)); // Output: 1  
    }  
}
```

```
213  
2314  
1
```

• Three Equal Parts

● Problem Statement (LeetCode #927):

Given a binary array `arr`, split the array into three non-empty parts such that all three parts represent the same binary number (ignoring leading zeros).

Return the indices `[i, j]` where:

- First part: `arr[0..i]`
- Second part: `arr[i+1..j]`
- Third part: `arr[j+1..n-1]`

If it's not possible, return `[-1, -1]`.

💡 Key Insight:

- Count total number of `1`s:
 - If total `1`s is not divisible by 3 → ✗ Not possible.
 - If total `1`s is 0 → ✓ Any split works: return `[0, n-1]`
- Let each part have `target = totalOnes / 3` number of `1`s.
- Find starting indices of each of the three parts with `target` `1`s.
- Then compare if the three parts are identical.

```
public class ThreeEqualParts {
    public static int[] threeEqualParts(int[] arr) {
        int totalOnes = 0;
        for (int bit : arr) {
            totalOnes += bit;
        }

        if (totalOnes == 0) {
            return new int[]{0, arr.length - 1}; // All zeros
        }

        if (totalOnes % 3 != 0) {
            return new int[]{-1, -1}; // Cannot divide 1s equally
        }

        int onesPerPart = totalOnes / 3;
        int n = arr.length;
        int first = -1, second = -1, third = -1;
        int count = 0;

        // Identify the starting index of each part
        for (int i = 0; i < n; i++) {
            if (arr[i] == 1) {
                count++;
                if (count == 1) first = i;
                else if (count == onesPerPart + 1) second = i;
                else if (count == 2 * onesPerPart + 1) third = i;
            }
        }

        // Compare the binary parts
        while (third < n) {
            if (arr[first] != arr[second] || arr[second] != arr[third]) {
                return new int[]{-1, -1};
            }
            first++;
            second++;
            third++;
        }
    }
}

// Driver code
public static void main(String[] args) {
    int[][] testCases = {
        {1, 0, 1, 0, 1}, // Output: [0, 3]
        {1, 1, 0, 1, 1}, // Output: [-1, -1]
        {0, 0, 0, 0, 0}, // Output: [0, 4]
        {1, 1, 0, 0, 1}, // Output: [0, 2]
    };

    for (int[] arr : testCases) {
        System.out.println("Input: " + Arrays.toString(arr));
        int[] res = threeEqualParts(arr);
        System.out.println("Output: " + Arrays.toString(res));
        System.out.println("-----");
    }
}
```

Sample Output:

```
makefile
```

```
Input: [1, 0, 1, 0, 1]
Output: [0, 3]

Input: [1, 1, 0, 1, 1]
Output: [-1, -1]

Input: [0, 0, 0, 0, 0]
Output: [0, 4]

Input: [1, 1, 0, 0, 1]
Output: [0, 2]
```

- Count of pairs that can be removed from Array without changing the Mean

Given an array `arr[]` of size `N`, the task is to find the number of pairs of the array upon removing which the `mean` (i.e. the average of all elements) of the array remains unchanged.

Examples:

`Input: N = 5, arr[] = {1, 4, 7, 3, 5}`

`Output: 2`

`Explanation: The required pairs of positions are – {0, 2} and {3, 4}.`

`Mean of original array = (1 + 4 + 7 + 3 + 5) / 5 = 4.`

`On deletion of elements at positions 0 and 2, array becomes {4, 3, 5}.`

`Mean of the new array = (4 + 3 + 5) / 3 = 4, which is same as the mean of original array.`

`On deletion of elements at positions 3 and 4, array becomes {1, 4, 7}.`

`Mean of the new array = (1 + 4 + 7) / 3 = 4, which is same as the mean of original array.`

`Input: N = 3, A = {50, 20, 10}`

`Output: 0`

`Explanation: No such pair is possible.`

Observation:

If a sum ($S = 2 * \text{mean}$) is subtracted from the original array (with initial mean = M), the mean of the new updated array will still remain as M .

Derivation:

- Sum of array (of size N) = $N * \text{mean} = N * M$
- new mean = $(\text{sum of array} - 2 * M) / (N - 2)$
 $= (N * M - 2 * M) / (N - 2)$
 $= M.$

```
public class MeanUnchangedPairs {
```

```
    public static void countAndPrintPairs(int[] arr) {
        int n = arr.length;
        long sum = 0;
        for (int num : arr) sum += num;

        if ((2 * sum) % n != 0) {
            System.out.println("No valid pairs that maintain the mean.");
            return;
        }

        long target = (2 * sum) / n;
        Map<Long, Integer> freq = new HashMap<>();

        for (int num : arr) {
            freq.put((long) num, freq.getOrDefault((long) num, 0) + 1);
        }

        System.out.println("Valid pairs:");
        for (int i = 0; i < n; i++) {
            long x = arr[i];
            freq.put(x, freq.get(x) - 1); // Temporarily remove this element
            long y = target - x;

            if (freq.getOrDefault(y, 0) > 0) {
                System.out.println("(" + x + ", " + y + ")");
                // decrease count for y to avoid duplicate printing
                freq.put(y, freq.get(y) - 1);
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 3, 2, 2, 1, 3};
        countAndPrintPairs(arr);
    }
}
```

Output:

lua

Valid pairs:
(1, 3)
(2, 2)
(1, 3)

Line Sweep

The Line Sweep Algorithm is a powerful technique used to solve a variety of problems involving intervals, rectangles, or events along a line (typically 1D or 2D). Let me show you the core idea, then give a Java implementation using a classic problem:

Problem Example:

"Given a list of intervals, find the maximum number of overlapping intervals."

Input:

```
java  
intervals = [[1, 3], [2, 5], [4, 6]]
```

Copy Edit

Output:

```
java  
Maximum overlap = 2
```

Copy Edit

Step 1: Building the Event Timeline

```
java  
TreeMap<Integer, Integer> timeline = new TreeMap<>();  
for (int[] interval : intervals) {  
    int start = interval[0];  
    int end = interval[1];  
    timeline.put(start, timeline.getOrDefault(start, 0) + 1);  
    timeline.put(end + 1, timeline.getOrDefault(end + 1, 0) - 1); // end + 1 closes the interval  
}
```

Copy Edit

This creates a sorted timeline of "events" using a `TreeMap`, which automatically keeps keys (time points) sorted.

- `+1` is added at the `start` time (an interval starts).
- `-1` is added at `end + 1` (one past the end), meaning the interval ends before that moment.

Why `end + 1`?

This ensures that the interval includes the end time. For example, an interval `[1, 3]` means the interval is active at time 3 – so we remove it after 3.

Timeline for `[[1, 3], [2, 5], [4, 6]]`:

Time	Change
1	+1
2	+1
4	+1
4	-1 (from 3+1)
6	-1
7	-1 (from 6+1)

Saved memory full

Sorting Events:

We sort the events by time. If two events have the same time, we process `start` before `end`, to ensure proper counting.

So the sorted events:

```
scss  
(1, +1)  -> interval starts  
(2, +1)  
(3, -1)  -> interval ends  
(4, +1)  
(5, -1)  
(6, -1)
```

Copy Edit

Why It Works:

We simulate a line sweeping from left to right, keeping track of how many intervals are currently active.

This allows us to efficiently compute overlaps without nested loops.

Step-by-Step Line Sweep:

We convert each interval into two events:

- A `start` point: when an interval begins.
- An `end` point: when an interval ends.

So, from `[[1, 3], [2, 5], [4, 6]]` we get the following events:

Time	Type
1	start
2	start
3	end
4	start
5	end
6	end

Sweep the Line:

We maintain a `currentOverlap` count and track `maxOverlap` as we process events:

Time	Event	Current Overlap	Max Overlap
1	+1	1	1
2	+1	2	2 <input checked="" type="checkbox"/>
3	-1	1	2
4	+1	2	2
5	-1	1	2
6	-1	0	2

```
java                                     ⌂ Copy ⌂ Edit

import java.util.*;

public class LineSweepMaxOverlap {
    public static int maxOverlap(int[][] intervals) {
        TreeMap<Integer, Integer> timeline = new TreeMap<>();

        // Step 1: Build the event timeline
        for (int[] interval : intervals) {
            int start = interval[0];
            int end = interval[1];
            timeline.put(start, timeline.getOrDefault(start, 0) + 1);
            timeline.put(end + 1, timeline.getOrDefault(end + 1, 0) - 1); // end + 1 to c
        }

        // Step 2: Sweep and track max overlap
        int active = 0, max = 0;
        for (int count : timeline.values()) {
            active += count;
            max = Math.max(max, active);
        }

        return max;
    }

    public static void main(String[] args) {
        int[][] intervals = { {1, 3}, {2, 5}, {4, 6} };
        System.out.println("Max Overlap: " + maxOverlap(intervals));
    }
}
```

✓ Minimum Meeting Rooms (1D Line Sweep variant)

✓ 1. Minimum Meeting Rooms (Interval Overlap via Line Sweep)

🔍 Problem:

You're given a list of meeting time intervals `[[start, end], ...]`. Return the **minimum number of meeting rooms required**.

💡 Approach:

- Treat each meeting's start as `+1 room needed`
- Treat each end as `-1 room released`
- Sort the timeline and keep a running count

```

import java.util.*;

public class MinMeetingRooms {
    public static int minMeetingRooms(int[][] intervals) {
        TreeMap<Integer, Integer> timeline = new TreeMap<>();

        for (int[] interval : intervals) {
            timeline.put(interval[0], timeline.getOrDefault(interval[0], 0) + 1);
            timeline.put(interval[1], timeline.getOrDefault(interval[1], 0) - 1);
        }

        int rooms = 0, maxRooms = 0;
        for (int count : timeline.values()) {
            rooms += count;
            maxRooms = Math.max(maxRooms, rooms);
        }

        return maxRooms;
    }

    public static void main(String[] args) {
        int[][] intervals = {{0, 30}, {5, 10}, {15, 20}};
        System.out.println("Min Rooms: " + minMeetingRooms(intervals));
    }
}

```

Output:

yaml Copy Edit

```
Min Rooms: 2
```

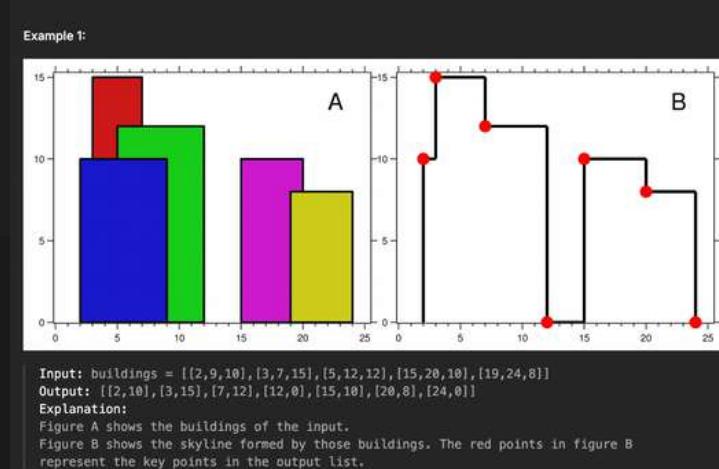
✓ 2. Skyline Problem (2D Line Sweep — Complex Variant)

🔍 Problem:

You are given a list of buildings represented as `[left, right, height]`. Return the **key points** of the skyline formed.

💡 Approach:

1. Create events:
 - Start of building → `(x, -height)`
 - End of building → `(x, +height)`
2. Sort events by `x`. For ties, process **starts before ends**.
3. Use a **Max-Heap** to keep track of current heights.
4. On change of max height → Add point to skyline.



```

public class SkylineProblem {
    public static List<int[]> getSkyline(int[][] buildings) {
        List<int[]> events = new ArrayList<>();
        for (int[] b : buildings) {
            events.add(new int[]{b[0], -b[2]}); // building start
            events.add(new int[]{b[1], b[2]}); // building end
        }

        // Sort by x; if tie, by height
        events.sort((a, b) ->
            a[0] != b[0] ? Integer.compare(a[0], b[0]) : Integer.compare(a[1], b[1]));
    }

    List<int[]> result = new ArrayList<>();
    PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder());
    maxHeap.add(0); // initial ground height

    int prevMax = 0;
    for (int[] event : events) {
        int x = event[0], h = event[1];

        if (h < 0) {
            maxHeap.add(-h); // start of building
        } else {
            maxHeap.remove(h); // end of building
        }

        int currMax = maxHeap.peek();
        if (currMax != prevMax) {
            result.add(new int[]{x, currMax});
            prevMax = currMax;
        }
    }

    return result;
}

```



```

public static void main(String[] args) {
    int[][] buildings = {{2, 9, 10}, {3, 7, 15}, {5, 12, 12}, {15, 20, 10}, {19, 24,
    List<int[]> skyline = getSkyline(buildings);
    for (int[] point : skyline) {
        System.out.println(Arrays.toString(point));
    }
}

```

Output:

csharp

[2, 10]
[3, 15]
[7, 12]
[12, 0]
[15, 10]
[20, 8]
[24, 0]

Summary:

Problem	Dimension	Key Data Structure	Optimization Idea
Max Overlap / Meeting Rooms	1D	TreeMap (event sweep)	Prefix sum of events
Skyline Problem	2D	Max-Heap (PQ)	Maintain current max height

Divide And Conquer

● What is Divide and Conquer?

Divide and Conquer breaks a problem into smaller subproblems, solves each recursively, and then combines the results.

Key Steps:

1. Divide: Break the input into parts.
2. Conquer: Recursively solve the subparts.
3. Combine: Merge sub-results to form the final result.

Common Examples & Patterns

Problem	Use of Divide & Conquer
Merge Sort	Divide array, sort halves, merge
Quick Sort	Partition around pivot, recurse
Binary Search	Split range in half each step
Maximum Subarray (Kadane)	Combine left, right, cross subarrays
Closest Pair of Points	Divide points by X, conquer, merge
Matrix Multiplication	Split into submatrices
K-th Largest Element	Use QuickSelect (a variant)
Strassen's Matrix Multiplication	Matrix divide & conquer

```
public class MergeSort {  
    public static void mergeSort(int[] arr, int l, int r) {  
        if (l < r) {  
            int mid = l + (r - l) / 2;  
  
            mergeSort(arr, l, mid);  
            mergeSort(arr, mid + 1, r);  
  
            merge(arr, l, mid, r);  
        }  
    }  
  
    private static void merge(int[] arr, int l, int m, int r) {  
        int[] left = Arrays.copyOfRange(arr, l, m + 1);  
        int[] right = Arrays.copyOfRange(arr, m + 1, r + 1);  
  
        int i = 0, j = 0, k = l;  
  
        while (i < left.length && j < right.length) {  
            arr[k++] = (left[i] <= right[j]) ? left[i++] : right[j++];  
        }  
  
        while (i < left.length) arr[k++] = left[i++];  
        while (j < right.length) arr[k++] = right[j++];  
    }  
  
    public static void main(String[] args) {  
        int[] arr = {5, 2, 4, 6, 1, 3};  
        mergeSort(arr, 0, arr.length - 1);  
        System.out.println(Arrays.toString(arr));  
    }  
}
```

Randomized

Randomized algorithms are often used in problems where a probabilistic approach can simplify the solution or improve the average-case performance. In LeetCode, there are a few problems that involve randomized algorithms either directly or indirectly. Here's a list of some classic examples:

1. Shuffle an Array (LeetCode 384)

- **Problem:** Given an array `nums`, implement an algorithm that randomly shuffles the array and returns the shuffled array.
- **Solution:** This problem can be solved using **Fisher-Yates shuffle**, which is a randomized algorithm. The Fisher-Yates shuffle ensures that each permutation of the array is equally likely.

```
import java.util.Random;

public class Solution {
    private int[] original;
    private Random rand;

    public Solution(int[] nums) {
        this.original = nums.clone();
        this.rand = new Random();
    }

    /** Resets the array to its original configuration and return it. */
    public int[] reset() {
        return original;
    }

    /** Returns a random shuffling of the array. */
    public int[] shuffle() {
        int[] shuffled = original.clone();
        for (int i = 0; i < shuffled.length; i++) {
            int j = rand.nextInt(i + 1);
            // Swap elements i and j
            int temp = shuffled[i];
            shuffled[i] = shuffled[j];
            shuffled[j] = temp;
        }
        return shuffled;
    }
}
```

2. Random Pick Index (LeetCode 398)

- **Problem:** Given an array of integers `nums`, write a function to pick a random index such that the element at that index is equal to a given target value.
- **Solution:** A solution to this problem uses the **Reservoir Sampling** algorithm, which is useful for situations where the size of the data is unknown, and we need to sample random elements.

```

import java.util.ArrayList;
import java.util.List;
import java.util.Random;

public class Solution {
    private int[] nums;
    private Random rand;

    public Solution(int[] nums) {
        this.nums = nums;
        this.rand = new Random();
    }

    public int pick(int target) {
        List<Integer> indices = new ArrayList<>();
        // Collect all indices of the target in the array
        for (int i = 0; i < nums.length; i++) {
            if (nums[i] == target) {
                indices.add(i);
            }
        }
        // Randomly pick one index
        return indices.get(rand.nextInt(indices.size()));
    }
}

```

3. Kth Largest Element in an Array (LeetCode 215)

- **Problem:** Given an integer array `nums` and an integer `k`, find the `k`th largest element in the array.
- **Solution:** You can use **Randomized QuickSelect** algorithm, which is a variant of QuickSort. Instead of fully sorting the array, you partition the array such that the `k`th largest element is placed at the correct index.

```

public class Solution {
    private Random rand = new Random();

    public int findKthLargest(int[] nums, int k) {
        return quickSelect(nums, 0, nums.length - 1, nums.length - k);
    }

    private int quickSelect(int[] nums, int left, int right, int index) {
        if (left == right) return nums[left];
        int pivotIndex = partition(nums, left, right);
        if (pivotIndex == index) return nums[pivotIndex];
        else if (pivotIndex < index) return quickSelect(nums, pivotIndex + 1, right, index);
        else return quickSelect(nums, left, pivotIndex - 1, index);
    }

    private int partition(int[] nums, int left, int right) {
        int pivotIndex = left + rand.nextInt(right - left + 1);
        int pivot = nums[pivotIndex];
        swap(nums, pivotIndex, right);
        int i = left;
        for (int j = left; j < right; j++) {
            if (nums[j] < pivot) {
                swap(nums, i, j);
                i++;
            }
        }
        swap(nums, i, right);
        return i;
    }

    private void swap(int[] nums, int i, int j) {
        int temp = nums[i];
        nums[i] = nums[j];
        nums[j] = temp;
    }
}

```

4. Linked List Random Node (LeetCode 382)

- **Problem:** Given a singly linked list, implement an algorithm to pick a random node from the list.
- **Solution:** This problem uses **Reservoir Sampling** to ensure that every node has an equal probability of being chosen. The algorithm randomly selects a node, with the probability of choosing each node being proportional to the number of nodes seen so far.

```
import java.util.Random;

public class Solution {
    private ListNode head;
    private Random rand;

    public Solution(ListNode head) {
        this.head = head;
        this.rand = new Random();
    }

    public int getRandom() {
        int result = head.val;
        ListNode current = head.next;
        int n = 2;
        while (current != null) {
            if (rand.nextInt(n) == 0) {
                result = current.val;
            }
            n++;
            current = current.next;
        }
        return result;
    }
}

class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}
```

5. Reservoir Sampling (General Problem)

- **Problem:** Given a stream of data (with unknown size), select a random sample of size k from the stream.
- **Solution:** Reservoir Sampling allows you to randomly sample from a stream without needing to know the total size in advance. You keep the first k elements, and for each subsequent element, you randomly decide whether to replace one of the current k elements with it.

```
public class ReservoirSampling {
    public int[] sample(int[] stream, int k) {
        Random rand = new Random();
        int[] reservoir = new int[k];

        // Fill the reservoir with the first k elements
        for (int i = 0; i < k; i++) {
            reservoir[i] = stream[i];
        }

        // Replace elements with a decreasing probability
        for (int i = k; i < stream.length; i++) {
            int j = rand.nextInt(i + 1);
            if (j < k) {
                reservoir[j] = stream[i];
            }
        }

        return reservoir;
    }
}
```

Reservoir Sampling

Reservoir Sampling is an important algorithm for randomly sampling elements from a stream of data, where the size of the data stream is not known in advance. It's commonly used in problems that involve random selection of elements or sampling from a large dataset with unknown size.

Here's a classic Reservoir Sampling problem from LeetCode:

Linked List Random Node (LeetCode 382)

This problem requires us to pick a random node from a singly linked list. The challenge is that we have to make this selection randomly, and every node should have an equal probability of being chosen.

Problem Explanation:

You are given a singly linked list with n nodes. Implement an algorithm to pick a random node from the list. You need to implement the `getRandom()` method such that each node has an equal probability of being selected.

Constraints:

- $1 \leq n \leq 10^4$
- The `getRandom` method should be called at most 10^4 times.

Approach:

You can solve this problem using the Reservoir Sampling algorithm. Here's how it works:

- Initialize a variable to store the randomly selected node.
- Traverse through the linked list, and for each node i (starting from the second node), randomly choose whether to replace the selected node with this node. The probability of replacing the current node should be $1/i$.

The intuition behind this is that each node should have an equal chance of being chosen, and the probability of choosing a node at each step is adjusted to maintain this fairness.

```
import java.util.Random;

// Definition for singly-linked list.
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

public class Solution {
    private ListNode head;
    private Random rand;

    // Constructor to initialize the linked list and Random object
    public Solution(ListNode head) {
        this.head = head;
        this.rand = new Random();
    }

    // The getRandom function that picks a random node from the linked list
    public int getRandom() {
        int result = head.val;
        ListNode current = head.next;
        int n = 2;

        // Traverse through the linked list
        while (current != null) {
            // With probability 1/n, replace the result with the current node's value
            if (rand.nextInt(n) == 0) {
                result = current.val;
            }
            n++;
            current = current.next;
        }
        return result;
    }
}
```

Explanation:

1. Initialization:

- The constructor `Solution(ListNode head)` initializes the linked list `head` and sets up a `Random` object to generate random numbers.

2. The `getRandom()` Method:

- We start by selecting the first node as the initial random node.
- As we traverse the linked list, we consider each node one by one.
- For the `i`-th node (where `i` is the index starting from 1), we generate a random number between 0 and `1`. If the random number is `0`, we replace the previously selected node with the current node. This ensures that each node has an equal probability of being chosen by the time we finish the traversal.
- Specifically, if `i` nodes have been seen, the probability of the `i`-th node being selected is `1/i`, which is the core idea behind Reservoir Sampling.

3. Why It Works:

- The algorithm ensures that after processing `i` nodes, each node has a probability of `1/i` to replace the previously selected node. As a result, by the time we reach the end of the list, every node has been equally likely to be selected.

4. Time and Space Complexity:

- Time Complexity:** $O(n)$, where `n` is the number of nodes in the linked list. We need to traverse through the entire list once.
- Space Complexity:** $O(1)$, since we are not using any additional data structures that grow with the size of the input.

Example Walkthrough:

Consider a linked list: `1 → 2 → 3 → 4 → 5`

- Initially, `result = 1` (the first node).
- After the second node (`2`), there's a 50% chance to update `result` to `2`.
- After the third node (`3`), there's a 1/3 chance to update `result` to `3`.
- After the fourth node (`4`), there's a 1/4 chance to update `result` to `4`.
- After the fifth node (`5`), there's a 1/5 chance to update `result` to `5`.

Rejection Sampling

★ What is Rejection Sampling?

- Rejection sampling is a technique where you generate random numbers and only accept them if they meet some condition.
- If not, you reject and try again.

★ Simple Intuition:

Imagine you want a random number between 1 to 5, but you can only generate numbers between 1 to 7.

Idea:

- Generate random numbers [1, 7].
- If it falls in [1, 5], keep it.
- If it's 6 or 7, reject and retry.

★ Examples on LeetCode:

Here are some problems that use rejection sampling:

Problem	Description
470. Implement Rand10() Using Rand7()	Use <code>rand7()</code> (returns 1-7) to make <code>rand10()</code> (returns 1-10). Classic rejection sampling.
478. Generate Random Point in a Circle	Generate a random (x, y) inside a circle. Rejection sampling is used because a random box covers the circle and points outside the circle are rejected.
497. Random Point in Non-overlapping Rectangles	Choose random points across rectangles — needs techniques similar to rejection or weighted random sampling.
528. Random Pick with Weight	Related concept: binary search after preparing prefix sums.
519. Random Flip Matrix	Pick random 0 in a matrix. Needs efficient mapping and randomization.

★ rand10() using rand7() Example:

You generate two random calls:

```
java ⌂ Copy ⌂ Edit  
int num = (rand7() - 1) * 7 + rand7(); // gives 1 to 49
```

If `num` is between 1 to 40 → good, because 40 is divisible cleanly into 10 groups.

- 1-10 → 1
- 11-20 → 2
- etc.

If `num > 49` → reject and retry.

★ Why Rejection Sampling is Important?

- ✓ It lets you convert random number ranges.
- ✓ It guarantees uniform distribution.
- ✓ It's simple and powerful when designed correctly.
- ✓ Even when retries happen, expected retries are low (math helps to balance it).

470. Implement Rand10() Using Rand7()

Problem:

You are given a function `rand7()` that generates random numbers between 1 and 7 (inclusive) uniformly. You are asked to implement `rand10()`, which generates random numbers between 1 and 10 uniformly.

Code:

```
java ⌂ Copy ⌂ Edit  
class Solution extends SolBase {  
    public int rand10() {  
        while (true) {  
            int num = (rand7() - 1) * 7 + rand7(); // Step 1  
            if (num <= 40) { // Step 2  
                return 1 + (num - 1) % 10; // Step 3  
            }  
        }  
    }  
}
```

Step-by-Step Explanation:

Step 1:

```
java ⌂ Copy ⌂ Edit  
int num = (rand7() - 1) * 7 + rand7();
```

- `rand7() - 1` gives a number between 0 and 6.
- Multiply by 7: `(rand7() - 1) * 7` gives values {0, 7, 14, 21, 28, 35, 42}.
- Then add another `rand7()` (which is 1-7).

Thus:

- We can generate numbers from 1 to 49.
- Each number from 1 to 49 is **equally likely**.

Step 2:

java

Copy Edit

```
if (num <= 40)
```

- We accept the number only if it's 1 to 40.
- Why?
 - Because 40 is divisible evenly by 10.
 - 40 numbers can be distributed equally among 10 outputs (4 numbers each).

If `num` is between 41-49, we **discard** and try again (`while (true)` will retry).

Step 3:

java

Copy Edit

```
return 1 + (num - 1) % 10;
```

- `(num - 1) % 10` gives a number between 0 and 9.
- Adding 1 shifts it to 1 and 10.

Thus, uniformly generating 1 to 10!

- `rand7()` returns 3 and 5.
- `num = (3-1) * 7 + 5 = 2*7 + 5 = 19`.

Now:

- 19 is ≤ 40 , so it is **accepted**.
- $(19-1)\%10 = 18\%10 = 8$, so we return $8+1=9$.

Thus, `rand10()` would return 9.

Why is this approach correct?

- ✓ `rand7()` gives each of 1-7 with equal probability.
- ✓ Mapping to 1-49 uniformly covers all numbers equally.
- ✓ Only accepting 1-40 ensures no bias.
- ✓ `(num-1)\%10 +1` spreads numbers 1-10 evenly.

Visualization:

Generated num (1-40)	Output
1-4	1
5-8	2
9-12	3
13-16	4
17-20	5
21-24	6
25-28	7
29-32	8
33-36	9
37-40	10

TL;DR:

- Make a bigger uniform space (1-49).
- Only take safe values (1-40).
- Then map them into 1-10 evenly.

Suffix Array

What is a Suffix Array?

A Suffix Array is a sorted array of all suffixes of a string.

Each suffix is represented by its starting index in the original string.

Example:

Let's take the string:

```
ini
s = "banana"
```

All suffixes are:

Index	Suffix
0	banana
1	anana
2	nana
3	na
4	a
5	

Now, sort these suffixes lexicographically (dictionary order):

Now, sort these suffixes lexicographically (dictionary order):

Sorted Suffix	Original Index
a	5
ana	3
anana	1
banana	0
na	4
nana	2

Thus, the Suffix Array is:

```
csharp
[5, 3, 1, 0, 4, 2]
```

This array stores the starting index of suffixes in sorted order.

Why is it useful?

Suffix Arrays are powerful because they allow:

- Finding substrings quickly.
- Solving Longest Repeated Substring.
- Solving Number of Distinct Substrings.
- Solving Pattern matching in $O(m + \log n)$ time.
- Used in bioinformatics for DNA sequence analysis!

⚡ How do we build it?

Idea:

- Sort all suffixes.
- But suffixes are strings => sorting directly is too slow ($O(N^2 \log N)$).
- So we compare suffixes smartly using ranking ideas.

In Java code (that I showed earlier):

- Rank each suffix by its first character initially.
- Then sort by first k characters at every step.
- At each round, double k (1, 2, 4, 8, ...).
- Recompute ranks after each round.

It's basically a form of **prefix doubling**.

Let's say we have `s = "banana"`

Round 1:

Rank by first character:

```
rust                                ⌂ Copy ⌂ Edit
a -> 97
b -> 98
n -> 110
```

Round 2:

Sort by first 2 characters:

(b, a), (a, n), (n, a)...

Continue doubling.

Finally you get the sorted order!

🔥 Summary Table

Concept	Meaning
What it stores	Sorted order of suffix starting indexes
How it is built	Smart sorting using ranks
Time complexity	$O(N \log^2 N)$ standard, $O(N \log N)$ optimized
Applications	Pattern search, Duplicate substring, Bioinformatics

What is an LCP Array?

LCP stands for Longest Common Prefix.

👉 The LCP Array tells you:

For every two *adjacent suffixes* in the Suffix Array, what is the length of the longest common prefix between them?

Example (Continuing from $s = "banana"$):

First, recall the Suffix Array:

javascript

 Copy  Edit

```
Suffix Array: [5, 3, 1, 0, 4, 2]
```

Which means suffixes in sorted order are:

Position	suffix starting at
0	5 ("a")
1	3 ("ana")
2	1 ("anana")
3	0 ("banana")
4	4 ("na")
5	2 ("nana")

Now, the LCP Array is built like this:

Compare	LCP Length
"a" vs "ana"	1 ("a")
"ana" vs "anana"	3 ("ana")
"anana" vs "banana"	0
"banana" vs "na"	0
"na" vs "nana"	2 ("na")

Thus, the LCP Array is:

csharp

 Copy  Edit

```
[1, 3, 0, 0, 2]
```

✓ Note:

- LCP Array has $n-1$ elements (because it's between adjacent suffixes).
- It measures *common prefix length* between adjacent suffixes.

💬 Why is LCP Array useful?

With Suffix + LCP Array together, you can solve:

- **Longest Repeated Substring** → Max value in LCP Array.
- **Number of Distinct Substrings** → Use total substrings minus repeated ones (from LCP).
- **Substring Search** → Much faster.

In short:

👉 Suffix Array + LCP Array = powerful string processing combo! 🔥

⚡ Quick High-Level Steps to Build LCP Array:

1. Build the Suffix Array.
2. Build an array `rank[i]` which stores the rank (position) of suffix starting at index `i`.
3. For each suffix:
 - Compare it with the previous suffix in the Suffix Array.
 - Count the length of common prefix.

Optimized build is in $O(N)$ using Kasai's Algorithm.

✍ Pseudocode of Kasai's Algorithm (for LCP):

```
java
int[] lcp = new int[n];
int[] rank = new int[n];

// Step 1: Build rank array
for (int i = 0; i < n; i++) {
    rank[suffixArr[i]] = i;
}

// Step 2: Build LCP array
int k = 0;
for (int i = 0; i < n; i++) {
    if (rank[i] == n - 1) {
        k = 0;
        continue;
    }
    int j = suffixArr[rank[i] + 1];
    while (i + k < n && j + k < n && s.charAt(i + k) == s.charAt(j + k)) {
        k++;
    }
    lcp[rank[i]] = k;
    if (k > 0) {
        k--;
    }
}
```

✓ This code builds the LCP array efficiently in $O(N)$ time!

Purpose:

Given a string and its suffix array, Kasai's algorithm efficiently constructs the LCP (Longest Common Prefix) array in $O(N)$ time.

🔧 Why do we need it?

- The Suffix Array tells us the order of suffixes sorted lexicographically.
- The LCP Array tells us for each consecutive pair of suffixes, how many starting characters they share.

Normally, comparing suffixes takes $O(N^2)$, but Kasai algorithm smartly reuses previous computations to achieve $O(N)$.

📦 How Kasai Works (Step-by-Step):

1. Create a rank array:

- `rank[i]` = the index of suffix starting at position `i` in the suffix array.
- Helps in finding **neighbor suffixes** quickly.

2. Iterate through the string:

- For each position `i`, find its next suffix (`suffixArr[rank[i] + 1]`).
- Compare characters one by one from the two suffixes starting at `i` and `j`.
- Keep a counter `k` for the number of matching characters.
- Store `k` in the `lcp[rank[i]]`.

3. Optimization:

- When moving from `i` to `i+1`, the matching prefix reduces by at most 1.
- So you start next comparison from `k-1` instead of starting fresh from `0`.

```
public static int[] buildLCP(String s, int[] suffixArr) {  
    int n = s.length();  
    int[] lcp = new int[n];  
    int[] rank = new int[n];  
  
    // Step 1: Build rank array  
    for (int i = 0; i < n; i++) {  
        rank[suffixArr[i]] = i;  
    }  
  
    int k = 0;  
  
    // Step 2: Build LCP array  
    for (int i = 0; i < n; i++) {  
        if (rank[i] == n - 1) {  
            k = 0;  
            continue;  
        }  
  
        int j = suffixArr[rank[i] + 1];  
  
        while (i + k < n && j + k < n && s.charAt(i + k) == s.charAt(j + k)) {  
            k++;  
        }  
  
        lcp[rank[i]] = k;  
    }  
}
```

```

        if (k > 0) {
            k--;
        }

        return lcp;
    }
}

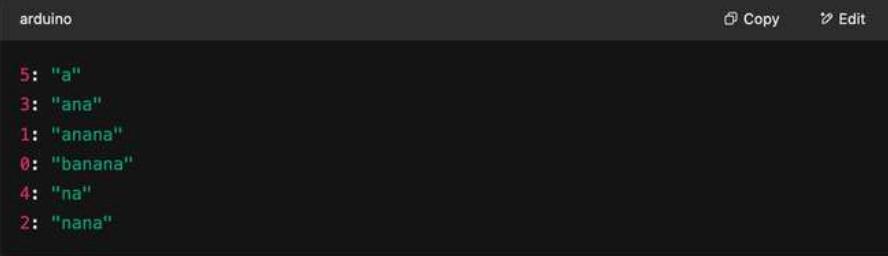
```

⌚ Time Complexity:

Step	Time
Building rank array	$O(N)$
Building LCP array	$O(N)$
Total	$O(N)$

For the string "banana" :

- Suffixes sorted:



The screenshot shows an Arduino IDE interface with a code editor containing the following code:

```

arduino
Copy Edit

5: "a"
3: "ana"
1: "anana"
0: "banana"
4: "na"
2: "nana"

```

- Suffix Array: [5, 3, 1, 0, 4, 2]
- Kasai's algorithm builds the LCP Array: [1, 3, 0, 0, 2, 0]

Meaning:

- "a" vs "ana" => LCP = 1
- "ana" vs "anana" => LCP = 3
- "anana" vs "banana" => LCP = 0
- "banana" vs "na" => LCP = 0
- "na" vs "nana" => LCP = 2
- Kasai efficiently computes the LCP array.
- It heavily reuses previous computation (carrying forward the matching character count).
- It's often used with Suffix Arrays to solve problems like:
 - Number of distinct substrings
 - Finding longest repeated substring
 - Compression problems


```

    // Build LCP array
    for (int i = 0; i < n; i++) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }

        int j = suffixArr[rank[i] + 1];

        while (i + k < n && j + k < n && s.charAt(i + k) == s.charAt(j + k)) {
            k++;
        }

        lcp[rank[i]] = k;

        if (k > 0) {
            k--;
        }
    }

    return lcp;
}

// Suffix class to store suffix string and original index
static class Suffix {
    String text;
    int index;

    Suffix(String text, int index) {
        this.text = text;
        this.index = index;
    }
}
}

// Driver Code
public static void main(String[] args) {
    String s = "banana";

    int[] suffixArr = buildSuffixArray(s);
    int[] lcp = buildLCP(s, suffixArr);

    System.out.println("Suffix Array:");
    for (int i = 0; i < suffixArr.length; i++) {
        System.out.println(suffixArr[i] + ": " + s.substring(suffixArr[i]));
    }

    System.out.println("\nLCP Array:");
    for (int i = 0; i < lcp.length; i++) {
        System.out.println("LCP[" + i + "] = " + lcp[i]);
    }
}
}

```

Suffix Array:

5: a	
3: ana	
1: anana	
0: banana	
4: na	
2: nana	

LCP Array:

LCP[0] = 1	
LCP[1] = 3	
LCP[2] = 0	
LCP[3] = 0	
LCP[4] = 2	
LCP[5] = 0	

You do:

1. Build a **rank array**:

`rank[i]` = index of suffix starting at `i` in `suffixArr`.

2. Initialize `k = 0` (current LCP length).

3. For each character position `i` from 0 to `n-1`:

- If `rank[i] == n-1`, skip (because there's no next suffix to compare).
- Find the next suffix `j = suffixArr[rank[i] + 1]`.
- While characters at `i+k` and `j+k` match, increment `k`.
- Set `lcp[rank[i]] = k`.
- Decrement `k` by 1 if `k > 0` (carry forward).

Number of Distinct Substrings.

Formula:

- Total number of substrings = $n * (n + 1) / 2$
- Total number of **distinct substrings** =
Total substrings – Sum of all LCPs

Because:

- $LCP[i]$ tells you how much **overlap** there is between adjacent suffixes.
- That overlapping part should **not** be double-counted.

```
import java.util.Arrays;
import java.util.Comparator;

public class DistinctSubstrings {

    public static int[] buildSuffixArray(String s) {
        int n = s.length();
        Suffix[] suffixes = new Suffix[n];

        for (int i = 0; i < n; i++) {
            suffixes[i] = new Suffix(s.substring(i), i);
        }

        Arrays.sort(suffixes, Comparator.comparing(a -> a.text));

        int[] suffixArr = new int[n];
        for (int i = 0; i < n; i++) {
            suffixArr[i] = suffixes[i].index;
        }

        return suffixArr;
    }

    public static int[] buildLCP(String s, int[] suffixArr) {
        int n = s.length();
        int[] lcp = new int[n];
        int[] rank = new int[n];

        for (int i = 0; i < n; i++) {
            rank[suffixArr[i]] = i;
        }

        int k = 0;
        for (int i = 0; i < n; i++) {
            if (rank[i] == n - 1) {
                k = 0;
                continue;
            }

            int j = suffixArr[rank[i] + 1];
            while (i + k < n && j + k < n && s.charAt(i + k) == s.charAt(j + k)) {
                k++;
            }

            lcp[rank[i]] = k;

            if (k > 0) {
                k--;
            }
        }

        return lcp;
    }
}
```

```

public static int countDistinctSubstrings(String s) {
    int n = s.length();
    int[] suffixArr = buildSuffixArray(s);
    int[] lcp = buildLCP(s, suffixArr);

    int totalSubstrings = n * (n + 1) / 2;
    int sumOfLCP = 0;
    for (int x : lcp) {
        sumOfLCP += x;
    }

    return totalSubstrings - sumOfLCP;
}

static class Suffix {
    String text;
    int index;

    Suffix(String text, int index) {
        this.text = text;
        this.index = index;
    }
}

public static void main(String[] args) {
    String s = "banana";
    int result = countDistinctSubstrings(s);
    System.out.println("Number of distinct substrings = " + result);
}
}

```

🏁 Output for "banana":

java

Number of **distinct substrings** = 15

✨ Quick Recap:

Step	Meaning
Build Suffix Array	Sort suffixes lexicographically
Build LCP Array	Find overlaps between adjacent suffixes
Use Formula	$(n * (n+1)/2) - \text{sum}(lcp)$

Example Walkthrough for "banana":

Suffix

a

ana

anana

banana

na

nana

- Total substrings = $6 * 7 / 2 = 21$
- Sum of LCPs = $1 + 3 + 0 + 0 + 2 = 6$
- Distinct substrings = $21 - 6 = 15$

Longest Duplicate Substring

Idea:

1. Suffix Array:

List all suffixes of the string in **sorted order**.

2. LCP Array:

For adjacent suffixes, find the **Longest Common Prefix**.

3. Answer:

The maximum value in the LCP array gives the length of the **longest duplicate substring**.

```
public static String longestDupSubstring(String s) {
    int[] suffix = buildSuffixArray(s);
    int[] lcp = buildLCPArray(s, suffix);

    int maxLen = 0;
    int startIndex = -1;
    for (int i = 0; i < lcp.length; i++) {
        if (lcp[i] > maxLen) {
            maxLen = lcp[i];
            startIndex = suffix[i];
        }
    }

    return maxLen > 0 ? s.substring(startIndex, startIndex + maxLen) : "";
}

private static int[] buildSuffixArray(String s) {
    int n = s.length();
    Suffix[] suffixes = new Suffix[n];

    for (int i = 0; i < n; i++) {
        suffixes[i] = new Suffix(s.substring(i), i);
    }

    Arrays.sort(suffixes);

    int[] suffixArr = new int[n];
    for (int i = 0; i < n; i++) {
        suffixArr[i] = suffixes[i].index;
    }

    return suffixArr;
}
```

```
private static int[] buildLCPArray(String s, int[] suffix) {
    int n = s.length();
    int[] lcp = new int[n];
    int[] rank = new int[n];

    for (int i = 0; i < n; i++) {
        rank[suffix[i]] = i;
    }

    int k = 0;
    for (int i = 0; i < n; i++) {
        if (rank[i] == 0) {
            k = 0;
            continue;
        }

        int j = suffix[rank[i] - 1];
        while (i + k < n && j + k < n && s.charAt(i + k) == s.charAt(j + k)) {
            k++;
        }
        lcp[rank[i]] = k;

        if (k > 0) {
            k--;
        }
    }

    return lcp;
}

private static class Suffix {
    String str;
    int index;

    Suffix(String str, int index) {
        this.str = str;
        this.index = index;
    }
}

public static void main(String[] args) {
    String s = "banana";
    System.out.println(longestDupSubstring(s)); // Output: "ana"
}
```

Example Walkthrough (for "banana"):

Suffixes after sorting:

makefile

 Copy

 Edit

```
0: a  
1: ana  
2: anana  
3: banana  
4: na  
5: nana
```

LCP array:

csharp

 Copy

 Edit

```
[0, 1, 3, 0, 0, 2]
```

(max LCP = 3 between "ana" and "anana" → substring = "ana")

So output = "ana"



Time Complexity:

Part	Complexity
Build Suffix Array (naive)	$O(N^2 \log N)$
Build LCP Array	$O(N)$

Longest Common Prefix (LCP across multiple strings) – using Suffix Array + LCP Array.

Idea

- **Suffix Array:**
Sort all suffixes of all the strings combined.
- **LCP Array:**
Find the longest common prefix between adjacent suffixes belonging to different strings.
- **Answer:**
The longest such common prefix is the answer.

Plan:

1. Merge strings with unique delimiters

We combine all strings with unique separators (like `#`, `$`, `%` etc.) because otherwise suffixes will overlap incorrectly.

Example:

Input: `["banana", "bandana"]`

Merged: `"banana#bandana$"`

(Note: `#` and `$` are characters that do not appear in original strings.)

2. Build Suffix Array

Build suffix array for the merged string.

3. Build LCP Array

Compute LCP for adjacent suffixes.

4. Check only between suffixes from different original strings

Because LCP between suffixes of the same string doesn't matter — we want LCP across *different strings*.

Time Complexity:

Part	Complexity
Building Suffix Array (naive way)	$O(N^2 \log N)$
Building LCP Array	$O(N)$

Input:

plaintext

Copy Edit

```
["banana", "bandana", "ban"]
```

Merged string:

plaintext

Copy Edit

```
"banana#bandana$ban%"
```

Suffix Array (after sorting):

plaintext

Copy Edit

```
%  
# (delimiter)  
$ (delimiter)  
ana#  
anana#  
and so on...
```

LCP array built.

Check LCPs between suffixes from *different original strings*.

"banana" and "bandana" have common "ban".

Thus output = "ban"

```
import java.util.*;

public class MultiStringLCP {

    static class Suffix implements Comparable<Suffix> {
        int index;
        String suffix;

        Suffix(int index, String suffix) {
            this.index = index;
            this.suffix = suffix;
        }

        public int compareTo(Suffix other) {
            return this.suffix.compareTo(other.suffix);
        }
    }

    // Helper to determine source string for a given suffix index
    static int getSourceIndex(int index, int[] boundaries) {
        for (int i = 0; i < boundaries.length - 1; i++) {
            if (index >= boundaries[i] && index < boundaries[i + 1] - 1) {
                return i;
            }
        }
        return -1;
    }

    // Build suffix array for a string
    public static int[] buildSuffixArray(String fullString) {
        int n = fullString.length();
        Suffix[] suffixes = new Suffix[n];
        for (int i = 0; i < n; i++) {
            suffixes[i] = new Suffix(i, fullString.substring(i));
        }
        Arrays.sort(suffixes);

        int[] suffixArr = new int[n];
        for (int i = 0; i < n; i++) {
            suffixArr[i] = suffixes[i].index;
        }
        return suffixArr;
    }
}
```

```

// Build LCP array using Kasai's algorithm
public static int[] buildLCPArray(String fullString, int[] suffixArr) {
    int n = fullString.length();
    int[] rank = new int[n];
    for (int i = 0; i < n; i++) {
        rank[suffixArr[i]] = i;
    }

    int[] lcp = new int[n];
    int k = 0;
    for (int i = 0; i < n; i++) {
        if (rank[i] == n - 1) {
            k = 0;
            continue;
        }

        int j = suffixArr[rank[i] + 1];
        while (i + k < n && j + k < n && fullString.charAt(i + k) == fullString.charAt(j + k)) {
            k++;
        }

        lcp[rank[i]] = k;
        if (k > 0) k--;
    }
    return lcp;
}

// Main logic to compute LCP across multiple strings
public static String longestCommonPrefix(String[] strings) {
    if (strings == null || strings.length == 0) return "";

    // Step 1: Concatenate strings with unique delimiters
    StringBuilder combined = new StringBuilder();
    List<Character> delimiters = new ArrayList<>(Arrays.asList('#', '$', '@', '%', '^', '&', '*', '!', '~', '|'));
    int[] boundaries = new int[strings.length + 1];
    boundaries[0] = 0;

    for (int i = 0; i < strings.length; i++) {
        combined.append(strings[i]).append(delimiters.get(i));
        boundaries[i + 1] = combined.length(); // start index of next
    }

    String fullString = combined.toString();
    int[] suffixArr = buildSuffixArray(fullString);
    int[] lcp = buildLCPArray(fullString, suffixArr);

    // Step 4: Find longest LCP between suffixes from different sources
    int maxLen = 0;
    int startIndex = -1;
    for (int i = 1; i < fullString.length(); i++) {
        int src1 = getSourceIndex(suffixArr[i - 1], boundaries);
        int src2 = getSourceIndex(suffixArr[i], boundaries);
        if (src1 != src2 && src1 != -1 && src2 != -1) {
            if (lcp[i - 1] > maxLen) {
                maxLen = lcp[i - 1];
                startIndex = suffixArr[i];
            }
        }
    }

    return (startIndex != -1 && maxLen > 0) ? fullString.substring(startIndex, startIndex + maxLen) : "";
}

public static void main(String[] args) {
    String[] strings = { "interspecies", "interstellar", "interstate" };
    System.out.println("LCP: " + longestCommonPrefix(strings)); // Output: "inters"
}

```

```

> p strings = {String[3]@450} ["interspecies", "interstellar", "interstate"]
> s combined = {StringBuilder@451} "interspecies#interstellar$interstate@"
> s delimiters = {ArrayList@452} size = 10
> s boundaries = {int[4]@453} [0, 13, 26, 37]

```

```

// Get the actual longest common prefix string using LCP array
public static String getLongestCommonPrefix(String s) {
    int[] suffixArr = buildSuffixArray(s);
    int[] lcp = buildLCP(s, suffixArr);

    int maxlen = 0;
    int index = 0;

    for (int i = 0; i < lcp.length; i++) {
        if (lcp[i] > maxlen) {
            maxlen = lcp[i];
            index = suffixArr[i];
        }
    }

    return s.substring(index, index + maxlen);
}

public static void main(String[] args) {
    String s = "banana";
    String lcp = getLongestCommonPrefix(s);
    System.out.println("Longest Common Prefix among suffixes: " + lcp);
}

```

String Matching

When we talk about String Matching problems on LeetCode, it refers to **finding patterns, comparing substrings, or analyzing relationships between one or more strings.**

In simple words:

- 👉 Search for a pattern inside a string (or compare multiple strings) — fast and efficiently.

★ Common Types of String Matching Problems:

Type	Example Problems	Description
Exact Match	<code>strStr()</code> (Find the Index of First Occurrence)	Find if one string is exactly inside another.
Substring Search	<code>Longest Duplicate Substring</code> , <code>Repeated Substring Pattern</code>	Find duplicates or repeated patterns.
Prefix/Suffix Matching	<code>Longest Common Prefix</code> , <code>Longest Happy Prefix</code>	Match prefixes or suffixes across multiple strings.
Pattern Matching with Wildcards	<code>Wildcard Matching</code> , <code>Regular Expression Matching</code>	Match strings allowing <code>?</code> , <code>*</code> , etc.
Anagram Matching	<code>Find All Anagrams in a String</code>	Match strings by their frequency counts (not exact order).

Knuth and Morris Algorithm

☰ Knuth-Morris-Pratt (KMP) Algorithm

KMP is a string matching algorithm that finds the first occurrence of a pattern (`P`) in a text (`T`) in $O(N + M)$ time.

(N = length of text, M = length of pattern)

Instead of checking all characters again when a mismatch happens, KMP uses precomputed information about the pattern itself to avoid unnecessary checks.

⌚ Why KMP?

- 👉 **Naive approach:** For each character in text, you check if the pattern starts there — $O(N \times M)$ worst case.
- 👉 **KMP:** Preprocessing saves work — $O(N + M)$ time!

How does KMP work?

Two major steps:

1. Preprocessing: Build an array called `LPS` (Longest Prefix which is also Suffix) for the pattern.
2. Searching: Use `LPS` to skip characters smartly when mismatch occurs during search.

Step 1: Build the LPS Array

- `LPS[i]` means: length of the longest prefix which is also suffix for the substring `pattern[0..i]`.
- It tells where to `restart` in the pattern when mismatch happens.

Example:

```
makefile          ⌂ Copy ⌂ Edit

Pattern: a b a b a c
LPS:      0 0 1 2 3 0
```

Meaning:

- At index 5 (char 'c'), there's no prefix==suffix, so `LPS[5] = 0`.
- At index 4 (char 'a'), the longest prefix==suffix is "aba", so `LPS[4] = 3`.

Step 2: Search Using the LPS

- If characters match, move ahead in both text and pattern.
- If mismatch happens after some matches:
 - Use `lps[j-1]` to reset the pattern pointer, **not** the text pointer.
 - This skips rechecking already matched characters.



Time and Space Complexity:

	Complexity
Preprocessing (<code>buildLPS</code>)	$O(M)$
Search	$O(N)$
Overall	$O(N + M)$
Extra Space	$O(M)$ (for LPS array)

🎯 Summary:

- ✓ Preprocessing the pattern with LPS
- ✓ Avoid rechecking text characters
- ✓ Search in **linear time** instead of quadratic

⭐ Why KMP is legendary:

It was one of the first algorithms to show that **pattern matching can be done in linear time** — a BIG breakthrough for Computer Science (1970s)! 

```
public class Solution {

    public static int strStr(String haystack, String needle) {
        if (needle.isEmpty()) {
            return 0;
        }

        int[] lps = buildLPS(needle);
        int i = 0; // pointer for haystack
        int j = 0; // pointer for needle

        while (i < haystack.length()) {
            if (haystack.charAt(i) == needle.charAt(j)) {
                i++;
                j++;
            }

            if (j == needle.length()) {
                return i - j; // match found
            } else if (i < haystack.length() && haystack.charAt(i) != needle.charAt(j)) {
                if (j != 0) {
                    j = lps[j - 1]; // use LPS to shift pattern
                } else {
                    i++;
                }
            }
        }
    }

    return -1; // no match found
}
}
```

```
private static int[] buildLPS(String pattern) {
    int[] lps = new int[pattern.length()];
    int len = 0; // length of the previous longest prefix suffix
    int i = 1;

    while (i < pattern.length()) {
        if (pattern.charAt(i) == pattern.charAt(len)) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1]; // try previous prefix
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }

    return lps;
}

public static void main(String[] args) {
    String haystack = "abxabcabcaby";
    String needle = "abcaby";
    int result = strStr(haystack, needle);
    System.out.println("First Occurrence Index: " + result);
}
```

Naive String Matching

In the naive approach, we try **every possible starting index** in the `text` where the `pattern` could match.

At every index, we **compare the characters one by one**.

Steps:

1. Start from index `i = 0` in `text`.
2. For each index `i`, try to match the pattern:
 - Compare `text[i + j]` with `pattern[j]` for all `j`.
3. If all characters match, **pattern is found**.
4. Otherwise, move to the next index `i + 1`.

```
public class NaiveStringMatching {  
    public static int strStr(String haystack, String needle) {  
        int n = haystack.length();  
        int m = needle.length();  
  
        if (m == 0) {  
            return 0; // Empty needle  
        }  
  
        for (int i = 0; i <= n - m; i++) {  
            int j;  
            for (j = 0; j < m; j++) {  
                if (haystack.charAt(i + j) != needle.charAt(j)) {  
                    break; // mismatch found  
                }  
            }  
            if (j == m) {  
                return i; // match found at index i  
            }  
        }  
  
        return -1; // no match found  
    }  
  
    public static void main(String[] args) {  
        String haystack = "abxabcabcaby";  
        String needle = "abcaby";  
        int result = strStr(haystack, needle);  
        System.out.println("First Occurrence Index: " + result);  
    }  
}
```

Time and Space Complexity:

Complexity	
Time (Worst Case)	$O(N \times M)$
Space	$O(1)$ (no extra space needed)

- N = length of text
 - M = length of pattern
-

📦 Example:

Text: "abcxabcdabcdabcy"

Pattern: "abcdabcy"

Try matching starting from each position until a match is found!

🎯 Why it's slower?

- Suppose Text = "aaaaaaaa..." (10000 a's)
- Pattern = "aaaab"

At each character, you almost match most of the pattern but fail at the last character.

➡ You end up checking almost the whole pattern M times at every index!

➡ Leads to $O(N \times M)$ time.

⭐ Quick Comparison:

Approach	Time Complexity	Space Complexity	Smartness
Naive	$O(N \times M)$	$O(1)$	Brute Force
KMP	$O(N + M)$	$O(M)$	Smart (uses pattern info)

Knuth-Morris-Pratt (KMP) Algorithm

KMP is a **string matching algorithm** that finds the first occurrence of a pattern (P) in a text (T) in $O(N + M)$ time.

(N = length of text, M = length of pattern)

Instead of checking all characters again when a mismatch happens, KMP uses **precomputed information** about the pattern itself to avoid unnecessary checks.

Why KMP?

- 👉 **Naive approach:** For each character in text, you check if the pattern starts there — $O(N \times M)$ worst case.
- 👉 **KMP:** Preprocessing saves work — $O(N + M)$ time!

How does KMP work?

Two major steps:

1. **Preprocessing:** Build an array called LPS (Longest Prefix which is also Suffix) for the pattern.
2. **Searching:** Use LPS to skip characters smartly when mismatch occurs during search.

Step 1: Build the LPS Array

- `LPS[i]` means: length of the longest prefix which is also suffix for the substring `pattern[0..i]`.
- It tells where to `restart` in the pattern when mismatch happens.

Example:

```
makefile
Pattern: a b a b a c
LPS:      0 0 1 2 3 0
```

Meaning:

- At index 5 (char 'c'), there's no prefix==suffix, so `LPS[5] = 0`.
- At index 4 (char 'a'), the longest prefix==suffix is "aba", so `LPS[4] = 3`.

Step 2: Search Using the LPS

- If characters match, move ahead in both text and pattern.
- If mismatch happens after some matches:
 - Use `lps[j-1]` to reset the pattern pointer, **not** the text pointer.
 - This skips rechecking already matched characters.

```
public class Solution {

    public static int strStr(String haystack, String needle) {
        if (needle.isEmpty()) {
            return 0;
        }

        int[] lps = buildLPS(needle);
        int i = 0; // pointer for haystack
        int j = 0; // pointer for needle

        while (i < haystack.length()) {
            if (haystack.charAt(i) == needle.charAt(j)) {
                i++;
                j++;
            }

            if (j == needle.length()) {
                return i - j; // match found
            } else if (i < haystack.length() && haystack.charAt(i) != needle.charAt(j)) {
                if (j != 0) {
                    j = lps[j - 1]; // use LPS to shift pattern
                } else {
                    i++;
                }
            }
        }

        return -1; // no match found
    }
}
```

```

private static int[] buildLPS(String pattern) {
    int[] lps = new int[pattern.length()];
    int len = 0; // length of the previous longest prefix suffix
    int i = 1;

    while (i < pattern.length()) {
        if (pattern.charAt(i) == pattern.charAt(len)) {
            len++;
            lps[i] = len;
            i++;
        } else {
            if (len != 0) {
                len = lps[len - 1]; // try previous prefix
            } else {
                lps[i] = 0;
                i++;
            }
        }
    }

    return lps;
}

public static void main(String[] args) {
    String haystack = "abxabcabcaby";
    String needle = "abcaby";
    int result = strStr(haystack, needle);
    System.out.println("First Occurrence Index: " + result);
}

```

i	char	lps[i]
0	a	0
1	b	0
2	a	1
3	b	2
4	a	3
5	c	0
6	a	1

Suppose at index 6, we find a mismatch between pattern and text.

- Current $\text{len} = 3$ (we matched "abc").
- But next char doesn't match.

Then we don't restart matching from 0.

Instead, we try with $\text{len} = \text{lps}[\text{len} - 1] = \text{lps}[2] = 1$.

Meaning:

🧠 Context

- You're **building or using the LPS array** (Longest Prefix Suffix array).
- **`len` = length of the longest proper prefix which is also a suffix** for the substring considered so far.
- **`lps[i]` = for the string from index `0` to `i`, what's the length of the longest prefix which is also a suffix.**

Meaning of this code:

- When you are comparing two characters and they don't match,
- Instead of starting all over again (going back to index 0 of pattern),
- You reuse the information from the LPS array to avoid redundant comparisons.

So when mismatch happens:

- 👉 Instead of trying a fresh start,
- 👉 You "fall back" to the last best prefix length,
- 👉 i.e., `len = lps[len - 1];`

You shrink your pattern intelligently, based on previously computed matches.

Suppose `pattern = "ababaca"`

and currently:

i	char	lps[i]
0	a	0
1	b	0
2	a	1
3	b	2
4	a	3
5	c	0
6	a	1

Suppose at index 5, we find a mismatch between pattern and text.

- Current `len = 3` (we matched "aba").
- But next char doesn't match.

Then we don't restart matching from 0.

Instead, we try with `len = lps[len - 1] = lps[2] = 1`.

Meaning:

Maybe the prefix "a" (length 1) can still help!

And you keep adjusting until `len = 0` or a match happens.

Visually:

```
pgsql
pattern = "ababc"
text window = "ababa"

Mismatch at i=4, len=3
=> Instead of restarting, fallback to lps[2]=1
=> Try match from 1-length prefix.
```

🎯 Rabin-Karp Algorithm

- Rabin-Karp is a **string matching algorithm**.
- It uses **hashing** to quickly compare the pattern with substrings of the text.
- **Idea:** Instead of comparing full strings every time, compare their **hash values** (integers).
- If hashes match, then **only do a real character-by-character comparison** to confirm.

🔥 What is a **spurious hit**?

Spurious hit = False alarm.

- 👉 Hash values match,
- 👉 but actual substrings don't match.

Because:

- Hashing can cause *collisions* (different strings having the same hash value).
- ✅ So Rabin-Karp **checks characters** again when hashes match.

Suppose:

```
text          ⚡ Copy ⚡ Edit
Text: "abcde"
Pattern: "abc"
```

Let's say we use a simple hash function like:

```
java          ⚡ Copy ⚡ Edit
hash(s) = sum of ASCII values of characters
```

- $\text{hash("abc")} = 97 + 98 + 99 = 294$
- $\text{hash("bcd")} = 98 + 99 + 100 = 297$
- $\text{hash("cde")} = 99 + 100 + 101 = 300$

No problem here.

BUT, in real cases, two *different* substrings might **coincidentally** have the same hash value.

In that case:

- Rabin-Karp thinks it found a match (hash equal),
- But after **comparing characters**, realizes they're different.

This mistake = **spurious hit**.

Yes, Rabin-Karp uses the rolling hash method – that is its core idea.



How Rabin-Karp handles spurious hits?

Step	Action
Hash matches	Check actual characters one-by-one
If strings match	Confirm pattern found
If strings differ	Ignore (spurious hit)

Thus, correctness is not affected — only performance may suffer slightly if too many spurious hits occur.



Important points about Spurious Hits

Cause	Hash collisions
Effect	Extra comparison step
Prevention	Use a good hash function (like rolling hash with prime mod)
Real-world	Rare if hash function is strong (but not impossible)



Summary

Term	Meaning
Rabin-Karp	Hash-based string matching
Spurious hit	Hashes match but actual strings don't
Action taken	Re-verify by comparing strings



Visual:

```
bash
[hash("pattern")] == [hash("some text substring")]
|
Compare actual characters
|
[Match] OR [Spurious Hit]
```

Would you also like a [short code snippet](#) showing an actual *spurious hit* happening in Rabin-Karp?

```
text
Pattern found at index 0
Spurious hit at index 9
Pattern found at index 12
```

(Notice the **spurious hit** printed at index 9 — means hash matched but actual characters didn't.)

```

public class RabinKarp {

    static final int ALPHABET_SIZE = 256; // Number of characters in the alphabet (ASCII)
    static final int PRIME_MOD = 101; // A prime number for modulo operation

    static void searchPattern(String pattern, String text) {
        int patternLength = pattern.length();
        int textLength = text.length();
        int patternHash = 0; // Hash value for pattern
        int windowHash = 0; // Hash value for current window of text
        int highOrderMultiplier = 1; // The value of pow(d, M-1) % q
        // Precompute (ALPHABET_SIZE^(patternLength-1)) % PRIME_MOD
        for (int i = 0; i < patternLength - 1; i++) {
            highOrderMultiplier = (highOrderMultiplier * ALPHABET_SIZE) % PRIME_MOD;
        }
        // Calculate the initial hash value for pattern and first window of text
        for (int i = 0; i < patternLength; i++) {
            patternHash = (ALPHABET_SIZE * patternHash + pattern.charAt(i)) % PRIME_MOD;
            windowHash = (ALPHABET_SIZE * windowHash + text.charAt(i)) % PRIME_MOD;
        }
        // Slide the window over the text
        for (int i = 0; i <= textLength - patternLength; i++) {
            // Check the hash values
            if (patternHash == windowHash) {
                boolean exactMatch = true;
                // Check characters one-by-one if hashes match
                for (int j = 0; j < patternLength; j++) {
                    if (text.charAt(i + j) != pattern.charAt(j)) {
                        exactMatch = false;
                        break;
                    }
                }
                if (exactMatch) {
                    System.out.println("Pattern found at index " + i);
                } else {
                    System.out.println("Spurious hit at index " + i);
                }
            }
            // Calculate hash for the next window
            if (i < textLength - patternLength) {
                windowHash = (ALPHABET_SIZE * (windowHash - text.charAt(i) * highOrderMultiplier) + text.charAt(i + patternLength)) % PRIME_MOD;
                // Ensure the hash value is positive
                if (windowHash < 0) {
                    windowHash += PRIME_MOD;
                }
            }
        }
    }

    public static void main(String[] args) {
        String text = "AABAACAADAABAABA";
        String pattern = "AABA";
        searchPattern(pattern, text);
    }
}

```

```

    // Slide over the text
    for (int i = 0; i <= n - m; i++) {
        if (pHash == tHash) {
            // Verify match to rule out spurious hit
            if (text.regionMatches(i, pattern, 0, m)) {
                System.out.println("Pattern found at index " + i);
            } else {
                spuriousHits++;
            }
        }
    }

```

◀ If a spurious hit occurs:

- Detect it (hashes match but characters differ).
- Ignore it, and just continue checking the next window.
- Optionally, you could switch to a fallback strategy (e.g., full match check) if spurious hits are too frequent — but that's rare.

Find All Anagrams Of A String

🔥 Problem: Find All Anagrams in a String

Given:

- Two strings `s` and `p`

Goal:

- Find all start indices of `p`'s anagrams in `s`.

Example:

```
vbnet Copy Edit
Input: s = "cbaebabacd", p = "abc"
Output: [0,6]
Explanation:
- "cha" at index 0 is an anagram of "abc"
- "bac" at index 6 is an anagram of "abc"
```

💡 Quick Explanation:

- `patternFreq`: frequency count of characters in `p`
- `windowFreq`: frequency count of current window in `s`
- Slide a window of size `p.length()` across `s`
- Compare frequency arrays using a helper method `matches()`
- If match → add starting index to result list ✓

⌚ Time Complexity:

- Building frequency arrays: $O(26)$ → constant
- Sliding the window: $O(n)$
- Comparing arrays at each step: $O(26)$ → constant

Total: $O(n)$ time and $O(1)$ space (since 26 characters only).

```
public class Solution {

    public static List<Integer> findAnagrams(String text, String pattern) {
        List<Integer> resultIndices = new ArrayList<>();
        if (text.length() < pattern.length()) {
            return resultIndices;
        }

        int[] patternFreq = new int[26];
        int[] windowFreq = new int[26];

        int windowHeight = pattern.length();

        // Build frequency array for the pattern
        for (char c : pattern.toCharArray()) {
            patternFreq[c - 'a']++;
        }

        // Build the first window
        for (int i = 0; i < windowHeight; i++) {
            windowFreq[text.charAt(i) - 'a']++;
        }

        // Compare first window
        if (matches(patternFreq, windowFreq)) {
            resultIndices.add(0);
        }

        // Slide the window
        for (int i = windowHeight; i < text.length(); i++) {
            // Add new character
            windowFreq[text.charAt(i) - 'a']++;
            // Remove character left out of window
            windowFreq[text.charAt(i - windowHeight) - 'a']--;

            if (matches(patternFreq, windowFreq)) {
                resultIndices.add(i - windowHeight + 1);
            }
        }

        return resultIndices;
    }

    // Helper function to compare two frequency arrays
    private static boolean matches(int[] freq1, int[] freq2) {
        for (int i = 0; i < 26; i++) {
            if (freq1[i] != freq2[i]) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        String text = "cbaebabacd";
        String pattern = "abc";

        List<Integer> anagramIndices = findAnagrams(text, pattern);
        System.out.println(anagramIndices); // Output: [0, 6]
    }
}
```

Deterministic And Non Deterministic Algorithms

Algorithms can be broadly classified into **deterministic** and **non-deterministic** based on how they behave for a given input. Let's go through both in detail:

1. Deterministic Algorithms

A **deterministic algorithm** is an algorithm that, given a particular input, will always produce the same output with a predictable execution flow.

Characteristics:

- **Predictable Outcome:** For every input, the outcome and execution path are always the same.
- **No Randomness:** The result is entirely determined by the input and the algorithm's logic.
- **Consistency:** If you run the algorithm multiple times with the same input, it will always produce the same output.

Example:

- **Binary Search:** Given a sorted array and a target value, binary search will consistently find the target (if present) or determine that it's absent in the array. The same input will always yield the same result.

```
java
int binarySearch(int[] arr, int target) {
    int low = 0, high = arr.length - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) {
            return mid;
        } else if (arr[mid] < target) {
            low = mid + 1;
        } else {
            high = mid - 1;
        }
    }
    return -1; // Target not found
}
```

Time Complexity:

- Typically $O(\log N)$ for algorithms like binary search, merge sort, etc.

2. Non-Deterministic Algorithms

A **non-deterministic algorithm** is an algorithm that may exhibit different outcomes for the same input on different runs. The algorithm's path or result is influenced by randomness or non-deterministic choices.

Characteristics:

- **Unpredictable Outcome:** The same input can lead to different outputs on different executions.
- **Randomness or Choices:** It may involve randomization, backtracking, or the algorithm may have multiple paths to explore.
- **Multiple Possible Solutions:** The algorithm might explore multiple possibilities and choose one at random or via some optimization criterion.

Example:

- **Randomized Algorithms:** These algorithms use random values to influence their behavior. For instance, the **Quicksort** algorithm uses a random pivot selection, which can lead to different performance outcomes (best case, worst case) depending on the chosen pivot.

```

// Randomized QuickSort
void randomizedQuickSort(int[] arr, int low, int high) {
    if (low < high) {
        int pivot = randomizedPartition(arr, low, high);
        randomizedQuickSort(arr, low, pivot - 1);
        randomizedQuickSort(arr, pivot + 1, high);
    }
}

int randomizedPartition(int[] arr, int low, int high) {
    int pivotIndex = low + (int) (Math.random() * (high - low + 1));
    int pivot = arr[pivotIndex];
    swap(arr, pivotIndex, high); // Swap pivot with high
    return partition(arr, low, high, pivot);
}

```

Time Complexity:

- Non-deterministic algorithms typically have randomized behavior, meaning their average-case time complexity is different from their worst-case complexity. For example, randomized quicksort has an average complexity of $O(N \log N)$, but its worst-case can be $O(N^2)$ depending on the pivot selection.

Time Complexity:

- Non-deterministic algorithms typically have randomized behavior, meaning their average-case time complexity is different from their worst-case complexity. For example, randomized quicksort has an average complexity of $O(N \log N)$, but its worst-case can be $O(N^2)$ depending on the pivot selection.

Comparison:

Feature	Deterministic Algorithms	Non-Deterministic Algorithms
Predictability	Always produces the same output for the same input.	May produce different results for the same input.
Control Flow	Follows a single, fixed sequence of operations.	May follow different paths or involve randomness.
Example	Merge Sort, Quick Sort, Binary Search.	Randomized QuickSort, Monte Carlo Methods, Backtracking.
Time Complexity	Fixed and predictable (e.g., $O(\log N)$, $O(N \log N)$).	Can vary depending on the execution path.
Common Use Cases	Applications where consistency is crucial.	Problems that benefit from randomization or exploration of multiple possibilities.

Key Differences:

- **Certainty of Result:**
 - **Deterministic algorithms** are completely predictable and are often used in scenarios requiring consistent results.
 - **Non-deterministic algorithms** can produce different outputs for the same input, which is useful for exploring large or complex search spaces (e.g., optimization problems).
- **Use of Randomness:**
 - Deterministic algorithms do not rely on random decisions.
 - Non-deterministic algorithms use randomness (or backtracking) to explore possible solutions or paths. This randomness can sometimes lead to faster average performance (like in randomized algorithms) or better approximations.
- **Efficiency and Complexity:**
 - **Deterministic algorithms** tend to be more efficient for problems where the behavior is predictable and the problem size is manageable.
 - **Non-deterministic algorithms** may offer efficient solutions for complex problems (like large search spaces), but their results are not guaranteed to be optimal or consistent on every run.

Summary:

- **Deterministic algorithms** are predictable, reliable, and efficient for well-defined problems.
- **Non-deterministic algorithms** use randomness or multiple paths to find solutions and are useful in cases where a deterministic solution might be inefficient or too complex.

NP Hard And NP Complete

NP-Hard vs NP-Complete

In computational complexity theory, **NP-Hard** and **NP-Complete** are classifications of problems based on their computational difficulty. Both terms are used to describe problems in the context of NP (Non-deterministic Polynomial time) problems, but they have distinct meanings.

1. NP-Hard

A problem is **NP-Hard** if it is at least as hard as the hardest problems in NP. This means that an **NP-Hard** problem may or may not be in NP, but it is at least as difficult to solve as any problem in NP.

Key Characteristics:

- **No Restriction on Solution Verification:** The key feature of NP-Hard problems is that they do not necessarily have a solution that can be verified in polynomial time. This distinguishes them from NP-Complete problems.
- **Harder Than NP Problems:** If you can solve an NP-Hard problem in polynomial time, you would be able to solve all NP problems in polynomial time. However, not all NP-Hard problems are in NP, meaning they may not have efficiently verifiable solutions.

Saved memory full ⓘ

Example:

- **Halting Problem:** One of the classic examples of an NP-Hard problem. It asks whether a given computer program will halt (finish execution) or run indefinitely. This problem is known to be undecidable, so it's not in NP, but it is NP-Hard because it is as hard as any problem in NP.

In Summary:

- NP-Hard problems are at least as hard as NP problems.
- NP-Hard problems may or may not be in NP (they can be outside of NP, like the Halting Problem).
- There is no requirement that solutions can be verified in polynomial time.

2. NP-Complete

A problem is **NP-Complete** if it is both:

1. In NP (meaning that given a solution, it can be verified in polynomial time).
2. NP-Hard (meaning that every problem in NP can be reduced to it in polynomial time).

Key Characteristics:

- **In NP:** This means that a potential solution to the problem can be verified in polynomial time.
- **NP-Hard:** The problem is at least as hard as the hardest problems in NP, meaning that if you can solve an NP-Complete problem in polynomial time, you can solve all NP problems in polynomial time.
- **Reduction:** An NP-Complete problem can be reduced from any other problem in NP in polynomial time.



Example:

- **Travelling Salesman Problem (TSP):** The decision version of TSP, where you are asked whether there exists a route with a total cost less than or equal to a given number, is NP-Complete. It is in NP because a proposed route can be verified quickly, and it is NP-Hard because any NP problem can be reduced to TSP.

In Summary:

- NP-Complete problems are both in NP and NP-Hard.
- NP-Complete problems are among the hardest problems in NP, and any NP problem can be transformed into an NP-Complete problem in polynomial time.
- If an NP-Complete problem can be solved in polynomial time, all NP problems can also be solved in polynomial time.

Comparison of NP-Hard vs NP-Complete

Feature	NP-Hard	NP-Complete
Problem Type	Can be outside of NP (not necessarily verifiable in polynomial time).	Must be in NP (verifiable in polynomial time).
Hardness	As hard as the hardest NP problems.	As hard as the hardest NP problems, but also in NP.
Reduction	All NP problems can be reduced to NP-Hard problems in polynomial time.	Every problem in NP can be reduced to an NP-Complete problem in polynomial time.
Example	Halting Problem, some search problems.	Traveling Salesman Problem (TSP), Knapsack Problem.
Solution Verification	Not guaranteed to have polynomial-time verification.	Solutions can be verified in polynomial time.
Relationship to NP	May or may not be in NP.	Is in NP and NP-Hard.

Summary:

- **NP-Hard** problems are the hardest problems that might not even belong to NP, meaning there's no guarantee that their solutions can be verified in polynomial time.
- **NP-Complete** problems are both in NP and NP-Hard. They are the hardest problems in NP, and solving any NP-Complete problem in polynomial time would imply that all NP problems can be solved in polynomial time.

In complexity-theoretic terms, the distinction between **deterministic** and **non-deterministic** algorithms underpins the classes **P**, **NP**, **NP-Hard**, and **NP-Complete**:

1. Deterministic vs. Non-Deterministic Machines

- **Deterministic Turing Machine (DTM):**
At each step, its transition (state, tape symbol → new state, symbol, head move) is uniquely determined.
→ Captures what we normally implement as "deterministic algorithms."
- **Non-Deterministic Turing Machine (NDTM):**
At each step, it may have multiple possible transitions. It "branches" into many computation paths simultaneously and **accepts** if any path accepts.
→ A theoretical model; we don't physically build these, but they define the class NP.

2. The Class P

- **P** = problems decidable by a DTM in polynomial time.
- Equivalently, the set of decision problems solvable by a deterministic algorithm in $O(n^k)$ time for some constant k.
- **Example:** Binary search, merge sort, graph connectivity can all be done in polynomial time by deterministic algorithms.

3. The Class NP

- **NP** = problems decidable by an NDTM in polynomial time.
- Equivalently, decision problems for which a given "certificate" (proof of a "yes" instance) can be verified in polynomial time by a DTM.
- **Example:**
 - **Subset-sum decision:** Given a set of numbers and a target, does some subset sum to the target?
 - Verification: Given a subset, you can sum and check in polynomial time.

4. NP-Hard and NP-Complete

- **NP-Hard:**

A problem H is NP-Hard if **every** problem in NP can be **reduced** to H in polynomial time.

- H may or may not itself be in NP—its solutions might not even be verifiable in polynomial time.

- **Relationship:** If you could solve H deterministically in polynomial time, you'd solve all of NP (and hence $P = NP$).

- **NP-Complete:**

A problem C is NP-Complete if:

1. $C \in NP$ (its solutions can be verified in polynomial time by a DTM), and

2. C is NP-Hard (every NP problem reduces to C).

- NP-Complete problems sit right at the boundary: they capture the “hardest” problems **inside** NP.

5. Putting It All Together

Machine / Algorithm Type	Class	Description
Deterministic poly-time (DTM)	P	Problems solvable by a deterministic algorithm in polynomial time.
Non-Deterministic poly-time (NDTM)	NP	Problems solvable by a non-deterministic machine in polynomial time (or verifiable by a DTM in poly-time).
NP-Hard	—	At least as hard as any NP problem (not necessarily in NP).
NP-Complete	$NP \cap NP\text{-Hard}$	The hardest problems within NP (verifiable in poly-time, and every NP reduces to them).

- All **deterministic** poly-time algorithms map into P.
- All **non-deterministic** poly-time processes map into NP.
- **NP-Complete** problems are those “stuck” at the NP boundary—they’re in NP (so verifiable by deterministic means) but as hard as any non-deterministic polynomial-time challenge.
- If you discover a **deterministic** poly-time algorithm for any NP-Complete problem, you would prove $P = NP$, because that single deterministic algorithm would collapse the gap between P and NP.

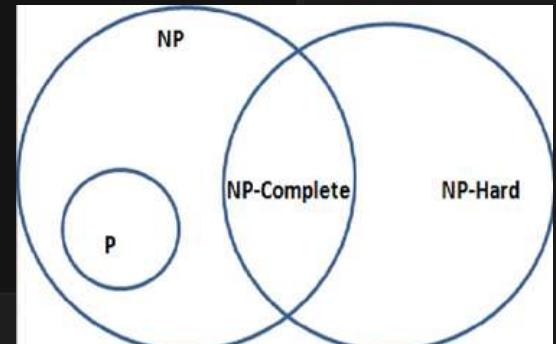
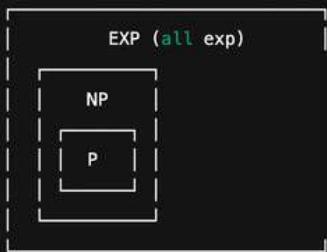
Visual Venn

Saved memory full

css

Copy

Edit



- $P \subseteq NP \subseteq EXP$
- NP-Hard sits “at least as hard as NP” (outside or overlapping NP).
- NP-Complete = $NP \cap NP\text{-Hard}$.

Bottom line:

- **Deterministic algorithms** define P;
- **Non-deterministic algorithms** define NP;
- **NP-Complete** sits at their intersection with NP-Hard, pinpointing the frontier where deterministic poly-time meets non-deterministic poly-time.

Types of TSP

1. Decision version (Is there a path with total cost $\leq K$?):
👉 This is **NP-Complete**.
2. Optimization version (What's the minimum cost route?):
👉 This is **NP-Hard** because it's even harder than the decision problem.

Why is the decision version of TSP NP-Complete?

To be **NP-Complete**, a problem must:

1. Be in NP \rightarrow We can verify a given solution in polynomial time.
2. Be as hard as any problem in NP \rightarrow Any NP problem can be reduced to it in polynomial time.

 For TSP:

- **Verification:** If someone gives you a tour (a specific order of cities) and a cost, you can **verify** whether the cost is $\leq K$ in **polynomial time** by summing the edge weights $\rightarrow \checkmark$ It's in NP.
- **Hardness:** The **Hamiltonian Cycle problem** (which is NP-Complete) can be **reduced to TSP** in polynomial time $\rightarrow \checkmark$ It's NP-Hard.

 Hence, the decision version of TSP is NP-Complete.

Connection to Deterministic & Non-Deterministic Algorithms

- **Deterministic algorithm** (like your Java backtracking code) takes factorial time \rightarrow impractical.
- **Non-deterministic algorithm** (theoretical, like NDTM) could "guess" the right tour instantly and verify it \rightarrow that's what makes it "in NP".

Summary Table

Version	Classification	🔗
Decision TSP ($\leq K$)	NP-Complete	
Optimization TSP (min)	NP-Hard	

Would you like an example showing reduction from Hamiltonian Cycle to TSP?

Rolling Hash

📚 What is Rolling Hash?

- It's a technique where **hashes** of substrings are updated efficiently when moving a window forward, without recomputing from scratch.
- Instead of recalculating the full hash, you **remove** the influence of the first character and **add** the new character in $O(1)$ time!

👉 Rolling Hash Formula

Suppose you have:

- a substring `S[i..i+M-1]` of length M .
- `hash(S[i..i+M-1])` → already known.

Then to find `hash(S[i+1..i+M])`:

```
csharp
newHash = (oldHash - (S[i] * base^(M-1))) * base + S[i+M]
newHash %= mod
```

Copy

Edit

where:

- `base` = size of alphabet (usually 26 for lowercase letters or 256 for ASCII)
- `mod` = a large prime number to avoid overflow (e.g., $1e9+7$, or 101 for Rabin-Karp)

🧠 Important points:

- **Sliding window:** Move character by character.
- **Efficient update:** Instead of recomputing full hash, update by adding and removing characters.
- **Spurious hits:** Sometimes, hash matches but actual strings don't → must **verify** characters after hash match.
- **Hash collision:** Different substrings can have the same hash, although very rare with good `base` and `mod`.

📈 Time Complexity:

- $O(n + m)$:
 - $O(m)$ for initial hashes
 - $O(n)$ for sliding the window

📦 Space Complexity:

- $O(1)$ (constant extra space)

```

public class Solution {
    private static final int BASE = 256; // ASCII characters
    private static final int MOD = 101; // A small prime for modulo

    public static List<Integer> findPattern(String text, String pattern) {
        List<Integer> result = new ArrayList<>();
        int n = text.length();
        int m = pattern.length();
        if (n < m) {
            return result;
        }
        int patternHash = 0; // Hash of pattern
        int windowHash = 0; // Hash of current window
        int highestPower = 1; // base^(m-1)
        // Precompute (base^(m-1)) % mod
        for (int i = 0; i < m - 1; i++) {
            highestPower = (highestPower * BASE) % MOD;
        }
        // Initial hash values for pattern and first window
        for (int i = 0; i < m; i++) {
            patternHash = (patternHash * BASE + pattern.charAt(i)) % MOD;
            windowHash = (windowHash * BASE + text.charAt(i)) % MOD;
        }
        // Slide the window
        for (int i = 0; i <= n - m; i++) {
            // If hash values match, do a character by character match
            if (patternHash == windowHash) {
                if (text.substring(i, i + m).equals(pattern)) {
                    result.add(i);
                }
            }
            // Calculate hash for next window
            if (i < n - m) {
                windowHash = (BASE * (windowHash - text.charAt(i) * highestPower) + text.charAt(i + m)) % MOD;
                // If hash is negative, make it positive
                if (windowHash < 0) {
                    windowHash += MOD;
                }
            }
        }
        return result;
    }

    public static void main(String[] args) {
        String text = "ababcabcababc";
        String pattern = "abc";
        List<Integer> indices = findPattern(text, pattern);
        System.out.println("Pattern found at indices: " + indices);
    }
}

```

⚡ Output

less

Copy

Edit

Pattern found at indices: [2, 5, 10]

🧠 Important points:

- **Sliding window:** Move character by character.
- **Efficient update:** Instead of recomputing full hash, update by adding and removing characters.
- **Spurious hits:** Sometimes, hash matches but actual strings don't → must verify characters after hash match.
- **Hash collision:** Different substrings can have the same hash, although very rare with good `base` and `mod`.

strStr() –

(find the first occurrence of needle in haystack) using Rolling Hash! 🚀

```
public class Solution {
    private static final int BASE = 256; // ASCII characters
    private static final int MOD = 1000000007; // A small prime for modulo

    public static List<Integer> findneedle(String haystack, String needle) {
        List<Integer> result = new ArrayList<>();
        int n = haystack.length();
        int m = needle.length();
        if (m == 0 || n < m) return result;
        int needleHash = 0; // Hash of needle
        int windowHash = 0; // Hash of current window
        int highestPower = 1; // BASE^(m-1)
        for (int i = 0; i < m - 1; i++) {
            highestPower = (highestPower * BASE) % MOD;
        }
        for (int i = 0; i < m; i++) {
            needleHash = (needleHash * BASE + needle.charAt(i)) % MOD;
            windowHash = (windowHash * BASE + haystack.charAt(i)) % MOD;
        }
        for (int i = 0; i <= n - m; i++) {
            if (needleHash == windowHash) {
                if (haystack.substring(i, i + m).equals(needle)) {
                    result.add(i);
                }
            }
            if (i < n - m) {
                windowHash = (BASE * (windowHash - haystack.charAt(i) * highestPower) + haystack.charAt(i + m)) % MOD;
                if (windowHash < 0) {
                    windowHash += MOD;
                }
            }
        }
    }

    return result;
}

public static void main(String[] args) {
    String haystack = "ababcabcababc";
    String needle = "abc";
    List<Integer> indices = findneedle(haystack, needle);
    System.out.println("needle found at indices: " + indices);
    // needle found at indices: [2, 5, 10]
}
}
```

findDuplicateSubstring

windowHash = (BASE * (windowHash - text.charAt(i - 1) * highestPower) + text.charAt(i + m - 1)) % MOD;

Great question!

✗ Why `text.charAt(i + m - 1)` instead of `text.charAt(i + m)`?

Let's break this down with an example and understand what `i`, `m`, and `text.charAt(...)` represent:

Rolling Hash Context

In the rolling hash (Rabin-Karp) technique, you're sliding a window of size `m` across the string `text`.

- `i` is the start index of the current window.
- So the window covers the substring from `text[i]` to `text[i + m - 1]`.

When you slide the window from position `i` to `i + 1`, you:

1. Remove `text[i - 1]` (the old character at the beginning of the previous window).
2. Add `text[i + m - 1]` (the new character at the end of the current window).

Why `i + m - 1`?

- For `i = 1`, the window is now `[1, 1 + m - 1] → length m`.
- `text.charAt(i + m - 1)` is the new character entering the window.
- If you used `i + m`, you'd go one character beyond the end of the window — off by one.

```
public class Solution {

    private static final int BASE = 256; // Alphabet size (ASCII)
    private static final int MOD = 1000000007; // Large prime

    public static String longestDupSubstring(String s) {
        int n = s.length();
        int low = 1, high = n - 1;
        int start = -1, maxLength = 0;

        while (low <= high) {
            int mid = low + (high - low) / 2;
            int idx = checkDuplicate(s, mid);

            if (idx != -1) {
                start = idx;
                maxLength = mid;
                low = mid + 1; // try for longer
            } else {
                high = mid - 1; // try shorter
            }
        }

        return start != -1 ? s.substring(start, start + maxLength) : "";
    }

    private static int checkDuplicate(String text, int m) {
        int n = text.length();
        long windowHash = 0;
        long highestPower = 1;
        // Compute BASE^(n-1) % MOD
        for (int i = 0; i < m - 1; i++) {
            highestPower = (highestPower * BASE) % MOD;
        }
        // Initial hash for the first window
        for (int i = 0; i < m; i++) {
            windowHash = (windowHash * BASE + text.charAt(i)) % MOD;
        }
        Set<Long> seen = new HashSet<>();
        seen.add(windowHash);
        // Slide the window
        for (int i = 1; i < n - m; i++) {
            windowHash = (BASE * (windowHash - text.charAt(i - 1) * highestPower) + text.charAt(i + m - 1)) % MOD;
            // Make sure hash is positive
            if (windowHash < 0) {
                windowHash += MOD;
            }
            if (seen.contains(windowHash)) {
                return i;
            }
            seen.add(windowHash);
        }
        return -1;
    }

    public static void main(String[] args) {
        String s = "banana";
        System.out.println("Largest Duplicate Substring: " + longestDupSubstring(s));
    }
}
```

Concurrency

Concurrency involves dealing with multiple tasks that are executing simultaneously and may interact with shared resources like variables or data structures. In programming challenges on LeetCode, concurrency problems typically require you to ensure that operations performed by multiple threads are synchronized and that shared resources are accessed safely to prevent issues like race conditions, data corruption, or inconsistent states.

Key Concepts in Concurrency Problems:

1. Threads: These are independent sequences of execution within a program. Each thread can execute code concurrently with other threads.
2. Synchronization: This refers to techniques used to control access to shared resources to avoid conflicts. Common synchronization mechanisms include locks (like 'synchronized' blocks in Java), mutexes, semaphores, and atomic operations.
3. Race Conditions: These occur when the outcome of a program depends on the sequence or timing of uncontrollable events, like the execution order of threads. Race conditions can lead to unpredictable results and bugs.
4. Atomicity: This refers to operations that are guaranteed to be executed as a single unit of work without interruption. In concurrent programming, atomic operations are crucial for ensuring data consistency.
5. Thread Safety: This concept ensures that data structures and operations can be used safely in multithreaded environments without causing unexpected behavior or errors.

Examples of Concurrency Problems on LeetCode:

- Producer-Consumer Problem: Threads that produce data and threads that consume data from a shared buffer.
- Readers-Writers Problem: Controlling access to a shared resource where multiple threads read and write data.
- Thread Synchronization: Ensuring that threads execute in a specific order or synchronize their actions using locks or other mechanisms.
- Concurrency in Algorithms: Problems where parallelism or concurrent execution can optimize performance, such as divide-and-conquer algorithms or parallel processing tasks.

Approaches to Solve Concurrency Problems:

- Locking: Use locks (e.g., 'synchronized' blocks in Java, 'Lock' interface) to synchronize access to shared resources.
- Atomic Operations: Utilize atomic classes ('AtomicInteger', 'AtomicReference', etc.) to ensure thread-safe operations on variables.
- Thread Communication: Use mechanisms like 'wait()', 'notify()', 'notifyAll()' in Java to coordinate actions between threads.
- Concurrency Utilities: Leverage language-specific concurrency utilities (e.g., Java 'ExecutorService', 'ThreadPoolExecutor') for managing and executing tasks concurrently.

Importance of Concurrency in LeetCode:

- Concurrency problems on LeetCode assess your ability to implement thread-safe solutions and handle complex scenarios where multiple threads interact.
- Understanding concurrency concepts and applying them correctly is crucial for writing efficient, scalable, and correct multithreaded programs.

By mastering concurrency concepts and practicing with problems on LeetCode, you can develop robust solutions that handle simultaneous execution gracefully and efficiently.

When you say "**Concurrency LeetCode**", you're referring to a special set of problems on LeetCode that test **multi-threaded programming** — especially about controlling the **order of execution across multiple threads**.

🧠 Main Topics Covered:

Concept	What it Means
Locks / Synchronized	Force only one thread to access critical code at a time
Semaphores	Control how many threads can access a resource
CountDownLatch	Wait for other threads to complete before proceeding
Barrier (CyclicBarrier)	Wait for a group of threads to reach a common point
Atomic Variables	Special classes like <code>AtomicInteger</code> for safe increments without explicit locking
Volatile Keyword	Ensures visibility of changes across threads immediately

Popular LeetCode Concurrency Problems:

Problem	Description
1114. Print in Order	Ensure <code>first()</code> , <code>second()</code> , <code>third()</code> print in order
1115. Print FooBar Alternately	Two threads print "Foo" and "Bar" alternately
1116. Print Zero Even Odd	Print numbers in a certain sequence
1195. Fizz Buzz Multithreaded	Four threads print numbers with "Fizz", "Buzz" rules
1226. The Dining Philosophers	Classic deadlock/avoidance problem
1188. Design Bounded Blocking Queue	Implement a thread-safe queue
1279. Traffic Light Controlled Intersection	Multiple cars crossing intersection without collision
1186. Building H2O	Synchronize printing "H2O" using threads

```
class Foo {
    private CountDownLatch secondLatch;
    private CountDownLatch thirdLatch;

    public Foo() {
        secondLatch = new CountDownLatch(1);
        thirdLatch = new CountDownLatch(1);
    }

    public void first(Runnable printFirst) throws InterruptedException {
        printFirst.run();
        secondLatch.countDown();
    }

    public void second(Runnable printSecond) throws InterruptedException {
        secondLatch.await();
        printSecond.run();
        thirdLatch.countDown();
    }

    public void third(Runnable printThird) throws InterruptedException {
        thirdLatch.await();
        printThird.run();
    }
}
```

Important Tools You Must Know:

- `synchronized`
- `Lock / ReentrantLock`
- `Semaphore`
- `CountDownLatch`
- `CyclicBarrier`
- `AtomicInteger`
- `Condition (advanced locks)`



Pro Tip:

In **concurrency problems**:

- It's **not** about how fast.
- It's about **order** and **correctness** of execution.
- **Deadlocks** and **race conditions** are the main traps.

Greedy

- Fractional Knapsack
- Activity Selection
- Job Sequencing With Deadlines
- Minimum Number Of Coins
- Minimum Equal Sum Possible With 3 Equal Stacks
- Gas Station
- Meeting Rooms II
- Non-overlapping Intervals
- Minimum Number of Arrows to Burst Balloons
- Can Place Flowers
- Partition Labels
- Boats to Save People

Fractional Knapsack

Problem Statement:

You have:

- n items, each with a `value[i]` and a `weight[i]`
- A knapsack (bag) with maximum capacity W

Goal:

- Maximize the total value you can put in the knapsack.
- You **can take fractions** of an item (i.e., not necessarily the whole item).

Approach (Greedy):

Pick items with the highest value per unit weight first.

- Sort items by `value/weight` in descending order.
- Then, for each item:
 - If the knapsack can hold the whole item, take it completely.
 - Otherwise, take the fraction that fits.

Time Complexity:

- Sorting: $O(n \log n)$
- Picking items: $O(n)$

Total = $O(n \log n)$

Key Difference with 0/1 Knapsack:

- In 0/1 knapsack you **cannot** take fractions (only full item or none).
- In fractional knapsack you **can**.

```
class Item {
    int value, weight;

    Item(int value, int weight) {
        this.value = value;
        this.weight = weight;
    }
}

public class Solution {

    static double fractionalKnapsack(int W, Item[] items) {
        Arrays.sort(items, (a, b) ->
            Double.compare((double)b.value / b.weight, (double)a.value / a.weight)
        );

        double totalValue = 0.0;

        for (Item item : items) {
            if (W >= item.weight) {
                W -= item.weight;
                totalValue += item.value;
            } else {
                totalValue += (double)item.value * ((double)W / item.weight);
                break;
            }
        }

        return totalValue;
    }

    public static void main(String[] args) {
        Item[] items = {
            new Item( value: 60, weight: 10),
            new Item( value: 100, weight: 20),
            new Item( value: 120, weight: 30)
        };
        int capacity = 50;

        double maxValue = fractionalKnapsack(capacity, items);
        System.out.println("Maximum value in knapsack = " + maxValue);
    }
}
// Maximum value in knapsack = 240.0
```

Activity Selection

Problem Statement:

You are given n activities with their **start** and **end** times.

Goal:

- Select the **maximum number of activities** that **don't overlap**.
- Only **one activity can happen at a time**.

Approach (Greedy):

Always pick the activity that **ends earliest**.

- Sort activities by **end time in ascending order**.
- Always select an activity if its start time is **after the end time of the last selected activity**.

```
public class Solution {  
  
    static class Activity {  
        int start, end;  
  
        Activity(int start, int end) {  
            this.start = start;  
            this.end = end;  
        }  
    }  
  
    static int maxActivities(Activity[] activities) {  
        Arrays.sort(activities, Comparator.comparingInt(a -> a.end));  
  
        int count = 0;  
        int lastEnd = -1;  
  
        for (Activity activity : activities) {  
            if (activity.start > lastEnd) {  
                count++;  
                lastEnd = activity.end;  
            }  
        }  
  
        return count;  
    }  
}
```

```
public static void main(String[] args) {
    Activity[] activities = {
        new Activity(1, 2),
        new Activity(3, 4),
        new Activity(0, 6),
        new Activity(5, 7),
        new Activity(8, 9),
        new Activity(5, 9)
    };

    System.out.println("Maximum number of activities = " + maxActivities(activities));
}
}
```

Output:

java

Copy Edit

```
Maximum number of activities = 4
```

Selected activities here:

(1,2), (3,4), (5,7), (8,9)

Minimum Number Of Coins

Problem Statement:

- You are given coins of certain denominations (e.g., [1, 2, 5, 10, 20, 50, 100, 500, 2000]).
- You need to make a target amount (e.g., 93).
- Goal:
Use the minimum number of coins to make that amount.

Approach (Greedy):

- Always pick the largest possible coin first.
- Sort coins in descending order.
- Keep picking the largest coin as long as it fits into the remaining amount.

Output:

```
swift
Minimum coins required:
50 20 20 2 1

(50 + 20 + 20 + 2 + 1 = 93)
```

```
=
public class Solution {

    static void findMinCoins(int[] coins, int amount) {
        Arrays.sort(coins); // Sort ascending first
        int n = coins.length;

        List<Integer> result = new ArrayList<>();

        for (int i = n - 1; i >= 0; i--) {
            while (amount >= coins[i]) {
                amount -= coins[i];
                result.add(coins[i]);
            }
        }

        System.out.println("Minimum coins required:");
        for (int coin : result) {
            System.out.print(coin + " ");
        }
    }

    public static void main(String[] args) {
        int[] coins = {1, 2, 5, 10, 20, 50, 100, 500, 2000};
        int amount = 93;

        findMinCoins(coins, amount);
    }
}
```

Job Sequencing With Deadlines

You are given n jobs.

Each job has:

- a deadline and
- a profit if it is completed before or on its deadline.

Goal:

- Schedule jobs to maximize the total profit.
- Each job takes exactly 1 unit of time.
- Only one job can be scheduled at a time.

Approach (Greedy):

Idea:

- Maximize profit by picking highest-profit jobs first, and
- Try to schedule them as late as possible (before their deadline).

Output:

```
bash                                     ⌂ Copy ⌂ Edit
Number of jobs done: 2
Total profit: 70
```

(Job 3 and Job 1 selected for max profit)

```
class Job {
    int id, deadline, profit;

    Job(int id, int deadline, int profit) {
        this.id = id;
        this.deadline = deadline;
        this.profit = profit;
    }
}

public class Solution {

    static int[] jobSequencing(Job[] jobs) {
        // Sort jobs by descending profit
        Arrays.sort(jobs, (a, b) -> b.profit - a.profit);
        int maxDeadline = 0;
        for (Job job : jobs) {
            maxDeadline = Math.max(maxDeadline, job.deadline);
        }
        int[] slots = new int[maxDeadline + 1]; // slots[0] is unused
        Arrays.fill(slots, val: -1); // -1 means slot is free
        int countJobs = 0, totalProfit = 0;

        for (Job job : jobs) {
            // Try to find a free slot for this job (starting from its deadline)
            for (int j = job.deadline; j > 0; j--) {
                if (slots[j] == -1) {
                    slots[j] = job.id; // assign job
                    countJobs++;
                    totalProfit += job.profit;
                    break;
                }
            }
        }
    }
}
```

```

        return new int[]{countJobs, totalProfit};
    }

    public static void main(String[] args) {
        Job[] jobs = {
            new Job( id: 1, deadline: 4, profit: 20),
            new Job( id: 2, deadline: 1, profit: 10),
            new Job( id: 3, deadline: 1, profit: 40),
            new Job( id: 4, deadline: 1, profit: 30)
        };

        int[] result = jobSequencing(jobs);
        System.out.println("Number of jobs done: " + result[0]);
        System.out.println("Total profit: " + result[1]);
    }
}

```

Minimum Equal Sum Possible With 3 Equal Stacks

Problem Statement:

- You are given 3 stacks of positive integers (stack1, stack2, stack3).
- Each element represents the **height** (or value).
- You can only **remove elements from the top** of a stack.
- **Goal:**
Find the **maximum possible equal sum** you can achieve across all 3 stacks after removing some elements.

Approach (Greedy):

Idea:

- Find the **current sums** of all 3 stacks.
- While the **sums are not equal**:
 - Remove the top element from the **stack with the highest sum**.
 - Stop when all three sums are equal.

Output:

bash

Copy Edit

```
Maximum possible equal sum: 5
```

```

public class Solution {

    static int maxEqualSum(int[] stack1, int[] stack2, int[] stack3) {
        Stack<Integer> s1 = buildStack(stack1);
        Stack<Integer> s2 = buildStack(stack2);
        Stack<Integer> s3 = buildStack(stack3);

        int sum1 = Arrays.stream(stack1).sum();
        int sum2 = Arrays.stream(stack2).sum();
        int sum3 = Arrays.stream(stack3).sum();

        while (true) {
            // If any stack is empty
            if (s1.isEmpty() || s2.isEmpty() || s3.isEmpty()) {
                return 0;
            }

            // If all sums are equal
            if (sum1 == sum2 && sum2 == sum3) {
                return sum1;
            }

            // Remove top from the stack with maximum sum
            if (sum1 >= sum2 && sum1 >= sum3) {
                sum1 -= s1.pop();
            } else if (sum2 >= sum1 && sum2 >= sum3) {
                sum2 -= s2.pop();
            } else {
                sum3 -= s3.pop();
            }
        }
    }

    private static Stack<Integer> buildStack(int[] arr) {
        Stack<Integer> stack = new Stack<>();
        // Push elements from end to start so first element is at bottom
        for (int i = arr.length - 1; i >= 0; i--) {
            stack.push(arr[i]);
        }
        return stack;
    }

    public static void main(String[] args) {
        int[] stack1 = {3, 2, 1, 1, 1};
        int[] stack2 = {4, 3, 2};
        int[] stack3 = {1, 1, 4, 1};

        int result = maxEqualSum(stack1, stack2, stack3);
        System.out.println("Maximum possible equal sum: " + result); // 5
    }
}

```

Time Complexity:

- $O(n_1 + n_2 + n_3)$
(where n_1, n_2, n_3 are the number of elements in each stack)

Gas Station

You are given:

- `gas[i]` : amount of gas at station i
- `cost[i]` : gas needed to travel from station i to $i+1$

Question:

- Can you complete the circuit once?
- If yes, return the starting gas station index.
- If not, return `-1`.

Important:

- You can only move forward station by station (station N connects to station 0 circularly).

```
public class Solution {

    public static int canCompleteCircuit(int[] gas, int[] cost) {
        int totalGas = 0, totalCost = 0;
        int tank = 0;
        int start = 0;

        for (int i = 0; i < gas.length; i++) {
            totalGas += gas[i];
            totalCost += cost[i];

            tank += gas[i] - cost[i];
            if (tank < 0) {
                // Cannot reach station  $i+1$ , change start to  $i+1$ 
                start = i + 1;
                tank = 0;
            }
        }

        return (totalGas >= totalCost) ? start : -1;
    }

    public static void main(String[] args) {
        int[] gas = {1, 2, 3, 4, 5};
        int[] cost = {3, 4, 5, 1, 2};

        int result = canCompleteCircuit(gas, cost);
        System.out.println("Starting station index: " + result); // 3
    }
}
```

Output:

perl

Copy Edit

```
Starting station index: 3
```

Explanation:

- Start from station 3:
 - $\text{gas}[3] = 4, \text{cost}[3] = 1 \rightarrow \text{tank} = 3$
 - $\text{gas}[4] = 5, \text{cost}[4] = 2 \rightarrow \text{tank} = 6$
 - $\text{gas}[0] = 1, \text{cost}[0] = 3 \rightarrow \text{tank} = 4$
 - $\text{gas}[1] = 2, \text{cost}[1] = 4 \rightarrow \text{tank} = 2$
 - $\text{gas}[2] = 3, \text{cost}[2] = 5 \rightarrow \text{tank} = 0$ ✓

Can Place Flowers

You are given an array `flowerbed` representing a row of flowers where:

- `flowerbed[i] == 0` means the plot is empty, and
- `flowerbed[i] == 1` means the plot contains a flower.

You have to plant a flower in an empty plot such that no two flowers are adjacent. The task is to determine if you can plant at least `n` flowers in the flowerbed.

Question:

Given the flowerbed and the number `n`, determine if you can plant `n` flowers in it without violating the "no adjacent flowers" rule.

```
public class Solution {

    public static boolean canPlaceFlowers(int[] flowerbed, int n) {
        int count = 0;

        for (int i = 0; i < flowerbed.length; i++) {
            // Check if the current plot is empty and the adjacent plots are also empty
            if (flowerbed[i] == 0
                && (i == 0 || flowerbed[i - 1] == 0) // Check left
                && (i == flowerbed.length - 1 || flowerbed[i + 1] == 0)) { // Check right
                flowerbed[i] = 1; // Plant a flower
                count++; // Increment the count of flowers planted
                if (count >= n) {
                    return true; // Return true if we've planted enough flowers
                }
                i++; // Skip the next plot as it cannot be used (adjacency rule)
            }
        }

        return count >= n;
    }

    public static void main(String[] args) {
        int[] flowerbed = {1, 0, 0, 0, 1};
        int n = 1;

        boolean result = canPlaceFlowers(flowerbed, n);
        System.out.println("Can place " + n + " flowers: " + result);
    }
}
```

Meeting Rooms II

Problem Statement:

- Given an array of meeting intervals `[[start1, end1], [start2, end2], ...]`
- Goal:** Find the **minimum number of meeting rooms** required so that no two meetings overlap.

Approach:

Main Idea:

- Sort the meetings by **start time**.
- Use a **min-heap (PriorityQueue)** to keep track of the **end times** of ongoing meetings.
- For each meeting:
 - If a room is free (meeting with the smallest end time \leq current start time), reuse it \rightarrow poll from heap.
 - Otherwise, need a new room.
- Push the current meeting's end time into heap.

Finally, the size of the heap tells you the minimum number of rooms needed.

```
public static int minMeetingRooms(int[][] intervals) {  
    if (intervals == null || intervals.length == 0) {  
        return 0;  
    }  
    // Sort meetings by start time  
    Arrays.sort(intervals, (a, b) -> a[0] - b[0]);  
    // Min-heap to track the earliest end time  
    PriorityQueue<Integer> heap = new PriorityQueue<>();  
    // Add end time of first meeting  
    heap.offer(intervals[0][1]);  
  
    for (int i = 1; i < intervals.length; i++) {  
        int currentStart = intervals[i][0];  
        int currentEnd = intervals[i][1];  
        // If earliest ending meeting is done, remove it  
        if (currentStart >= heap.peek()) {  
            heap.poll();  
        }  
        // Add the current meeting's end time  
        heap.offer(currentEnd);  
    }  
    // Heap size = number of meeting rooms needed  
    return heap.size();  
}  
  
public static void main(String[] args) {  
    int[][] meetings = {  
        {0, 30},  
        {5, 10},  
        {15, 20}  
    };  
    int result = minMeetingRooms(meetings);  
    System.out.println("Minimum meeting rooms needed: " + result); // ?  
}
```

Output:

yaml

Copy Edit

Minimum meeting rooms needed: 2

Non-overlapping Intervals

- Given an array of intervals `[[start1, end1], [start2, end2], ...]`
- Find the minimum number of intervals you need to remove to make the rest non-overlapping.

Approach:

Main Idea (Greedy):

- Sort intervals by their end times (smallest end time first).
- Pick intervals that do not overlap greedily.
- Remove others (count how many are overlapping).

Why sort by end time?

- Choosing the interval with the earliest ending gives maximum room for the next meetings → less chance of overlapping.

```
public class Solution {

    public static int eraseOverlapIntervals(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return 0;
        }

        // Sort intervals by end time
        Arrays.sort(intervals, (a, b) -> a[1] - b[1]);

        int count = 0;
        int prevEnd = intervals[0][1];

        for (int i = 1; i < intervals.length; i++) {
            if (intervals[i][0] < prevEnd) {
                // Overlapping → remove this interval
                count++;
            } else {
                // No overlap → update previous end
                prevEnd = intervals[i][1];
            }
        }

        return count;
    }

    public static void main(String[] args) {
        int[][] intervals = {
            {1, 2},
            {2, 3},
            {3, 4},
            {1, 3}
        };

        int result = eraseOverlapIntervals(intervals);
        System.out.println("Minimum intervals to remove: " + result);
    }
}
```

Output:

arduino

Copy Edit

Minimum intervals to remove: 1

Boats to Save People

You are given an array `people` where `people[i]` is the weight of the `i`th person, and an integer `limit` representing the maximum weight a boat can carry.

Each boat can carry at most two people at the same time, provided the sum of their weights is less than or equal to `limit`.

Find the minimum number of boats needed to save everyone.

```
public class BoatsToSavePeople {

    public static int numRescueBoats(int[] people, int limit) {
        Arrays.sort(people); // Sort people by weight

        int left = 0;
        int right = people.length - 1;
        int boats = 0;

        while (left <= right) {
            if (people[left] + people[right] <= limit) {
                left++; // Lightest person is saved along with the heaviest
            }
            right--; // Heaviest person always boards
            boats++; // One boat used
        }

        return boats;
    }

    public static void main(String[] args) {
        int[] people = {3, 5, 3, 4};
        int limit = 5;

        int result = numRescueBoats(people, limit);
        System.out.println("Minimum number of boats needed: " + result);
    }
}
```

Output:

```
typescript
Minimum number of boats needed: 4
```

Minimum Number of Arrows to Burst Balloons

Problem Statement:

You are given an array of balloons where each balloon is represented as:

```
balloons[i] = [start_i, end_i]
```

Each balloon represents a range on the x-axis.

- You can shoot an arrow at any x-coordinate.
- A balloon is burst if the arrow hits any point inside its interval.
- Find the minimum number of arrows needed to burst all balloons.

```
import java.util.*;

public class MinArrowsToBurstBalloons {

    public static int findMinArrowShots(int[][] points) {
        if (points == null || points.length == 0) {
            return 0;
        }

        // Sort balloons by end coordinate
        Arrays.sort(points, (a, b) -> Integer.compare(a[1], b[1]));

        int arrows = 1;
        int arrowPos = points[0][1];

        for (int i = 1; i < points.length; i++) {
            if (points[i][0] > arrowPos) {
                // Need a new arrow
                arrows++;
                arrowPos = points[i][1];
            }
        }

        return arrows;
    }

    public static void main(String[] args) {
        int[][] balloons = {
            {10, 16},
            {2, 8},
            {1, 6},
            {7, 12}
        };

        int result = findMinArrowShots(balloons);
        System.out.println("Minimum arrows needed: " + result);
    }
}
```

Output:

yaml

Minimum arrows needed: 2

Quick Dry Run:

Balloons after sorting by end time:

- [1,6], [2,8], [7,12], [10,16]

Processing:

- Shoot at end=6.
- [2,8]: 2 <= 6 → no need for new arrow.
- [7,12]: 7 > 6 → need new arrow at 12.
- [10,16]: 10 <= 12 → no need for new arrow.

Final: 2 arrows needed.

Partition Labels

You are given a string `s` and you need to partition it into as many parts as possible so that each letter appears in at most one part.

- Return the sizes of these parts.

Approach:

✓ Main Idea:

- The key idea is to find the **last occurrence** of each character in the string.
- Then, for each partition, you keep expanding until you encounter the **last occurrence** of any character that was previously encountered in the current partition.
- Once you've processed all characters in the partition, mark the end of the partition.
- This is essentially about tracking intervals where each character's last occurrence is included.

```
public class Solution {  
  
    public static List<Integer> partitionLabels(String s) {  
        // Step 1: Record the last occurrence of each character.  
        Map<Character, Integer> lastOccurrence = new HashMap<>();  
        for (int i = 0; i < s.length(); i++) {  
            lastOccurrence.put(s.charAt(i), i);  
        }  
        List<Integer> result = new ArrayList<>();  
        int start = 0;  
        int end = 0;  
        // Step 2: Traverse through the string to find partitions  
        for (int i = 0; i < s.length(); i++) {  
            // Extend the end of the current partition to include the last occurrence of the character  
            end = Math.max(end, lastOccurrence.get(s.charAt(i)));  
            // Step 3: When we reach the end of the current partition  
            if (i == end) {  
                // Record the size of the partition  
                result.add(i - start + 1);  
                start = i + 1; // Move start to the next partition  
            }  
        }  
        return result;  
    }  
  
    public static void main(String[] args) {  
        String s = "abac";  
        List<Integer> result = partitionLabels(s);  
        System.out.println("Partition sizes: " + result);  
    }  
}
```

✓ Output:

less Copy Edit
Partition sizes: [3, 1]

String: abac

1. Last occurrence:

- a → index 2
- b → index 1
- c → index 3

2. Processing the string:

- i = 0, char = 'a', end = max(0, 2) = 2
- i = 1, char = 'b', end = max(2, 1) = 2
- i = 2, char = 'a', end = max(2, 2) = 2
- i = 3, char = 'c', end = max(2, 3) = 3
- i = 3 (end reached), partition size = 4 (from index 0 to 3).

Thus, the result is [3, 1].

Hash Table

1. Two Sum (Hash Map)
2. Design HashMap
3. Group Anagrams
4. Insert Delete GetRandom O(1)
5. Ransom Note
6. 4Sum II
7. Valid Anagram

Two Sum (Hash Map)

```
import java.util.HashMap;
import java.util.Map;

public class TwoSumHashMap {

    public static int[] twoSum(int[] nums, int target) {
        Map<Integer, Integer> map = new HashMap<>(); // Stores number -> index

        for (int i = 0; i < nums.length; i++) {
            int complement = target - nums[i];

            if (map.containsKey(complement)) {
                return new int[] { map.get(complement), i }; // Found the pair
            }

            map.put(nums[i], i); // Store the current number and its index
        }

        throw new IllegalArgumentException("No two sum solution found");
    }

    public static void main(String[] args) {
        int[] nums = {2, 7, 11, 15};
        int target = 9;

        int[] result = twoSum(nums, target);
        System.out.println("Indices: " + result[0] + ", " + result[1]);
    }
}
```

Group Anagrams

```
import java.util.*;

public class GroupAnagrams {

    public static List<List<String>> groupAnagrams(String[] strs) {
        if (strs == null || strs.length == 0) {
            return new ArrayList<>();
        }

        Map<String, List<String>> map = new HashMap<>();

        for (String s : strs) {
            char[] charArray = s.toCharArray();
            Arrays.sort(charArray);
            String sorted = new String(charArray);

            if (!map.containsKey(sorted)) {
                map.put(sorted, new ArrayList<>());
            }
            map.get(sorted).add(s);
        }

        return new ArrayList<>(map.values());
    }

    public static void main(String[] args) {
        String[] strs = {"eat", "tea", "tan", "ate", "nat", "bat"};

        List<List<String>> result = groupAnagrams(strs);
        for (List<String> group : result) {
            System.out.println(group);
        }
    }
}
```

Design HashMap

```
class MyHashMap {  
    private static class Node {  
        int key, value;  
        Node next;  
  
        Node(int key, int value) {  
            this.key = key;  
            this.value = value;  
        }  
    }  
  
    private final int SIZE = 10000; // Number of buckets  
    private Node[] buckets;  
  
    public MyHashMap() {  
        buckets = new Node[SIZE];  
    }  
  
    private int hash(int key) {  
        return key % SIZE;  
    }  
  
    public void put(int key, int value) {  
        int index = hash(key);  
        if (buckets[index] == null) {  
            buckets[index] = new Node(-1, -1); // dummy head  
        }  
        Node prev = find(buckets[index], key);  
        if (prev.next == null) {  
            prev.next = new Node(key, value);  
        } else {  
            prev.next.value = value; // Update existing  
        }  
    }  
  
    public int get(int key) {  
        int index = hash(key);  
        if (buckets[index] == null) {  
            return -1;  
        }  
        Node prev = find(buckets[index], key);  
        if (prev.next == null) {  
            return -1;  
        }  
        return prev.next.value;  
    }  
}
```

```
public void remove(int key) {
    int index = hash(key);
    if (buckets[index] == null) {
        return;
    }
    Node prev = find(buckets[index], key);
    if (prev.next == null) {
        return;
    }
    prev.next = prev.next.next;
}

// Helper: Find previous node of the key if present
private Node find(Node head, int key) {
    Node curr = head, prev = null;
    while (curr != null && curr.key != key) {
        prev = curr;
        curr = curr.next;
    }
    return prev;
}

public static void main(String[] args) {
    MyHashMap map = new MyHashMap();
    map.put(1, 1);
    map.put(2, 2);
    System.out.println(map.get(1)); // returns 1
    System.out.println(map.get(3)); // returns -1 (not found)
    map.put(2, 1); // update the value
    System.out.println(map.get(2)); // returns 1
    map.remove(key: 2);
    System.out.println(map.get(2)); // returns -1 (removed)
}
}
```

Valid Anagram

Problem Statement:

Given two strings `s` and `t`, return `true` if `t` is an anagram of `s`, and `false` otherwise.

Example:

```
text          ⌂ Copy ⌂ Edit

Input: s = "anagram", t = "nagaram"
Output: true

Input: s = "rat", t = "car"
Output: false
```

Solution 1: Using a character count array (Best for lowercase letters)

Java code:

```
java          ⌂ Copy ⌂ Edit

public boolean isAnagram(String s, String t) {
    if (s.length() != t.length()) {
        return false;
    }

    int[] count = new int[26]; // Only lowercase English letters

    for (int i = 0; i < s.length(); i++) {
        count[s.charAt(i) - 'a']++;
        count[t.charAt(i) - 'a']--;
    }

    for (int c : count) {
        if (c != 0) {
            return false;
        }
    }

    return true;
}
```

4Sum II

```
public class Solution {
    public int fourSumCount(int[] nums1, int[] nums2, int[] nums3, int[] nums4) {
        Map<Integer, Integer> sumMap = new HashMap<>();

        for (int a : nums1) {
            for (int b : nums2) {
                int sum = a + b;
                sumMap.put(sum, sumMap.getOrDefault(sum, defaultValue: 0) + 1);
            }
        }

        int count = 0;

        for (int c : nums3) {
            for (int d : nums4) {
                int target = -(c + d);
                count += sumMap.getOrDefault(target, defaultValue: 0);
            }
        }

        return count;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();
        int[] nums1 = {1, 2};
        int[] nums2 = {-2, -1};
        int[] nums3 = {-1, 2};
        int[] nums4 = {0, 2};
        int result = solution.fourSumCount(nums1, nums2, nums3, nums4);
        System.out.println("Number of tuples: " + result); // 2
    }
}
```

Insert Delete GetRandom O(1)

```
public class Solution {

    private List<Integer> list;
    private Map<Integer, Set<Integer>> map;

    public Solution() {
        list = new ArrayList<>();
        map = new HashMap<>();
    }

    public boolean insert(int val) {
        // If val is not present, the first insertion is considered.
        boolean contains = map.containsKey(val);
        // Add val's index in the map.
        map.putIfAbsent(val, new HashSet<>());
        map.get(val).add(list.size()); // Map the index where val is inserted in list.
        list.add(val); // Insert val in the list.
        return !contains; // Return true if it's the first insertion.
    }

    public boolean remove(int val) {
        // If val is not in the map, we can't remove it.
        if (!map.containsKey(val)) {
            return false;
        }
        // Get the index of the value we want to delete (remove one of them).
        int removeIndex = map.get(val).iterator().next();
        map.get(val).remove(removeIndex);
        if (map.get(val).isEmpty()) {
            map.remove(val); // Clean up map if val is completely removed.
        }
        // Swap the element with the last element in the list.
        if (removeIndex != list.size() - 1) {
            int lastElement = list.get(list.size() - 1);
            list.set(removeIndex, lastElement);
            map.get(lastElement).remove(list.size() - 1);
            map.get(lastElement).add(removeIndex); // Update the map with new index for the last element.
        }
        list.remove(index: list.size() - 1); // Remove the last element (which was swapped).
        return true;
    }

    public int getRandom() {
        Random rand = new Random();
        return list.get(rand.nextInt(list.size())); // Get random element in O(1)
    }

    public static void main(String[] args) {
        Solution collection = new Solution();
        // Test Insert
        System.out.println(collection.insert( val: 1)); // true, since 1 is inserted.
        System.out.println(collection.insert( val: 1)); // false, 1 is already in the collection.
        System.out.println(collection.insert( val: 2)); // true, since 2 is inserted.
        // Test GetRandom
        System.out.println(collection.getRandom()); // Returns 1 or 2 randomly.
        // Test Remove
        System.out.println(collection.remove( val: 1)); // true, removes 1 from the collection.
        System.out.println(collection.remove( val: 1)); // true, removes another 1 (since duplicate is present).
        System.out.println(collection.remove( val: 3)); // false, 3 is not in the collection.
        // Test GetRandom again
        System.out.println(collection.getRandom()); // Should return 2, since it's the only remaining element.
    }
}
```

Ransom Note

```
public class Solution {

    // Using frequency array
    public boolean canConstructUsingArray(String ransomNote, String magazine) {
        if (ransomNote.length() > magazine.length()) {
            return false;
        }

        int[] count = new int[26]; // For lowercase English letters

        // Count frequency of each letter in magazine
        for (char c : magazine.toCharArray()) {
            count[c - 'a']++;
        }

        // Check if ransomNote can be formed with those frequencies
        for (char c : ransomNote.toCharArray()) {
            if (count[c - 'a'] == 0) {
                return false; // Not enough letters in magazine
            }
            count[c - 'a']--;
        }

        return true;
    }

    // Using HashMap
    public boolean canConstructUsingHashMap(String ransomNote, String magazine) {
        HashMap<Character, Integer> map = new HashMap<>();

        // Count frequency of characters in magazine
        for (char c : magazine.toCharArray()) {
            map.put(c, map.getOrDefault(c, defaultValue: 0) + 1);
        }

        // Check if ransomNote can be constructed
        for (char c : ransomNote.toCharArray()) {
            if (!map.containsKey(c) || map.get(c) == 0) {
                return false; // Not enough characters in magazine
            }
            map.put(c, map.get(c) - 1);
        }

        return true;
    }

    public static void main(String[] args) {
        Solution rn = new Solution();
        String ransomNote1 = "aa";
        String magazine1 = "aab";
        System.out.println("Can construct ransomNote1 using frequency array: " +
                           rn.canConstructUsingArray(ransomNote1, magazine1)); // true
        String ransomNote2 = "a";
        String magazine2 = "b";
        System.out.println("Can construct ransomNote2 using frequency array: " +
                           rn.canConstructUsingArray(ransomNote2, magazine2)); // false
        String ransomNote3 = "rat";
        String magazine3 = "car";
        System.out.println("Can construct ransomNote3 using HashMap: " +
                           rn.canConstructUsingHashMap(ransomNote3, magazine3)); // false
        String ransomNote4 = "aabb";
        String magazine4 = "aabbbc";
        System.out.println("Can construct ransomNote4 using HashMap: " +
                           rn.canConstructUsingHashMap(ransomNote4, magazine4)); // true
    }
}
```

Segment Tree

A **Segment Tree** is a powerful data structure used for answering **range queries** (like sum, minimum, maximum, GCD, etc.) and for **updating** elements efficiently. It is especially useful when the data doesn't change frequently and you're performing multiple queries.

◆ Segment Tree Summary

- **Use cases:** Range sum, range minimum/maximum, GCD, LCM, frequency count, etc.
- **Operations:**
 - Build the tree
 - Query the tree for a range
 - Update a value

🔧 Segment Tree: Basics

- For an array of size n , the Segment Tree is typically built in a **tree form**, where:
 - Each **leaf node** represents an element of the array.
 - Each **internal node** represents a value over a segment (like sum, min, max, etc.) of the array.

⌚ Time Complexities

Operation	Time Complexity
Build Tree	$O(n)$
Query Range	$O(\log n)$
Update Element	$O(\log n)$

Segment Tree Operations

Let's assume we're working with **range sum**.

1. Build Tree

Build a tree from the input array.

2. Query

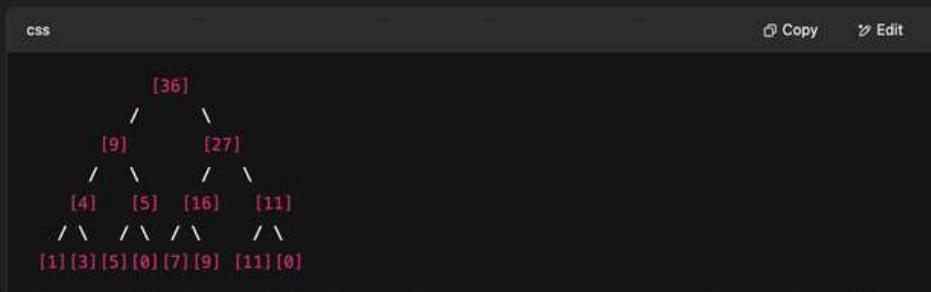
Get sum in range `[l, r]`.

3. Update

Change an element at index `i` to a new value and update the tree.

Tree Structure Example

For array `nums = [1, 3, 5, 7, 9, 11]`
Tree structure (node = sum of subarray):



Variants

- **Range Min/Max:** Replace sum logic with `Math.min/Math.max`.
- **Lazy Propagation:** For efficient range updates.
- **Segment Tree with Objects:** To store multiple values like min+max+sum etc.

What is the sum from index l to r?

```
public class Solution {  
  
    private static class SegmentTree {  
        int[] tree;  
        int n;  
  
        public SegmentTree(int[] nums) {  
            n = nums.length;  
            tree = new int[4 * n];  
            build(nums, start: 0, end: n - 1, node: 0);  
        }  
  
        private void build(int[] nums, int start, int end, int node) {  
            if (start == end) {  
                tree[node] = nums[start];  
                return;  
            }  
            int mid = (start + end) / 2;  
            build(nums, start, mid, node: 2 * node + 1);  
            build(nums, start: mid + 1, end, node: 2 * node + 2);  
            tree[node] = tree[2 * node + 1] + tree[2 * node + 2]; // for sum  
        }  
  
        public int query(int l, int r) {  
            return queryUtil( start: 0, end: n - 1, l, r, node: 0);  
        }  
  
        private int queryUtil(int start, int end, int l, int r, int node) {  
            if (r < start || end < l) return 0; // completely outside  
            if (l <= start && end <= r) return tree[node]; // completely inside  
            int mid = (start + end) / 2;  
            return queryUtil(start, mid, l, r, node: 2 * node + 1) +  
                queryUtil( start: mid + 1, end, l, r, node: 2 * node + 2);  
        }  
    }  
}
```

```

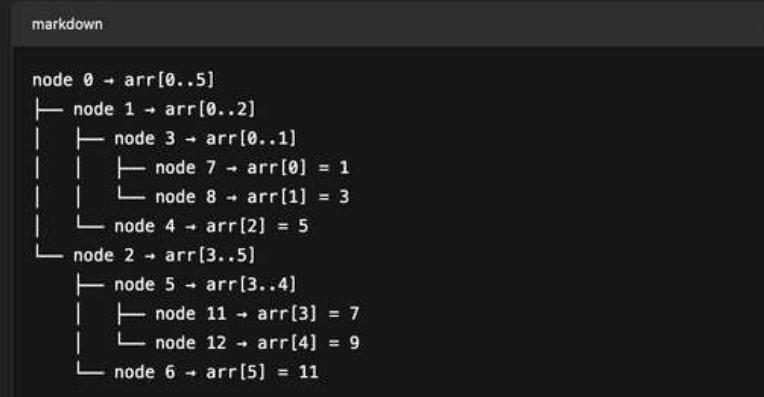
        public void update(int index, int value) {
            updateUtil(start: 0, end: n - 1, index, value, node: 0);
        }

        private void updateUtil(int start, int end, int idx, int val, int node) {
            if (start == end) {
                tree[node] = val;
                return;
            }
            int mid = (start + end) / 2;
            if (idx <= mid)
                updateUtil(start, mid, idx, val, node: 2 * node + 1);
            else
                updateUtil(start: mid + 1, end, idx, val, node: 2 * node + 2);
            tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
        }
    }

    public static void main(String[] args) {
        int[] nums = {1, 3, 5, 7, 9, 11};
        SegmentTree st = new SegmentTree(nums);
        System.out.println(st.query(1, 3)); // sum from index 1 to 3: 3+5+7 = 15
        st.update(index: 1, value: 10); // change index 1 to value 10
        System.out.println(st.query(1, 3)); // now 10+5+7 = 22
    }
}

```

We store the tree in an array `tree[]`, built as a binary tree:



What is 1-Based Indexing?

In 1-based indexing, array indices start from 1 instead of 0. This means:

- `arr[1]` refers to the first element
- `arr[2]` refers to the second element
- ...
- `arr[n]` refers to the nth element

Comparison with 0-Based Indexing:

Concept	0-Based Indexing	1-Based Indexing
First element index	<code>arr[0]</code>	<code>arr[1]</code>
Last element index	<code>arr[n-1]</code>	<code>arr[n]</code>
Common in	C, C++, Java, Python	Competitive programming, some Fenwick Tree logic

- `tree[4 * n]`:
 - ✓ This is just an upper bound to safely store all nodes (including internal and leaf nodes) of the segment tree.
 - It does not mean there will always be a value at `tree[4 * n]`. Most of the space is unused.
 - You only need about $2 \cdot 2^{\lceil \log_2(n) \rceil} - 1$ nodes in general, but `4 * n` is safe and standard in contests.
- `tree[n + 1]`:
 - ⚠ Not a meaningful index unless you're doing a special or custom implementation (e.g. storing values at leaves from `tree[n]` to `tree[2n-1]` in a 0-indexed segment tree).
 - In iterative segment trees, `tree[n + 1]` could represent the leaf node for `arr[1]` (if the leaf nodes start from `tree[n]` onward).

Why Use 1-Based Indexing in Fenwick Trees?

Because of how the Least Significant Bit (LSB) logic works using `i & -i`, 1-based indexing:

- Avoids infinite loops at index 0
- Makes the update/query logic cleaner

```

int i;
while (i > 0) {
    tree[i] += val;
    i -= i & -i;
}

```

Here, `0` is always `& 1`, and `0` is not used at all—it's treated as a dummy base.

2. In Fenwick Tree (Binary Indexed Tree):

Structure:

- Uses a 1D array `tree[]` of size `n + 1` for 1-based indexing.
- `tree[n + 1]`:
 - ✗ Not used. The valid range is `tree[1]` to `tree[n]`.
 - `tree[n + 1]` is out of bounds unless allocated explicitly.
 - In rare cases, it might be used for buffering or boundary checks, but it holds no meaning in standard operations.
- `tree[4 * n]`:
 - ✗ Not required at all.
 - Unlike segment trees, Fenwick Trees require only `n + 1` space (1-based index), so `4 * n` is overkill.
 - If you use `tree[4 * n]` in a Fenwick Tree, you're wasting space.

Step 1: Build Segment Tree

We store sum at each node.

java

Copy Edit

```
SegmentTree st = new SegmentTree([1, 3, 5, 7, 9, 11]);
```

- `tree[0]` = sum of [1, 3, 5, 7, 9, 11] = 36
- `tree[1]` = sum of [1, 3, 5] = 9
 - `tree[3]` = [1, 3] = 4
 - `tree[7]` = 1
 - `tree[8]` = 3
 - `tree[4]` = 5
- `tree[2]` = sum of [7, 9, 11] = 27
 - `tree[5]` = sum [7, 9] = 16
 - `tree[11]` = 7
 - `tree[12]` = 9
 - `tree[6]` = 11

Final Tree:

java

Copy Edit

```
tree = [36, 9, 27, 4, 5, 16, 11, 1, 3, _, _, 7, 9, _, _, 11]
```

Step 2: Query(1, 3)

Query sum in range index 1 to 3 in `nums`.

Breakdown:

1. `node 0` → range [0, 5], query [1, 3] → split into:
 - `node 1` → [0, 2]
 - `node 2` → [3, 5]
2. From `node 1`:
 - `node 3` → [0, 1] → only [1] is in range
 - `node 7` = 1 (index 0, out of range)
 - `node 8` = 3 ✓ in range → add 3
 - `node 4` = 5 ✓ index 2 → add 5 → sum from `node 1` = $3 + 5 = 8$
3. From `node 2`:
 - `node 5` → [3, 4] → index 3 in range → `node 11` = 7 ✓ → add 7

Total = $3 + 5 + 7 = 15$

Output: 15

✓ Step 3: Update index 1 → 10

```
java
```

Copy Edit

```
st.update(1, 10);
```

- Traverse from root:
 - node 0 → range [0, 5]
 - node 1 → [0, 2]
 - node 3 → [0, 1]
 - node 8 = index 1 → update to 10
 - Update node 3 = $1 + 10 = 11$
 - Update node 1 = $11 + 5 = 16$
 - Update node 0 = $16 + 27 = 43$

✓ Tree after update:

```
java
```

Copy Edit

```
tree = [43, 16, 27, 11, 5, 16, 11, 1, 10, -, -, 7, 9, -, -, 11]
```

✓ Step 4: Query(1, 3) again

Repeat step 2, now with updated value.

- Index 1 = 10
- Sum should be $10 + 5 + 7 = 22$

✓ Output: 22

Final Summary

Operation	Result
Query(1, 3) before update	15
Update index 1 → 10	-
Query(1, 3) after update	22

What is the minimum value between l and r? (Range Minimum Query (RMQ))

To find the minimum value between indices `l` and `r` in an array efficiently, you can use a Segment Tree built for the minimum operation (instead of sum). This is commonly known as a Range Minimum Query (RMQ).

✓ Problem

Given an array `arr[]`, and two indices `l` and `r`, return:

```
min(arr[l], arr[l+1], ..., arr[r])
```

✓ Naive Approach

Loop through `arr[l..r]` and keep track of the minimum.

- Time Complexity: $O(r - l + 1)$
- Not efficient for multiple queries.

✓ Efficient Approach — Segment Tree (for Min)

Segment Tree Construction

Each node stores the minimum of a range `[start, end]`.

⌚ Query(l, r)

- Recursively check left and right children.
- If the segment is completely within $[l, r]$, return the node's min.
- If no overlap, return `Integer.MAX_VALUE`.
- Otherwise, merge left and right: `Math.min(left, right)`.

```
public class Solution {

    private static class SegmentTree {
        int[] tree;
        int n;

        public SegmentTree(int[] nums) {
            n = nums.length;
            tree = new int[4 * n];
            build(nums, start: 0, end: n - 1, node: 0);
            System.out.println(Arrays.toString(tree));
        }

        private void build(int[] arr, int start, int end, int node) {
            if (start == end) {
                tree[node] = arr[start];
            } else {
                int mid = (start + end) / 2;
                build(arr, start, mid, node: 2 * node + 1);
                build(arr, start: mid + 1, end, node: 2 * node + 2);
                tree[node] = Math.min(tree[2 * node + 1], tree[2 * node + 2]);
            }
        }

        public int query(int l, int r) {
            return queryUtil( start: 0, end: n - 1, l, r, node: 0);
        }

        private int queryUtil(int start, int end, int l, int r, int node) {
            // No overlap
            if (r < start || l > end) return Integer.MAX_VALUE;
            // Complete overlap
            if (l <= start && end <= r) return tree[node];
            // Partial overlap
            int mid = (start + end) / 2;
            int leftMin = queryUtil(start, mid, l, r, node: 2 * node + 1);
            int rightMin = queryUtil( start: mid + 1, end, l, r, node: 2 * node + 2);
            return Math.min(leftMin, rightMin);
        }
    }

    public static void main(String[] args) {
        int[] nums = {1, 3, 2, 7, 9, 11};
        SegmentTree st = new SegmentTree(nums);
        System.out.println(st.query( l: 1, r: 4)); // Output: 2
        System.out.println(st.query( l: 2, r: 5)); // Output: 2
        System.out.println(st.query( l: 3, r: 3)); // Output: 7
    }
}
```

Minimum/Maximum

```
class Solution {
    int[] minTree, maxTree;
    int n;

    Solution(int[] arr) {
        n = arr.length;
        minTree = new int[4 * n];
        maxTree = new int[4 * n];
        build(arr, start: 0, end: n - 1, node: 0);
    }

    void build(int[] arr, int start, int end, int node) {
        if (start == end) {
            minTree[node] = arr[start];
            maxTree[node] = arr[start];
        } else {
            int mid = (start + end) / 2;
            build(arr, start, mid, node: 2 * node + 1);
            build(arr, start: mid + 1, end, node: 2 * node + 2);
            minTree[node] = Math.min(minTree[2 * node + 1], minTree[2 * node + 2]);
            maxTree[node] = Math.max(maxTree[2 * node + 1], maxTree[2 * node + 2]);
        }
    }

    int queryMin(int l, int r) {
        return queryUtilMin( start: 0, end: n - 1, l, r, node: 0);
    }

    int queryMax(int l, int r) {
        return queryUtilMax( start: 0, end: n - 1, l, r, node: 0);
    }

    int queryUtilMin(int start, int end, int l, int r, int node) {
        if (r < start || l > end) return Integer.MAX_VALUE;
        if (l <= start && end <= r) return minTree[node];
        int mid = (start + end) / 2;
        return Math.min(
            queryUtilMin(start, mid, l, r, node: 2 * node + 1),
            queryUtilMin( start: mid + 1, end, l, r, node: 2 * node + 2)
        );
    }

    int queryUtilMax(int start, int end, int l, int r, int node) {
        if (r < start || l > end) return Integer.MIN_VALUE;
        if (l <= start && end <= r) return maxTree[node];
        int mid = (start + end) / 2;
        return Math.max(
            queryUtilMax(start, mid, l, r, node: 2 * node + 1),
            queryUtilMax( start: mid + 1, end, l, r, node: 2 * node + 2)
        );
    }

    public static void main(String[] args) {
        int[] arr = {1, 3, 2, 7, 9, 11};
        Solution st = new Solution(arr);

        System.out.println(st.queryMin(l: 1, r: 4)); // Output: 2
        System.out.println(st.queryMax(l: 1, r: 4)); // Output: 9
        System.out.println(st.queryMin(l: 0, r: 5)); // Output: 1
        System.out.println(st.queryMax(l: 2, r: 5)); // Output: 11
    }
}
```

GCD

GCD (Greatest Common Divisor)

The Greatest Common Divisor (GCD) of two numbers is the largest positive integer that divides both numbers without leaving a remainder.

Methods to Find GCD

1. Euclidean Algorithm (Efficient)

- Uses the property:

$$\gcd(a, b) = \gcd(b, a \% b)$$

- Base case: $\gcd(a, 0) = a$

Time Complexity:

- $O(\log(\min(a, b)))$

```
java
public class GCDEExample {

    // Recursive Euclidean Algorithm
    public static int gcd(int a, int b) {
        if (b == 0) return a;
        return gcd(b, a % b);
    }

    // Iterative version
    public static int gcdIterative(int a, int b) {
        while (b != 0) {
            int temp = b;
            b = a % b;
            a = temp;
        }
        return a;
    }

    // Driver code
    public static void main(String[] args) {
        int a = 36, b = 60;

        System.out.println("Recursive GCD of " + a + " and " + b + " = " + gcd(a, b));
        System.out.println("Iterative GCD of " + a + " and " + b + " = " + gcdIterative(a
    }
}
```

Output:

```
mathematica
Recursive GCD of 36 and 60 = 12
Iterative GCD of 36 and 60 = 12
```

Use Cases

- Reducing fractions (e.g., simplify 36/60 → divide by GCD 12 → 3/5)
- LCM calculation:

$$\text{LCM}(a, b) = \frac{a \cdot b}{\text{GCD}(a, b)}$$

```
public class Solution {  
    int[] tree;  
    int n;  
  
    public Solution(int[] arr) {  
        this.n = arr.length;  
        this.tree = new int[4 * n];  
        build(arr, start: 0, end: n - 1, node: 0);  
    }  
  
    private int gcd(int a, int b) {  
        if (b == 0) return a;  
        return gcd(b, b: a % b);  
    }  
  
    private void build(int[] arr, int start, int end, int node) {  
        if (start == end) {  
            tree[node] = arr[start];  
        } else {  
            int mid = (start + end) / 2;  
            build(arr, start, mid, node: 2 * node + 1);  
            build(arr, start: mid + 1, end, node: 2 * node + 2);  
            tree[node] = gcd(tree[2 * node + 1], tree[2 * node + 2]);  
        }  
    }  
  
    public int queryGCD(int l, int r) {  
        return queryUtil( start: 0, end: n - 1, l, r, node: 0);  
    }  
}
```

```
public class Solution {
    int[] tree;
    int n;

    public Solution(int[] arr) {
        this.n = arr.length;
        this.tree = new int[4 * n];
        build(arr, start: 0, end: n - 1, node: 0);
    }

    private int gcd(int a, int b) {
        if (b == 0) return a;
        return gcd(b, b: a % b);
    }

    private void build(int[] arr, int start, int end, int node) {
        if (start == end) {
            tree[node] = arr[start];
        } else {
            int mid = (start + end) / 2;
            build(arr, start, mid, node: 2 * node + 1);
            build(arr, start: mid + 1, end, node: 2 * node + 2);
            tree[node] = gcd(tree[2 * node + 1], tree[2 * node + 2]);
        }
    }

    public int queryGCD(int l, int r) {
        return queryUtil( start: 0, end: n - 1, l, r, node: 0);
    }
}
```

Output

scss

 Copy

 Edit

```
GCD(1, 3): 3
GCD(0, 4): 3
GCD(1, 3) after update: 3
```

307. Range Sum Query - Mutable – (Basics of build, query, update)

307. Range Sum Query - Mutable

Medium

Given an integer array `nums`, handle multiple queries of the following types:

Update the value of an element in `nums`.

Calculate the sum of the elements of `nums` between indices `left` and `right` inclusive where `left <= right`.

Implement the `NumArray` class:

- `NumArray(int[] nums)` Initializes the object with the integer array `nums`.
- `void update(int index, int val)` Updates the value of `nums[index]` to be `val`.
- `int sumRange(int left, int right)` Returns the sum of the elements of `nums` between indices `left` and `right` inclusive (i.e. `nums[left] + nums[left + 1] + ... + nums[right]`).

Example 1:

```
Input["NumArray", "sumRange", "update", "sumRange"]
[[[1, 3, 5]], [0, 2], [1, 2], [0, 2]]
Output
[null, 9, null, 8]
Explanation
NumArray numArray = new NumArray([1, 3, 5]);
numArray.sumRange(0, 2); // return 1 + 3 + 5 = 9
numArray.update(1, 2); // nums = [1, 2, 5]
numArray.sumRange(0, 2); // return 1 + 2 + 5 = 8
```

```
class NumArray {
    int[] tree;
    int n;

    public NumArray(int[] nums) {
        n = nums.length;
        tree = new int[4 * n];
        build(nums, start: 0, end: n - 1, node: 0);
    }

    // Build segment tree
    private void build(int[] nums, int start, int end, int node) {
        if (start == end) {
            tree[node] = nums[start];
        } else {
            int mid = (start + end) / 2;
            build(nums, start, mid, node: 2 * node + 1);
            build(nums, start: mid + 1, end, node: 2 * node + 2);
            tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
        }
    }

    // Update a value at index
    public void update(int index, int val) {
        updateUtil(start: 0, end: n - 1, index, val, node: 0);
    }
}
```

```

    private void updateUtil(int start, int end, int idx, int val, int node) {
        if (start == end) {
            tree[node] = val;
        } else {
            int mid = (start + end) / 2;
            if (idx <= mid) {
                updateUtil(start, mid, idx, val, node: 2 * node + 1);
            } else {
                updateUtil( start: mid + 1, end, idx, val, node: 2 * node + 2);
            }
            tree[node] = tree[2 * node + 1] + tree[2 * node + 2];
        }
    }

    // Query sum from left to right
    public int sumRange(int left, int right) {
        return queryUtil( start: 0, end: n - 1, left, right, node: 0);
    }

    private int queryUtil(int start, int end, int l, int r, int node) {
        if (r < start || end < l) return 0; // no overlap
        if (l <= start && end <= r) return tree[node]; // complete overlap

        // partial overlap
        int mid = (start + end) / 2;
        int leftSum = queryUtil(start, mid, l, r, node: 2 * node + 1);
        int rightSum = queryUtil( start: mid + 1, end, l, r, node: 2 * node + 2);
        return leftSum + rightSum;
    }
}

```

315. Count of Smaller Numbers After Self

Given an integer array `nums`, return an integer array `counts` where `counts[i]` is the number of smaller elements to the right of `nums[i]`.

Example 1:

```

Input: nums = [5,2,6,1]
Output: [2,1,1,0]
Explanation:
To the right of 5 there are 2 smaller elements (2 and 1).
To the right of 2 there is only 1 smaller element (1).
To the right of 6 there is 1 smaller element (1).
To the right of 1 there is 0 smaller element.

```

Example 2:

```

Input: nums = [-1]
Output: [0]

```

Example 3:

```

Input: nums = [-1,-1]
Output: [0,0]

```

Constraints:

- $1 \leq \text{nums.length} \leq 10^5$
- $-10^4 \leq \text{nums}[i] \leq 10^4$

```

public class Solution {
    class SegmentTree {
        int[] tree;
        int size;

        public SegmentTree(int size) {
            this.size = size;
            this.tree = new int[4 * size];
        }

        // Increment count at index
        public void update(int index, int start, int end, int pos) {
            if (start == end) {
                tree[index]++;
            } else {
                int mid = (start + end) / 2;
                if (pos <= mid) {
                    update(index: 2 * index + 1, start, mid, pos);
                } else {
                    update(index: 2 * index + 2, start: mid + 1, end, pos);
                }
                tree[index] = tree[2 * index + 1] + tree[2 * index + 2];
            }
        }

        // Query how many numbers < pos
        public int query(int index, int start, int end, int l, int r) {
            if (r < start || end < l) return 0; // no overlap
            if (l <= start && end <= r) return tree[index]; // full overlap

            int mid = (start + end) / 2;
            int left = query(index: 2 * index + 1, start, mid, l, r);
            int right = query(index: 2 * index + 2, start: mid + 1, end, l, r);
            return left + right;
        }
    }

    public List<Integer> countSmaller(int[] nums) {
        List<Integer> result = new ArrayList<>();

        // 1. Coordinate Compression
        TreeSet<Integer> set = new TreeSet<>();
        for (int num : nums) set.add(num);

        Map<Integer, Integer> map = new HashMap<>();
        int rank = 0;
        for (int num : set) {
            map.put(num, rank++);
        }

        // 2. Build segment tree
        SegmentTree segTree = new SegmentTree(rank);

        // 3. Traverse from right to left
        for (int i = nums.length - 1; i >= 0; i--) {
            int index = map.get(nums[i]);
            int count = segTree.query(index: 0, start: 0, end: rank - 1, l: 0, r: index - 1);
            result.add(count);
            segTree.update(index: 0, start: 0, end: rank - 1, index);
        }

        // Reverse the result
        Collections.reverse(result);
        return result;
    }

    public static void main(String[] args) {
        Solution solution = new Solution();

        // Example input
        int[] nums = {5, 2, 6, 1};

        // Call the method
        List<Integer> result = solution.countSmaller(nums);

        // Output the result
        System.out.println("Count of smaller elements after self:");
        System.out.println(result); // Expected Output: [2, 1, 1, 0]
    }
}

```

715. Range Module

A Range Module is a module that tracks ranges of numbers. Design a data structure to track the ranges represented as half-open intervals and query about them.

A half-open interval `[left, right)` denotes all the real numbers x where $\text{left} \leq x < \text{right}$.

Implement the `RangeModule` class:

- `RangeModule()` Initializes the object of the data structure.
- `void addRange(int left, int right)` Adds the half-open interval `[left, right)`, tracking every real number in that interval. Adding an interval that partially overlaps with currently tracked numbers should add any numbers in the interval `[left, right)` that are not already tracked.
- `boolean queryRange(int left, int right)` Returns `true` if every real number in the interval `[left, right)` is currently being tracked, and `false` otherwise.
- `void removeRange(int left, int right)` Stops tracking every real number currently being tracked in the half-open interval `[left, right)`.

Example 1:

Input["RangeModule", "addRange", "removeRange", "queryRange", "queryRange", "queryRange"]

[[], [10, 20], [14, 16], [10, 14], [13, 15], [16, 17]]

Output

[null, null, null, true, false, true]

Explanation

```
RangeModule rangeModule = new RangeModule();
rangeModule.addRange(10, 20);
rangeModule.removeRange(14, 16);
rangeModule.queryRange(10, 14); // return True,(Every number in [10, 14) is being
tracked)
rangeModule.queryRange(13, 15); // return False,(Numbers like 14, 14.03, 14.17 in [13,
15) are not being tracked)
rangeModule.queryRange(16, 17); // return True, (The number 16 in [16, 17) is still being
tracked, despite the remove operation)
```

```
class RangeModule {
    class Node {
        int start, end;
        Node left, right;
        boolean tracked;

        Node(int start, int end) {
            this.start = start;
            this.end = end;
            this.tracked = false;
        }

        private void pushDown() {
            if (left == null && end - start > 1) {
                int mid = start + (end - start) / 2;
                left = new Node(start, mid);
                right = new Node(mid, end);
                left.tracked = tracked;
                right.tracked = tracked;
            }
        }

        public void addRange(int l, int r) {
            if (r <= start || end <= l) return;
            if (l <= start && end <= r) {
                tracked = true;
                left = null;
                right = null;
                return;
            }
            pushDown();
            left.addRange(l, r);
            right.addRange(l, r);
            tracked = left.tracked && right.tracked;
        }
    }
}
```

```
        public boolean queryRange(int l, int r) {
            if (r <= start || end <= l) return true;
            if (l <= start && end <= r) return tracked;
            if (left == null) return tracked;
            return left.queryRange(l, r) && right.queryRange(l, r);
        }
    }

    Node root;

    public RangeModule() {
        root = new Node(0, 1_000_000_000); // Initialize with max constraint
    }

    public void addRange(int left, int right) {
        root.addRange(left, right);
    }

    public boolean queryRange(int left, int right) {
        return root.queryRange(left, right);
    }

    public void removeRange(int left, int right) {
        root.removeRange(left, right);
    }
}

/**
 * Your RangeModule object will be instantiated and called as such:
 * RangeModule obj = new RangeModule();
 * obj.addRange(left,right);
 * boolean param_2 = obj.queryRange(left,right);
 * obj.removeRange(left,right);
 */

```

732. My Calendar III – (Interval management using Segment Tree)

732. My Calendar III

Solved

Hard

A k -booking happens when k events have some non-empty intersection (i.e., there is some time that is common to all k events.)

You are given some events $[\text{startTime}, \text{endTime}]$, after each given event, return an integer k representing the maximum k -booking between all the previous events.

Implement the `MyCalendarThree` class:

- `MyCalendarThree()` Initializes the object.
- `int book(int startTime, int endTime)` Returns an integer k representing the largest integer such that there exists a k -booking in the calendar.

Example 1:

Input
["MyCalendarThree", "book", "book", "book", "book", "book", "book", "book"]
[[], [10, 20], [50, 60], [10, 40], [5, 15], [5, 10], [25, 55]]
Output
[null, 1, 1, 2, 3, 3, 3]

Explanation
MyCalendarThree myCalendarThree = new MyCalendarThree();
myCalendarThree.book(10, 20); // return 1
myCalendarThree.book(50, 60); // return 1
myCalendarThree.book(10, 40); // return 2
myCalendarThree.book(5, 15); // return 3
myCalendarThree.book(5, 10); // return 3
myCalendarThree.book(25, 55); // return 3

```
java
import java.util.*;

class MyCalendarThree {

    List<int[]> bookings;

    public MyCalendarThree() {
        bookings = new ArrayList<>();
    }

    public int book(int start, int end) {
        bookings.add(new int[]{start, end});
        int maxOverlap = 0;

        for (int i = 0; i < bookings.size(); i++) {
            int[] interval1 = bookings.get(i);
            int count = 0;

            for (int[] interval2 : bookings) {
                if (interval1[0] < interval2[1] & interval2[0] < interval1[1]) {
                    count++;
                }
            }

            maxOverlap = Math.max(maxOverlap, count);
        }

        return maxOverlap;
    }
}
```

```

class MyCalendarThree {

    class Node {
        int start, end, max, lazy;
        Node left, right;
        Node(int start, int end) {
            this.start = start;
            this.end = end;
            this.max = 0;
            this.lazy = 0;
        }

        private void push() {
            if (left == null) {
                int mid = start + (end - start) / 2;
                left = new Node(start, mid);
                right = new Node(mid + 1, end);
            }
            if (lazy != 0) {
                left.max += lazy;
                left.lazy += lazy;
                right.max += lazy;
                right.lazy += lazy;
                lazy = 0;
            }
        }
    }

    public void update(int l, int r) {
        if (r < start || end < l) return;
        if (l <= start && end <= r) {
            max++;
            lazy++;
            return;
        }
        push();
        left.update(l, r);
        right.update(l, r);
        max = Math.max(left.max, right.max);
    }
}

```

```

Node root;
int MAX_TIME = (int) 1e9;

public MyCalendarThree() {
    root = new Node(0, MAX_TIME);
}

public int book(int start, int end) {
    root.update(start, end - 1);
    return root.max;
}
}

/**
 * Your MyCalendarThree object will be instantiated and called as such:
 * MyCalendarThree obj = new MyCalendarThree();
 * int param_1 = obj.book(startTime,endTime);
 */

```

This function updates the segment $[l, r]$ by incrementing the number of overlapping bookings.

- Base Case 1: if $(r < start \text{ || } end < l)$ → no overlap → do nothing.
- Base Case 2: if $(l <= start \text{ && } end <= r)$ → full overlap → increment `max` and `lazy`.
- Partial Overlap: Use `push()` to propagate and split, then recursively update children and recalculate `max`.

- `start`, `end`: Range that this segment tree node covers.
- `max`: Maximum number of overlapping bookings in this node's range.
- `lazy`: Value for lazy propagation (delayed updates).
- `left`, `right`: Children of the current node.

Binary Indexed Tree (Fenwick Tree)

A Fenwick Tree, also known as a Binary Indexed Tree (BIT), is a data structure that efficiently supports:

- Prefix sum queries (e.g., sum of elements from index 0 to i)
- Point updates (e.g., add a value to element at index i)

Both operations can be done in $O(\log n)$ time using bit manipulation.

🔍 Why it's called "Binary Indexed Tree"

The name **Binary Indexed Tree** comes from how it uses binary representation of indices to efficiently store and query cumulative information.

Let's say we're working with a 1-based index system. For a given index i:

- $i \& -i$ gives the least significant set bit (LSSB).
- This tells us the size of the range the current node ($\text{bit}[i]$) represents.

For example:

If $i = 6$, binary is 0110,

- $i \& -i = 0010 \rightarrow$ value is 2
- So, $\text{bit}[6]$ stores sum of 2 elements: $\text{arr}[5] + \text{arr}[6]$

Each index in the BIT stores the sum of a range of elements, and the range size is determined by the least significant set bit of the index.

◆ Fenwick Tree (Binary Indexed Tree)

Feature	Description
Structure	Array-based (implicit tree using index bits)
Operations	Efficient for prefix sum / cumulative frequency
Time Complexity	$O(\log N)$ for both <code>update()</code> and <code>query()</code>
Space Complexity	$O(N)$
Best for	Point updates and prefix/range sum queries
Limitations	Cannot handle complex operations like range minimum/maximum, GCD, or arbitrary associative functions easily
Implementation	Simpler and lighter than segment tree

◆ Segment Tree

Feature	Description	🔗
Structure	Tree (explicitly constructed with nodes or arrays)	🔗
Operations	Generic: Supports range queries (sum, min, max, GCD, XOR, etc.) and point/range updates	🔗
Time Complexity	$O(\log N)$ for both <code>update()</code> and <code>query()</code>	🔗
Space Complexity	$O(4N)$ for array-based implementation	🔗
Best for	Range queries (min/max/sum/etc.) and lazy propagation for range updates	🔗
Limitations	Slightly heavier in memory and more complex to implement	🔗

💡 Intuition Behind It

A Fenwick Tree breaks down a prefix sum into a few relevant ranges (based on powers of 2). It only needs to look at $O(\log n)$ such ranges, which is what makes it efficient.

For example, if we want prefix sum till index 13:

- We look at `bit[13]`, `bit[12]`, `bit[8]`, etc., jumping back by `i & -i` each time.

⚙️ Internal Working

Update Operation:

To add value `x` at index `i`:

```
java Copy Edit
while (i <= n) {
    bit[i] += x;
    i += i & -i; // jump to parent
}
```

Query Operation:

To get sum of elements from 1 to `i`:

```
java Copy Edit
while (i > 0) {
    sum += bit[i];
    i -= i & -i; // jump to ancestor that contributed to i
}
```

💡 Why use Fenwick Tree?

- Much faster than a naive prefix sum array for **dynamic updates**.
- Lighter and easier to implement than a Segment Tree in many cases.
- Useful in problems involving:
 - Inversion Count
 - K-th smallest/largest elements
 - Dynamic frequency counting
 - 2D Range Queries (with 2D Fenwick Tree)

✍️ Example Visualization

Let `arr = [1, 7, 3, 0, 7, 8, 3, 2, 6, 2]` (1-indexed).

We build `bit[]` such that:

- `bit[1] = arr[1]`
- `bit[2] = arr[1] + arr[2]`
- `bit[4] = arr[1] + arr[2] + arr[3] + arr[4]`
- `bit[8] = arr[1] to arr[8]`

So, we can quickly query or update sums of intervals like `[1..i]` by combining few such blocks.

Use Case:

Supports:

- `update(index, value)` – Adds `value` at index `i`
- `query(index)` – Returns the prefix sum from `1` to `index`

```
class FenwickTree {  
    private int[] tree;  
    private int size;  
  
    public FenwickTree(int n) {  
        this.size = n;  
        this.tree = new int[n + 1]; // 1-based indexing  
    }  
  
    // Adds 'value' to index 'i'  
    public void update(int i, int value) {  
        while (i <= size) {  
            tree[i] += value;  
            i += i & -i; // move to next responsible index  
        }  
    }  
  
    // Returns prefix sum from index 1 to i  
    public int query(int i) {  
        int sum = 0;  
        while (i > 0) {  
            sum += tree[i];  
            i -= i & -i; // move to parent index  
        }  
        return sum;  
    }  
  
    // Returns sum in range [l, r]  
    public int rangeQuery(int l, int r) {  
        return query(r) - query(l - 1);  
    }  
  
    public static void main(String[] args) {  
        int[] arr = {0, 3, 2, -1, 6, 5, 4}; // index 0 is dummy for 1-based indexing  
        int n = arr.length - 1;  
        FenwickTree ft = new FenwickTree(n);  
        for (int i = 1; i <= n; i++) {  
            ft.update(i, arr[i]);  
        }  
        // Prefix sum from index 1 to 5  
        System.out.println("Prefix sum [1..5]: " + ft.query(5)); // Output: Prefix sum [1..5]: 15  
        // Range sum from index 2 to 5  
        System.out.println("Range sum [2..5]: " + ft.rangeQuery(2, 5)); // Output: Range sum [2..5]: 12  
        // Update index 3 by +2  
        ft.update(3, value: 2);  
        System.out.println("Prefix sum [1..5] after update: " + ft.query(5)); // Output: Prefix sum [1..5] after update: 17  
    }  
}
```

Input Array:

java

Copy Edit

```
int[] arr = {0, 3, 2, -1, 6, 5, 4};
// Index: 0 1 2 3 4 5 6
// Note: index 0 is dummy (for 1-based indexing)
```

So $n = 6$.

Step 1: Build Fenwick Tree

We'll insert values one by one using `update(i, arr[i])`

✓ Update(1, 3):

- $i = 1 \rightarrow tree[1] += 3 \rightarrow tree[1] = 3$
- $i = 2 \rightarrow tree[2] += 3 \rightarrow tree[2] = 3$
- $i = 4 \rightarrow tree[4] += 3 \rightarrow tree[4] = 3$
- $i = 8 > 6 \rightarrow done$

tree = [0, 3, 3, 0, 3, 0, 0]

✓ Update(2, 2):

- $i = 2 \rightarrow tree[2] += 2 \rightarrow tree[2] = 5$
- $i = 4 \rightarrow tree[4] += 2 \rightarrow tree[4] = 5$
- $i = 8 > 6 \rightarrow done$

tree = [0, 3, 5, 0, 5, 0, 0]

✓ Update(3, -1):

- $i = 3 \rightarrow tree[3] += -1 \rightarrow tree[3] = -1$
- $i = 4 \rightarrow tree[4] += -1 \rightarrow tree[4] = 4$
- $i = 8 > 6 \rightarrow done$

tree = [0, 3, 5, -1, 4, 0, 0]

✓ Update(4, 6):

- $i = 4 \rightarrow tree[4] += 6 \rightarrow tree[4] = 10$
- $i = 8 > 6 \rightarrow done$

tree = [0, 3, 5, -1, 10, 0, 0]

✓ Update(5, 5):

- $i = 5 \rightarrow tree[5] += 5 \rightarrow tree[5] = 5$
- $i = 6 \rightarrow tree[6] += 5 \rightarrow tree[6] = 5$
- $i = 8 > 6 \rightarrow done$

tree = [0, 3, 5, -1, 10, 5, 5]

✓ Update(6, 4):

- $i = 6 \rightarrow tree[6] += 4 \rightarrow tree[6] = 9$
- $i = 8 > 6 \rightarrow done$

Final tree[] = [0, 3, 5, -1, 10, 5, 9]

✗ **Query 1:** query(5)

Compute prefix sum up to index 5:

- `i = 5 → sum += tree[5] = 5`
- `i = 4 → sum += tree[4] = 10 → sum = 15`
- `i = 0 → done`

✓ Result: Prefix sum [1...5] = 15

✗ **Query 2:** rangeQuery(2, 5) = query(5) - query(1)

query(5):

- From before → 15

query(1):

- `i = 1 → sum += tree[1] = 3`
- `i = 0 → done`

✓ rangeQuery = 15 - 3 = 12

💡 **Update:** update(3, +2)

- `i = 3 → tree[3] += 2 → now = 1`
- `i = 4 → tree[4] += 2 → now = 12`
- `i = 8 → done`

✓ Updated tree[] = [0, 3, 5, 1, 12, 5, 9]

// Fenwick Tree structure after building with arr[1..5]
// Each line shows which tree[i] contributes to which others
// via $i += i \& -i$ (during update)
// Think of this like a "parent to children" relation

```
tree[1]
  └── contributes to tree[2]
      └── contributes to tree[4]
```

```
tree[2]
  └── contributes to tree[4]
```

```
tree[3]
  └── contributes to tree[4]
```

```
tree[4] // receives values from tree[1], tree[2], tree[3]
```

```
tree[5] // standalone (no further contributions since 6 > size)
```

✓ **Summary Output**

```
java
[0, 3, 5, -1, 10, 5, 9]          // Original tree
Prefix sum [1...5]: 15
Range sum [2...5]: 12
Prefix sum [1...5] after update: 17
```

// So visually, it's like:

```
tree[1]
  └── tree[2]
      └── tree[4]
tree[3]
  └── tree[4]
tree[5]
```

- `tree[2]` = sum of values at index `1` and `2`
- `tree[4]` = sum of values at indices `1` to `4`, using the update propagation logic

```

class FenwickTree {
    private int[] tree;
    private int n;

    public FenwickTree(int[] arr) {
        this.n = arr.length;
        this.tree = new int[n + 1];
        build(arr); // O(n log n)
    }

    // Gets the lowest bit (last set bit)
    private int lowbit(int x) {
        return x & (-x);
    }

    // Build the tree in O(n log n)
    private void build(int[] arr) {
        for (int i = 0; i < n; i++) {
            update(i + 1, arr[i]); // +1 for 1-based indexing
        }
    }

    // Update index i by adding value (1-based)
    public void update(int i, int value) {
        while (i <= n) {
            tree[i] += value;
            i += lowbit(i);
        }
    }

    // Query prefix sum from 1 to i
    public int query(int i) {
        int sum = 0;
        while (i > 0) {
            sum += tree[i];
            i -= lowbit(i);
        }
        return sum;
    }

    // Range sum from l to r (inclusive)
    public int rangeQuery(int l, int r) {
        return query(r) - query(l - 1);
    }
}

```

```

java
public class Main {
    public static void main(String[] args) {
        int[] arr = {3, 2, -1, 6, 5, 4}; // 0-based array
        FenwickTree ft = new FenwickTree(arr);

        System.out.println("Prefix sum [1...5]: " + ft.query(5));           // 3+2+(-1)+6+5
        System.out.println("Range sum [2...5]: " + ft.rangeQuery(2, 5)); // 2+(-1)+6+5 =

        ft.update(3, 2); // arr[2] = -1 + 2 = 1
        System.out.println("Prefix sum [1...5] after update: " + ft.query(5)); // now = 1
    }
}

```

⌚ Time Complexity

Operation	Time
lowbit()	O(1)
build()	O(n log n)
update()	O(log n)
query()	O(log n)

307. Range Sum Query - Mutable

Problem Summary

Implement a data structure that supports:

- `update(index, val)` — Updates the value at `index`.
- `sumRange(left, right)` — Returns the sum of elements between indices `left` and `right` inclusive.

```
class NumArray {  
    private int[] nums;  
    private int[] tree;  
    private int n;  
  
    public NumArray(int[] nums) {  
        this.n = nums.length;  
        this.nums = new int[n];  
        this.tree = new int[n + 1];  
  
        for (int i = 0; i < n; i++) {  
            update(i, nums[i]);  
        }  
    }  
  
    // Gets the least significant bit  
    private int lowbit(int x) {  
        return x & (-x);  
    }  
  
    // Updates the Fenwick Tree and internal array  
    public void update(int index, int val) {  
        int delta = val - nums[index];  
        nums[index] = val;  
        index++; // Convert to 1-based index  
        while (index <= n) {  
            tree[index] += delta;  
            index += lowbit(index);  
        }  
    }  
  
    // Prefix sum query from 0 to i  
    private int query(int index) {  
        index++; // Convert to 1-based index  
        int sum = 0;  
        while (index > 0) {  
            sum += tree[index];  
            index -= lowbit(index);  
        }  
        return sum;  
    }  
  
    // Sum range query from left to right  
    public int sumRange(int left, int right) {  
        return query(right) - query(left - 1);  
    }  
  
    public static void main(String[] args) {  
        int[] arr = {1, 3, 5};  
        NumArray numArray = new NumArray(arr);  
  
        System.out.println(numArray.sumRange( left: 0, right: 2)); // Output: 9  
        numArray.update( index: 1, val: 2); // arr = [1, 2, 5]  
        System.out.println(numArray.sumRange( left: 0, right: 2)); // Output: 8  
    }  
}
```

update

```
java  
  
public void update(int index, int val) {  
    int delta = val - nums[index];  
    nums[index] = val;  
    updateTree(index + 1, delta);  
}
```

[Copy](#) [Edit](#)

- Updates the value at `index` to `val`.
- Calculates the change (`delta`) needed.
- Applies `delta` to the Fenwick Tree.
- `index + 1` due to 1-based indexing.

query

```
java  
  
private int query(int i) {  
    int sum = 0;  
    while (i > 0) {  
        sum += tree[i];  
        i -= i & -i;  
    }  
    return sum;  
}
```

[Copy](#) [Edit](#)

- Computes prefix sum from index 1 to `i` using the Fenwick Tree.
- `i -= i & -i` moves to parent.

Why `delta` is needed:

The original value at `nums[1]` was 3, and you're changing it to 2.

So the difference (delta) is:

```
java  
  
delta = new_value - old_value  
      = 2 - 3  
      = -1
```

[Copy](#) [Edit](#)

To keep the prefix sums accurate, you must subtract 1 from all prefix sums in the tree that include `nums[1]`.

Thus, you can't just replace 3 with 2 in the tree — you add `-1` (`delta`) to the tree nodes that include index 1.

Why not directly set the value in the tree?

Because the tree is built using overlapping partial sums, directly setting values would corrupt the entire structure.

Example:

- `tree[2]` may represent the sum of `nums[1] + nums[2]`
- If you change `nums[1]` from 3 → 2, only part of `tree[2]` should be changed (by `delta -1`)

Thus, we propagate `delta` using:

```
java  
  
updateTree(index + 1, delta);
```

[Copy](#) [Edit](#)

update in FenwickTree class (Your simpler version):

```
java
// Adds 'value' to index 'i'
public void update(int i, int value) {
    while (i <= size) {
        tree[i] += value;
        i += i & -i;
    }
}
```

+ What it does:

This method adds a value (e.g., `+2`, `-3`) to the original index and updates the relevant prefix sums in the BIT.

Use Case:

- This is used during the initial construction (`build`) of the BIT.
- Or when you're modifying an element by adding a delta.
- It does not track the original array. It only works well when:
 - You know the amount you're adding (`+value`)
 - You don't care about the original value of the element

No need for `delta`:

You're already passing the delta directly. Example:

```
java
ft.update(3, 2); // means: add 2 to index 3
```

update in NumArray (LeetCode 307) with tracking:

```
java
public void update(int index, int val) {
    int delta = val - nums[index]; // calculate difference
    nums[index] = val;
    updateTree(index + 1, delta); // apply delta
}
```

+ What it does:

This method sets the element at `index` to a new value `val`. It calculates the difference (`delta`) between old and new values and then applies this delta using BIT logic.

Use Case:

- When you're doing a set operation like `arr[i] = 10`, and not just `arr[i] += x`
- You must maintain a separate `nums[]` array to compute the delta
- Ideal for use cases like Leetcode 307: `update(i, val)`

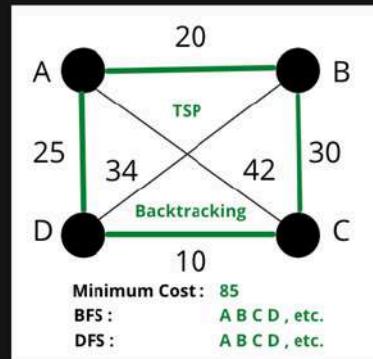
Summary Comparison

Feature	Simple FenwickTree Update	NumArray Update (Leetcode 307)
Operation type	Add delta	Set to new value
Requires original array?	No	Yes (for delta calculation)
Used in build?	Yes	Yes
Used in <code>set()</code> operations?	Not suitable	Yes
Value passed	Delta directly (<code>+value</code>)	Target value, then compute delta

- Count of Smaller Numbers After Self
- Count of Range Sum
- Reverse Pairs
- Create Sorted Array through Instructions
- Inversion Count
- K-th smallest/largest elements
- Dynamic frequency counting

Travelling Salesman Problem

Traveling Salesman Problem Statement



Given a set of cities and distance between every pair of cities, the problem is to find the shortest possible route that visits every city exactly once and returns back to the starting point.

```
public class TSPDFS {  
    static int n;  
    static int[][] graph;  
    static boolean[] visited;  
    static int minCost = Integer.MAX_VALUE;  
  
    public static int tsp(int[][] inputGraph) {  
        n = inputGraph.length;  
        graph = inputGraph;  
        visited = new boolean[n];  
        minCost = Integer.MAX_VALUE;  
        dfs(current: 0, count: 1, cost: 0, start: 0); // Start from node 0  
        return minCost == Integer.MAX_VALUE ? -1 : minCost;  
    }  
  
    private static void dfs(int current, int count, int cost, int start) {  
        if (count == n && graph[current][start] > 0) {  
            minCost = Math.min(minCost, cost + graph[current][start]);  
            return;  
        }  
  
        visited[current] = true;  
        for (int next = 0; next < n; next++) {  
            if (!visited[next] && graph[current][next] > 0) {  
                dfs(next, count + 1, cost + graph[current][next], start);  
            }  
        }  
        visited[current] = false; // Backtrack  
    }  
  
    public static void main(String[] args) {  
        int[][] graph = {  
            {0, 10, 15, 20},  
            {10, 0, 35, 25},  
            {15, 35, 0, 30},  
            {20, 25, 30, 0}  
        };  
  
        System.out.println("Minimum cost: " + tsp(graph)); // Output: 80  
    }  
}
```

In your DFS-based TSP code:

```
java  
if (!visited[next] && graph[current][next] > 0)
```

Copy

Edit

• Why `graph[current][next] > 0`?

This condition checks whether there's a valid edge between the current city and the `next` city. Here's what it's doing:

✓ `graph[current][next] > 0`:

- Ensures that there's an edge with a positive cost from `current` to `next`.
- Prevents trying to travel on non-existent paths, especially if the graph is sparse (i.e., not fully connected).
- Avoids invalid or unintended behavior like adding `0` to the cost or accessing undefined paths.

* What if we skip this condition?

If you remove `graph[current][next] > 0`, you risk:

- Traversing to cities with no direct connection.
- Adding 0 to the cost (which might be incorrect).
- Completing an invalid tour that doesn't exist in the original graph.

↗ Example:

Let's say your graph looks like:

```
java  
int[][] graph = {  
    {0, 10, 0, 20},  
    {10, 0, 35, 25},  
    {0, 35, 0, 30},  
    {20, 25, 30, 0}  
};
```

Copy

Edit

Here:

- `graph[0][2] == 0`: No direct edge between city 0 and city 2.
- `graph[2][0] == 0`: Also no path back.

Without `graph[current][next] > 0`, DFS might still try to go from 0 to 2 (which is invalid).

✓ Alternative:

If you want to allow 0-cost edges only when valid, you should use a sentinel value like `-1` or `Integer.MAX_VALUE` to represent no path. Then the condition becomes:

```
java  
if (!visited[next] && graph[current][next] != -1)
```

Copy

Edit

```

public class TSPHeldKarp {

    public static int tsp(int[][] graph) {
        int n = graph.length;
        int[][] dp = new int[1 << n][n];
        // Initialize with infinity
        for (int[] row : dp)
            Arrays.fill(row, val: Integer.MAX_VALUE / 2); // To prevent overflow
        // Base case: start from city 0
        dp[1][0] = 0;
        for (int mask = 1; mask < (1 << n); mask++) {
            for (int u = 0; u < n; u++) {
                if ((mask & (1 << u)) == 0) continue; // city u not in subset

                for (int v = 0; v < n; v++) {
                    if ((mask & (1 << v)) != 0 || graph[u][v] == 0) continue;
                    int nextMask = mask | (1 << v);
                    dp[nextMask][v] = Math.min(dp[nextMask][v], dp[mask][u] + graph[u][v]);
                }
            }
        }
        // Complete the cycle by returning to the starting city (0)
        int minCost = Integer.MAX_VALUE;
        for (int i = 1; i < n; i++) {
            if (graph[i][0] > 0)
                minCost = Math.min(minCost, dp[(1 << n) - 1][i] + graph[i][0]);
        }
        return minCost;
    }

    public static void main(String[] args) {
        int[][] graph = {
            {0, 10, 15, 20},
            {10, 0, 35, 25},
            {15, 35, 0, 30},
            {20, 25, 30, 0}
        };

        System.out.println("Minimum cost: " + tsp(graph)); // Output: 80
    }
}

```

💡 Key Concepts

- **State:** `dp[mask][i]` represents the minimum cost to reach city `i` having visited the cities in `mask`.
- **mask:** a bitmask of visited cities (e.g., `01101` means cities 0, 2, and 3 are visited).
- **Transition:** For each subset of visited cities, and for each city `i` in that subset, update `dp[mask][i]` by trying all `j ≠ i` such that `j` is in `mask`.

⌚ Time & Space Complexity

- **Time:** $O(n^2 * 2^n)$ — exponential but efficient for $n \leq 20$
- **Space:** $O(n * 2^n)$

📌 Output

yaml

Copy Edit

Minimum cost: 80

1. Count number of subMatrixes with all 1

```
public int numSubmat(int[][] mat) {
    int m = mat.length, n = mat[0].length;
    int[][] rowOnes = new int[m][n];
    int count = 0;

    // Step 1: Build row-wise consecutive 1s
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (mat[i][j] == 1) {
                rowOnes[i][j] = (j == 0) ? 1 : rowOnes[i][j - 1] + 1;
            }
        }
    }

    // Step 2: For each cell, go upwards and find minimum width to count rectangles
    for (int j = 0; j < n; j++) {
        for (int i = 0; i < m; i++) {
            if (mat[i][j] == 1) {
                int minWidth = rowOnes[i][j];
                for (int k = i; k >= 0 && rowOnes[k][j] > 0; k--) {
                    minWidth = Math.min(minWidth, rowOnes[k][j]);
                    count += minWidth;
                }
            }
        }
    }

    return count;
}
```

```
public class ManufacturingUnit {
    public static long getMaximumQuantity(List<Integer> power, List<Integer> quantity, int maxPower) {
        int n = power.size();
        long[][] dp = new long[n + 1][maxPower + 1];

        for (int i = 1; i <= n; i++) {
            for (int j = 0; j <= maxPower; j++) {
                dp[i][j] = dp[i - 1][j];

                if (j >= power.get(i - 1)) {
                    dp[i][j] = Math.max(dp[i][j], dp[i - 1][j - power.get(i - 1)] + quantity.get(i - 1));
                }
            }
        }

        return dp[n][maxPower];
    }

    public static void main(String[] args) {
        List<Integer> power = List.of(2, 2, 2);
        List<Integer> quantity = List.of(1, 2, 3);
        int maxPower = 4;

        long result = getMaximumQuantity(power, quantity, maxPower);
        System.out.println(result);
    }
}
```

dp[i][j] represents the maximum total quantity of goods that can be produced using the first i machines (from power and quantity lists) and considering a total power constraint of j.