LeetCode Blind 75

(Array)

Two Sum

The "Two Sum" problem is a classic algorithmic problem where you are given an array of integers and a target sum. The task is to find two distinct indices in the array such that the numbers at those indices add up to the target sum. Below are different approaches to solve this problem in Java, along with explanations.

1. Brute Force Approach

This is the simplest approach where we check every possible pair in the array to see if they add up to the target.

```java
public int[] twoSum(int[] nums, int target) {
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
            if (nums[i] + nums[j] == target) {
                return new int[]{i, j};
            }
        }
    }
    throw new IllegalArgumentException("No two sum solution");
}
```

Explanation:
- We use two nested loops to iterate through all possible pairs of indices `(i, j)`.
- For each pair, we check if the sum of `nums[i]` and `nums[j]` equals the target.
- If a valid pair is found, we return their indices.
- If no pair is found, we throw an exception.

Time Complexity: $O(n^2)$
- The outer loop runs `n` times, and the inner loop runs `n-1` times, resulting in $O(n^2)$.

Space Complexity: O(1)
- No additional space is used.

2. Two-Pass Hash Table
This approach uses a hash table (HashMap) to store the array elements and their indices. We first populate the hash table and then check for the complement of each element.

```java
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        map.put(nums[i], i);
    }
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement) && map.get(complement) != i) {
            return new int[]{i, map.get(complement)};
        }
    }
    throw new IllegalArgumentException("No two sum solution");
}
```

Explanation:
- In the first pass, we store each element and its index in a HashMap.
- In the second pass, for each element, we calculate its complement (`target - nums[i]`) and check if it exists in the HashMap.
- If the complement exists and is not the same element, we return the indices.

Time Complexity: O(n)
- We traverse the array twice, but each lookup in the HashMap is O(1).

Space Complexity: O(n)
- We use extra space to store the HashMap.

3. One-Pass Hash Table
This is an optimized version of the two-pass hash table approach. We check for the complement while populating the HashMap.

```java
public int[] twoSum(int[] nums, int target) {
    Map<Integer, Integer> map = new HashMap<>();
    for (int i = 0; i < nums.length; i++) {
        int complement = target - nums[i];
        if (map.containsKey(complement)) {
            return new int[]{map.get(complement), i};
        }
        map.put(nums[i], i);
    }
    throw new IllegalArgumentException("No two sum solution");
}
```

Explanation:
- As we iterate through the array, we calculate the complement for the current element.
- If the complement is already in the HashMap, we return the indices.
- Otherwise, we add the current element and its index to the HashMap.

Time Complexity: O(n)
- We traverse the array only once.

Space Complexity: O(n)
- We use extra space to store the HashMap.

4. Sorting with Two Pointers
This approach involves sorting the array and using two pointers to find the pair. However, this approach does not return the original indices, so it is not suitable if the problem requires indices.

```java
public int[] twoSum(int[] nums, int target) {
    Arrays.sort(nums);
    int left = 0, right = nums.length - 1;
    while (left < right) {
        int sum = nums[left] + nums[right];
        if (sum == target) {
            return new int[]{left, right};
        } else if (sum < target) {
            left++;
        } else {
            right--;
        }
    }
    throw new IllegalArgumentException("No two sum solution");
}
```

Explanation:
- First, we sort the array.
- We use two pointers, `left` and `right`, initialized to the start and end of the array.
- We check the sum of the elements at these pointers:
  - If the sum equals the target, we return the indices.
  - If the sum is less than the target, we move the `left` pointer to the right.
  - If the sum is greater than the target, we move the `right` pointer to the left.

Time Complexity: O(n log n)
- Sorting takes O(n log n), and the two-pointer traversal takes O(n).

Space Complexity: O(1)
- No additional space is used (ignoring the space required for sorting).

Which Approach to Use?
- If you need to return the original indices, use the One-Pass Hash Table approach.
- If you only need the values and not the indices, the Sorting with Two Pointers approach is efficient.
- The Brute Force approach is simple but inefficient for large datasets.

Let me know if you need further clarification!

Best Time to Buy and Sell Stock I

The "Best Time to Buy and Sell Stock I" problem is a classic algorithmic problem where you are given an array `prices` where `prices[i]` is the price of a given stock on the `i-th` day. The goal is to find the maximum profit you can achieve by buying on one day and selling on a later day. If no profit can be achieved, return `0`.

Here are different approaches to solve this problem in Java:

1. Brute Force Approach
This approach involves checking all possible pairs of buy and sell days to find the maximum profit.

```java
public int maxProfit(int[] prices) {
    int maxProfit = 0;
    for (int i = 0; i < prices.length; i++) {
        for (int j = i + 1; j < prices.length; j++) {
            int profit = prices[j] - prices[i];
            if (profit > maxProfit) {
                maxProfit = profit;
            }
        }
    }
    return maxProfit;
}
```

Explanation:
- Use two nested loops to compare every possible pair of buy and sell days.
- Calculate the profit for each pair and keep track of the maximum profit.
- Time Complexity: O(n^2) (inefficient for large datasets).

2. One Pass (Optimal Approach)
This approach involves iterating through the array once, keeping track of the minimum price and the maximum profit.

```
public int maxProfit(int[] prices) {
    int minPrice = Integer.MAX_VALUE;
    int maxProfit = 0;
    for (int price : prices) {
        if (price < minPrice) {
            minPrice = price; // Update the minimum price
        } else if (price - minPrice > maxProfit) {
            maxProfit = price - minPrice; // Update the maximum profit
        }
    }
    return maxProfit;
}
```

Explanation:
- Traverse the array while keeping track of the minimum price encountered so far (`minPrice`).
- Calculate the profit if the stock is sold on the current day (`price - minPrice`).
- Update `maxProfit` if the current profit is greater than the previously recorded maximum profit.
- Time Complexity: O(n) (efficient).

3. Dynamic Programming (Kadane's Algorithm)
This approach uses a modified version of Kadane's Algorithm to solve the problem.

```
public int maxProfit(int[] prices) {
    int maxCur = 0, maxSoFar = 0;
    for (int i = 1; i < prices.length; i++) {
        maxCur = Math.max(0, maxCur + prices[i] - prices[i - 1]);
        maxSoFar = Math.max(maxSoFar, maxCur);
    }
    return maxSoFar;
}
```

Explanation:

- Treat the problem as finding the maximum subarray sum of the differences between consecutive days (`prices[i] - prices[i-1]`).
- Use Kadane's Algorithm to track the maximum sum of any contiguous subarray.
- Time Complexity: O(n).

4. Divide and Conquer
This approach divides the problem into smaller subproblems and combines the results.

```
public int maxProfit(int[] prices) {
    return helper(prices, 0, prices.length - 1);
}

private int helper(int[] prices, int left, int right) {
    if (left >= right) {
        return 0;
    }
    int mid = left + (right - left) / 2;
    int leftProfit = helper(prices, left, mid);
    int rightProfit = helper(prices, mid + 1, right);
    int crossProfit = findMax(prices, mid + 1, right) - findMin(prices,
left, mid);
    return Math.max(Math.max(leftProfit, rightProfit), crossProfit);
}

private int findMin(int[] prices, int left, int right) {
    int min = Integer.MAX_VALUE;
    for (int i = left; i <= right; i++) {
        if (prices[i] < min) {
            min = prices[i];
        }
    }
    return min;
}

private int findMax(int[] prices, int left, int right) {
    int max = Integer.MIN_VALUE;
    for (int i = left; i <= right; i++) {
        if (prices[i] > max) {
            max = prices[i];
        }
    }
    return max;
}
```

Explanation:
- Divide the array into two halves and recursively find the maximum profit in each half.
- Calculate the maximum profit that crosses the midpoint by finding the minimum in the left half and the maximum in the right half.
- Combine the results to get the overall maximum profit.
- Time Complexity: O(n log n).

5. Using Auxiliary Arrays

This approach uses two auxiliary arrays to store the minimum prices and maximum profits up to each day.

```
public int maxProfit(int[] prices) {
    if (prices.length == 0) return 0;
    int[] minPrices = new int[prices.length];
    int[] maxProfits = new int[prices.length];
    minPrices[0] = prices[0];
    maxProfits[0] = 0;
    for (int i = 1; i < prices.length; i++) {
        minPrices[i] = Math.min(minPrices[i - 1], prices[i]);
        maxProfits[i] = Math.max(maxProfits[i - 1], prices[i] -
minPrices[i]);
    }
    return maxProfits[prices.length - 1];
}
```

Explanation:
- Use `minPrices` to store the minimum price up to each day.
- Use `maxProfits` to store the maximum profit up to each day.
- Time Complexity: O(n).

Summary of Approaches

Approach	Time Complexity	Space Complexity	Description
Brute Force	O(n²)	O(1)	Check all pairs of buy and sell days.
One Pass (Optimal)	O(n)	O(1)	Track minimum price and maximum profit.
Dynamic Programming	O(n)	O(1)	Use Kadane's Algorithm for maximum subarray.
Divide and Conquer	O(n log n)	O(log n)	Recursively divide and combine results.
Auxiliary Arrays	O(n)	O(n)	Use arrays to store min prices and profits.

The One Pass (Optimal) approach is the most efficient and widely used solution for this problem.

Best Time to Buy and Sell Stock II

The "Best Time to Buy and Sell Stock II" problem is a popular coding question where you are given an array `prices` where `prices[i]` is the price of a given stock on the `i-th` day. The goal is to maximize your profit by buying and selling the stock multiple times. You can complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times), but you must sell the stock before you buy again.

Here are three different approaches to solve this problem in Java:

Approach 1: Greedy Algorithm
The greedy approach involves buying and selling the stock whenever there is a profit opportunity. If the price on the next day is higher than the current day, we buy today and sell tomorrow.

Explanation:
1. Iterate through the `prices` array.
2. If the price on the next day (`prices[i+1]`) is greater than the current day (`prices[i]`), add the difference to the profit.
3. Repeat until the end of the array.

```java
public int maxProfit(int[] prices) {
    int profit = 0;
    for (int i = 0; i < prices.length - 1; i++) {
        if (prices[i + 1] > prices[i]) {
            profit += prices[i + 1] - prices[i];
        }
    }
    return profit;
}
```

Time Complexity:
- O(n), where `n` is the length of the `prices` array.

Space Complexity:
- O(1), as no extra space is used.

Approach 2: Peak-Valley Approach

This approach identifies the peaks (highest prices) and valleys (lowest prices) in the stock prices. We buy at valleys and sell at peaks to maximize profit.

Explanation:
1. Initialize `profit` to 0.
2. Iterate through the `prices` array.
3. Find the lowest price (valley) and the highest price (peak) after it.
4. Add the difference between the peak and valley to the profit.
5. Repeat until the end of the array.

```
public int maxProfit(int[] prices) {
    int profit = 0;
    int i = 0;
    while (i < prices.length - 1) {
        // Find the valley (lowest price)
        while (i < prices.length - 1 && prices[i] >= prices[i + 1]) {
            i++;
        }
        int valley = prices[i];

        // Find the peak (highest price after the valley)
        while (i < prices.length - 1 && prices[i] <= prices[i + 1]) {
            i++;
        }
        int peak = prices[i];

        // Add the profit
        profit += peak - valley;
    }
    return profit;
}
```

Time Complexity:
- O(n), where `n` is the length of the `prices` array.

Space Complexity:
- O(1), as no extra space is used.

Approach 3: Dynamic Programming
This approach uses dynamic programming to keep track of the maximum profit at each step. We maintain two states:
1. `cash`: The maximum profit if we do not hold any stock.
2. `hold`: The maximum profit if we hold a stock.

Explanation:
1. Initialize `cash` to 0 (no profit initially) and `hold` to `Integer.MIN_VALUE` (impossible to hold a stock initially).
2. Iterate through the `prices` array.
3. Update `cash` to be the maximum of the current `cash` or the profit from selling the stock (`hold + prices[i]`).
4. Update `hold` to be the maximum of the current `hold` or the profit from buying the stock (`cash - prices[i]`).
5. Return `cash` at the end.

```java
public int maxProfit(int[] prices) {
    int cash = 0; // Profit if we do not hold any stock
    int hold = Integer.MIN_VALUE; // Profit if we hold a stock

    for (int price : prices) {
        int prevCash = cash;
        cash = Math.max(cash, hold + price); // Sell the stock
        hold = Math.max(hold, prevCash - price); // Buy the stock
    }

    return cash;
}
```

Time Complexity:
- O(n), where `n` is the length of the `prices` array.

Space Complexity:
- O(1), as no extra space is used.

Comparison of Approaches:
1. Greedy Algorithm:
   - Simplest and most intuitive.
   - Works well for this problem because we can make multiple transactions.

2. Peak-Valley Approach:
   - More explicit in identifying buying and selling points.
   - Slightly more complex than the greedy approach.

3. Dynamic Programming:
   - More general and can be extended to more complex problems (e.g., with transaction limits or cooldown periods).
   - Slightly harder to understand but very powerful.

For this specific problem, the Greedy Algorithm is the most efficient and easiest to implement. However, the other approaches are useful for understanding different problem-solving techniques.

Best Time to Buy and Sell Stock III

The "Best Time to Buy and Sell Stock III" problem is a popular coding interview question. The problem statement is as follows:

Problem Statement:
You are given an array `prices` where `prices[i]` is the price of a given stock on the `i-th` day. You are allowed to complete at most two transactions (i.e., buy one and sell one share of the stock twice). Find the maximum profit you can achieve.

Constraints:
- You cannot engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).
- You may not buy and sell on the same day.

Approaches to Solve the Problem

There are multiple ways to solve this problem. Below are two common approaches:

1. Dynamic Programming Approach

This approach uses dynamic programming to keep track of the maximum profit at each step. We break the problem into smaller subproblems and solve them iteratively.

Explanation:
1. We define four states:
   - `buy1`: Maximum profit after the first buy.
   - `sell1`: Maximum profit after the first sell.
   - `buy2`: Maximum profit after the second buy.
   - `sell2`: Maximum profit after the second sell.

2. Initialize:
   - `buy1` and `buy2` to `-prices[0]` because we are spending money to buy the stock.
   - `sell1` and `sell2` to `0` because no profit has been made yet.

3. Iterate through the prices array and update the states:
   - `buy1`: Maximum of the previous `buy1` or buying the stock on the current day.
   - `sell1`: Maximum of the previous `sell1` or selling the stock on the current day.

  - `buy2`: Maximum of the previous `buy2` or buying the stock on the current day using the profit from `sell1`.
  - `sell2`: Maximum of the previous `sell2` or selling the stock on the current day using the profit from `buy2`.

4. The final answer will be `sell2`, which represents the maximum profit after at most two transactions.

```
public int maxProfit(int[] prices) {
    if (prices == null || prices.length == 0) {
        return 0;
    }

    int buy1 = -prices[0], sell1 = 0;
    int buy2 = -prices[0], sell2 = 0;

    for (int i = 1; i < prices.length; i++) {
        buy1 = Math.max(buy1, -prices[i]); // First buy
        sell1 = Math.max(sell1, buy1 + prices[i]); // First sell
        buy2 = Math.max(buy2, sell1 - prices[i]); // Second buy
        sell2 = Math.max(sell2, buy2 + prices[i]); // Second sell
    }

    return sell2;
}
```

Time Complexity: O(n)
- We iterate through the array once.

Space Complexity: O(1)
- We use a constant amount of extra space.

2. Divide and Conquer Approach

This approach divides the problem into two parts:
1. Find the maximum profit for the first transaction (from day 0 to day i).
2. Find the maximum profit for the second transaction (from day i+1 to day n-1).

Explanation:
1. Create two arrays:
  - `leftProfit`: Stores the maximum profit for the first transaction from day 0 to day i.
  - `rightProfit`: Stores the maximum profit for the second transaction from day i to day n-1.

2. Fill the `leftProfit` array:
   - Track the minimum price so far and calculate the profit for each day.

3. Fill the `rightProfit` array:
   - Track the maximum price so far and calculate the profit for each day.

4. Combine the results:
   - The maximum profit is the sum of `leftProfit[i]` and `rightProfit[i]` for each day.

```java
public int maxProfit(int[] prices) {
    if (prices == null || prices.length == 0) {
        return 0;
    }

    int n = prices.length;

    // Left profit array
    int[] leftProfit = new int[n];
    int minPrice = prices[0];
    for (int i = 1; i < n; i++) {
        minPrice = Math.min(minPrice, prices[i]);
        leftProfit[i] = Math.max(leftProfit[i - 1], prices[i] - minPrice);
    }

    // Right profit array
    int[] rightProfit = new int[n];
    int maxPrice = prices[n - 1];
    for (int i = n - 2; i >= 0; i--) {
        maxPrice = Math.max(maxPrice, prices[i]);
        rightProfit[i] = Math.max(rightProfit[i + 1], maxPrice -
prices[i]);
    }

    // Combine results
    int maxProfit = 0;
    for (int i = 0; i < n; i++) {
        maxProfit = Math.max(maxProfit, leftProfit[i] + rightProfit[i]);
    }

    return maxProfit;
}
```

Time Complexity: O(n)
- We iterate through the array three times.

Space Complexity: O(n)
- We use two additional arrays of size `n`.

Comparison of Approaches:
1. Dynamic Programming Approach:
   - More efficient in terms of space complexity (O(1)).
   - Easier to implement and understand.

2. Divide and Conquer Approach:
   - Uses additional space (O(n)).
   - Provides a clear breakdown of profits for each transaction.

Example:

Input:

int[] 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.
- Buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 3.
- Total profit = 3 + 3 = 6.

Both approaches are valid, but the dynamic programming approach is generally preferred due to its efficiency.

Best Time to Buy and Sell Stock IV

The "Best Time to Buy and Sell Stock IV" problem is a classic dynamic programming problem. The goal is to maximize the profit by performing at most `k` transactions (buying and selling is considered one transaction). Below, I'll explain two approaches to solve this problem in Java: Dynamic Programming and Optimized Dynamic Programming.

Problem Statement
You are given an array `prices` where `prices[i]` is the price of a given stock on the `i-th` day. You are also given an integer `k`, which represents the maximum number of transactions you can perform. Find the maximum profit you can achieve.

Approach 1: Dynamic Programming (2D DP Table)

Explanation
1. We use a 2D DP table `dp[i][j]` where:
   - `i` represents the number of transactions (`0` to `k`).
   - `j` represents the day (`0` to `n-1`).
   - `dp[i][j]` stores the maximum profit achievable with at most `i` transactions up to day `j`.

2. The recurrence relation is:
   - If no transaction is performed on day `j`, then `dp[i][j] = dp[i][j-1]`.
   - If a transaction is performed on day `j`, then we need to find the best day `m` to buy (`0 <= m < j`) and sell on day `j`. The profit is `prices[j] - prices[m] + dp[i-1][m-1]`.

3. The final answer is `dp[k][n-1]`.

```java
public int maxProfit(int k, int[] prices) {
    int n = prices.length;
    if (n == 0 || k == 0) return 0;

    // Create a DP table
    int[][] dp = new int[k + 1][n];

    // Fill the DP table
    for (int i = 1; i <= k; i++) {
        int maxDiff = -prices[0]; // Initialize maxDiff for the first day
        for (int j = 1; j < n; j++) {
            dp[i][j] = Math.max(dp[i][j - 1], prices[j] + maxDiff);
            maxDiff = Math.max(maxDiff, dp[i - 1][j - 1] - prices[j]);
        }
    }

    return dp[k][n - 1];
}
```

Time Complexity
- Time: `O(k * n)`, where `k` is the number of transactions and `n` is the number of days.
- Space: `O(k * n)` for the DP table.

Approach 2: Optimized Dynamic Programming (1D DP Table)

Explanation
1. We optimize the space complexity by using a 1D DP array instead of a 2D table.
2. We maintain two arrays:

  - `dp`: Represents the maximum profit with at most `i` transactions up to day `j`.
  - `maxDiff`: Represents the maximum value of `dp[i-1][m-1] - prices[m]` for all `m < j`.

3. The recurrence relation is updated to:
  - `dp[j] = Math.max(dp[j - 1], prices[j] + maxDiff)`.
  - `maxDiff = Math.max(maxDiff, dp[j - 1] - prices[j])`.

```java
public int maxProfit(int k, int[] prices) {
    int n = prices.length;
    if (n == 0 || k == 0) return 0;

    // If k >= n/2, we can perform as many transactions as we want
    if (k >= n / 2) {
        int maxProfit = 0;
        for (int i = 1; i < n; i++) {
            if (prices[i] > prices[i - 1]) {
                maxProfit += prices[i] - prices[i - 1];
            }
        }
        return maxProfit;
    }

    // Create a 1D DP array
    int[] dp = new int[n];

    // Perform k transactions
    for (int i = 1; i <= k; i++) {
        int maxDiff = -prices[0];
        int[] newDp = new int[n];
        for (int j = 1; j < n; j++) {
            newDp[j] = Math.max(newDp[j - 1], prices[j] + maxDiff);
            maxDiff = Math.max(maxDiff, dp[j - 1] - prices[j]);
        }
        dp = newDp;
    }

    return dp[n - 1];
}
```

Time Complexity
- Time: `O(k * n)`.
- Space: `O(n)` for the 1D DP array.

Approach 3: Greedy (When k >= n/2)

Explanation
1. If `k >= n/2`, it means we can perform as many transactions as we want. In this case, the problem reduces to finding all profitable transactions (buy low, sell high).
2. We iterate through the prices and add the profit whenever the price increases.

```java
public int maxProfit(int k, int[] prices) {
    int n = prices.length;
    if (n == 0 || k == 0) return 0;

    // If k >= n/2, perform as many transactions as possible
    if (k >= n / 2) {
        int maxProfit = 0;
        for (int i = 1; i < n; i++) {
            if (prices[i] > prices[i - 1]) {
                maxProfit += prices[i] - prices[i - 1];
            }
        }
        return maxProfit;
    }

    // Otherwise, use the DP approach
    return maxProfitDP(k, prices);
}

private int maxProfitDP(int k, int[] prices) {
    int n = prices.length;
    int[] dp = new int[n];
    for (int i = 1; i <= k; i++) {
        int maxDiff = -prices[0];
        int[] newDp = new int[n];
        for (int j = 1; j < n; j++) {
            newDp[j] = Math.max(newDp[j - 1], prices[j] + maxDiff);
            maxDiff = Math.max(maxDiff, dp[j - 1] - prices[j]);
        }
        dp = newDp;
    }
    return dp[n - 1];
}
```

Time Complexity
- Time: `O(n)` when `k >= n/2`, otherwise `O(k * n)`.
- Space: `O(n)`.

Summary
- Use Dynamic Programming for the general case.
- Optimize space complexity using a 1D DP array.
- Use a Greedy approach when `k >= n/2` to simplify the problem.

Let me know if you need further clarification!

Contains Duplicate

The "Contains Duplicate" problem is a common coding challenge where you need to determine if an array contains any duplicate elements. Below are different approaches to solve this problem in Java, along with explanations.

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

Approach 1: Brute Force
- Explanation: Compare each element with every other element in the array to check for duplicates.
- Time Complexity: \(O(n^2)\) (inefficient for large arrays).
- Space Complexity: \(O(1)\) (no extra space used).

```java
public boolean containsDuplicate(int[] nums) {
    for (int i = 0; i < nums.length; i++) {
        for (int j = i + 1; j < nums.length; j++) {
            if (nums[i] == nums[j]) {
                return true;
            }
        }
    }
    return false;
}
```

Approach 2: Sorting
- Explanation: Sort the array first. If there are duplicates, they will be adjacent to each other. Then, iterate through the sorted array to check for adjacent duplicates.
- Time Complexity: \(O(n \log n)\) (due to sorting).
- Space Complexity: \(O(1)\) (if sorting is done in-place).

```java
public boolean containsDuplicate(int[] nums) {
    Arrays.sort(nums); // Sort the array
    for (int i = 0; i < nums.length - 1; i++) {
        if (nums[i] == nums[i + 1]) {
            return true;
        }
    }
    return false;
}
```

Approach 3: HashSet

- Explanation: Use a `HashSet` to store unique elements. If an element is already in the set, it means there is a duplicate.
- Time Complexity: \(O(n)\) (average case for HashSet operations).
- Space Complexity: \(O(n)\) (extra space for the HashSet).

```java
public boolean containsDuplicate(int[] nums) {
    Set<Integer> set = new HashSet<>();
    for (int num : nums) {
        if (set.contains(num)) {
            return true;
        }
        set.add(num);
    }
    return false;
}
```

Approach 4: HashMap (Frequency Count)

- Explanation: Use a `HashMap` to count the frequency of each element. If any element's frequency exceeds 1, there is a duplicate.
- Time Complexity: \(O(n)\).
- Space Complexity: \(O(n)\).

```java
public boolean containsDuplicate(int[] nums) {
    Map<Integer, Integer> frequencyMap = new HashMap<>();
    for (int num : nums) {
        frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        if (frequencyMap.get(num) > 1) {
            return true;
        }
    }
    return false;
}
```

Approach 5: Stream API (Java 8+)
- Explanation: Use Java's Stream API to convert the array into a set and compare its size with the original array's length.
- Time Complexity: \(O(n)\).
- Space Complexity: \(O(n)\).

```
public boolean containsDuplicate(int[] nums) {
    return Arrays.stream(nums).distinct().count() < nums.length;
}
```

Approach 6: Bit Manipulation (Limited Use Case)
- Explanation: Use bit manipulation to track seen elements. This approach is limited to small ranges of integers and is not practical for general cases.
- Time Complexity: \(O(n)\).
- Space Complexity: \(O(1)\).

```
public boolean containsDuplicate(int[] nums) {
    int checker = 0;
    for (int num : nums) {
        int mask = 1 << num;
        if ((checker & mask) > 0) {
            return true;
        }
        checker |= mask;
    }
    return false;
}
```

Summary
- Brute Force: Simple but inefficient for large arrays.
- Sorting: Efficient and uses constant space.
- HashSet: Most efficient in terms of time complexity.
- HashMap: Useful if you need frequency counts.
- Stream API: Concise and modern but may have overhead.
- Bit Manipulation: Limited use case, not practical for general problems.

For most cases, the HashSet approach is the best choice due to its optimal time complexity and simplicity.

Product of Array Except Self

The "Product of Array Except Self" problem is a common coding interview question. The goal is to return an array where each element at index `i` is equal to the product of all the elements in the original array except the element at index `i`. Here are two approaches to solve this problem in Java:

Approach 1: Using Division (Not Recommended for Edge Cases)
This approach calculates the product of all elements in the array and then divides it by the current element to get the result. However, this approach fails if the array contains a `0` or if the product overflows.

Steps:
1. Calculate the total product of all elements in the array.
2. For each element, divide the total product by the current element to get the result.

```java
public int[] productExceptSelf(int[] nums) {
    int n = nums.length;
    int[] result = new int[n];
    int totalProduct = 1;

    // Calculate the total product of all elements
    for (int num : nums) {
        totalProduct *= num;
    }

    // Fill the result array
    for (int i = 0; i < n; i++) {
        result[i] = totalProduct / nums[i];
    }

    return result;
}
```

Limitations:
- Fails if the array contains a `0` (division by zero).
- May cause integer overflow if the product is too large.

Approach 2: Using Prefix and Suffix Products (Optimal)
This approach avoids division and handles edge cases like zeros. It uses two auxiliary arrays:
1. `prefix`: Stores the product of all elements to the left of the current index.
2. `suffix`: Stores the product of all elements to the right of the current index.

The result for each index is the product of the corresponding `prefix` and `suffix`.

Steps:
1. Initialize `prefix` and `suffix` arrays.
2. Fill the `prefix` array by multiplying elements from the left.
3. Fill the `suffix` array by multiplying elements from the right.
4. Multiply `prefix` and `suffix` to get the result.

```java
public int[] productExceptSelf(int[] nums) {
    int n = nums.length;
    int[] result = new int[n];

    // Initialize prefix and suffix arrays
    int[] prefix = new int[n];
    int[] suffix = new int[n];

    // Fill prefix array
    prefix[0] = 1;
    for (int i = 1; i < n; i++) {
        prefix[i] = prefix[i - 1] * nums[i - 1];
    }

    // Fill suffix array
    suffix[n - 1] = 1;
    for (int i = n - 2; i >= 0; i--) {
        suffix[i] = suffix[i + 1] * nums[i + 1];
    }

    // Calculate result
    for (int i = 0; i < n; i++) {
        result[i] = prefix[i] * suffix[i];
    }

    return result;
}
```

Time Complexity: O(n)
- We traverse the array three times: once for `prefix`, once for `suffix`, and once for the result.
- Space Complexity: O(n) (due to `prefix` and `suffix` arrays).

Approach 3: Space-Optimized (Using Single Output Array)
This approach optimizes space by using the output array to store the `prefix` and then calculating the `suffix` on the fly.

Steps:
1. Initialize the result array with `1`.
2. Traverse the array from left to right, storing the `prefix` product in the result array.
3. Traverse the array from right to left, multiplying the `suffix` product with the `prefix` stored in the result array.

```java
public int[] productExceptSelf(int[] nums) {
    int n = nums.length;
    int[] result = new int[n];

    // Fill result array with prefix products
    result[0] = 1;
    for (int i = 1; i < n; i++) {
        result[i] = result[i - 1] * nums[i - 1];
    }

    // Calculate suffix products and multiply with prefix
    int suffix = 1;
    for (int i = n - 1; i >= 0; i--) {
        result[i] *= suffix;
        suffix *= nums[i];
    }

    return result;
}
```

Time Complexity: O(n)
- We traverse the array twice: once for `prefix` and once for `suffix`.
- Space Complexity: O(1) (excluding the output array).

Explanation of Approach 3:
1. Prefix Calculation:
  - Start with `result[0] = 1` because there are no elements to the left of the first element.
  - For each index `i`, `result[i]` stores the product of all elements to the left of `i`.

2. Suffix Calculation:
  - Use a variable `suffix` to store the product of all elements to the right of the current index.
  - Multiply `result[i]` by `suffix` to get the final result.
  - Update `suffix` by multiplying it with `nums[i]`.

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

1. Prefix Calculation:
   - `result = [1, 1, 2, 6]`

2. Suffix Calculation:
   - Start with `suffix = 1`.
   - At index `3`: `result[3] = 6 * 1 = 6`, `suffix = 1 * 4 = 4`.
   - At index `2`: `result[2] = 2 * 4 = 8`, `suffix = 4 * 3 = 12`.
   - At index `1`: `result[1] = 1 * 12 = 12`, `suffix = 12 * 2 = 24`.
   - At index `0`: `result[0] = 1 * 24 = 24`, `suffix = 24 * 1 = 24`.

Final Output: `[24, 12, 8, 6]`

Conclusion:
- Approach 1 is simple but not robust due to division and edge cases.
- Approach 2 is optimal and handles all cases but uses extra space.
- Approach 3 is the most efficient, using constant space (excluding the output array). It is the recommended solution for interviews.

Maximum Subarray

The "Maximum Subarray" problem is a classic algorithmic problem where the goal is to find the contiguous subarray within a one-dimensional array of numbers that has the largest sum. Below are different approaches to solve this problem in Java, along with explanations.

1. Brute Force Approach
This approach checks all possible subarrays and calculates their sums to find the maximum.

```java
public class MaximumSubarray {
    public static int maxSubArray(int[] nums) {
        int maxSum = Integer.MIN_VALUE;
        for (int i = 0; i < nums.length; i++) {
            int currentSum = 0;
            for (int j = i; j < nums.length; j++) {
                currentSum += nums[j];
                if (currentSum > maxSum) {
                    maxSum = currentSum;
                }
            }
        }
        return maxSum;
    }
    public static void main(String[] args) {
        int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        System.out.println("Maximum Subarray Sum: " + maxSubArray(nums));
    }
}
```

Explanation:
- Use two nested loops to generate all possible subarrays.
- The outer loop fixes the starting point of the subarray.
- The inner loop calculates the sum of the subarray starting from the fixed point.
- Track the maximum sum found.

Time Complexity: O(n^2)
Space Complexity: O(1)

2. Kadane's Algorithm (Dynamic Programming)
This is an optimized approach that solves the problem in linear time.

```java
public class MaximumSubarray {
    public static int maxSubArray(int[] nums) {
        int maxSum = nums[0];
        int currentSum = nums[0];
        for (int i = 1; i < nums.length; i++) {
            currentSum = Math.max(nums[i], currentSum + nums[i]);
            maxSum = Math.max(maxSum, currentSum);
        }
        return maxSum;
    }

    public static void main(String[] args) {
        int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        System.out.println("Maximum Subarray Sum: " + maxSubArray(nums));
    }
}
```

Explanation:
- Iterate through the array while maintaining two variables:
  - `currentSum`: The sum of the current subarray.
  - `maxSum`: The maximum sum found so far.
- At each step, decide whether to start a new subarray or continue the existing one by comparing the current element with the sum of the current subarray.

- Update `maxSum` if the `currentSum` is greater.

Time Complexity: O(n)
Space Complexity: O(1)

3. Divide and Conquer Approach
This approach divides the array into smaller subarrays, solves them recursively, and combines the results.

```
public class MaximumSubarray {
    public static int maxSubArray(int[] nums) {
        return divideAndConquer(nums, 0, nums.length - 1);
    }

    private static int divideAndConquer(int[] nums, int left, int right) {
        if (left == right) {
            return nums[left];
        }
        int mid = left + (right - left) / 2;
        int leftMax = divideAndConquer(nums, left, mid);
        int rightMax = divideAndConquer(nums, mid + 1, right);
        int crossMax = maxCrossingSum(nums, left, mid, right);
        return Math.max(Math.max(leftMax, rightMax), crossMax);
    }

    private static int maxCrossingSum(int[] nums, int left, int mid, int
right) {
        int leftSum = Integer.MIN_VALUE;
        int sum = 0;
        for (int i = mid; i >= left; i--) {
            sum += nums[i];
            if (sum > leftSum) {
                leftSum = sum;
            }
        }
        int rightSum = Integer.MIN_VALUE;
        sum = 0;
        for (int i = mid + 1; i <= right; i++) {
            sum += nums[i];
            if (sum > rightSum) {
                rightSum = sum;
            }
        }
        return leftSum + rightSum;
    }

    public static void main(String[] args) {
        int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        System.out.println("Maximum Subarray Sum: " + maxSubArray(nums));
    }
}
```

Explanation:
- Divide the array into two halves.
- Recursively find the maximum subarray sum in the left and right halves.
- Find the maximum subarray sum that crosses the midpoint.
- Combine the results to get the overall maximum.

Time Complexity: O(n log n)
Space Complexity: O(log n) (due to recursion stack)

4. Dynamic Programming with Tabulation
This approach uses a dynamic programming table to store intermediate results.

```
public class MaximumSubarray {
    public static int maxSubArray(int[] nums) {
        int n = nums.length;
        int[] dp = new int[n];
        dp[0] = nums[0];
        int maxSum = dp[0];
        for (int i = 1; i < n; i++) {
            dp[i] = Math.max(nums[i], dp[i - 1] + nums[i]);
            maxSum = Math.max(maxSum, dp[i]);
        }
        return maxSum;
    }

    public static void main(String[] args) {
        int[] nums = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
        System.out.println("Maximum Subarray Sum: " + maxSubArray(nums));
    }
}
```

Explanation:
- Use a `dp` array where `dp[i]` represents the maximum subarray sum ending at index `i`.

- At each step, decide whether to start a new subarray or continue the previous one.
- Track the maximum value in the `dp` array.

Time Complexity: O(n)
Space Complexity: O(n)

Comparison of Approaches

Approach	Time Complexity	Space Complexity	Use Case
Brute Force	O(n²)	O(1)	Small arrays, simple to implement
Kadane's Algorithm	O(n)	O(1)	Optimal for most cases
Divide and Conquer	O(n log n)	O(log n)	Educational, less efficient
Dynamic Programming	O(n)	O(n)	Alternative to Kadane's Algorithm

Kadane's Algorithm is the most efficient and widely used solution for this problem.

Maximum Product Subarray

The "Maximum Product Subarray" problem is a classic algorithmic challenge where you are given an array of integers, and you need to find the contiguous subarray that yields the maximum product. Below, I'll explain two different approaches to solve this problem in Java:

Approach 1: Dynamic Programming (Kadane's Algorithm Variant)

This approach uses dynamic programming to keep track of the maximum and minimum product at each step. The reason for tracking the minimum product is that a negative number can turn a minimum product into a maximum product if multiplied by another negative number.

Steps:
1. Initialize `maxProduct`, `minProduct`, and `result` to the first element of the array.
2. Iterate through the array starting from the second element.
3. At each step, calculate the new maximum and minimum products by considering:
   - The current number itself.
   - The product of the current number and the previous maximum product.
   - The product of the current number and the previous minimum product.
4. Update the `result` with the maximum value encountered so far.

```java
public class MaximumProductSubarray {
    public static int maxProduct(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int maxProduct = nums[0];
        int minProduct = nums[0];
        int result = nums[0];

        for (int i = 1; i < nums.length; i++) {
            int current = nums[i];
            // Swap maxProduct and minProduct if current number is negative
            if (current < 0) {
                int temp = maxProduct;
                maxProduct = minProduct;
                minProduct = temp;
            }

            // Update maxProduct and minProduct
            maxProduct = Math.max(current, maxProduct * current);
            minProduct = Math.min(current, minProduct * current);

            // Update the result
            result = Math.max(result, maxProduct);
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {2, 3, -2, 4};
        System.out.println("Maximum Product Subarray: " +
maxProduct(nums)); // Output: 6
    }
}
```

Explanation:
- We maintain `maxProduct` and `minProduct` to handle cases where negative numbers can flip the minimum product into a maximum product.
- The `result` variable keeps track of the overall maximum product found so far.
- Time Complexity: O(n), where `n` is the length of the array.
- Space Complexity: O(1), as we use only a few variables.

Approach 2: Two-Pass Traversal

This approach involves traversing the array twice: once from the left and once from the right. The idea is to handle cases where the maximum product is obtained by skipping zeros or negative numbers.

Steps:
1. Traverse the array from the left, calculating the product as you go. Reset the product to 1 if you encounter a zero.
2. Traverse the array from the right, calculating the product as you go. Reset the product to 1 if you encounter a zero.
3. Keep track of the maximum product encountered during both traversals.

```java
public class MaximumProductSubarray {
    public static int maxProduct(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }

        int maxProduct = Integer.MIN_VALUE;
        int product = 1;

        // Traverse from left to right
        for (int i = 0; i < nums.length; i++) {
            product *= nums[i];
            maxProduct = Math.max(maxProduct, product);
            if (nums[i] == 0) {
                product = 1; // Reset product if zero is encountered
            }
        }

        // Traverse from right to left
        product = 1;
        for (int i = nums.length - 1; i >= 0; i--) {
            product *= nums[i];
            maxProduct = Math.max(maxProduct, product);
            if (nums[i] == 0) {
                product = 1; // Reset product if zero is encountered
            }
        }

        return maxProduct;
    }

    public static void main(String[] args) {
        int[] nums = {2, 3, -2, 4};
        System.out.println("Maximum Product Subarray: " +
maxProduct(nums)); // Output: 6
    }
}
```

Explanation:
- By traversing the array twice (left to right and right to left), we ensure that we consider all possible subarrays, including those that skip zeros or negative numbers.
- Time Complexity: O(n), as we traverse the array twice.
- Space Complexity: O(1), as we use only a few variables.

Comparison of Approaches:
1. Dynamic Programming Approach:
   - More efficient and concise.
   - Handles all edge cases (negative numbers, zeros) in a single pass.
   - Preferred for most scenarios.

2. Two-Pass Traversal Approach:
   - Easier to understand and implement.
   - Requires two traversals but still efficient.

Both approaches are optimal and solve the problem in linear time. Choose the one that best fits your understanding and requirements.

Find Minimum in Rotated Sorted Array

Problem Statement:
Given a sorted array that has been rotated at some pivot point, find the minimum element in the array. The array may contain duplicates.

Example:
Input: [3, 4, 5, 1, 2]
Output: 1

Approaches:

1. Linear Search (Brute Force)
- Explanation: Iterate through the array and find the minimum element.
- Time Complexity: O(n)
- Space Complexity: O(1)

```java
public int findMinLinear(int[] nums) {
    int min = nums[0];
    for (int i = 1; i < nums.length; i++) {
        if (nums[i] < min) {
            min = nums[i];
        }
    }
    return min;
}
```

2. Binary Search (Optimal for Sorted Rotated Array)
- Explanation: Use binary search to find the pivot point where the array is rotated. The minimum element is at the pivot point.
- Time Complexity: O(log n)
- Space Complexity: O(1)

```java
public int findMinBinarySearch(int[] nums) {
    int left = 0;
    int right = nums.length - 1;

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

        if (nums[mid] > nums[right]) {
            // The minimum is in the right half
            left = mid + 1;
        } else {
            // The minimum is in the left half (including mid)
            right = mid;
        }
    }

    // When left == right, we've found the minimum element
    return nums[left];
}
```

3. Recursive Binary Search
- Explanation: Similar to the iterative binary search, but implemented recursively.
- Time Complexity: O(log n)
- Space Complexity: O(log n) due to recursion stack

```
public int findMinRecursive(int[] nums) {
    return findMinRecursiveHelper(nums, 0, nums.length - 1);
}

private int findMinRecursiveHelper(int[] nums, int left, int right) {
    if (left == right) {
        return nums[left];
    }

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

    if (nums[mid] > nums[right]) {
        return findMinRecursiveHelper(nums, mid + 1, right);
    } else {
        return findMinRecursiveHelper(nums, left, mid);
    }
}
```

4. Handling Duplicate
- Explanation: If the array contains duplicates, the binary search approach may not work in O(log n) time in the worst case. In such cases, you may need to fall back to linear search or modify the binary search to handle duplicates.
- Time Complexity: O(n) in the worst case due to duplicates
- Space Complexity: O(1)

```
public int findMinWithDuplicates(int[] nums) {
    int left = 0;
    int right = nums.length - 1;

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

        if (nums[mid] > nums[right]) {
            left = mid + 1;
        } else if (nums[mid] < nums[right]) {
            right = mid;
        } else {
            // When nums[mid] == nums[right], we can't determine the side,
so we reduce the search space by 1
            right--;
        }
    }
    return nums[left];
}
```

Summary:
- Linear Search: Simple but not optimal for large arrays.
- Binary Search: Optimal for sorted rotated arrays without duplicates.
- Recursive Binary Search: Same as binary search but implemented recursively.
- Handling Duplicates: Requires a modified approach to handle cases where duplicates are present.

The binary search approach is generally the most efficient for this problem, especially when the array does not contain duplicates.

Search in Rotated Sorted Array

Problem Statement:
Given a sorted array that has been rotated at some pivot point, write a function to search for a target value in the array. If the target exists in the array, return its index; otherwise, return `-1`. The array does not contain duplicates.

Example:
Input: `nums = [4,5,6,7,0,1,2]`, `target = 0`
Output: `4` (index of `0` in the array)

Approaches:

1. Linear Search (Brute Force)
- Explanation: Iterate through the array and check if the current element matches the target.
- Time Complexity: `O(n)`
- Space Complexity: `O(1)`

```
public int search(int[] nums, int target) {
    for (int i = 0; i < nums.length; i++) {
        if (nums[i] == target) {
            return i;
        }
    }
    return -1;
}
```

2. Binary Search (Optimal Approach)
- Explanation: Since the array is sorted but rotated, we can still use binary search with some modifications. The key idea is to determine which part of the array (left or right of the mid) is sorted and then decide where to search for the target.
- Steps:
  1. Find the middle element.
  2. Check if the left half is sorted (`nums[left] <= nums[mid]`).
     - If the target lies in the sorted left half, search in the left half.
     - Otherwise, search in the right half.
  3. If the right half is sorted (`nums[mid] <= nums[right]`):
     - If the target lies in the sorted right half, search in the right half.
     - Otherwise, search in the left half.
- Time Complexity: `O(log n)`
- Space Complexity: `O(1)`

```java
public int search(int[] nums, int target) {
    int left = 0, right = nums.length - 1;

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

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

        // Check if the left half is sorted
        if (nums[left] <= nums[mid]) {
            // Check if the target lies in the left half
            if (nums[left] <= target && target < nums[mid]) {
                right = mid - 1; // Search in the left half
            } else {
                left = mid + 1; // Search in the right half
            }
        } else {
            // Check if the target lies in the right half
            if (nums[mid] < target && target <= nums[right]) {
                left = mid + 1; // Search in the right half
            } else {
                right = mid - 1; // Search in the left half
            }
        }
    }

    return -1; // Target not found
}
```

3. Find Pivot + Binary Search
- Explanation: First, find the pivot (the smallest element in the array, which is the point of rotation). Then, perform binary search on the appropriate subarray.
- Steps:
  1. Find the pivot using binary search.
  2. Perform binary search on the left or right subarray based on the pivot.
- Time Complexity: `O(log n)`
- Space Complexity: `O(1)`

```
public int search(int[] nums, int target) {
    int pivot = findPivot(nums);

    // If no pivot, the array is not rotated
    if (pivot == -1) {
        return binarySearch(nums, target, 0, nums.length - 1);
    }

    // Decide which subarray to search
    if (nums[pivot] == target) {
        return pivot;
    } else if (nums[0] <= target) {
        return binarySearch(nums, target, 0, pivot - 1);
    } else {
        return binarySearch(nums, target, pivot + 1, nums.length - 1);
    }
}

private int findPivot(int[] nums) {
    int left = 0, right = nums.length - 1;

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

        if (mid < right && nums[mid] > nums[mid + 1]) {
            return mid + 1; // Pivot found
        } else if (mid > left && nums[mid - 1] > nums[mid]) {
            return mid; // Pivot found
        } else if (nums[left] >= nums[mid]) {
            right = mid - 1; // Search in the left half
        } else {
            left = mid + 1; // Search in the right half
        }
    }

    return -1; // No pivot found
}

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

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

    return -1; // Target not found
}
```

Summary:
- **Linear Search**: Simple but inefficient for large arrays.
- **Binary Search**: Efficient and optimal for this problem.
- **Find Pivot + Binary Search**: Another efficient approach but involves two passes of binary search.

The Binary Search (Optimal Approach) is the most commonly used solution due to its simplicity and efficiency.

---

3 Sum

The "3Sum" problem is a classic algorithmic challenge where you are given an array of integers, and you need to find all unique triplets in the array that sum up to zero (or a specified target). Below are different approaches to solve the 3Sum problem in Java, along with explanations.

1. Brute Force Approach
This is the simplest approach, but it is inefficient with a time complexity of \(O(n^3)\).

Steps:
1. Use three nested loops to iterate through all possible triplets.
2. Check if the sum of the triplet equals zero.
3. Avoid duplicates by sorting the triplet and storing it in a `Set`.

Code:

```
import java.util.*;

public class ThreeSumBruteForce {
    public List<List<Integer>> threeSum(int[] nums) {
        Set<List<Integer>> result = new HashSet<>();
        int n = nums.length;

        for (int i = 0; i < n - 2; i++) {
            for (int j = i + 1; j < n - 1; j++) {
                for (int k = j + 1; k < n; k++) {
                    if (nums[i] + nums[j] + nums[k] == 0) {
                        List<Integer> triplet = Arrays.asList(nums[i],
nums[j], nums[k]);
                        Collections.sort(triplet); // Sort to avoid
duplicates
                        result.add(triplet);
                    }
                }
            }
        }

        return new ArrayList<>(result);
    }

    public static void main(String[] args) {
        ThreeSumBruteForce solution = new ThreeSumBruteForce();
        int[] nums = {-1, 0, 1, 2, -1, -4};
        System.out.println(solution.threeSum(nums));
    }
}
```

Explanation:

- The outer loop iterates through the first element of the triplet.
- The middle loop iterates through the second element.
- The inner loop iterates through the third element.
- If the sum of the triplet is zero, it is added to the result set after sorting to avoid duplicates.

2. Two-Pointer Approach
This is an optimized approach with a time complexity of \(O(n^2)\).

Steps:
1. Sort the array.
2. Use a loop to fix the first element of the triplet.
3. Use two pointers (one at the start and one at the end) to find the other two elements that sum to zero with the fixed element.
4. Skip duplicates to avoid redundant triplets.

Code:

```java
import java.util.*;

public class ThreeSumTwoPointer {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums); // Sort the array
        List<List<Integer>> result = new ArrayList<>();
        int n = nums.length;

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

            int left = i + 1, right = n - 1;
            while (left < right) {
                int sum = nums[i] + nums[left] + nums[right];
                if (sum == 0) {
                    result.add(Arrays.asList(nums[i], nums[left], 
nums[right]));
                    while (left < right && nums[left] == nums[left + 1]) 
left++; // Skip duplicates
                    while (left < right && nums[right] == nums[right - 1]) 
right--; // Skip duplicates
                    left++;
                    right--;
                } else if (sum < 0) {
                    left++;
                } else {
                    right--;
                }
            }
        }

        return result;
    }

    public static void main(String[] args) {
        ThreeSumTwoPointer solution = new ThreeSumTwoPointer();
        int[] nums = {-1, 0, 1, 2, -1, -4};
        System.out.println(solution.threeSum(nums));
    }
}
```

Explanation:
- The array is sorted to make it easier to find triplets.
- The outer loop fixes the first element of the triplet.
- The two-pointer technique is used to find the other two elements.
- Duplicates are skipped to ensure unique triplets.

3. Hash Set Approach
This approach uses a hash set to store complements, reducing the time complexity to \(O(n^2)\).

Steps:
1. Sort the array.
2. Use a loop to fix the first element of the triplet.
3. Use a hash set to store the second element and find the third element as the complement.

Code:

```java
public class ThreeSumHashSet {
    public List<List<Integer>> threeSum(int[] nums) {
        Arrays.sort(nums); // Sort the array
        List<List<Integer>> result = new ArrayList<>();
        int n = nums.length;

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

            Set<Integer> set = new HashSet<>();
            for (int j = i + 1; j < n; j++) {
                int complement = -nums[i] - nums[j];
                if (set.contains(complement)) {
                    result.add(Arrays.asList(nums[i], nums[j],
complement));
                    while (j + 1 < n && nums[j] == nums[j + 1]) j++; //
Skip duplicates
                }
                set.add(nums[j]);
            }
        }

        return result;
    }

    public static void main(String[] args) {
        ThreeSumHashSet solution = new ThreeSumHashSet();
        int[] nums = {-1, 0, 1, 2, -1, -4};
        System.out.println(solution.threeSum(nums));
    }
}
```

<u>Explanation</u>:
- The array is sorted to handle duplicates.
- The outer loop fixes the first element of the triplet.
- A hash set is used to store the second element and find the third element as the complement.
- Duplicates are skipped to ensure unique triplets.

Comparison of Approaches

Approach	Time Complexity	Space Complexity	Pros	Cons
Brute Force	$O(n^3)$	$O(1)$	Simple to implement	Inefficient for large inputs
Two-Pointer	$O(n^2)$	$O(1)$	Efficient and avoids duplicates	Requires sorting
Hash Set	$O(n^2)$	$O(n)$	Efficient and avoids duplicates	Requires extra space for hash set

Conclusion:
- The Two-Pointer Approach is the most efficient and commonly used method for solving the 3Sum problem.
- The Hash Set Approach is also efficient but uses extra space.
- The Brute Force Approach is only suitable for small inputs due to its high time complexity.

---

4 Sum

The "4Sum" problem is an extension of the "3Sum" problem. The goal is to find all unique quadruplets in an array that sum up to a target value. Below are different approaches to solve the 4Sum problem in Java, along with explanations.

Approach 1: Brute Force (Naive)
This approach involves checking all possible quadruplets in the array and checking if their sum equals the target.

```java
import java.util.ArrayList;
import java.util.List;

public class FourSumBruteForce {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> result = new ArrayList<>();
        int n = nums.length;
        for (int i = 0; i < n - 3; i++) {
            for (int j = i + 1; j < n - 2; j++) {
                for (int k = j + 1; k < n - 1; k++) {
                    for (int l = k + 1; l < n; l++) {
                        if (nums[i] + nums[j] + nums[k] + nums[l] ==
target) {
                            List<Integer> quadruplet = new ArrayList<>();
                            quadruplet.add(nums[i]);
                            quadruplet.add(nums[j]);
                            quadruplet.add(nums[k]);
                            quadruplet.add(nums[l]);
                            result.add(quadruplet);
                        }
                    }
                }
            }
        }
        return result;
    }
    public static void main(String[] args) {
        FourSumBruteForce solution = new FourSumBruteForce();
        int[] nums = {1, 0, -1, 0, -2, 2};
        int target = 0;
        System.out.println(solution.fourSum(nums, target));
    }
}
```

Explanation:
- Time Complexity: \(O(n^4)\) because of four nested loops.
- Space Complexity: \(O(1)\) (excluding the space for the result).
- This approach is inefficient for large arrays.

Approach 2: Sorting + Two Pointers
This approach involves sorting the array and using two pointers to find the quadruplets efficiently.

```
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

public class FourSumTwoPointers {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> result = new ArrayList<>();
        int n = nums.length;
        if (n < 4) return result;

        Arrays.sort(nums); // Sort the array

        for (int i = 0; i < n - 3; i++) {
            if (i > 0 && nums[i] == nums[i - 1]) continue; // Skip
duplicates

            for (int j = i + 1; j < n - 2; j++) {
                if (j > i + 1 && nums[j] == nums[j - 1]) continue; // Skip
duplicates

                int left = j + 1, right = n - 1;
                while (left < right) {
                    long sum = (long) nums[i] + nums[j] + nums[left] +
nums[right];
                    if (sum == target) {
                        result.add(Arrays.asList(nums[i], nums[j],
nums[left], nums[right]));
                        while (left < right && nums[left] == nums[left +
1]) left++; // Skip duplicates
                        while (left < right && nums[right] == nums[right -
1]) right--; // Skip duplicates
                        left++;
                        right--;
                    } else if (sum < target) {
                        left++;
                    } else {
                        right--;
                    }
                }
            }
        }
        return result;
    }

    public static void main(String[] args) {
        FourSumTwoPointers solution = new FourSumTwoPointers();
        int[] nums = {1, 0, -1, 0, -2, 2};
        int target = 0;
        System.out.println(solution.fourSum(nums, target));
    }
}
```

Explanation:
- Time Complexity: \(O(n^3)\) due to sorting (\(O(n \log n)\)) and nested loops.
- Space Complexity: \(O(1)\) (excluding the space for the result).
- This approach is efficient and avoids duplicates by skipping over repeated elements.

Approach 3: Using Hash Set
This approach uses a hash set to store pairs of numbers and check for the remaining two numbers.

```
import java.util.*;

public class FourSumHashSet {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> result = new ArrayList<>();
        int n = nums.length;
        if (n < 4) return result;

        Arrays.sort(nums); // Sort the array
        Set<List<Integer>> set = new HashSet<>(); // To avoid duplicates

        for (int i = 0; i < n - 3; i++) {
            for (int j = i + 1; j < n - 2; j++) {
                int left = j + 1, right = n - 1;
                while (left < right) {
                    long sum = (long) nums[i] + nums[j] + nums[left] +
nums[right];
                    if (sum == target) {
                        set.add(Arrays.asList(nums[i], nums[j], nums[left],
nums[right]));
                        left++;
                        right--;
                    } else if (sum < target) {
                        left++;
                    } else {
                        right--;
                    }
                }
            }
        }
        result.addAll(set);
        return result;
    }

    public static void main(String[] args) {
        FourSumHashSet solution = new FourSumHashSet();
        int[] nums = {1, 0, -1, 0, -2, 2};
        int target = 0;
        System.out.println(solution.fourSum(nums, target));
    }
}
```

Explanation:
- Time Complexity: \(O(n^3)\) due to nested loops.
- Space Complexity: \(O(n)\) for the hash set.
- This approach avoids duplicates using a hash set but is less efficient than the two-pointer approach.

Approach 4: Recursive Backtracking

This approach uses recursion to generate all possible quadruplets and checks if they sum to the target.

```
import java.util.*;

public class FourSumBacktracking {
    public List<List<Integer>> fourSum(int[] nums, int target) {
        List<List<Integer>> result = new ArrayList<>();
        Arrays.sort(nums); // Sort the array
        backtrack(result, new ArrayList<>(), nums, target, 0);
        return result;
    }

    private void backtrack(List<List<Integer>> result, List<Integer> temp,
int[] nums, long target, int start) {
        if (temp.size() == 4) {
            if (target == 0) {
                result.add(new ArrayList<>(temp));
            }
            return;
        }

        for (int i = start; i < nums.length; i++) {
            if (i > start && nums[i] == nums[i - 1]) continue; // Skip
duplicates
            temp.add(nums[i]);
            backtrack(result, temp, nums, target - nums[i], i + 1);
            temp.remove(temp.size() - 1);
        }
    }

    public static void main(String[] args) {
        FourSumBacktracking solution = new FourSumBacktracking();
        int[] nums = {1, 0, -1, 0, -2, 2};
        int target = 0;
        System.out.println(solution.fourSum(nums, target));
    }
}
```

Explanation:
- Time Complexity: \(O(n^3)\) due to recursion and backtracking.
- Space Complexity: \(O(n)\) for the recursion stack.
- This approach is flexible but less efficient than the two-pointer approach.

Summary:
- Brute Force: Simple but inefficient (\(O(n^4)\)).
- Sorting + Two Pointers: Efficient and avoids duplicates (\(O(n^3)\)).
- Hash Set: Avoids duplicates but uses extra space (\(O(n^3)\)).
- Backtracking: Flexible but less efficient (\(O(n^3)\)).

The Sorting + Two Pointers approach is the most efficient and commonly used solution for the 4Sum problem.

Container With Most Water

The "Container With Most Water" problem is a classic algorithmic challenge. The problem statement is as follows:

Problem Statement:
Given an array of integers `height` where each element represents the height of a vertical line at position `i`, find two lines that, together with the x-axis, form a container that holds the most water. You cannot slant the container.

Example:

Input: height = [1, 8, 6, 2, 5, 4, 8, 3, 7]
Output: 49
Explanation: The maximum area is formed between the lines at positions 1 and 8 (height[1] = 8 and height[8] = 7). The area is min(8, 7) * (8 - 1) = 7 * 7 = 49.

Approaches to Solve the Problem

1. Brute Force Approach
- Idea: Calculate the area for every possible pair of lines and keep track of the maximum area found.
- Time Complexity: $O(n^2)$
- Space Complexity: O(1)

```
public int maxArea(int[] height) {
        int maxArea = 0;
        for (int i = 0; i < height.length; i++) {
            for (int j = i + 1; j < height.length; j++) {
                int currentArea = Math.min(height[i], height[j]) * (j - i);
                maxArea = Math.max(maxArea, currentArea);
            }
        }
        return maxArea;
    }
```

Explanation:
- We use two nested loops to consider every possible pair of lines.
- For each pair, we calculate the area using the formula `min(height[i], height[j]) * (j - i)`.
- We keep updating `maxArea` if the current area is greater than the previously recorded maximum.

2. Two-Pointer Approach
- Idea: Use two pointers, one at the beginning (`left`) and one at the end (`right`) of the array. Move the pointer pointing to the shorter line towards the center, as moving the longer line would not increase the area.
- Time Complexity: O(n)
- Space Complexity: O(1)

```
   public int maxArea(int[] height) {
        int maxArea = 0;
        int left = 0;
        int right = height.length - 1;

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

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

        return maxArea;
    }
```

Explanation:
- We initialize two pointers, `left` at the start and `right` at the end of the array.
- We calculate the area formed by the lines at `left` and `right`.
- We move the pointer pointing to the shorter line inward because the area is limited by the shorter line, and moving the longer line would not help in finding a larger area.

- We repeat this process until the two pointers meet.

Comparison of Approaches

- Brute Force: Simple to implement but inefficient for large input sizes due to its O(n²) time complexity.
- Two-Pointer: Much more efficient with O(n) time complexity, making it suitable for large datasets.

Conclusion

The Two-Pointer approach is the optimal solution for the "Container With Most Water" problem, providing a significant performance improvement over the brute force method. It efficiently finds the maximum area by intelligently narrowing down the search space.

(Binary)

Sum of Two Integers

The "Sum of Two Integers" problem is a classic problem that involves adding two integers without using the `+` or `-` operators. This problem is often used to test understanding of bitwise operations.

Problem Statement:
Given two integers `a` and `b`, return the sum of the two integers without using the `+` or `-` operators.

Example:

Input: a = 1, b = 2
Output: 3

Approaches to Solve the Problem

1. Iterative Approach Using Bitwise Operations
- Idea: Use bitwise operations to simulate the addition process. The sum of two bits can be obtained using the XOR (`^`) operation, and the carry can be obtained using the AND (`&`) operation followed by a left shift (`<<`).
- Time Complexity: O(n), where `n` is the number of bits in the integers.
- Space Complexity: O(1)

```
    public int getSum(int a, int b) {
        while (b != 0) {
            int carry = a & b;  // Carry is AND of a and b
            a = a ^ b;          // Sum of a and b without considering carry
            b = carry << 1;      // Carry is shifted by 1 so that it can be
added to the next higher bit
        }
        return a;
    }
```

Explanation:
  - We use a loop to continue the process until there is no carry (`b == 0`).
  - The variable `carry` holds the common set bits of `a` and `b`, which are the bits that need to be carried over.
  - The variable `a` is updated to hold the sum of `a` and `b` without considering the carry.
  - The variable `b` is updated to hold the carry, shifted left by 1 bit, so it can be added to the next higher bit in the next iteration.

2. Recursive Approach Using Bitwise Operations
  - Idea: The same logic as the iterative approach, but implemented recursively.
  - Time Complexity: O(n), where `n` is the number of bits in the integers.
  - Space Complexity: O(n) due to the recursion stack.

```
public int getSum(int a, int b) {
        if (b == 0) {
            return a;
        }
        return getSum(a ^ b, (a & b) << 1);
    }
```

Explanation:
  - The base case for the recursion is when `b == 0`, in which case `a` holds the final sum.
  - The recursive case involves computing the sum of `a` and `b` without considering the carry (`a ^ b`) and the carry (`(a & b) << 1`), and then calling the function recursively with these new values.

Comparison of Approaches

- Iterative Approach: More efficient in terms of space complexity since it does not use the recursion stack. It is generally preferred for its simplicity and lower space usage.

- Recursive Approach: Easier to understand and implement but uses additional space due to the recursion stack. It is a good demonstration of how recursion can be used to solve problems involving bitwise operations.

Conclusion

Both approaches effectively solve the problem using bitwise operations, with the iterative approach being more space-efficient. Understanding these methods provides insight into how addition can be performed at the bit level, which is a fundamental concept in computer science.

Number of 1 Bits

The "Number of 1 Bits" problem, also known as the "Hamming Weight" problem, involves counting the number of `1` bits in the binary representation of a given integer.

Problem Statement:
Write a function that takes an unsigned integer and returns the number of `1` bits it has (also known as the Hamming weight).

Example:

Input: n = 11 (Binary: 1011)
Output: 3
Explanation: The binary representation of 11 has three '1' bits.

Approaches to Solve the Problem

1. Iterative Approach Using Bitwise Operations
- Idea: Repeatedly check the least significant bit (LSB) of the number and shift the number to the right until the number becomes zero.
- Time Complexity: O(k), where `k` is the number of bits in the integer (typically 32 for a 32-bit integer).
- Space Complexity: O(1)

```java
public int hammingWeight(int n) {
       int count = 0;
       while (n != 0) {
           count += n & 1;  // Add 1 if LSB is 1
           n = n >>> 1;     // Unsigned right shift
       }
       return count;
   }
```

Explanation:
  - We initialize a counter `count` to zero.
  - We use a loop to check the LSB of `n` by performing a bitwise AND with `1`. If the result is `1`, we increment the counter.
  - We then perform an unsigned right shift (`>>>`) on `n` to discard the LSB and continue the process until `n` becomes zero.
  - The unsigned right shift ensures that the sign bit is not propagated, which is important for handling negative numbers correctly.

2. Optimized Approach Using Brian Kernighan's Algorithm
  - Idea: Repeatedly flip the least significant `1` bit of the number and count how many times this operation is performed until the number becomes zero.
  - Time Complexity: O(m), where `m` is the number of `1` bits in the integer.
  - Space Complexity: O(1)

```
public int hammingWeight(int n) {
      int count = 0;
      while (n != 0) {
          n = n & (n - 1);  // Flip the least significant 1 bit
          count++;
      }
      return count;
  }
```

Explanation:
  - We initialize a counter `count` to zero.
  - We use a loop to repeatedly perform the operation `n = n & (n - 1)`, which flips the least significant `1` bit of `n` to `0`.
  - Each time this operation is performed, we increment the counter.
  - The loop continues until `n` becomes zero, at which point the counter holds the number of `1` bits.

Comparison of Approaches

- Iterative Approach: Simple and straightforward, but it always iterates over all bits of the integer, even if there are no more `1` bits left.
- Brian Kernighan's Algorithm: More efficient as it only iterates over the number of `1` bits, making it faster for numbers with fewer `1` bits.

Conclusion

Both approaches effectively solve the problem, with Brian Kernighan's Algorithm being more efficient for numbers with fewer `1` bits. Understanding these methods provides insight into how bitwise operations can be used to manipulate and analyze binary data efficiently.

Counting Bits

The "Counting Bits" problem involves generating an array where each element at index `i` represents the number of `1` bits in the binary representation of `i`.

Problem Statement:
Given an integer `n`, return an array `ans` of length `n + 1` such that for each `i` (`0 <= i <= n`), `ans[i]` is the number of `1`'s in the binary representation of `i`.

Example:

Input: n = 5
Output: [0, 1, 1, 2, 1, 2]
Explanation:
0 --> 0 (0 ones)
1 --> 1 (1 one)
2 --> 10 (1 one)
3 --> 11 (2 ones)
4 --> 100 (1 one)
5 --> 101 (2 ones)

Approaches to Solve the Problem

1. Naive Approach (Using Hamming Weight for Each Number)
  - Idea: For each number from `0` to `n`, calculate the number of `1` bits using the Hamming Weight method (as in the previous problem).
  - Time Complexity: O(n * k), where `k` is the number of bits in the largest number (typically 32 for a 32-bit integer).
  - Space Complexity: O(1) (excluding the output array)

```
public int[] countBits(int n) {
      int[] ans = new int[n + 1];
      for (int i = 0; i <= n; i++) {
          ans[i] = hammingWeight(i);
      }
      return ans;
  }
```

```
    private int hammingWeight(int n) {
        int count = 0;
        while (n != 0) {
            count += n & 1;  // Add 1 if LSB is 1
            n = n >>> 1;     // Unsigned right shift
        }
        return count;
    }
```

Explanation:
- We initialize an array `ans` of size `n + 1`.
- For each number `i` from `0` to `n`, we calculate the number of `1` bits using the `hammingWeight` method.
- The `hammingWeight` method counts the number of `1` bits by checking the LSB and shifting the number right until it becomes zero.

2. Dynamic Programming with Bit Manipulation
- Idea: Use dynamic programming to build the solution. The number of `1` bits in `i` can be derived from the number of `1` bits in `i / 2` (or `i >> 1`) plus the LSB of `i`.
- Time Complexity: O(n)
- Space Complexity: O(1) (excluding the output array)

```
public int[] countBits(int n) {
        int[] ans = new int[n + 1];
        for (int i = 1; i <= n; i++) {
            ans[i] = ans[i >> 1] + (i & 1);
        }
        return ans;
    }
```

Explanation:
- We initialize an array `ans` of size `n + 1`.
- For each number `i` from `1` to `n`, we calculate the number of `1` bits using the formula `ans[i] = ans[i >> 1] + (i & 1)`.
- `i >> 1` is equivalent to `i / 2`, which gives the number of `1` bits in `i / 2`.
- `i & 1` gives the LSB of `i`, which is `1` if `i` is odd and `0` if `i` is even.
- This approach leverages the previously computed results to build the solution efficiently.

3. Dynamic Programming with Last Set Bit
- Idea: Use dynamic programming to build the solution. The number of `1` bits in `i` can be derived from the number of `1` bits in `i & (i - 1)` plus `1`.
- Time Complexity: O(n)

- Space Complexity: O(1) (excluding the output array)

```
public int[] countBits(int n) {
     int[] ans = new int[n + 1];
     for (int i = 1; i <= n; i++) {
         ans[i] = ans[i & (i - 1)] + 1;
     }
     return ans;
 }
```

Explanation:
- We initialize an array `ans` of size `n + 1`.
- For each number `i` from `1` to `n`, we calculate the number of `1` bits using the formula `ans[i] = ans[i & (i - 1)] + 1`.
- `i & (i - 1)` drops the lowest set bit of `i`, so `ans[i & (i - 1)]` gives the number of `1` bits in `i` without the lowest set bit.
- Adding `1` accounts for the dropped bit.
- This approach also leverages previously computed results to build the solution efficiently.

Comparison of Approaches

- Naive Approach: Simple but inefficient for large `n` due to its O(n * k) time complexity.
- Dynamic Programming with Bit Manipulation: Efficient with O(n) time complexity, leveraging the relationship between a number and its half.
- Dynamic Programming with Last Set Bit: Also efficient with O(n) time complexity, leveraging the relationship between a number and the number with its lowest set bit dropped.

Conclusion

The dynamic programming approaches are the most efficient for solving the "Counting Bits" problem, with both methods providing O(n) time complexity. The choice between the two dynamic programming methods can depend on personal preference or specific use cases, but both are highly effective and widely used.

Missing Number

The "Missing Number" problem involves finding the missing number in a sequence of integers from `0` to `n` where one number is missing.

Problem Statement:
Given an array `nums` containing `n` distinct numbers in the range `[0, n]`, return the only number in the range that is missing from the array.

Example:

Input: nums = [3, 0, 1]
Output: 2
Explanation: The numbers in the range [0, 3] are 0, 1, 2, 3. The missing number is 2.

Approaches to Solve the Problem

1. Summation Approach

- Idea: Calculate the expected sum of numbers from `0` to `n` using the formula `n * (n + 1) / 2`. Subtract the sum of the elements in the array from this expected sum to find the missing number.
- Time Complexity: O(n)
- Space Complexity: O(1)

```
public int missingNumber(int[] nums) {
    int n = nums.length;
    int expectedSum = n * (n + 1) / 2;
    int actualSum = 0;
    for (int num : nums) {
        actualSum += num;
    }
    return expectedSum - actualSum;
}
```

Explanation:

- We calculate the expected sum of numbers from `0` to `n` using the formula `n * (n + 1) / 2`.
- We calculate the actual sum of the elements in the array.
- The difference between the expected sum and the actual sum gives the missing number.

2. Bit Manipulation Approach (XOR)

- Idea: Use the XOR operation to find the missing number. XOR all numbers from `0` to `n` and XOR all numbers in the array. The result will be the missing number.
- Time Complexity: O(n)
- Space Complexity: O(1)

```
public int missingNumber(int[] nums) {
     int missing = nums.length;
     for (int i = 0; i < nums.length; i++) {
         missing ^= i ^ nums[i];
     }
```

```
        return missing;
    }
```

Explanation:
- We initialize `missing` with `n` (the length of the array).
- We iterate through the array and perform XOR operations between `missing`, the index `i`, and the element `nums[i]`.
- The result of these XOR operations will be the missing number because XORing a number with itself results in `0`, and XORing `0` with a number results in the number itself.

3. HashSet Approach
- Idea: Use a HashSet to store all numbers in the array. Then, iterate through the range `[0, n]` and check which number is missing from the HashSet.
- Time Complexity: O(n)
- Space Complexity: O(n)

```
    public int missingNumber(int[] nums) {
        Set<Integer> numSet = new HashSet<>();
        for (int num : nums) {
            numSet.add(num);
        }
        for (int i = 0; i <= nums.length; i++) {
            if (!numSet.contains(i)) {
                return i;
            }
        }
        return -1; // This line will never be reached if input is valid
    }
```

Explanation:
- We create a HashSet and add all elements from the array to it.
- We iterate through the range `[0, n]` and check if each number is present in the HashSet.
- The first number that is not in the HashSet is the missing number.

Comparison of Approaches

- Summation Approach: Efficient with O(n) time complexity and O(1) space complexity. It is simple and easy to implement.
- Bit Manipulation Approach (XOR): Also efficient with O(n) time complexity and O(1) space complexity. It is elegant and avoids potential integer overflow issues that can occur with the summation approach.

- HashSet Approach: Easy to understand but uses O(n) additional space, making it less efficient in terms of space complexity compared to the other two approaches.

Conclusion

The summation and bit manipulation approaches are the most efficient for solving the "Missing Number" problem, with both methods providing O(n) time complexity and O(1) space complexity. The choice between these methods can depend on personal preference or specific use cases, but both are highly effective and widely used. The HashSet approach, while easy to understand, is less efficient in terms of space complexity.

Reverse Bits

The "Reverse Bits" problem involves reversing the bits of a given 32-bit unsigned integer.

Problem Statement:
Reverse the bits of a given 32-bit unsigned integer.

Example:

Input: n = 00000010100101000001111010011100
Output: 964176192 (00111001011110000010100101000000)
Explanation: The input binary string 00000010100101000001111010011100 represents the unsigned integer 43261596, so return 964176192 which its binary representation is 00111001011110000010100101000000.

Approaches to Solve the Problem

1. Bit by Bit Reversal
- Idea: Iterate through the bits of the input number and construct the reversed number bit by bit.
- Time Complexity: O(1) (since the number of bits is fixed at 32)
- Space Complexity: O(1)

```
public int reverseBits(int n) {
    int result = 0;
    for (int i = 0; i < 32; i++) {
        result = (result << 1) | (n & 1);
        n = n >>> 1;
    }
    return result;
}
```

Explanation:
   - We initialize `result` to `0`.
   - We iterate 32 times (since it's a 32-bit integer).
   - In each iteration, we shift `result` left by 1 bit and OR it with the LSB of `n`.
   - We then shift `n` right by 1 bit (using unsigned right shift `>>>` to handle the sign bit correctly).
   - After 32 iterations, `result` will hold the reversed bits of `n`.

2. Divide and Conquer with Bit Masking
   - Idea: Use a divide and conquer approach to reverse the bits. Swap the left and right halves of the bits, then swap the halves of those halves, and so on.
   - Time Complexity: O(1) (since the number of bits is fixed at 32)
   - Space Complexity: O(1)

```
 public int reverseBits(int n) {
        n = (n >>> 16) | (n << 16); // Swap left and right 16 bits
        n = ((n & 0xff00ff00) >>> 8) | ((n & 0x00ff00ff) << 8); // Swap left and right 8 bits within each 16 bits
        n = ((n & 0xf0f0f0f0) >>> 4) | ((n & 0x0f0f0f0f) << 4); // Swap left and right 4 bits within each 8 bits
        n = ((n & 0xcccccccc) >>> 2) | ((n & 0x33333333) << 2); // Swap left and right 2 bits within each 4 bits
        n = ((n & 0xaaaaaaaa) >>> 1) | ((n & 0x55555555) << 1); // Swap left and right 1 bit within each 2 bits
        return n;
    }
```

Explanation:
   - We start by swapping the left and right 16 bits of the number.
   - Then, we swap the left and right 8 bits within each 16-bit half.
   - Next, we swap the left and right 4 bits within each 8-bit segment.
   - Then, we swap the left and right 2 bits within each 4-bit segment.
   - Finally, we swap the left and right 1 bit within each 2-bit segment.
   - This approach effectively reverses the bits of the number.

Comparison of Approaches

- Bit by Bit Reversal: Simple and straightforward, easy to understand and implement. It iterates through each bit individually, making it clear how the reversal is achieved.
- Divide and Conquer with Bit Masking: More efficient in terms of the number of operations, as it reduces the problem size logarithmically. It is a more advanced technique that leverages bit masking and shifting to achieve the result in fewer steps.

Conclusion

Both approaches effectively solve the "Reverse Bits" problem, with the bit by bit reversal being simpler and the divide and conquer approach being more efficient in terms of the number of operations. The choice between these methods can depend on personal preference or specific use cases, but both are highly effective and widely used.

(Dynamic Programming)

Climbing Stairs

The "Climbing Stairs" problem is a classic dynamic programming problem that involves finding the number of distinct ways to climb to the top of a staircase with `n` steps, where you can take either 1 or 2 steps at a time.

Problem Statement:
You are climbing a staircase. It takes `n` steps to reach the top. Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

Example:

Input: n = 3
Output: 3
Explanation: There are three ways to climb to the top.
1. 1 step + 1 step + 1 step
2. 1 step + 2 steps
3. 2 steps + 1 step

Approaches to Solve the Problem

1. Recursive Approach
- Idea: Use recursion to explore all possible ways to climb the stairs. The number of ways to reach the nth step is the sum of the ways to reach the (n-1)th step and the (n-2)th step.
- Time Complexity: O(2^n) (exponential due to repeated calculations)
- Space Complexity: O(n) (due to the recursion stack)

```
public int climbStairs(int n) {
     if (n == 0 || n == 1) {
         return 1;
```

```
        }
        return climbStairs(n - 1) + climbStairs(n - 2);
    }
```

Explanation:
- The base cases are when `n` is `0` or `1`, in which case there is only one way to climb the stairs (by taking no steps or one step, respectively).
- For `n > 1`, the number of ways to climb the stairs is the sum of the ways to climb `n-1` stairs and the ways to climb `n-2` stairs.
- This approach is simple but inefficient due to the exponential time complexity caused by repeated calculations.

2. Dynamic Programming (Memoization)
- Idea: Use memoization to store the results of subproblems to avoid redundant calculations in the recursive approach.
- Time Complexity: O(n)
- Space Complexity: O(n)

```
 public int climbStairs(int n) {
        int[] memo = new int[n + 1];
        return climbStairsMemo(n, memo);
    }

    private int climbStairsMemo(int n, int[] memo) {
        if (n == 0 || n == 1) {
            return 1;
        }
        if (memo[n] > 0) {
            return memo[n];
        }
        memo[n] = climbStairsMemo(n - 1, memo) + climbStairsMemo(n - 2,
memo);
        return memo[n];
    }
```

Explanation:
- We use an array `memo` to store the results of subproblems.
- The base cases are the same as in the recursive approach.
- Before making a recursive call, we check if the result is already in the `memo` array. If it is, we return the stored result.
- If the result is not in the `memo` array, we compute it, store it in the array, and then return it.
- This approach reduces the time complexity to O(n) by avoiding redundant calculations.

3. Dynamic Programming (Tabulation)
- Idea: Use an iterative approach to build up the solution from the base cases.
- Time Complexity: O(n)
- Space Complexity: O(n)

```java
public int climbStairs(int n) {
     if (n == 0 || n == 1) {
         return 1;
     }
     int[] dp = new int[n + 1];
     dp[0] = 1;
     dp[1] = 1;
     for (int i = 2; i <= n; i++) {
         dp[i] = dp[i - 1] + dp[i - 2];
     }
     return dp[n];
 }
```

Explanation:
- We initialize a `dp` array where `dp[i]` represents the number of ways to climb `i` stairs.
- The base cases are `dp[0] = 1` and `dp[1] = 1`.
- We iterate from `2` to `n` and fill the `dp` array using the relation `dp[i] = dp[i - 1] + dp[i - 2]`.
- The final result is `dp[n]`.

4. Space-Optimized Dynamic Programming
- Idea: Since the current value depends only on the previous two values, we can optimize space by using just two variables instead of an entire array.
- Time Complexity: O(n)
- Space Complexity: O(1)

```java
public int climbStairs(int n) {
     if (n == 0 || n == 1) {
         return 1;
     }
     int prev1 = 1, prev2 = 1;
     for (int i = 2; i <= n; i++) {
         int current = prev1 + prev2;
         prev2 = prev1;
         prev1 = current;
     }
     return prev1;
```

```
    }
```

Explanation:
- We use two variables `prev1` and `prev2` to store the number of ways to climb `i-1` and `i-2` stairs, respectively.
- We iterate from `2` to `n` and compute the current value as the sum of `prev1` and `prev2`.
- We then update `prev2` to be the old `prev1` and `prev1` to be the current value.
- The final result is `prev1`.

Comparison of Approaches

- Recursive Approach: Simple but inefficient due to exponential time complexity.
- Dynamic Programming (Memoization): More efficient with O(n) time complexity, but uses O(n) space for the memoization array.
- Dynamic Programming (Tabulation): Efficient with O(n) time complexity and O(n) space, but can be optimized further.
- Space-Optimized Dynamic Programming: Most efficient with O(n) time complexity and O(1) space.

Conclusion

The space-optimized dynamic programming approach is the most efficient for solving the "Climbing Stairs" problem, providing O(n) time complexity and O(1) space complexity. The choice between the methods can depend on personal preference or specific use cases, but the space-optimized approach is generally preferred for its efficiency.

Coin Change

The "Coin Change" problem is a classic dynamic programming problem that involves finding the minimum number of coins required to make up a given amount of money using a given set of coin denominations.

Problem Statement:
You are given an integer array `coins` representing coins of different denominations and an integer `amount` representing a total amount of money. Return the fewest number of coins that you need to make up that amount. If that amount of money cannot be made up by any combination of the coins, return `-1`.

Example:

Input: coins = [1, 2, 5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1

Approaches to Solve the Problem

1. Recursive Approach

- Idea: Use recursion to explore all possible combinations of coins to make up the given amount.
- Time Complexity: O(2^n) (exponential due to repeated calculations)
- Space Complexity: O(n) (due to the recursion stack)

```java
public int coinChange(int[] coins, int amount) {
        return coinChangeHelper(coins, amount);
   }

   private int coinChangeHelper(int[] coins, int amount) {
        if (amount == 0) {
            return 0;
        }
        int minCoins = Integer.MAX_VALUE;
        for (int coin : coins) {
            if (amount - coin >= 0) {
                int result = coinChangeHelper(coins, amount - coin);
                if (result != -1) {
                    minCoins = Math.min(minCoins, result + 1);
                }
            }
        }
        return (minCoins == Integer.MAX_VALUE) ? -1 : minCoins;
   }
```

Explanation:

- The base case is when `amount` is `0`, in which case no coins are needed.
- For each coin, we recursively check if it can be used to make up the remaining amount.
- We keep track of the minimum number of coins required.
- If no combination of coins can make up the amount, we return `-1`.

2. Dynamic Programming (Memoization)

- Idea: Use memoization to store the results of subproblems to avoid redundant calculations in the recursive approach.
- Time Complexity: O(n * m) (where `n` is the amount and `m` is the number of coin denominations)
- Space Complexity: O(n)

```
public int coinChange(int[] coins, int amount) {
        int[] memo = new int[amount + 1];
        return coinChangeMemo(coins, amount, memo);
    }

    private int coinChangeMemo(int[] coins, int amount, int[] memo) {
        if (amount == 0) {
            return 0;
        }
        if (memo[amount] != 0) {
            return memo[amount];
        }
        int minCoins = Integer.MAX_VALUE;
        for (int coin : coins) {
            if (amount - coin >= 0) {
                int result = coinChangeMemo(coins, amount - coin, memo);
                if (result != -1) {
                    minCoins = Math.min(minCoins, result + 1);
                }
            }
        }
        memo[amount] = (minCoins == Integer.MAX_VALUE) ? -1 : minCoins;
        return memo[amount];
    }
```

Explanation:
- We use a `memo` array to store the results of subproblems.
- The base case is the same as in the recursive approach.
- Before making a recursive call, we check if the result is already in the `memo` array. If it is, we return the stored result.
- If the result is not in the `memo` array, we compute it, store it in the array, and then return it.
- This approach reduces the time complexity to O(n * m) by avoiding redundant calculations.

3. Dynamic Programming (Tabulation)
- Idea: Use an iterative approach to build up the solution from the base case.
- Time Complexity: O(n * m)
- Space Complexity: O(n)

```
public int coinChange(int[] coins, int amount) {
        int[] dp = new int[amount + 1];
        Arrays.fill(dp, amount + 1);
        dp[0] = 0;
        for (int i = 1; i <= amount; i++) {
            for (int coin : coins) {
                if (i - coin >= 0) {
                    dp[i] = Math.min(dp[i], dp[i - coin] + 1);
                }
            }
        }
        return (dp[amount] > amount) ? -1 : dp[amount];
    }
```

Explanation:
- We initialize a `dp` array where `dp[i]` represents the minimum number of coins needed to make up the amount `i`.
- We fill the `dp` array with `amount + 1` (a value larger than the maximum possible number of coins) to represent infinity.
- The base case is `dp[0] = 0` (no coins are needed to make up the amount `0`).
- We iterate through each amount from `1` to `amount` and for each coin, we update the `dp` array if using that coin results in a smaller number of coins.
- The final result is `dp[amount]`, or `-1` if `dp[amount]` is still `amount + 1` (meaning it's not possible to make up the amount with the given coins).

Comparison of Approaches

- Recursive Approach: Simple but inefficient due to exponential time complexity.
- Dynamic Programming (Memoization): More efficient with O(n * m) time complexity, but uses O(n) space for the memoization array.
- Dynamic Programming (Tabulation): Efficient with O(n * m) time complexity and O(n) space, and is often easier to implement iteratively.

Conclusion

The dynamic programming tabulation approach is the most efficient for solving the "Coin Change" problem, providing O(n * m) time complexity and O(n) space complexity. The choice between the methods can depend on personal preference or specific use cases, but the tabulation approach is generally preferred for its efficiency and simplicity.

Another Approach
The "Coin Change" problem is a classic dynamic programming problem where you are given a set of coin denominations and an amount, and you need to find the minimum number of coins required to make up that amount. The "take or skip" approach is a common way to solve this problem using recursion with memoization or dynamic programming.

Problem Statement:

Given an array of coin denominations `coins` and an integer `amount`, return the minimum number of coins needed to make up that amount. If it is not possible to make up the amount with the given coins, return `-1`.

Example:

Input: coins = [1, 2, 5], amount = 11
Output: 3
Explanation: 11 = 5 + 5 + 1

Approaches to Solve the Problem

1. Recursive Approach (Take or Skip)

- Idea: For each coin, decide whether to take it or skip it. If you take it, subtract its value from the amount and recursively solve for the remaining amount. If you skip it, move to the next coin.
- Time Complexity: O(2^n) (exponential due to repeated calculations)
- Space Complexity: O(n) (due to the recursion stack)

```
public int coinChange(int[] coins, int amount) {
       return coinChangeHelper(coins, amount, 0);
   }

   private int coinChangeHelper(int[] coins, int amount, int index) {
       if (amount == 0) {
           return 0;
       }
       if (index >= coins.length || amount < 0) {
           return -1;
       }
       int take = coinChangeHelper(coins, amount - coins[index], index);
       int skip = coinChangeHelper(coins, amount, index + 1);
       if (take == -1) {
           return skip;
       }
       if (skip == -1) {
           return take + 1;
       }
       return Math.min(take + 1, skip);
   }
```

Explanation:

- The base case is when `amount` is `0`, in which case no more coins are needed.

- If the index is out of bounds or the amount becomes negative, return `-1` indicating it's not possible to make up the amount.
- For each coin, decide whether to take it or skip it.
- If you take the coin, recursively solve for the remaining amount.
- If you skip the coin, move to the next coin.
- Return the minimum number of coins needed, considering both options.

2. Dynamic Programming (Memoization)

- Idea: Use memoization to store the results of subproblems to avoid redundant calculations in the recursive approach.
- Time Complexity: O(n * m) (where `n` is the amount and `m` is the number of coins)
- Space Complexity: O(n * m)

```java
 public int coinChange(int[] coins, int amount) {
       Integer[][] memo = new Integer[coins.length][amount + 1];
       return coinChangeMemo(coins, amount, 0, memo);
   }

   private int coinChangeMemo(int[] coins, int amount, int index,
Integer[][] memo) {
       if (amount == 0) {
           return 0;
       }
       if (index >= coins.length || amount < 0) {
           return -1;
       }
       if (memo[index][amount] != null) {
           return memo[index][amount];
       }
       int take = coinChangeMemo(coins, amount - coins[index], index,
memo);
       int skip = coinChangeMemo(coins, amount, index + 1, memo);
       if (take == -1) {
           memo[index][amount] = skip;
       } else if (skip == -1) {
           memo[index][amount] = take + 1;
       } else {
           memo[index][amount] = Math.min(take + 1, skip);
       }
       return memo[index][amount];
   }
```

Explanation:

- We use a 2D array `memo` to store the results of subproblems.
- The base cases are the same as in the recursive approach.
- Before making a recursive call, we check if the result is already in the `memo` array. If it is, we return the stored result.
- If the result is not in the `memo` array, we compute it, store it in the array, and then return it.
- This approach reduces the time complexity to O(n * m) by avoiding redundant calculations.

3. Dynamic Programming (Tabulation)
- Idea: Use an iterative approach to build up the solution from the base cases.
- Time Complexity: O(n * m)
- Space Complexity: O(n)

```
public int coinChange(int[] coins, int amount) {
       int[] dp = new int[amount + 1];
       Arrays.fill(dp, amount + 1);
       dp[0] = 0;
       for (int coin : coins) {
           for (int i = coin; i <= amount; i++) {
               dp[i] = Math.min(dp[i], dp[i - coin] + 1);
           }
       }
       return dp[amount] > amount ? -1 : dp[amount];
   }
```

Explanation:
- We initialize a `dp` array where `dp[i]` represents the minimum number of coins needed to make up the amount `i`.
- We fill the `dp` array with `amount + 1` (a value larger than the maximum possible number of coins).
- The base case is `dp[0] = 0` since no coins are needed to make up the amount `0`.
- For each coin, we iterate through the amounts from the coin's value to the target amount and update the `dp` array.
- The final result is `dp[amount]`, or `-1` if it's not possible to make up the amount.

Comparison of Approaches

- Recursive Approach: Simple but inefficient due to exponential time complexity.
- Dynamic Programming (Memoization): More efficient with O(n * m) time complexity, but uses O(n * m) space for the memoization array.
- Dynamic Programming (Tabulation): Efficient with O(n * m) time complexity and O(n) space, making it the most practical solution for large inputs.

Conclusion

The dynamic programming tabulation approach is the most efficient for solving the "Coin Change" problem, providing O(n * m) time complexity and O(n) space complexity. The choice between the methods can depend on personal preference or specific use cases, but the tabulation approach is generally preferred for its efficiency and simplicity.

Longest Increasing Subsequence

The "Longest Increasing Subsequence" (LIS) problem is a classic dynamic programming problem where you need to find the length of the longest subsequence of a given sequence such that all elements of the subsequence are sorted in increasing order.

Problem Statement:
Given an integer array `nums`, return the length of the longest strictly increasing subsequence.

Example:

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.

Approaches to Solve the Problem

1. Dynamic Programming Approach
- Idea: Use dynamic programming to store the length of the longest increasing subsequence ending at each index. Iterate through the array and for each element, check all previous elements to find the longest subsequence that can be extended by the current element.
- Time Complexity: $O(n^2)$
- Space Complexity: O(n)

```
public int lengthOfLIS(int[] nums) {
      if (nums == null || nums.length == 0) {
          return 0;
      }
      int[] dp = new int[nums.length];
      Arrays.fill(dp, 1);
      int maxLength = 1;
      for (int i = 1; i < nums.length; i++) {
          for (int j = 0; j < i; j++) {
              if (nums[i] > nums[j]) {
                  dp[i] = Math.max(dp[i], dp[j] + 1);
```

```
                }
            }
            maxLength = Math.max(maxLength, dp[i]);
        }
        return maxLength;
    }
```

Explanation:
- We initialize a `dp` array where `dp[i]` represents the length of the longest increasing subsequence ending at index `i`.
- We fill the `dp` array with `1` since each element is a subsequence of length `1` by itself.
- For each element at index `i`, we check all previous elements at indices `j < i`. If `nums[i] > nums[j]`, we update `dp[i]` to be the maximum of its current value and `dp[j] + 1`.
- We keep track of the maximum length found in the `dp` array.
- The final result is the maximum value in the `dp` array.

2. Binary Search with Dynamic Programming Approach
- Idea: Use a combination of binary search and dynamic programming to efficiently find the length of the longest increasing subsequence. Maintain a list that stores the smallest tail value for all increasing subsequences of different lengths.
- Time Complexity: O(n log n)
- Space Complexity: O(n)

```
public int lengthOfLIS(int[] nums) {
        int[] tails = new int[nums.length];
        int size = 0;
        for (int num : nums) {
            int i = 0, j = size;
            while (i != j) {
                int mid = (i + j) / 2;
                if (tails[mid] < num) {
                    i = mid + 1;
                } else {
                    j = mid;
                }
            }
            tails[i] = num;
            if (i == size) {
                size++;
            }
        }
        return size;
    }
```

Explanation:
   - We initialize a `tails` array to store the smallest tail values for all increasing subsequences of different lengths.
   - We iterate through each number in `nums` and use binary search to find the position at which the number can be placed in the `tails` array.
   - If the number is greater than all elements in `tails`, it extends the longest subsequence, so we append it to `tails` and increment the size.
   - If the number is not greater than all elements, it replaces the first element in `tails` that is greater than or equal to it.
   - The final result is the size of the `tails` array, which represents the length of the longest increasing subsequence.

Comparison of Approaches

- Dynamic Programming Approach: Simple and straightforward, but less efficient with $O(n^2)$ time complexity. It is easy to understand and implement.
- Binary Search with Dynamic Programming Approach: More efficient with O(n log n) time complexity, making it suitable for larger inputs. It leverages binary search to optimize the process of finding the correct position for each element.

Conclusion

The binary search with dynamic programming approach is the most efficient for solving the "Longest Increasing Subsequence" problem, providing O(n log n) time complexity. The choice between the methods can depend on personal preference or specific use cases, but the binary search approach is generally preferred for its efficiency, especially for larger datasets.

Longest Common Subsequence

The "Longest Common Subsequence" (LCS) problem is a classic dynamic programming problem where you need to find the length of the longest subsequence that is common to two given strings. A subsequence is a sequence that appears in the same relative order but not necessarily contiguous.

Problem Statement:
Given two strings `text1` and `text2`, return the length of their longest common subsequence. If there is no common subsequence, return `0`.

Example:

Input: text1 = "abcde", text2 = "ace"
Output: 3
Explanation: The longest common subsequence is "ace" and its length is 3.

Approaches to Solve the Problem

1. Recursive Approach

- Idea: Use recursion to explore all possible subsequences of the two strings. If the characters match, move to the next character in both strings. If they don't match, consider two options: move to the next character in `text1` or move to the next character in `text2`.
- Time Complexity: O(2^(m+n)) (exponential due to repeated calculations)
- Space Complexity: O(m + n) (due to the recursion stack)

```
public int longestCommonSubsequence(String text1, String text2) {
       return lcsHelper(text1, text2, text1.length(), text2.length());
   }

   private int lcsHelper(String text1, String text2, int m, int n) {
       if (m == 0 || n == 0) {
           return 0;
       }
       if (text1.charAt(m - 1) == text2.charAt(n - 1)) {
           return 1 + lcsHelper(text1, text2, m - 1, n - 1);
       } else {
           return Math.max(lcsHelper(text1, text2, m - 1, n),
lcsHelper(text1, text2, m, n - 1));
       }
   }
```

Explanation:

- The base case is when either of the strings is empty, in which case the LCS length is `0`.
- If the last characters of both strings match, we add `1` to the result and recursively compute the LCS for the remaining strings.
- If the last characters do not match, we consider two options: exclude the last character of `text1` or exclude the last character of `text2`, and take the maximum of the two results.
- This approach is simple but inefficient due to exponential time complexity caused by repeated calculations.

2. Dynamic Programming (Memoization)

- Idea: Use memoization to store the results of subproblems to avoid redundant calculations in the recursive approach.

- Time Complexity: O(m * n)
- Space Complexity: O(m * n)

```
 public int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length();
        int n = text2.length();
        int[][] memo = new int[m + 1][n + 1];
        for (int[] row : memo) {
            Arrays.fill(row, -1);
        }
        return lcsMemo(text1, text2, m, n, memo);
    }

    private int lcsMemo(String text1, String text2, int m, int n, int[][]
memo) {
        if (m == 0 || n == 0) {
            return 0;
        }
        if (memo[m][n] != -1) {
            return memo[m][n];
        }
        if (text1.charAt(m - 1) == text2.charAt(n - 1)) {
            memo[m][n] = 1 + lcsMemo(text1, text2, m - 1, n - 1, memo);
        } else {
            memo[m][n] = Math.max(lcsMemo(text1, text2, m - 1, n, memo),
lcsMemo(text1, text2, m, n - 1, memo));
        }
        return memo[m][n];
    }
```

Explanation:
- We use a 2D array `memo` to store the results of subproblems.
- The base case is the same as in the recursive approach.
- Before making a recursive call, we check if the result is already in the `memo` array. If it is, we return the stored result.
- If the result is not in the `memo` array, we compute it, store it in the array, and then return it.
- This approach reduces the time complexity to O(m * n) by avoiding redundant calculations.

3. Dynamic Programming (Tabulation)
- Idea: Use an iterative approach to build up the solution from the base cases.
- Time Complexity: O(m * n)
- Space Complexity: O(m * n)

```
  public int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length();
        int n = text2.length();
        int[][] dp = new int[m + 1][n + 1];
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                    dp[i][j] = dp[i - 1][j - 1] + 1;
                } else {
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
                }
            }
        }
        return dp[m][n];
   }
```

Explanation:
- We initialize a `dp` array where `dp[i][j]` represents the length of the LCS of the first `i` characters of `text1` and the first `j` characters of `text2`.
- We iterate through each character of `text1` and `text2`.
- If the characters match, we set `dp[i][j]` to `dp[i - 1][j - 1] + 1`.
- If the characters do not match, we set `dp[i][j]` to the maximum of `dp[i - 1][j]` and `dp[i][j - 1]`.
- The final result is `dp[m][n]`.

4. Space-Optimized Dynamic Programming
- Idea: Since the current row in the `dp` array only depends on the previous row, we can optimize space by using a 1D array.
- Time Complexity: O(m * n)
- Space Complexity: O(n)

```
public int longestCommonSubsequence(String text1, String text2) {
        int m = text1.length();
        int n = text2.length();
        int[] dp = new int[n + 1];
        for (int i = 1; i <= m; i++) {
            int prev = 0;
            for (int j = 1; j <= n; j++) {
                int temp = dp[j];
                if (text1.charAt(i - 1) == text2.charAt(j - 1)) {
                    dp[j] = prev + 1;
                } else {
                    dp[j] = Math.max(dp[j], dp[j - 1]);
                }
                prev = temp;
            }
        }
        return dp[n];
    }
```

Explanation:
  - We use a 1D array `dp` where `dp[j]` represents the length of the LCS of the first `i` characters of `text1` and the first `j` characters of `text2`.
  - We iterate through each character of `text1` and `text2`.
  - If the characters match, we set `dp[j]` to `prev + 1`.
  - If the characters do not match, we set `dp[j]` to the maximum of `dp[j]` and `dp[j - 1]`.
  - We use a `prev` variable to store the value of `dp[j]` before it is updated.
  - The final result is `dp[n]`.

Comparison of Approaches

- Recursive Approach: Simple but inefficient due to exponential time complexity.
- Dynamic Programming (Memoization): More efficient with O(m * n) time complexity, but uses O(m * n) space for the memoization array.
- Dynamic Programming (Tabulation): Efficient with O(m * n) time complexity and O(m * n) space, but can be optimized further.
- Space-Optimized Dynamic Programming: Most efficient with O(m * n) time complexity and O(n) space.

Conclusion

The space-optimized dynamic programming approach is the most efficient for solving the "Longest Common Subsequence" problem, providing O(m * n) time complexity and O(n) space complexity. The choice between the methods can depend on personal preference or specific use cases, but the space-optimized approach is generally preferred for its efficiency.

Word Break Problem

The "Word Break" problem is a classic dynamic programming problem where you need to determine if a given string can be segmented into a sequence of one or more dictionary words.

Problem Statement:
Given a string `s` and a dictionary of words `wordDict`, return `true` if `s` can be segmented into a space-separated sequence of one or more dictionary words. Otherwise, return `false`.

Example:

Input: s = "leetcode", wordDict = ["leet", "code"]
Output: true
Explanation: "leetcode" can be segmented as "leet code".

Approaches to Solve the Problem

1. Recursive Approach

- Idea: Use recursion to explore all possible ways to segment the string. For each prefix of the string, check if it is in the dictionary and recursively check the remaining suffix.
- Time Complexity: O(2^n) (exponential due to repeated calculations)
- Space Complexity: O(n) (due to the recursion stack)

```
  public boolean wordBreak(String s, List<String> wordDict) {
       return wordBreakHelper(s, new HashSet<>(wordDict), 0);
   }

   private boolean wordBreakHelper(String s, Set<String> wordDict, int
start) {
       if (start == s.length()) {
           return true;
       }
       for (int end = start + 1; end <= s.length(); end++) {
           if (wordDict.contains(s.substring(start, end)) &&
wordBreakHelper(s, wordDict, end)) {
               return true;
           }
       }
       return false;
   }
```

Explanation:

- The base case is when `start` reaches the end of the string, in which case we return `true`.
- For each possible prefix of the string starting at `start`, we check if it is in the dictionary.
- If it is, we recursively check the remaining suffix.
- If any recursive call returns `true`, we return `true`.
- This approach is simple but inefficient due to exponential time complexity caused by repeated calculations.

2. Dynamic Programming (Memoization)

- Idea: Use memoization to store the results of subproblems to avoid redundant calculations in the recursive approach.

  - Time Complexity: O(n²) (where `n` is the length of the string)
  - Space Complexity: O(n)

```
   public boolean wordBreak(String s, List<String> wordDict) {
        return wordBreakMemo(s, new HashSet<>(wordDict), 0, new
Boolean[s.length()]);
    }

    private boolean wordBreakMemo(String s, Set<String> wordDict, int start,
Boolean[] memo) {
        if (start == s.length()) {
            return true;
        }
        if (memo[start] != null) {
            return memo[start];
        }
        for (int end = start + 1; end <= s.length(); end++) {
            if (wordDict.contains(s.substring(start, end)) &&
wordBreakMemo(s, wordDict, end, memo)) {
                memo[start] = true;
                return true;
            }
        }
        memo[start] = false;
        return false;
    }
```

Explanation:
  - We use a `memo` array to store the results of subproblems.
  - The base case is the same as in the recursive approach.
  - Before making a recursive call, we check if the result is already in the `memo` array. If it is, we return the stored result.
  - If the result is not in the `memo` array, we compute it, store it in the array, and then return it.
  - This approach reduces the time complexity to O(n²) by avoiding redundant calculations.

3. Dynamic Programming (Tabulation)
  - Idea: Use an iterative approach to build up the solution from the base cases. Create a `dp` array where `dp[i]` is `true` if the substring `s[0..i-1]` can be segmented into dictionary words.
  - Time Complexity: O(n²)
  - Space Complexity: O(n)

```
  public boolean wordBreak(String s, List<String> wordDict) {
```

```java
        Set<String> wordDictSet = new HashSet<>(wordDict);
        boolean[] dp = new boolean[s.length() + 1];
        dp[0] = true;
        for (int i = 1; i <= s.length(); i++) {
            for (int j = 0; j < i; j++) {
                if (dp[j] && wordDictSet.contains(s.substring(j, i))) {
                    dp[i] = true;
                    break;
                }
            }
        }
        return dp[s.length()];
    }
```

Explanation:

- We initialize a `dp` array where `dp[i]` represents whether the substring `s[0..i-1]` can be segmented into dictionary words.
- The base case is `dp[0] = true` since an empty string can always be segmented.
- For each position `i` in the string, we check all possible prefixes `s[j..i-1]` where `j < i`.
- If `dp[j]` is `true` and the prefix `s[j..i-1]` is in the dictionary, we set `dp[i]` to `true`.
- The final result is `dp[s.length()]`.

Comparison of Approaches

- Recursive Approach: Simple but inefficient due to exponential time complexity.
- Dynamic Programming (Memoization): More efficient with $O(n^2)$ time complexity, but uses O(n) space for the memoization array.
- Dynamic Programming (Tabulation): Efficient with $O(n^2)$ time complexity and O(n) space, making it the most practical solution for large inputs.

Conclusion

The dynamic programming tabulation approach is the most efficient for solving the "Word Break" problem, providing $O(n^2)$ time complexity and O(n) space complexity. The choice between the methods can depend on personal preference or specific use cases, but the tabulation approach is generally preferred for its efficiency and simplicity.

Combination Sum

The "Combination Sum" problem is a classic backtracking problem where you need to find all unique combinations of numbers in a given array that sum up to a target value. Each number in the array can be used an unlimited number of times.

Problem Statement:
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.

Example:

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.

Approaches to Solve the Problem

1. Backtracking (Recursive) Approach
- Idea: Use recursion to explore all possible combinations. For each candidate, decide whether to include it in the current combination or not. If included, subtract its value from the target and continue the search. If not, move to the next candidate.
- Time Complexity: O(N^(T/M + 1)) (where `N` is the number of candidates, `T` is the target, and `M` is the minimal value among candidates)
- Space Complexity: O(T/M) (due to the recursion stack)

```
public List<List<Integer>> combinationSum(int[] candidates, int target) {
      List<List<Integer>> result = new ArrayList<>();
      backtrack(result, new ArrayList<>(), candidates, target, 0);
      return result;
  }

  private void backtrack(List<List<Integer>> result, List<Integer>
tempList, int[] candidates, int remain, int start) {
      if (remain < 0) {
          return;
      } else if (remain == 0) {
          result.add(new ArrayList<>(tempList));
      } else {
          for (int i = start; i < candidates.length; i++) {
              tempList.add(candidates[i]);
```

```
                backtrack(result, tempList, candidates, remain -
candidates[i], i); // not i + 1 because we can reuse same elements
                tempList.remove(tempList.size() - 1);
            }
        }
    }
```

Explanation:
- We initialize a `result` list to store all valid combinations.
- We use a `tempList` to build the current combination.
- The base case is when `remain` (the remaining target) is less than `0`, in which case we return.
- If `remain` is `0`, we add the current combination (`tempList`) to the `result`.
- For each candidate, we add it to `tempList` and recursively call `backtrack` with the updated `remain` and the same start index (since we can reuse elements).
- After the recursive call, we remove the last element from `tempList` to backtrack and try the next candidate.

2. Dynamic Programming Approach
- Idea: Use dynamic programming to build up the solution. Create a `dp` array where `dp[i]` is a list of combinations that sum up to `i`. Iterate through each candidate and update the `dp` array.
- Time Complexity: O(N * T^2) (where `N` is the number of candidates and `T` is the target)
- Space Complexity: O(T^2)

```
   public List<List<Integer>> combinationSum(int[] candidates, int target) {
        List<List<Integer>>[] dp = new ArrayList[target + 1];
        for (int i = 0; i <= target; i++) {
            dp[i] = new ArrayList<>();
        }
        dp[0].add(new ArrayList<>());
        for (int candidate : candidates) {
            for (int i = candidate; i <= target; i++) {
                for (List<Integer> prev : dp[i - candidate]) {
                    List<Integer> temp = new ArrayList<>(prev);
                    temp.add(candidate);
                    dp[i].add(temp);
                }
            }
        }
        return dp[target];
    }
```

Explanation:
   - We initialize a `dp` array where `dp[i]` is a list of combinations that sum up to `i`.
   - `dp[0]` is initialized with an empty combination.
   - For each candidate, we iterate through the `dp` array from the candidate's value to the target.
   - For each position `i`, we add the candidate to each combination in `dp[i - candidate]` and add the new combination to `dp[i]`.
   - The final result is `dp[target]`, which contains all combinations that sum up to the target.

Comparison of Approaches

- Backtracking (Recursive) Approach: Simple and intuitive, but can be inefficient for large targets due to its exponential time complexity. It is easy to implement and understand.
- Dynamic Programming Approach: More efficient for larger targets, but uses more space and is more complex to implement. It builds up the solution iteratively and avoids redundant calculations.

Conclusion

The backtracking approach is generally preferred for its simplicity and ease of implementation, especially for smaller inputs. However, for larger targets, the dynamic programming approach can be more efficient. The choice between the methods can depend on the specific requirements and constraints of the problem.

House Robber

The "House Robber" problem is a classic dynamic programming problem where you need to determine the maximum amount of money you can rob from a row of houses without robbing two adjacent houses.

Problem Statement:
Given an integer array `nums` representing the amount of money in each house, return the maximum amount of money you can rob tonight without alerting the police (i.e., without robbing two adjacent houses).

Example:

Input: nums = [2,7,9,3,1]
Output: 12
Explanation: Rob house 1 (money = 2), rob house 3 (money = 9), and rob house 5 (money = 1).
Total amount you can rob = 2 + 9 + 1 = 12.

Approaches to Solve the Problem

1. Recursive Approach
- Idea: Use recursion to explore all possible ways to rob the houses. For each house, decide whether to rob it or skip it. If you rob it, you cannot rob the next house. If you skip it, you can consider robbing the next house.
- Time Complexity: O(2^n) (exponential due to repeated calculations)
- Space Complexity: O(n) (due to the recursion stack)

```
public int rob(int[] nums) {
       return robHelper(nums, 0);
   }

   private int robHelper(int[] nums, int index) {
       if (index >= nums.length) {
           return 0;
       }
       int robCurrent = nums[index] + robHelper(nums, index + 2);
       int skipCurrent = robHelper(nums, index + 1);
       return Math.max(robCurrent, skipCurrent);
   }
```

Explanation:
- The base case is when `index` is out of bounds, in which case we return `0`.
- For each house, we consider two options: rob the current house and skip the next house, or skip the current house and consider the next house.
- We return the maximum of the two options.
- This approach is simple but inefficient due to exponential time complexity caused by repeated calculations.

2. Dynamic Programming (Memoization)
- Idea: Use memoization to store the results of subproblems to avoid redundant calculations in the recursive approach.
- Time Complexity: O(n)
- Space Complexity: O(n)

```
 public int rob(int[] nums) {
       Integer[] memo = new Integer[nums.length];
       return robMemo(nums, 0, memo);
   }

   private int robMemo(int[] nums, int index, Integer[] memo) {
       if (index >= nums.length) {
           return 0;
       }
       if (memo[index] != null) {
           return memo[index];
       }
       int robCurrent = nums[index] + robMemo(nums, index + 2, memo);
       int skipCurrent = robMemo(nums, index + 1, memo);
       memo[index] = Math.max(robCurrent, skipCurrent);
       return memo[index];
   }
```

Explanation:
- We use a `memo` array to store the results of subproblems.
- The base case is the same as in the recursive approach.
- Before making a recursive call, we check if the result is already in the `memo` array. If it is, we return the stored result.
- If the result is not in the `memo` array, we compute it, store it in the array, and then return it.
- This approach reduces the time complexity to O(n) by avoiding redundant calculations.

3. Dynamic Programming (Tabulation)
- Idea: Use an iterative approach to build up the solution from the base cases. Create a `dp` array where `dp[i]` represents the maximum amount of money that can be robbed up to the ith house.
- Time Complexity: O(n)
- Space Complexity: O(n)

```
public int rob(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        if (nums.length == 1) {
            return nums[0];
        }
        int[] dp = new int[nums.length];
        dp[0] = nums[0];
        dp[1] = Math.max(nums[0], nums[1]);
        for (int i = 2; i < nums.length; i++) {
            dp[i] = Math.max(dp[i - 1], dp[i - 2] + nums[i]);
        }
        return dp[nums.length - 1];
```

```
    }
```

Explanation:
- We initialize a `dp` array where `dp[i]` represents the maximum amount of money that can be robbed up to the ith house.
- The base cases are `dp[0] = nums[0]` and `dp[1] = Math.max(nums[0], nums[1])`.
- For each house, we consider two options: rob the current house and add its value to the amount robbed up to two houses before, or skip the current house and take the amount robbed up to the previous house.
- We return the last element in the `dp` array, which represents the maximum amount of money that can be robbed up to the last house.

4. Space-Optimized Dynamic Programming
- Idea: Since the current value in the `dp` array only depends on the previous two values, we can optimize space by using two variables instead of an entire array.
- Time Complexity: O(n)
- Space Complexity: O(1)

```
public int rob(int[] nums) {
       if (nums == null || nums.length == 0) {
           return 0;
       }
       if (nums.length == 1) {
           return nums[0];
       }
       int prev2 = nums[0];
       int prev1 = Math.max(nums[0], nums[1]);
       for (int i = 2; i < nums.length; i++) {
           int current = Math.max(prev1, prev2 + nums[i]);
           prev2 = prev1;
           prev1 = current;
       }
       return prev1;
    }
```

Explanation:
- We use two variables `prev2` and `prev1` to store the maximum amounts robbed up to two houses before and the previous house, respectively.
- The base cases are `prev2 = nums[0]` and `prev1 = Math.max(nums[0], nums[1])`.
- For each house, we compute the current maximum amount as the maximum of `prev1` and `prev2 + nums[i]`.

  - We then update `prev2` to be the old `prev1` and `prev1` to be the current value.
  - The final result is `prev1`.

Comparison of Approaches

- Recursive Approach: Simple but inefficient due to exponential time complexity.
- Dynamic Programming (Memoization): More efficient with O(n) time complexity, but uses O(n) space for the memoization array.
- Dynamic Programming (Tabulation): Efficient with O(n) time complexity and O(n) space, but can be optimized further.
- Space-Optimized Dynamic Programming: Most efficient with O(n) time complexity and O(1) space.

Conclusion

The space-optimized dynamic programming approach is the most efficient for solving the "House Robber" problem, providing O(n) time complexity and O(1) space complexity. The choice between the methods can depend on personal preference or specific use cases, but the space-optimized approach is generally preferred for its efficiency.

House Robber II

The "House Robber II" problem is an extension of the "House Robber" problem where the houses are arranged in a circle. This means that the first and last houses are adjacent, and you cannot rob both of them.

Problem Statement:
Given an integer array `nums` representing the amount of money in each house, return the maximum amount of money you can rob tonight without alerting the police (i.e., without robbing two adjacent houses). The houses are arranged in a circle, meaning the first and last houses are adjacent.

Example:

Input: nums = [2,3,2]
Output: 3
Explanation: You cannot rob house 1 (money = 2) and then rob house 3 (money = 2) because they are adjacent. The maximum amount you can rob is 3 (house 2).

Approaches to Solve the Problem

1. Dynamic Programming Approach
  - Idea: Since the houses are arranged in a circle, we can break the problem into two subproblems:

1. Rob houses from the first house to the second-to-last house.
2. Rob houses from the second house to the last house.
The final result is the maximum of these two subproblems.

- Time Complexity: O(n)
- Space Complexity: O(n)

```java
  public int rob(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        if (nums.length == 1) {
            return nums[0];
        }
        return Math.max(robHelper(nums, 0, nums.length - 2), robHelper(nums,
1, nums.length - 1));
    }

    private int robHelper(int[] nums, int start, int end) {
        if (start == end) {
            return nums[start];
        }
        int[] dp = new int[nums.length];
        dp[start] = nums[start];
        dp[start + 1] = Math.max(nums[start], nums[start + 1]);
        for (int i = start + 2; i <= end; i++) {
            dp[i] = Math.max(dp[i - 1], dp[i - 2] + nums[i]);
        }
        return dp[end];
    }
```

Explanation:

- We handle the edge cases where the array is empty or has only one element.
- We break the problem into two subproblems: robbing houses from the first to the second-to-last house and robbing houses from the second to the last house.
- We use a helper function `robHelper` to compute the maximum amount of money that can be robbed for a given range of houses.
- The helper function uses dynamic programming to build up the solution from the base cases.
- The final result is the maximum of the two subproblems.

2. Space-Optimized Dynamic Programming Approach

- Idea: Since the current value in the `dp` array only depends on the previous two values, we can optimize space by using two variables instead of an entire array.
- Time Complexity: O(n)

- Space Complexity: O(1)

```
 public int rob(int[] nums) {
        if (nums == null || nums.length == 0) {
            return 0;
        }
        if (nums.length == 1) {
            return nums[0];
        }
        return Math.max(robHelper(nums, 0, nums.length - 2), robHelper(nums,
1, nums.length - 1));
    }

    private int robHelper(int[] nums, int start, int end) {
        if (start == end) {
            return nums[start];
        }
        int prev2 = nums[start];
        int prev1 = Math.max(nums[start], nums[start + 1]);
        for (int i = start + 2; i <= end; i++) {
            int current = Math.max(prev1, prev2 + nums[i]);
            prev2 = prev1;
            prev1 = current;
        }
        return prev1;
    }
```

Explanation:
- We handle the edge cases where the array is empty or has only one element.
- We break the problem into two subproblems: robbing houses from the first to the second-to-last house and robbing houses from the second to the last house.
- We use a helper function `robHelper` to compute the maximum amount of money that can be robbed for a given range of houses.
- The helper function uses two variables `prev2` and `prev1` to store the maximum amounts robbed up to two houses before and the previous house, respectively.
- For each house, we compute the current maximum amount as the maximum of `prev1` and `prev2 + nums[i]`.
- We then update `prev2` to be the old `prev1` and `prev1` to be the current value.
- The final result is the maximum of the two subproblems.

Comparison of Approaches

- Dynamic Programming Approach: Simple and straightforward, but uses O(n) space for the `dp` array.
- Space-Optimized Dynamic Programming Approach: More efficient with O(1) space complexity, making it the preferred approach for large inputs.

Conclusion

The space-optimized dynamic programming approach is the most efficient for solving the "House Robber II" problem, providing O(n) time complexity and O(1) space complexity. The choice between the methods can depend on personal preference or specific use cases, but the space-optimized approach is generally preferred for its efficiency.

Decode Ways

The "Decode Ways" problem is a classic dynamic programming problem. The problem statement is as follows:

Problem Statement:
Given a string `s` containing only digits, return the number of ways to decode it. The mapping is:
- 'A' -> 1
- 'B' -> 2
- ...
- 'Z' -> 26

For example:
- Input: `s = "12"`
- Output: `2` (Because "12" can be decoded as "AB" (1 2) or "L" (12)).

Approaches to Solve the Problem

1. Recursive Approach (Top-Down)
This approach involves breaking the problem into smaller subproblems. For each character or pair of characters, we recursively check if it can be decoded.

Explanation:
- If the string is empty, there is 1 way to decode it (empty string).
- If the first character is '0', it cannot be decoded, so return 0.
- If the string has only one character and it's not '0', return 1.
- For two characters, check if they form a valid number (between 10 and 26). If yes, add the ways to decode the remaining string.

Code:

```java
public int numDecodings(String s) {
    return helper(s, 0);
}

private int helper(String s, int index) {
    // Base case: If we reach the end of the string, return 1
    if (index == s.length()) {
        return 1;
    }
    // If the current character is '0', it cannot be decoded
    if (s.charAt(index) == '0') {
        return 0;
    }
    // Decode single character
    int ways = helper(s, index + 1);
    // Decode two characters if possible
    if (index + 1 < s.length() && Integer.parseInt(s.substring(index, index
+ 2)) <= 26) {
        ways += helper(s, index + 2);
    }
    return ways;
}
```

Time Complexity: O(2^n) (Exponential due to overlapping subproblems).

2. Dynamic Programming (Bottom-Up)
This approach uses memoization to store the results of subproblems and avoid redundant calculations.

Explanation:
- Use a `dp` array where `dp[i]` represents the number of ways to decode the substring `s[0..i-1]`.
- Initialize `dp[0] = 1` (empty string has 1 way to decode).
- Iterate through the string and update the `dp` array based on valid single-digit and two-digit decodings.

Code:

```java
public int numDecodings(String s) {
    if (s == null || s.length() == 0) {
        return 0;
    }
    int n = s.length();
    int[] dp = new int[n + 1];
    dp[0] = 1; // Empty string has 1 way to decode
    dp[1] = s.charAt(0) == '0' ? 0 : 1; // First character

    for (int i = 2; i <= n; i++) {
        // Single digit
        int oneDigit = Integer.parseInt(s.substring(i - 1, i));
        if (oneDigit >= 1 && oneDigit <= 9) {
            dp[i] += dp[i - 1];
        }
        // Two digits
        int twoDigits = Integer.parseInt(s.substring(i - 2, i));
        if (twoDigits >= 10 && twoDigits <= 26) {
            dp[i] += dp[i - 2];
        }
    }
    return dp[n];
}
```

Time Complexity: O(n), where `n` is the length of the string.
Space Complexity: O(n) for the `dp` array.

3. Optimized Dynamic Programming (Space Optimization)
We can optimize the space complexity by using only two variables instead of the entire `dp` array.

Explanation:
- Use `prev1` to store the number of ways to decode the previous character.
- Use `prev2` to store the number of ways to decode the character before the previous one.
- Update these variables iteratively.

Code:

```java
public int numDecodings(String s) {
    if (s == null || s.length() == 0 || s.charAt(0) == '0') {
        return 0;
    }
    int n = s.length();
    int prev1 = 1; // Ways to decode empty string
    int prev2 = 1; // Ways to decode first character

    for (int i = 1; i < n; i++) {
        int current = 0;
        // Single digit
        int oneDigit = Integer.parseInt(s.substring(i, i + 1));
        if (oneDigit >= 1 && oneDigit <= 9) {
            current += prev2;
        }
        // Two digits
        int twoDigits = Integer.parseInt(s.substring(i - 1, i + 1));
        if (twoDigits >= 10 && twoDigits <= 26) {
            current += prev1;
        }
        prev1 = prev2;
        prev2 = current;
    }
    return prev2;
}
```

Time Complexity: O(n).
Space Complexity: O(1).
 Summary
- Recursive Approach: Simple but inefficient due to overlapping subproblems.
- Dynamic Programming: Efficient with O(n) time and space.
- Optimized DP: Most efficient with O(n) time and O(1) space.

The optimized DP approach is the best for this problem.

Unique Paths

The "Unique Paths" problem is another classic dynamic programming problem. The problem statement is as follows:

Problem Statement:
Given a `m x n` grid, find the number of unique paths from the top-left corner to the bottom-right corner. You can only move either down or right at any point in time.

For example:
- Input: `m = 3`, `n = 2`
- Output: `3` (Because there are 3 unique paths: Right -> Down -> Down, Down -> Right -> Down, Down -> Down -> Right).

Approaches to Solve the Problem

1. Recursive Approach (Top-Down)
This approach involves breaking the problem into smaller subproblems. For each cell, we recursively calculate the number of ways to reach the bottom-right corner.

Explanation:
- If we are at the bottom-right corner, there is only 1 way to stay there.
- If we are at the last row or last column, there is only 1 way to move (right or down).
- For other cells, the number of unique paths is the sum of paths from the cell below and the cell to the right.

Code:

```java
public int uniquePaths(int m, int n) {
    return helper(0, 0, m, n);
}

private int helper(int row, int col, int m, int n) {
    // Base case: If we reach the bottom-right corner
    if (row == m - 1 && col == n - 1) {
        return 1;
    }
    // If we go out of bounds
    if (row >= m || col >= n) {
        return 0;
    }
    // Move down + move right
    return helper(row + 1, col, m, n) + helper(row, col + 1, m, n);
}
```

Time Complexity: O(2^(m+n)) (Exponential due to overlapping subproblems).

2. Dynamic Programming (Bottom-Up)
This approach uses memoization to store the results of subproblems and avoid redundant calculations.

Explanation:
- Use a `dp` table where `dp[i][j]` represents the number of unique paths to reach cell `(i, j)`.
- Initialize the first row and first column to 1 because there is only 1 way to reach any cell in the first row or first column.
- For other cells, `dp[i][j] = dp[i-1][j] + dp[i][j-1]`.

Code:

```
public int uniquePaths(int m, int n) {
    int[][] dp = new int[m][n];

    // Fill the first row with 1
    for (int i = 0; i < m; i++) {
        dp[i][0] = 1;
    }
    // Fill the first column with 1
    for (int j = 0; j < n; j++) {
        dp[0][j] = 1;
    }

    // Fill the rest of the table
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
        }
    }
    return dp[m - 1][n - 1];
}
```

Time Complexity: O(m * n).
Space Complexity: O(m * n).

3. Optimized Dynamic Programming (Space Optimization)
We can optimize the space complexity by using a single array instead of the entire `dp` table.

Explanation:
- Use a 1D array `dp` where `dp[j]` represents the number of unique paths to reach the current cell.
- Update the array iteratively for each row.

Code:

```
public int uniquePaths(int m, int n) {
    int[] dp = new int[n];
    // Initialize the first row with 1
    for (int j = 0; j < n; j++) {
        dp[j] = 1;
    }

    // Fill the rest of the table
    for (int i = 1; i < m; i++) {
        for (int j = 1; j < n; j++) {
            dp[j] += dp[j - 1];
        }
    }
    return dp[n - 1];
}
```

Time Complexity: O(m * n).
Space Complexity: O(n).

4. Combinatorial Approach (Mathematical)
This approach uses combinatorics to solve the problem. The number of unique paths is equal to the number of ways to choose `(m-1)` down moves or `(n-1)` right moves from `(m+n-2)` total moves.

Explanation:
- The total number of moves required is `(m + n - 2)`.
- The number of unique paths is the binomial coefficient: `C((m + n - 2), (m - 1))`.

Code:

```
public int uniquePaths(int m, int n) {
    // Calculate the binomial coefficient C((m + n - 2), (m - 1))
    int total = m + n - 2;
    int k = m - 1;
    long result = 1;
    // Calculate C(total, k)
    for (int i = 1; i <= k; i++) {
        result = result * (total - k + i) / i;
    }
    return (int) result;
}
```

Time Complexity: O(min(m, n)).
Space Complexity: O(1).

Summary

- Recursive Approach: Simple but inefficient due to overlapping subproblems.
- Dynamic Programming: Efficient with O(m * n) time and space.
- Optimized DP: More efficient with O(m * n) time and O(n) space.
- Combinatorial Approach: Most efficient with O(min(m, n)) time and O(1) space.

The combinatorial approach is the best for this problem if you are comfortable with combinatorics. Otherwise, the optimized DP approach is a good choice.

Jump Game

The "Jump Game" problem is a popular problem that tests your understanding of greedy algorithms and dynamic programming. The problem statement is as follows:

Problem Statement:
Given an array of non-negative integers `nums`, where each element represents the maximum jump length from that position, determine if you can reach the last index starting from the first index.

For example:
- Input: `nums = [2, 3, 1, 1, 4]`
- Output: `true` (Because you can jump from index 0 to 1, then to the last index).

Approaches to Solve the Problem

1. Greedy Approach
This approach involves iterating through the array and keeping track of the farthest reachable index. If the farthest reachable index is greater than or equal to the last index, return `true`.

Explanation:
- Initialize `maxReach` to 0.
- Iterate through the array and update `maxReach` as `maxReach = max(maxReach, i + nums[i])`.
- If at any point `maxReach >= last index`, return `true`.
- If the loop finishes and `maxReach` is not enough to reach the last index, return `false`.

Code:

```java
public boolean canJump(int[] nums) {
    int maxReach = 0;
    int lastIndex = nums.length - 1;
    for (int i = 0; i <= maxReach; i++) {
        maxReach = Math.max(maxReach, i + nums[i]);
        if (maxReach >= lastIndex) {
            return true;
        }
    }
    return false;
}
```

Time Complexity: O(n), where `n` is the length of the array.
Space Complexity: O(1).

2. Dynamic Programming (Bottom-Up)
This approach uses a `dp` array to store whether each index is reachable.

Explanation:
- Initialize a `dp` array where `dp[i]` represents whether the index `i` is reachable.
- Set `dp[0] = true` because the first index is always reachable.
- For each index `i`, check all previous indices `j` and update `dp[i]` if `dp[j]` is true and `j + nums[j] >= i`.

Code:

```java
public boolean canJump(int[] nums) {
    int n = nums.length;
    boolean[] dp = new boolean[n];
    dp[0] = true; // First index is always reachable

    for (int i = 1; i < n; i++) {
        for (int j = 0; j < i; j++) {
            if (dp[j] && j + nums[j] >= i) {
                dp[i] = true;
                break;
            }
        }
    }
    return dp[n - 1];
}
```

Time Complexity: O(n^2).

Space Complexity: O(n).

3. Backtracking (Recursive)
This approach involves recursively checking all possible jumps from each position.

Explanation:
- Start from the first index and try all possible jumps.
- If you reach the last index, return `true`.
- If you cannot reach the last index from any jump, return `false`.

Code:

```
public boolean canJump(int[] nums) {
    return backtrack(nums, 0);
}

private boolean backtrack(int[] nums, int position) {
    // Base case: If we reach the last index
    if (position == nums.length - 1) {
        return true;
    }
    // Try all possible jumps
    int maxJump = Math.min(position + nums[position], nums.length - 1);
    for (int i = position + 1; i <= maxJump; i++) {
        if (backtrack(nums, i)) {
            return true;
        }
    }
    return false;
}
```

Time Complexity: O(2^n) (Exponential due to overlapping subproblems).
Space Complexity: O(n) (Recursion stack).

4. Memoization (Top-Down Dynamic Programming)
This approach optimizes the backtracking approach by storing the results of subproblems.

Explanation:
- Use a `memo` array to store whether each index can reach the last index.
- If a subproblem is already solved, return the result from `memo`.

Code:

```java
public boolean canJump(int[] nums) {
    int[] memo = new int[nums.length];
    Arrays.fill(memo, -1); // -1: unvisited, 0: false, 1: true
    return backtrackWithMemo(nums, 0, memo);
}

private boolean backtrackWithMemo(int[] nums, int position, int[] memo) {
    // Base case: If we reach the last index
    if (position == nums.length - 1) {
        return true;
    }
    // If already computed, return the result
    if (memo[position] != -1) {
        return memo[position] == 1;
    }
    // Try all possible jumps
    int maxJump = Math.min(position + nums[position], nums.length - 1);
    for (int i = position + 1; i <= maxJump; i++) {
        if (backtrackWithMemo(nums, i, memo)) {
            memo[position] = 1;
            return true;
        }
    }
    memo[position] = 0;
    return false;
}
```

Time Complexity: O(n^2) (Each position is visited at most once).
Space Complexity: O(n) (Recursion stack and memo array).

Summary
- Greedy Approach: Most efficient with O(n) time and O(1) space.
- Dynamic Programming: Less efficient with O(n^2) time and O(n) space.
- Backtracking: Inefficient with O(2^n) time and O(n) space.
- Memoization: Optimized backtracking with O(n^2) time and O(n) space.

The Greedy Approach is the best solution for this problem due to its efficiency and simplicity.

(Graph)

Clone Graph

Cloning a graph is a common problem in computer science, especially in the context of graph traversal and deep copying. Below, I'll explain different approaches to clone a graph in Java, along with their explanations.

Problem Definition:
Given a reference of a node in a connected undirected graph, return a deep copy (clone) of the graph. Each node in the graph contains a value (`int`) and a list (`List<Node>`) of its neighbors.

Node Class Definition:

```java
class Node {
    public int val;
    public List<Node> neighbors;

    public Node(int val) {
        this.val = val;
        neighbors = new ArrayList<>();
    }
}
```

Approach 1: Depth-First Search (DFS) with Recursion
This approach uses DFS to traverse the graph and recursively clone each node and its neighbors.

Steps:
1. Use a `HashMap<Node, Node>` to map original nodes to their clones.
2. Start DFS from the given node.
3. For each node, create a clone and add it to the map.
4. Recursively clone all neighbors and add them to the clone's neighbor list.

Code:

```java
import java.util.*;

class Solution {
    private Map<Node, Node> visited = new HashMap<>();
    public Node cloneGraph(Node node) {
        if (node == null) return null;

        // If the node is already visited, return its clone
        if (visited.containsKey(node)) {
            return visited.get(node);
        }

        // Create a clone of the current node
        Node cloneNode = new Node(node.val);
        visited.put(node, cloneNode);

        // Recursively clone all neighbors
        for (Node neighbor : node.neighbors) {
            cloneNode.neighbors.add(cloneGraph(neighbor));
        }

        return cloneNode;
    }
}
```

Explanation:
- Time Complexity: O(N + E), where N is the number of nodes and E is the number of edges.
- Space Complexity: O(N) for the recursion stack and the hash map.
- This approach is simple and intuitive but may cause a stack overflow for very large graphs due to recursion.

Approach 2: Breadth-First Search (BFS) with a Queue
This approach uses BFS to traverse the graph level by level and clone nodes iteratively.

Steps:
1. Use a `HashMap<Node, Node>` to map original nodes to their clones.
2. Use a queue to perform BFS.
3. Start by cloning the given node and adding it to the queue.
4. For each node in the queue, clone its neighbors and add them to the queue if they haven't been visited.

Code:

```
import java.util.*;
```

```java
class Solution {
    public Node cloneGraph(Node node) {
        if (node == null) return null;

        // Map to store original nodes and their clones
        Map<Node, Node> visited = new HashMap<>();
        Queue<Node> queue = new LinkedList<>();

        // Clone the starting node
        Node cloneNode = new Node(node.val);
        visited.put(node, cloneNode);
        queue.add(node);

        // Perform BFS
        while (!queue.isEmpty()) {
            Node current = queue.poll();

            // Clone neighbors
            for (Node neighbor : current.neighbors) {
                if (!visited.containsKey(neighbor)) {
                    // Clone the neighbor and add it to the map
                    visited.put(neighbor, new Node(neighbor.val));
                    queue.add(neighbor);
                }
                // Add the cloned neighbor to the current clone's neighbors
                visited.get(current).neighbors.add(visited.get(neighbor));
            }
        }

        return cloneNode;
    }
}
```

Explanation:
- Time Complexity: O(N + E), where N is the number of nodes and E is the number of edges.
- Space Complexity: O(N) for the queue and the hash map.
- This approach avoids recursion and is more suitable for large graphs.

Approach 3: Iterative Depth-First Search (DFS) with a Stack
This approach uses an explicit stack to perform DFS iteratively.

Steps:

1. Use a `HashMap<Node, Node>` to map original nodes to their clones.
2. Use a stack to perform DFS.
3. Start by cloning the given node and adding it to the stack.
4. For each node in the stack, clone its neighbors and add them to the stack if they haven't been visited.

Code:

```java
import java.util.*;

class Solution {
    public Node cloneGraph(Node node) {
        if (node == null) return null;

        // Map to store original nodes and their clones
        Map<Node, Node> visited = new HashMap<>();
        Stack<Node> stack = new Stack<>();

        // Clone the starting node
        Node cloneNode = new Node(node.val);
        visited.put(node, cloneNode);
        stack.push(node);

        // Perform DFS
        while (!stack.isEmpty()) {
            Node current = stack.pop();

            // Clone neighbors
            for (Node neighbor : current.neighbors) {
                if (!visited.containsKey(neighbor)) {
                    // Clone the neighbor and add it to the map
                    visited.put(neighbor, new Node(neighbor.val));
                    stack.push(neighbor);
                }
                // Add the cloned neighbor to the current clone's neighbors
                visited.get(current).neighbors.add(visited.get(neighbor));
            }
        }

        return cloneNode;
    }
}
```

Explanation:
- Time Complexity: O(N + E), where N is the number of nodes and E is the number of edges.
- Space Complexity: O(N) for the stack and the hash map.
- This approach is similar to BFS but uses a stack instead of a queue.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Recursive DFS	O(N + E)	O(N)	Simple and intuitive	May cause stack overflow
BFS	O(N + E)	O(N)	No recursion, suitable for large graphs	Slightly more complex
Iterative DFS	O(N + E)	O(N)	No recursion, uses explicit stack	Similar to BFS, but less common

Conclusion:
- Use Recursive DFS for small graphs or when simplicity is preferred.
- Use BFS or Iterative DFS for large graphs to avoid stack overflow.
- All approaches have the same time and space complexity, so the choice depends on the specific use case and constraints.

Course Schedule

The Course Schedule problem is a classic example of detecting cycles in a directed graph. It is commonly solved using topological sorting or cycle detection algorithms. Below, I'll explain different approaches to solve this problem in Java, along with their explanations.

Problem Definition:
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] = [a, b]` indicates that you must take course `b` before course `a`. Return `true` if you can finish all courses (i.e., no cycles in the graph), otherwise return `false`.

Approach 1: Topological Sorting using Kahn's Algorithm (BFS)
This approach uses Kahn's algorithm, which is based on BFS, to perform topological sorting and detect cycles.

Steps:
1. Build the graph using an adjacency list and calculate the in-degree of each node.
2. Use a queue to process nodes with an in-degree of 0.
3. Perform BFS: Dequeue a node, reduce the in-degree of its neighbors, and enqueue neighbors with an in-degree of 0.

4. If all nodes are processed, the graph has no cycles; otherwise, a cycle exists.

Code:

```
import java.util.*;

class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        // Step 1: Build the graph and calculate in-degrees
        List<List<Integer>> graph = new ArrayList<>();
        int[] inDegree = new int[numCourses];
        for (int i = 0; i < numCourses; i++) {
            graph.add(new ArrayList<>());
        }
        for (int[] edge : prerequisites) {
            int course = edge[0];
            int prerequisite = edge[1];
            graph.get(prerequisite).add(course);
            inDegree[course]++;
        }

        // Step 2: Initialize the queue with nodes having in-degree 0
        Queue<Integer> queue = new LinkedList<>();
        for (int i = 0; i < numCourses; i++) {
            if (inDegree[i] == 0) {
                queue.add(i);
            }
        }

        // Step 3: Perform BFS
        int count = 0; // Count of processed nodes
        while (!queue.isEmpty()) {
            int node = queue.poll();
            count++;
            for (int neighbor : graph.get(node)) {
                inDegree[neighbor]--;
                if (inDegree[neighbor] == 0) {
                    queue.add(neighbor);
                }
            }
        }

        // Step 4: Check if all nodes were processed
        return count == numCourses;
    }
}
```

Explanation:
- Time Complexity: O(V + E), where V is the number of courses (nodes) and E is the number of prerequisites (edges).
- Space Complexity: O(V + E) for the graph and in-degree array.
- This approach is efficient and works well for large graphs.

Approach 2: Depth-First Search (DFS) with Cycle Detection
This approach uses DFS to detect cycles in the graph. If a cycle is found, it means it's impossible to finish all courses.

Steps:
1. Build the graph using an adjacency list.
2. Use a `visited` array to track the state of each node:
   - `0`: Not visited.
   - `1`: Being visited (in the current DFS path).
   - `2`: Fully visited.
3. Perform DFS for each unvisited node. If a node being visited is encountered again, a cycle exists.

Code:

```
import java.util.*;

class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        // Step 1: Build the graph
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < numCourses; i++) {
            graph.add(new ArrayList<>());
        }
        for (int[] edge : prerequisites) {
            int course = edge[0];
            int prerequisite = edge[1];
            graph.get(prerequisite).add(course);
        }

        // Step 2: Initialize visited array
        int[] visited = new int[numCourses];
        // Step 3: Perform DFS for each unvisited node
        for (int i = 0; i < numCourses; i++) {
            if (visited[i] == 0 && hasCycle(graph, visited, i)) {
                return false;
            }
        }

        return true;
    }

    private boolean hasCycle(List<List<Integer>> graph, int[] visited, int
node) {
        // Mark the node as being visited
        visited[node] = 1;

        // Visit all neighbors
        for (int neighbor : graph.get(node)) {
            if (visited[neighbor] == 1) {
                // Cycle detected
                return true;
            }
            if (visited[neighbor] == 0 && hasCycle(graph, visited,
neighbor)) {
                return true;
            }
        }

        // Mark the node as fully visited
        visited[node] = 2;
        return false;
    }
}
```

Explanation:
- Time Complexity: O(V + E), where V is the number of courses (nodes) and E is the number of prerequisites (edges).
- Space Complexity: O(V + E) for the graph and visited array.
- This approach is intuitive and leverages DFS for cycle detection.

Approach 3: Iterative Depth-First Search (DFS) with Stack
This approach uses an explicit stack to perform DFS iteratively and detect cycles.

Steps:
1. Build the graph using an adjacency list.
2. Use a `visited` array to track the state of each node.
3. Use a stack to perform DFS iteratively.
4. If a node being visited is encountered again, a cycle exists.

Code:

```java
import java.util.*;

class Solution {
    public boolean canFinish(int numCourses, int[][] prerequisites) {
        // Step 1: Build the graph
        List<List<Integer>> graph = new ArrayList<>();
        for (int i = 0; i < numCourses; i++) {
            graph.add(new ArrayList<>());
        }
        for (int[] edge : prerequisites) {
            int course = edge[0];
            int prerequisite = edge[1];
            graph.get(prerequisite).add(course);
        }

        // Step 2: Initialize visited array
        int[] visited = new int[numCourses];

        // Step 3: Perform DFS for each unvisited node
        for (int i = 0; i < numCourses; i++) {
            if (visited[i] == 0 && hasCycle(graph, visited, i)) {
                return false;
            }
        }

        return true;
    }

    private boolean hasCycle(List<List<Integer>> graph, int[] visited, int
start) {
        Stack<Integer> stack = new Stack<>();
        stack.push(start);

        while (!stack.isEmpty()) {
            int node = stack.peek();

            if (visited[node] == 0) {
                // Mark the node as being visited
                visited[node] = 1;
            } else if (visited[node] == 1) {
                // Cycle detected
                return true;
            } else {
                // Node is fully visited
                stack.pop();
                continue;
            }

            // Push unvisited neighbors onto the stack
            for (int neighbor : graph.get(node)) {
                if (visited[neighbor] != 2) {
                    stack.push(neighbor);
                }
            }
        }

        // Mark the node as fully visited
        visited[start] = 2;
        return false;
    }
}
```

Explanation:
- Time Complexity: O(V + E), where V is the number of courses (nodes) and E is the number of prerequisites (edges).
- Space Complexity: O(V + E) for the graph and stack.
- This approach avoids recursion and is suitable for large graphs.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
BFS (Kahn's)	O(V + E)	O(V + E)	Efficient, no recursion	Requires in-degree calculation
Recursive DFS	O(V + E)	O(V + E)	Intuitive, easy to implement	May cause stack overflow

| Iterative DFS | O(V + E) | O(V + E) | No recursion, explicit stack | Slightly more complex |

Conclusion:
- Use BFS (Kahn's Algorithm) for an efficient and non-recursive solution.
- Use Recursive DFS for simplicity and readability.
- Use Iterative DFS to avoid recursion and handle large graphs.
- All approaches have the same time and space complexity, so the choice depends on the specific use case and constraints.

Pacific Atlantic Water Flow

The Pacific Atlantic Water Flow problem is a matrix traversal problem that involves determining which cells in a matrix can flow water to both the Pacific and Atlantic oceans. This problem can be solved using Depth-First Search (DFS) or Breadth-First Search (BFS) to explore the reachable cells from both oceans.

Problem Definition:
Given an `m x n` matrix of non-negative integers representing the height of each cell, the Pacific Ocean touches the left and top edges of the matrix, and the Atlantic Ocean touches the right and bottom edges. Water can only flow from a cell to adjacent cells (up, down, left, right) with equal or lower height. Return a list of coordinates where water can flow to both the Pacific and Atlantic oceans.

Approach 1: Depth-First Search (DFS) from Ocean Boundaries
This approach uses DFS to explore all cells that can flow water to the Pacific Ocean and all cells that can flow water to the Atlantic Ocean. The intersection of these two sets of cells is the answer.

Steps:
1. Create two boolean matrices, `pacificReachable` and `atlanticReachable`, to track cells that can reach the Pacific and Atlantic oceans, respectively.
2. Perform DFS from all cells on the Pacific boundary (left and top edges) and mark reachable cells in `pacificReachable`.
3. Perform DFS from all cells on the Atlantic boundary (right and bottom edges) and mark reachable cells in `atlanticReachable`.
4. Collect all cells that are marked in both `pacificReachable` and `atlanticReachable`.

Code:

```
import java.util.*;

```

```java
class Solution {
    private int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    public List<List<Integer>> pacificAtlantic(int[][] heights) {
        List<List<Integer>> result = new ArrayList<>();
        if (heights == null || heights.length == 0) return result;

        int m = heights.length, n = heights[0].length;
        boolean[][] pacificReachable = new boolean[m][n];
        boolean[][] atlanticReachable = new boolean[m][n];

        // Perform DFS from Pacific boundary (left and top edges)
        for (int i = 0; i < m; i++) {
            dfs(heights, pacificReachable, i, 0);
        }
        for (int j = 0; j < n; j++) {
            dfs(heights, pacificReachable, 0, j);
        }

        // Perform DFS from Atlantic boundary (right and bottom edges)
        for (int i = 0; i < m; i++) {
            dfs(heights, atlanticReachable, i, n - 1);
        }
        for (int j = 0; j < n; j++) {
            dfs(heights, atlanticReachable, m - 1, j);
        }

        // Collect cells that can reach both oceans
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (pacificReachable[i][j] && atlanticReachable[i][j]) {
                    result.add(Arrays.asList(i, j));
                }
            }
        }

        return result;
    }

    private void dfs(int[][] heights, boolean[][] reachable, int i, int j)
{
        reachable[i][j] = true;
        for (int[] dir : directions) {
            int x = i + dir[0];
            int y = j + dir[1];

            if (x >= 0 && x < heights.length && y >= 0 && y <
heights[0].length &&
                !reachable[x][y] && heights[x][y] >= heights[i][j]) {
                dfs(heights, reachable, x, y);
            }
        }
    }
}
```

Explanation:
- Time Complexity: O(M * N), where M is the number of rows and N is the number of columns.
- Space Complexity: O(M * N) for the `reachable` matrices and the recursion stack.
- This approach is efficient and leverages DFS to explore reachable cells.

Approach 2: Breadth-First Search (BFS) from Ocean Boundaries
This approach uses BFS to explore all cells that can flow water to the Pacific and Atlantic oceans. The intersection of these two sets of cells is the answer.

Steps:
1. Create two boolean matrices, `pacificReachable` and `atlanticReachable`, to track cells that can reach the Pacific and Atlantic oceans, respectively.
2. Perform BFS from all cells on the Pacific boundary (left and top edges) and mark reachable cells in `pacificReachable`.
3. Perform BFS from all cells on the Atlantic boundary (right and bottom edges) and mark reachable cells in `atlanticReachable`.
4. Collect all cells that are marked in both `pacificReachable` and `atlanticReachable`.

Code:

```
import java.util.*;

class Solution {
    private int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    public List<List<Integer>> pacificAtlantic(int[][] heights) {
        List<List<Integer>> result = new ArrayList<>();
        if (heights == null || heights.length == 0) return result;
        int m = heights.length, n = heights[0].length;
        boolean[][] pacificReachable = new boolean[m][n];
        boolean[][] atlanticReachable = new boolean[m][n];
        Queue<int[]> pacificQueue = new LinkedList<>();
        Queue<int[]> atlanticQueue = new LinkedList<>();

        // Initialize queues with Pacific boundary cells
        for (int i = 0; i < m; i++) {
            pacificQueue.add(new int[]{i, 0});
            pacificReachable[i][0] = true;
        }
        for (int j = 0; j < n; j++) {
            pacificQueue.add(new int[]{0, j});
            pacificReachable[0][j] = true;
        }

        // Initialize queues with Atlantic boundary cells
        for (int i = 0; i < m; i++) {
            atlanticQueue.add(new int[]{i, n - 1});
            atlanticReachable[i][n - 1] = true;
        }
        for (int j = 0; j < n; j++) {
            atlanticQueue.add(new int[]{m - 1, j});
            atlanticReachable[m - 1][j] = true;
        }

        // Perform BFS for Pacific
        bfs(heights, pacificQueue, pacificReachable);

        // Perform BFS for Atlantic
        bfs(heights, atlanticQueue, atlanticReachable);

        // Collect cells that can reach both oceans
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (pacificReachable[i][j] && atlanticReachable[i][j]) {
                    result.add(Arrays.asList(i, j));
                }
            }
        }

        return result;
    }
    private void bfs(int[][] heights, Queue<int[]> queue, boolean[][]
reachable) {
        while (!queue.isEmpty()) {
            int[] cell = queue.poll();
            int i = cell[0], j = cell[1];

            for (int[] dir : directions) {
                int x = i + dir[0];
                int y = j + dir[1];

                if (x >= 0 && x < heights.length && y >= 0 && y <
heights[0].length &&
                    !reachable[x][y] && heights[x][y] >= heights[i][j]) {
                    reachable[x][y] = true;
                    queue.add(new int[]{x, y});
                }
            }
        }
    }
}
```

Explanation:
- Time Complexity: O(M * N), where M is the number of rows and N is the number of columns.
- Space Complexity: O(M * N) for the `reachable` matrices and the queues.
- This approach avoids recursion and is suitable for large matrices.

Approach 3: Iterative Depth-First Search (DFS) with Stack
This approach uses an explicit stack to perform DFS iteratively and explore reachable cells.

Steps:
1. Create two boolean matrices, `pacificReachable` and `atlanticReachable`, to track cells that can reach the Pacific and Atlantic oceans, respectively.
2. Perform iterative DFS from all cells on the Pacific boundary (left and top edges) and mark reachable cells in `pacificReachable`.
3. Perform iterative DFS from all cells on the Atlantic boundary (right and bottom edges) and mark reachable cells in `atlanticReachable`.
4. Collect all cells that are marked in both `pacificReachable` and `atlanticReachable`.

Code:

```java
import java.util.*;

class Solution {
    private int[][] directions = {{0, 1}, {1, 0}, {0, -1}, {-1, 0}};

    public List<List<Integer>> pacificAtlantic(int[][] heights) {
        List<List<Integer>> result = new ArrayList<>();
        if (heights == null || heights.length == 0) return result;

        int m = heights.length, n = heights[0].length;
        boolean[][] pacificReachable = new boolean[m][n];
        boolean[][] atlanticReachable = new boolean[m][n];
        Stack<int[]> pacificStack = new Stack<>();
        Stack<int[]> atlanticStack = new Stack<>();

        // Initialize stacks with Pacific boundary cells
        for (int i = 0; i < m; i++) {
            pacificStack.push(new int[]{i, 0});
            pacificReachable[i][0] = true;
        }
        for (int j = 0; j < n; j++) {
            pacificStack.push(new int[]{0, j});
            pacificReachable[0][j] = true;
        }

        // Initialize stacks with Atlantic boundary cells
        for (int i = 0; i < m; i++) {
            atlanticStack.push(new int[]{i, n - 1});
            atlanticReachable[i][n - 1] = true;
        }
        for (int j = 0; j < n; j++) {
            atlanticStack.push(new int[]{m - 1, j});
            atlanticReachable[m - 1][j] = true;
        }

        // Perform iterative DFS for Pacific
        dfs(heights, pacificStack, pacificReachable);

        // Perform iterative DFS for Atlantic
        dfs(heights, atlanticStack, atlanticReachable);

        // Collect cells that can reach both oceans
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (pacificReachable[i][j] && atlanticReachable[i][j]) {
                    result.add(Arrays.asList(i, j));
                }
            }
        }

        return result;
    }

    private void dfs(int[][] heights, Stack<int[]> stack, boolean[][]
reachable) {
        while (!stack.isEmpty()) {
            int[] cell = stack.pop();
            int i = cell[0], j = cell[1];

            for (int[] dir : directions) {
                int x = i + dir[0];
                int y = j + dir[1];

                if (x >= 0 && x < heights.length && y >= 0 && y <
heights[0].length &&
                    !reachable[x][y] && heights[x][y] >= heights[i][j]) {
                    reachable[x][y] = true;
                    stack.push(new int[]{x, y});
                }
            }
        }
    }
}
```

Explanation:
- Time Complexity: O(M * N), where M is the number of rows and N is the number of columns.
- Space Complexity: O(M * N) for the `reachable` matrices and the stacks.
- This approach avoids recursion and is suitable for large matrices.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Recursive DFS	O(M * N)	O(M * N)	Intuitive, easy to implement	May cause stack overflow

| BFS | O(M * N) | O(M * N) | No recursion, suitable for large matrices | Requires explicit queue |
| Iterative DFS | O(M * N) | O(M * N) | No recursion, explicit stack | Slightly more complex |

Conclusion:
- Use Recursive DFS for simplicity and readability.
- Use BFS or Iterative DFS to avoid recursion and handle large matrices.
- All approaches have the same time and space complexity, so the choice depends on the specific use case and constraints.

Number of Islands

The Number of Islands problem is a classic matrix traversal problem that involves counting the number of connected regions (islands) in a 2D grid. This problem can be solved using Depth-First Search (DFS), Breadth-First Search (BFS), or Union-Find (Disjoint Set Union). Below, I'll explain different approaches to solve this problem in Java, along with their explanations.

Problem Definition:
Given a 2D grid map of `'1'`s (land) and `'0'`s (water), count the number of islands. An island is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are surrounded by water.

Approach 1: Depth-First Search (DFS)
This approach uses DFS to explore all connected land cells (forming an island) and marks them as visited.

Steps:
1. Iterate through each cell in the grid.
2. If a cell contains `'1'` (land), increment the island count and perform DFS to mark all connected land cells as visited.
3. Return the total island count.

Code:

```
class Solution {
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0) return 0;

        int numIslands = 0;
```

```java
        int m = grid.length, n = grid[0].length;

        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, n = grid[0].length;

        // Check boundaries and if the cell is land
        if (i < 0 || i >= m || j < 0 || j >= n || grid[i][j] == '0') {
            return;
        }

        // Mark the cell as visited
        grid[i][j] = '0';

        // Explore all four directions
        dfs(grid, i + 1, j);
        dfs(grid, i - 1, j);
        dfs(grid, i, j + 1);
        dfs(grid, i, j - 1);
    }
}
```

Explanation:
- Time Complexity: O(M * N), where M is the number of rows and N is the number of columns.
- Space Complexity: O(M * N) in the worst case (due to recursion stack).
- This approach is simple and intuitive but may cause a stack overflow for very large grids due to recursion.

Approach 2: Breadth-First Search (BFS)
This approach uses BFS to explore all connected land cells (forming an island) and marks them as visited.

Steps:
1. Iterate through each cell in the grid.
2. If a cell contains `'1'` (land), increment the island count and perform BFS to mark all connected land cells as visited.
3. Return the total island count.

Code:

```
import java.util.LinkedList;
import java.util.Queue;

class Solution {
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0) return 0;

        int numIslands = 0;
        int m = grid.length, n = grid[0].length;
        int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1') {
                    numIslands++;
                    grid[i][j] = '0'; // Mark as visited
                    Queue<int[]> queue = new LinkedList<>();
                    queue.add(new int[]{i, j});

                    // Perform BFS
                    while (!queue.isEmpty()) {
                        int[] cell = queue.poll();
                        int x = cell[0], y = cell[1];

                        for (int[] dir : directions) {
                            int newX = x + dir[0];
                            int newY = y + dir[1];

                            if (newX >= 0 && newX < m && newY >= 0 && newY
< n && grid[newX][newY] == '1') {
                                grid[newX][newY] = '0'; // Mark as visited
                                queue.add(new int[]{newX, newY});
                            }
                        }
                    }
                }
            }
        }

        return numIslands;
    }
}
```

Explanation:
- Time Complexity: O(M * N), where M is the number of rows and N is the number of columns.
- Space Complexity: O(min(M, N)) for the queue in the worst case.
- This approach avoids recursion and is suitable for large grids.

Approach 3: Union-Find (Disjoint Set Union)
This approach uses the Union-Find data structure to count the number of islands by connecting adjacent land cells.

Steps:
1. Initialize a parent array to represent each cell as its own parent.
2. Iterate through each cell in the grid.
3. If a cell contains `'1'` (land), check its adjacent cells and union them if they are also land.
4. Count the number of unique parents (islands).

Code:

```
class Solution {
    public int numIslands(char[][] grid) {
        if (grid == null || grid.length == 0) return 0;

        int m = grid.length, n = grid[0].length;
        int[] parent = new int[m * n];
        int[][] directions = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
        int numIslands = 0;

        // Initialize parent array
        for (int i = 0; i < m * n; i++) {
            parent[i] = i;
        }

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (grid[i][j] == '1') {
                    numIslands++;
                    int current = i * n + j;

                    for (int[] dir : directions) {
                        int x = i + dir[0];
                        int y = j + dir[1];

                        if (x >= 0 && x < m && y >= 0 && y < n &&
grid[x][y] == '1') {
                            int neighbor = x * n + y;
                            int parentCurrent = find(parent, current);
                            int parentNeighbor = find(parent, neighbor);

                            if (parentCurrent != parentNeighbor) {
                                parent[parentNeighbor] = parentCurrent;
                                numIslands--;
                            }
                        }
                    }
                }
            }
        }

        return numIslands;
    }

    private int find(int[] parent, int i) {
        if (parent[i] != i) {
            parent[i] = find(parent, parent[i]); // Path compression
        }
        return parent[i];
    }
}
```

Explanation:
- Time Complexity: O(M * N * α(M * N)), where α is the inverse Ackermann function (very small, nearly constant).
- Space Complexity: O(M * N) for the parent array.
- This approach is efficient and leverages the Union-Find data structure for dynamic connectivity.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
DFS	O(M * N)	O(M * N) (recursion)	Simple and intuitive	May cause stack overflow
BFS	O(M * N)	O(min(M, N))	No recursion, suitable for large grids	Requires explicit queue
Union-Find	O(M * N * α(M * N))	O(M * N)	Efficient, dynamic connectivity	Slightly more complex

Conclusion:
- Use DFS for simplicity and readability.
- Use BFS to avoid recursion and handle large grids.
- Use Union-Find for dynamic connectivity and efficient performance.
- All approaches have the same time complexity, so the choice depends on the specific use case and constraints.

Longest Consecutive Sequence

The Longest Consecutive Sequence problem involves finding the length of the longest sequence of consecutive integers in an unsorted array. This problem can be solved using sorting, hash sets, or Union-Find (Disjoint Set Union). Below, I'll explain different approaches to solve this problem in Java, along with their explanations.

Problem Definition:
Given an unsorted array of integers `nums`, return the length of the longest sequence of consecutive integers. For example, for the input `[100, 4, 200, 1, 3, 2]`, the longest consecutive sequence is `[1, 2, 3, 4]`, so the output is `4`.

Approach 1: Sorting
This approach sorts the array and then iterates through it to find the longest consecutive sequence.

Steps:
1. Sort the array.
2. Iterate through the sorted array and count the length of consecutive sequences.
3. Track the maximum length found.

Code:

```java
import java.util.Arrays;
```

```java
class Solution {
    public int longestConsecutive(int[] nums) {
        if (nums == null || nums.length == 0) return 0;

        Arrays.sort(nums);
        int longestStreak = 1;
        int currentStreak = 1;

        for (int i = 1; i < nums.length; i++) {
            if (nums[i] != nums[i - 1]) { // Skip duplicates
                if (nums[i] == nums[i - 1] + 1) {
                    currentStreak++;
                    longestStreak = Math.max(longestStreak, currentStreak);
                } else {
                    currentStreak = 1;
                }
            }
        }

        return longestStreak;
    }
}
```

Explanation:
- Time Complexity: O(N log N), where N is the number of elements in the array (due to sorting).
- Space Complexity: O(1) (if sorting is done in-place).
- This approach is simple but inefficient for large arrays due to the sorting step.

Approach 2: Hash Set
This approach uses a hash set to store all elements and then iterates through the array to find the longest consecutive sequence.

Steps:
1. Add all elements to a hash set for O(1) lookups.
2. Iterate through the array and for each element, check if it is the start of a sequence (i.e., `num - 1` is not in the set).
3. If it is the start of a sequence, count the length of the sequence by incrementing `num` until `num + 1` is not in the set.
4. Track the maximum length found.

Code:

```java
import java.util.HashSet;
import java.util.Set;

class Solution {
    public int longestConsecutive(int[] nums) {
        if (nums == null || nums.length == 0) return 0;

        Set<Integer> numSet = new HashSet<>();
        for (int num : nums) {
            numSet.add(num);
        }

        int longestStreak = 0;

        for (int num : numSet) {
            // Check if the current number is the start of a sequence
            if (!numSet.contains(num - 1)) {
                int currentNum = num;
                int currentStreak = 1;

                // Count the length of the sequence
                while (numSet.contains(currentNum + 1)) {
                    currentNum++;
                    currentStreak++;
                }

                longestStreak = Math.max(longestStreak, currentStreak);
            }
        }

        return longestStreak;
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of elements in the array (each element is processed at most twice).
- Space Complexity: O(N) for the hash set.
- This approach is efficient and avoids the need for sorting.

Approach 3: Union-Find (Disjoint Set Union)
This approach uses the Union-Find data structure to group consecutive elements and find the size of the largest group.

Steps:
1. Initialize a parent array and a size array to represent each element as its own parent and track the size of each group.
2. Add all elements to a hash set for O(1) lookups.
3. For each element, union it with its adjacent elements (`num - 1` and `num + 1`) if they exist in the set.
4. Track the maximum size of any group.

Code:

```java
import java.util.HashMap;
import java.util.Map;

class Solution {
    public int longestConsecutive(int[] nums) {
        if (nums == null || nums.length == 0) return 0;

        Map<Integer, Integer> parent = new HashMap<>();
        Map<Integer, Integer> size = new HashMap<>();
        int longestStreak = 1;

        // Initialize parent and size maps
        for (int num : nums) {
            parent.put(num, num);
            size.put(num, 1);
        }

        for (int num : nums) {
            if (parent.containsKey(num - 1)) {
                union(parent, size, num, num - 1);
            }
            if (parent.containsKey(num + 1)) {
                union(parent, size, num, num + 1);
            }
            longestStreak = Math.max(longestStreak, size.get(find(parent,
num)));
```

```
        }

        return longestStreak;
    }

    private int find(Map<Integer, Integer> parent, int num) {
        if (parent.get(num) != num) {
            parent.put(num, find(parent, parent.get(num))); // Path
compression
        }
        return parent.get(num);
    }

    private void union(Map<Integer, Integer> parent, Map<Integer, Integer>
size, int x, int y) {
        int rootX = find(parent, x);
        int rootY = find(parent, y);

        if (rootX != rootY) {
            // Union by size
            if (size.get(rootX) < size.get(rootY)) {
                parent.put(rootX, rootY);
                size.put(rootY, size.get(rootY) + size.get(rootX));
            } else {
                parent.put(rootY, rootX);
                size.put(rootX, size.get(rootX) + size.get(rootY));
            }
        }
    }
}
```

Explanation:
- Time Complexity: O(N * α(N)), where α is the inverse Ackermann function (very small, nearly constant).
- Space Complexity: O(N) for the parent and size maps.
- This approach is efficient and leverages the Union-Find data structure for dynamic connectivity.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Sorting	O(N log N)	O(1)	Simple and intuitive	Inefficient for large arrays
Hash Set	O(N)	O(N)	Efficient, avoids sorting	Requires extra space for set
Union-Find	O(N * α(N))	O(N)	Efficient, dynamic connectivity	Slightly more complex

Conclusion:
- Use Sorting for simplicity and small arrays.
- Use Hash Set for efficient performance and large arrays.
- Use Union-Find for dynamic connectivity and efficient performance.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Alien Dictionary (Leetcode Premium)

The Alien Dictionary problem is a topological sorting problem where you are given a list of words sorted in an alien language, and you need to determine the order of characters in that language. This problem can be solved using Kahn's Algorithm (BFS-based topological sorting) or DFS-based topological sorting.

Problem Definition:
You are given a list of words sorted lexicographically in an alien language. Return a string of unique letters in the alien language's dictionary order. If no valid order exists, return an empty string.

Example:
Input: `["wrt", "wrf", "er", "ett", "rftt"]`
Output: `"wertf"`

Approach 1: Kahn's Algorithm (BFS-based Topological Sorting)
This approach uses BFS to perform topological sorting and determine the order of characters.

Steps:
1. Build the Graph:
   - Compare adjacent words to find the first differing character and establish edges in the graph.
   - Calculate the in-degree of each character.
2. Perform BFS:
   - Use a queue to process characters with an in-degree of 0.
   - Add characters to the result in the order they are processed.

   - Reduce the in-degree of neighboring characters and enqueue them if their in-degree becomes 0.
3. Check for Validity:
   - If the result contains all unique characters, return the result.
   - Otherwise, return an empty string (indicating a cycle).

Code:

```java
import java.util.*;

class Solution {
    public String alienOrder(String[] words) {
        // Step 1: Initialize graph and in-degree map
        Map<Character, Set<Character>> graph = new HashMap<>();
        Map<Character, Integer> inDegree = new HashMap<>();

        // Initialize in-degree for all unique characters
        for (String word : words) {
            for (char c : word.toCharArray()) {
                inDegree.put(c, 0);
            }
        }

        // Step 2: Build the graph and update in-degrees
        for (int i = 0; i < words.length - 1; i++) {
            String word1 = words[i];
            String word2 = words[i + 1];

            // Check if word2 is a prefix of word1 (invalid case)
            if (word1.length() > word2.length() && word1.startsWith(word2))
{
                return "";
            }

            // Find the first differing character
            for (int j = 0; j < Math.min(word1.length(), word2.length());
j++) {
                char c1 = word1.charAt(j);
                char c2 = word2.charAt(j);

                if (c1 != c2) {
                    // Add edge from c1 to c2
                    if (!graph.containsKey(c1)) {
                        graph.put(c1, new HashSet<>());
                    }
                    if (!graph.get(c1).contains(c2)) {
                        graph.get(c1).add(c2);
                        inDegree.put(c2, inDegree.get(c2) + 1);
                    }
                    break;
                }
            }
        }

        // Step 3: Perform BFS (Kahn's Algorithm)
        Queue<Character> queue = new LinkedList<>();
        for (Map.Entry<Character, Integer> entry : inDegree.entrySet()) {
            if (entry.getValue() == 0) {
                queue.add(entry.getKey());
            }
        }

        StringBuilder result = new StringBuilder();
        while (!queue.isEmpty()) {
            char c = queue.poll();
            result.append(c);

            if (graph.containsKey(c)) {
                for (char neighbor : graph.get(c)) {
                    inDegree.put(neighbor, inDegree.get(neighbor) - 1);
                    if (inDegree.get(neighbor) == 0) {
                        queue.add(neighbor);
                    }
                }
            }
        }

        // Step 4: Check if all characters are included
        if (result.length() != inDegree.size()) {
            return "";
        }

        return result.toString();
    }
}
```

Explanation:
- Time Complexity: O(C), where C is the total number of characters in all words.
- Space Complexity: O(1) (since there are at most 26 characters).
- This approach is efficient and avoids recursion.

Approach 2: DFS-based Topological Sorting
This approach uses DFS to perform topological sorting and determine the order of characters.

Steps:
1. Build the Graph:
   - Compare adjacent words to find the first differing character and establish edges in the graph.
2. Perform DFS:
   - Use a `visited` array to track the state of each character:
     - `0`: Not visited.
     - `1`: Being visited (in the current DFS path).
     - `2`: Fully visited.
   - If a cycle is detected, return an empty string.
3. Construct the Result:
   - Add characters to the result in reverse post-order.

Code:

```java
import java.util.*;

class Solution {
    private Map<Character, Set<Character>> graph = new HashMap<>();
    private Map<Character, Integer> visited = new HashMap<>();
    private StringBuilder result = new StringBuilder();

    public String alienOrder(String[] words) {
        // Step 1: Initialize visited map
        for (String word : words) {
            for (char c : word.toCharArray()) {
                visited.put(c, 0);
            }
        }

        // Step 2: Build the graph
        for (int i = 0; i < words.length - 1; i++) {
            String word1 = words[i];
            String word2 = words[i + 1];
            // Check if word2 is a prefix of word1 (invalid case)
            if (word1.length() > word2.length() && word1.startsWith(word2))
{
                return "";
            }

            // Find the first differing character
            for (int j = 0; j < Math.min(word1.length(), word2.length());
j++) {
                char c1 = word1.charAt(j);
                char c2 = word2.charAt(j);

                if (c1 != c2) {
                    // Add edge from c1 to c2
                    if (!graph.containsKey(c1)) {
                        graph.put(c1, new HashSet<>());
                    }
                    graph.get(c1).add(c2);
                    break;
                }
            }
        }

        // Step 3: Perform DFS
        for (char c : visited.keySet()) {
            if (visited.get(c) == 0 && !dfs(c)) {
                return "";
            }
        }

        return result.reverse().toString();
    }

    private boolean dfs(char c) {
        visited.put(c, 1); // Mark as being visited

        if (graph.containsKey(c)) {
            for (char neighbor : graph.get(c)) {
                if (visited.get(neighbor) == 1) {
                    return false; // Cycle detected
                }
                if (visited.get(neighbor) == 0 && !dfs(neighbor)) {
                    return false;
                }
            }
        }

        visited.put(c, 2); // Mark as fully visited
        result.append(c);
        return true;
    }
}
```

Explanation:
- Time Complexity: O(C), where C is the total number of characters in all words.
- Space Complexity: O(1) (since there are at most 26 characters).
- This approach is intuitive and leverages DFS for cycle detection.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Kahn's (BFS)	O(C)	O(1)	Efficient, no recursion	Requires in-degree calculation
DFS	O(C)	O(1)	Intuitive, cycle detection	May cause stack overflow

Conclusion:
- Use Kahn's Algorithm for an efficient and non-recursive solution.
- Use DFS for simplicity and cycle detection.
- Both approaches have the same time and space complexity, so the choice depends on the specific use case and constraints.

Graph Valid Tree (Leetcode Premium)

The Graph Valid Tree problem involves determining whether a given undirected graph is a valid tree. A graph is a valid tree if it is connected and has no cycles. This problem can be solved using Union-Find (Disjoint Set Union) or Depth-First Search (DFS).

Problem Definition:
Given `n` nodes labeled from `0` to `n - 1` and a list of undirected edges, determine if these edges form a valid tree.

# Example:
Input: `n = 5`, `edges = [[0, 1], [0, 2], [0, 3], [1, 4]]`

Output: `true`

Approach 1: Union-Find (Disjoint Set Union)
This approach uses the Union-Find data structure to detect cycles and ensure the graph is connected.

Steps:
1. Initialize a parent array where each node is its own parent.
2. Iterate through the edges and union the two nodes of each edge.
3. If the two nodes of an edge already have the same parent, a cycle is detected.
4. After processing all edges, ensure that all nodes have the same parent (i.e., the graph is connected).

Code:

```java
class Solution {
    public boolean validTree(int n, int[][] edges) {
        // A valid tree must have exactly n-1 edges
        if (edges.length != n - 1) return false;

        int[] parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i; // Each node is its own parent initially
        }

        for (int[] edge : edges) {
            int x = edge[0];
            int y = edge[1];

            int parentX = find(parent, x);
            int parentY = find(parent, y);

            // If both nodes have the same parent, a cycle is detected
            if (parentX == parentY) {
                return false;
            }

            // Union the two nodes
            parent[parentY] = parentX;
        }

        // Check if all nodes have the same parent (graph is connected)
        int root = find(parent, 0);
        for (int i = 1; i < n; i++) {
            if (find(parent, i) != root) {
                return false;
            }
        }

        return true;
    }

    private int find(int[] parent, int i) {
        if (parent[i] != i) {
            parent[i] = find(parent, parent[i]); // Path compression
        }
        return parent[i];
    }
}
```

Explanation:
- Time Complexity: O(N * α(N)), where N is the number of nodes and α is the inverse Ackermann function (very small, nearly constant).
- Space Complexity: O(N) for the parent array.
- This approach is efficient and leverages the Union-Find data structure for cycle detection and connectivity.

Approach 2: Depth-First Search (DFS)
This approach uses DFS to detect cycles and ensure the graph is connected.

Steps:
1. Build an adjacency list to represent the graph.
2. Use a `visited` array to track visited nodes.
3. Perform DFS starting from node `0` and check for cycles.
4. Ensure all nodes are visited (i.e., the graph is connected).

Code:

```
import java.util.*;

class Solution {
    public boolean validTree(int n, int[][] edges) {
        // A valid tree must have exactly n-1 edges
        if (edges.length != n - 1) return false;
```

```java
        // Build adjacency list
        List<List<Integer>> adjList = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            adjList.add(new ArrayList<>());
        }
        for (int[] edge : edges) {
            adjList.get(edge[0]).add(edge[1]);
            adjList.get(edge[1]).add(edge[0]);
        }

        // Track visited nodes
        boolean[] visited = new boolean[n];

        // Check for cycles
        if (hasCycle(adjList, visited, 0, -1)) {
            return false;
        }

        // Check if all nodes are visited
        for (boolean v : visited) {
            if (!v) {
                return false;
            }
        }

        return true;
    }

    private boolean hasCycle(List<List<Integer>> adjList, boolean[]
visited, int node, int parent) {
        visited[node] = true;

        for (int neighbor : adjList.get(node)) {
            if (!visited[neighbor]) {
                if (hasCycle(adjList, visited, neighbor, node)) {
                    return true;
                }
            } else if (neighbor != parent) {
                // If the neighbor is visited and not the parent, a cycle
exists
                return true;
            }
        }

        return false;
    }
}
```

Explanation:
- Time Complexity: O(N + E), where N is the number of nodes and E is the number of edges.
- Space Complexity: O(N) for the adjacency list and visited array.
- This approach is intuitive and leverages DFS for cycle detection and connectivity.

Approach 3: Breadth-First Search (BFS)
This approach uses BFS to detect cycles and ensure the graph is connected.

Steps:
1. Build an adjacency list to represent the graph.
2. Use a `visited` array to track visited nodes.
3. Perform BFS starting from node `0` and check for cycles.
4. Ensure all nodes are visited (i.e., the graph is connected).

Code:

```
import java.util.*;

class Solution {
    public boolean validTree(int n, int[][] edges) {
        // A valid tree must have exactly n-1 edges
        if (edges.length != n - 1) return false;

        // Build adjacency list
        List<List<Integer>> adjList = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            adjList.add(new ArrayList<>());
        }
        for (int[] edge : edges) {
            adjList.get(edge[0]).add(edge[1]);
            adjList.get(edge[1]).add(edge[0]);
        }

        // Track visited nodes
        boolean[] visited = new boolean[n];
```

```java
        // Perform BFS
        Queue<Integer> queue = new LinkedList<>();
        queue.add(0);
        visited[0] = true;

        while (!queue.isEmpty()) {
            int node = queue.poll();

            for (int neighbor : adjList.get(node)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.add(neighbor);
                }
            }
        }

        // Check if all nodes are visited
        for (boolean v : visited) {
            if (!v) {
                return false;
            }
        }

        return true;
    }
}
```

Explanation:
- Time Complexity: O(N + E), where N is the number of nodes and E is the number of edges.
- Space Complexity: O(N) for the adjacency list and visited array.
- This approach avoids recursion and is suitable for large graphs.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Union-Find	O(N * α(N))	O(N)	Efficient, dynamic connectivity	Slightly more complex
DFS	O(N + E)	O(N)	Intuitive, cycle detection	May cause stack overflow

| BFS | O(N + E) | O(N) | No recursion, suitable for large graphs | Requires explicit queue |

Conclusion:
- Use Union-Find for efficient cycle detection and connectivity.
- Use DFS for simplicity and cycle detection.
- Use BFS to avoid recursion and handle large graphs.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Number of Connected Components in an Undirected Graph (Leetcode Premium)

The Number of Connected Components in an Undirected Graph problem involves determining the number of connected components in an undirected graph. This problem can be solved using Depth-First Search (DFS), Breadth-First Search (BFS), or Union-Find (Disjoint Set Union).

Problem Definition:
Given `n` nodes labeled from `0` to `n - 1` and a list of undirected edges, determine the number of connected components in the graph.

Example:
Input: `n = 5`, `edges = [[0, 1], [1, 2], [3, 4]]`
Output: `2` (There are two connected components: `[0, 1, 2]` and `[3, 4]`.)

Approach 1: Depth-First Search (DFS)
This approach uses DFS to traverse the graph and count the number of connected components.

Steps:
1. Build an adjacency list to represent the graph.
2. Use a `visited` array to track visited nodes.
3. Iterate through all nodes, and for each unvisited node, perform DFS to mark all reachable nodes as visited.
4. Increment the count of connected components for each DFS call.

Code:

```java
import java.util.*;

class Solution {
    public int countComponents(int n, int[][] edges) {
        // Build adjacency list
        List<List<Integer>> adjList = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            adjList.add(new ArrayList<>());
        }
        for (int[] edge : edges) {
            adjList.get(edge[0]).add(edge[1]);
            adjList.get(edge[1]).add(edge[0]);
        }

        // Track visited nodes
        boolean[] visited = new boolean[n];
        int count = 0;

        // Perform DFS for each unvisited node
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {
                dfs(adjList, visited, i);
                count++;
            }
        }

        return count;
    }

    private void dfs(List<List<Integer>> adjList, boolean[] visited, int
node) {
        visited[node] = true;

        for (int neighbor : adjList.get(node)) {
            if (!visited[neighbor]) {
                dfs(adjList, visited, neighbor);
            }
        }
    }
}
```

Explanation:
- Time Complexity: O(N + E), where N is the number of nodes and E is the number of edges.
- Space Complexity: O(N) for the adjacency list and visited array.
- This approach is simple and intuitive but may cause a stack overflow for large graphs due to recursion.

Approach 2: Breadth-First Search (BFS)
This approach uses BFS to traverse the graph and count the number of connected components.

Steps:
1. Build an adjacency list to represent the graph.
2. Use a `visited` array to track visited nodes.
3. Iterate through all nodes, and for each unvisited node, perform BFS to mark all reachable nodes as visited.
4. Increment the count of connected components for each BFS call.

Code:

```java
import java.util.*;

class Solution {
    public int countComponents(int n, int[][] edges) {
        // Build adjacency list
        List<List<Integer>> adjList = new ArrayList<>();
        for (int i = 0; i < n; i++) {
            adjList.add(new ArrayList<>());
        }
        for (int[] edge : edges) {
            adjList.get(edge[0]).add(edge[1]);
            adjList.get(edge[1]).add(edge[0]);
        }

        // Track visited nodes
        boolean[] visited = new boolean[n];
        int count = 0;

        // Perform BFS for each unvisited node
        for (int i = 0; i < n; i++) {
            if (!visited[i]) {
                bfs(adjList, visited, i);
                count++;
            }
        }

        return count;
    }

    private void bfs(List<List<Integer>> adjList, boolean[] visited, int
start) {
        Queue<Integer> queue = new LinkedList<>();
        queue.add(start);
        visited[start] = true;

        while (!queue.isEmpty()) {
            int node = queue.poll();

            for (int neighbor : adjList.get(node)) {
                if (!visited[neighbor]) {
                    visited[neighbor] = true;
                    queue.add(neighbor);
                }
            }
        }
    }
}
```

Explanation:
- Time Complexity: O(N + E), where N is the number of nodes and E is the number of edges.
- Space Complexity: O(N) for the adjacency list and visited array.
- This approach avoids recursion and is suitable for large graphs.

Approach 3: Union-Find (Disjoint Set Union)
This approach uses the Union-Find data structure to group nodes into connected components and count the number of unique parents.

Steps:
1. Initialize a parent array where each node is its own parent.
2. Iterate through the edges and union the two nodes of each edge.
3. Count the number of unique parents (connected components).

Code:

```
class Solution {
    public int countComponents(int n, int[][] edges) {
        int[] parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i; // Each node is its own parent initially
        }

        // Union nodes for each edge
        for (int[] edge : edges) {
            int x = edge[0];
            int y = edge[1];

            int parentX = find(parent, x);
            int parentY = find(parent, y);

            if (parentX != parentY) {
                parent[parentY] = parentX; // Union the two nodes
            }
        }

        // Count the number of unique parents
        int count = 0;
        for (int i = 0; i < n; i++) {
            if (parent[i] == i) {
                count++;
            }
        }

        return count;
    }

    private int find(int[] parent, int i) {
        if (parent[i] != i) {
            parent[i] = find(parent, parent[i]); // Path compression
        }
        return parent[i];
    }
}
```

Explanation:
- Time Complexity: O(N * α(N)), where N is the number of nodes and α is the inverse Ackermann function (very small, nearly constant).
- Space Complexity: O(N) for the parent array.
- This approach is efficient and leverages the Union-Find data structure for dynamic connectivity.

Comparison of Approaches:
| Approach | Time Complexity | Space Complexity | Pros | Cons
|
|-|--||-|-|

DFS	O(N + E)	O(N)	Simple and intuitive	May cause stack overflow
BFS	O(N + E)	O(N)	No recursion, suitable for large graphs	Requires explicit queue
Union-Find	O(N * α(N))	O(N)	Efficient, dynamic connectivity	Slightly more complex

Conclusion:
- Use DFS for simplicity and readability.
- Use BFS to avoid recursion and handle large graphs.
- Use Union-Find for efficient performance and dynamic connectivity.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

(Interval)

Insert Interval

The Insert Interval problem involves inserting a new interval into a list of non-overlapping intervals, merging overlapping intervals if necessary. This problem can be solved using a linear scan or binary search for optimization.

Problem Definition:
Given a list of non-overlapping intervals sorted by their start times, insert a new interval into the list and merge any overlapping intervals. Return the updated list of intervals.

Example:
Input: `intervals = [[1, 3], [6, 9]]`, `newInterval = [2, 5]`
Output: `[[1, 5], [6, 9]]`

Approach 1: Linear Scan
This approach iterates through the intervals and merges the new interval with overlapping intervals.

Steps:
1. Initialize an empty result list.
2. Iterate through the intervals:
   - If the current interval ends before the new interval starts, add it to the result.
   - If the current interval starts after the new interval ends, add the new interval to the result and then add the remaining intervals.
   - If the current interval overlaps with the new interval, merge them by updating the new interval's start and end.

3. Add the new interval to the result if it hasn't been added yet.

Code:

```java
import java.util.*;

class Solution {
    public int[][] insert(int[][] intervals, int[] newInterval) {
        List<int[]> result = new ArrayList<>();
        int i = 0;
        int n = intervals.length;

        // Add all intervals before the new interval
        while (i < n && intervals[i][1] < newInterval[0]) {
            result.add(intervals[i]);
            i++;
        }

        // Merge overlapping intervals
        while (i < n && intervals[i][0] <= newInterval[1]) {
            newInterval[0] = Math.min(newInterval[0], intervals[i][0]);
            newInterval[1] = Math.max(newInterval[1], intervals[i][1]);
            i++;
        }
        result.add(newInterval);

        // Add remaining intervals
        while (i < n) {
            result.add(intervals[i]);
            i++;
        }

        // Convert the result list to a 2D array
        return result.toArray(new int[result.size()][]);
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of intervals.
- Space Complexity: O(N) for the result list.
- This approach is simple and works well for small to medium-sized inputs.

Approach 2: Binary Search for Optimization
This approach uses binary search to find the position where the new interval should be inserted, reducing the number of comparisons.

Steps:
1. Use binary search to find the insertion position for the new interval.
2. Merge overlapping intervals as in the linear scan approach.
3. Add the merged interval and the remaining intervals to the result.

Code:

```
import java.util.*;

class Solution {
    public int[][] insert(int[][] intervals, int[] newInterval) {
        List<int[]> result = new ArrayList<>();
        int n = intervals.length;

        // Find the insertion position using binary search
        int left = 0, right = n;
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (intervals[mid][1] < newInterval[0]) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }

        // Add all intervals before the insertion position
        for (int i = 0; i < left; i++) {
            result.add(intervals[i]);
        }

        // Merge overlapping intervals
        int start = newInterval[0];
        int end = newInterval[1];
        while (left < n && intervals[left][0] <= end) {
            start = Math.min(start, intervals[left][0]);
            end = Math.max(end, intervals[left][1]);
            left++;
        }
        result.add(new int[]{start, end});
        // Add remaining intervals
        for (int i = left; i < n; i++) {
            result.add(intervals[i]);
        }

        // Convert the result list to a 2D array
        return result.toArray(new int[result.size()][]);
    }
}
```

Explanation:
- Time Complexity: O(log N + N), where N is the number of intervals.
- Space Complexity: O(N) for the result list.
- This approach is more efficient for large inputs due to the use of binary search.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Linear Scan	O(N)	O(N)	Simple and intuitive	Less efficient for large inputs
Binary Search	O(log N + N)	O(N)	More efficient for large inputs	Slightly more complex

Conclusion:
- Use Linear Scan for simplicity and small to medium-sized inputs.
- Use Binary Search for optimization and large inputs.
- Both approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Merge Intervals

The Merge Intervals problem involves merging overlapping intervals in a list of intervals. This problem can be solved using sorting and a linear scan or using a Union-Find approach for more complex scenarios.

Problem Definition:
Given a list of intervals, merge all overlapping intervals and return a list of non-overlapping intervals.

Example:

Input: `intervals = [[1, 3], [2, 6], [8, 10], [15, 18]]`
Output: `[[1, 6], [8, 10], [15, 18]]`

Approach 1: Sorting and Linear Scan
This approach sorts the intervals by their start times and then merges overlapping intervals using a linear scan.

Steps:
1. Sort the intervals by their start times.
2. Initialize a result list with the first interval.
3. Iterate through the sorted intervals:
   - If the current interval overlaps with the last interval in the result list, merge them by updating the end time.
   - Otherwise, add the current interval to the result list.
4. Return the result list.

Code:

```
import java.util.*;

class Solution {
    public int[][] merge(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return new int[0][];
        }

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

        List<int[]> result = new ArrayList<>();
        int[] currentInterval = intervals[0];
        result.add(currentInterval);

        for (int[] interval : intervals) {
            int currentEnd = currentInterval[1];
            int nextStart = interval[0];
            int nextEnd = interval[1];

            if (currentEnd >= nextStart) {
                // Overlapping intervals, merge them
                currentInterval[1] = Math.max(currentEnd, nextEnd);
            } else {
                // Non-overlapping interval, add to result
                currentInterval = interval;
                result.add(currentInterval);
            }
        }

        // Convert the result list to a 2D array
        return result.toArray(new int[result.size()][]);
    }
}
```

Explanation:
- Time Complexity: O(N log N), where N is the number of intervals (due to sorting).
- Space Complexity: O(N) for the result list.
- This approach is simple and efficient for most cases.

Approach 2: Union-Find (Disjoint Set Union)
This approach uses the Union-Find data structure to group overlapping intervals and merge them.

Steps:
1. Sort the intervals by their start times.
2. Initialize a parent array where each interval is its own parent.
3. Iterate through the intervals and union overlapping intervals.
4. Group intervals by their parent and merge them.

Code:

```
import java.util.*;

class Solution {
    public int[][] merge(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return new int[0][];
        }

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

        int n = intervals.length;
        int[] parent = new int[n];
        for (int i = 0; i < n; i++) {
            parent[i] = i; // Each interval is its own parent initially
```

```java
        }

        // Union overlapping intervals
        for (int i = 1; i < n; i++) {
            if (intervals[i][0] <= intervals[i - 1][1]) {
                parent[i] = i - 1;
            }
        }

        // Merge intervals based on parent
        List<int[]> result = new ArrayList<>();
        int i = n - 1;
        while (i >= 0) {
            int start = intervals[i][0];
            int end = intervals[i][1];
            while (i >= 0 && parent[i] != i) {
                i--;
                end = Math.max(end, intervals[i][1]);
            }
            result.add(new int[]{start, end});
            i--;
        }

        // Reverse the result to get the correct order
        Collections.reverse(result);

        // Convert the result list to a 2D array
        return result.toArray(new int[result.size()][]);
    }
}
```

Explanation:
- Time Complexity: O(N log N), where N is the number of intervals (due to sorting).
- Space Complexity: O(N) for the parent array and result list.
- This approach is more complex but can be extended to handle more advanced scenarios.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Sorting + Scan	O(N log N)	O(N)	Simple and efficient	Requires sorting

| Union-Find | O(N log N) | O(N) | Can handle advanced scenarios | More complex |

Conclusion:
- Use Sorting + Linear Scan for simplicity and efficiency.
- Use Union-Find for more complex scenarios or when additional operations are needed.
- Both approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Non-overlapping Intervals

The Non-overlapping Intervals problem involves finding the minimum number of intervals to remove to make the rest of the intervals non-overlapping. This problem can be solved using a greedy approach or dynamic programming.

Problem Definition:
Given a list of intervals, find the minimum number of intervals to remove to make the rest of the intervals non-overlapping.

Example:
Input: `intervals = [[1, 2], [2, 3], [3, 4], [1, 3]]`
Output: `1` (Remove `[1, 3]` to make the intervals non-overlapping.)

Approach 1: Greedy Approach (Sort by End Time)
This approach sorts the intervals by their end times and selects the earliest-ending interval at each step, removing overlapping intervals.

Steps:
1. Sort the intervals by their end times.
2. Initialize a variable to track the end of the last selected interval.
3. Iterate through the sorted intervals:
   - If the current interval does not overlap with the last selected interval, select it.
   - Otherwise, increment the count of intervals to remove.
4. Return the count of intervals to remove.

Code:

```java
import java.util.*;

class Solution {
    public int eraseOverlapIntervals(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return 0;
        }

        // Sort intervals by end time
        Arrays.sort(intervals, (a, b) -> Integer.compare(a[1], b[1]));

        int count = 0;
        int lastEnd = intervals[0][1];

        for (int i = 1; i < intervals.length; i++) {
            if (intervals[i][0] < lastEnd) {
                // Overlapping interval, increment count
                count++;
            } else {
                // Non-overlapping interval, update lastEnd
                lastEnd = intervals[i][1];
            }
        }

        return count;
    }
}
```

Explanation:
- Time Complexity: O(N log N), where N is the number of intervals (due to sorting).
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and works well for most cases.

Approach 2: Greedy Approach (Sort by Start Time)
This approach sorts the intervals by their start times and uses a greedy strategy to select non-overlapping intervals.

Steps:
1. Sort the intervals by their start times.
2. Initialize a variable to track the end of the last selected interval.
3. Iterate through the sorted intervals:
   - If the current interval does not overlap with the last selected interval, select it.
   - Otherwise, choose the interval with the smaller end time to minimize overlaps.
4. Return the count of intervals to remove.

Code:

```java
import java.util.*;

class Solution {
    public int eraseOverlapIntervals(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return 0;
        }

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

        int count = 0;
        int lastEnd = intervals[0][1];

        for (int i = 1; i < intervals.length; i++) {
            if (intervals[i][0] < lastEnd) {
                // Overlapping interval, increment count
                count++;
                // Choose the interval with the smaller end time
                lastEnd = Math.min(lastEnd, intervals[i][1]);
            } else {
                // Non-overlapping interval, update lastEnd
                lastEnd = intervals[i][1];
            }
        }

        return count;
    }
}
```

Explanation:
- Time Complexity: O(N log N), where N is the number of intervals (due to sorting).
- Space Complexity: O(1) (no extra space is used).
- This approach is also efficient and provides an alternative way to solve the problem.

Approach 3: Dynamic Programming
This approach uses dynamic programming to find the maximum number of non-overlapping intervals and then calculates the minimum number of intervals to remove.

Steps:
1. Sort the intervals by their start times.
2. Use a dynamic programming array `dp` where `dp[i]` represents the maximum number of non-overlapping intervals up to the `i-th` interval.

3. Iterate through the intervals and update the `dp` array by checking for non-overlapping intervals.
4. The result is the total number of intervals minus the maximum value in the `dp` array.

Code:

```java
import java.util.*;

class Solution {
    public int eraseOverlapIntervals(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return 0;
        }

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

        int n = intervals.length;
        int[] dp = new int[n];
        Arrays.fill(dp, 1); // Each interval is a valid subset by itself

        int maxNonOverlapping = 1;

        for (int i = 1; i < n; i++) {
            for (int j = 0; j < i; j++) {
                if (intervals[j][1] <= intervals[i][0]) {
                    dp[i] = Math.max(dp[i], dp[j] + 1);
                }
            }
            maxNonOverlapping = Math.max(maxNonOverlapping, dp[i]);
        }

        return n - maxNonOverlapping;
    }
}
```

Explanation:
- Time Complexity: O(N^2), where N is the number of intervals (due to nested loops).
- Space Complexity: O(N) for the `dp` array.
- This approach is less efficient but demonstrates the use of dynamic programming for interval problems.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Greedy (Sort by End)	O(N log N)	O(1)	Efficient and simple	Requires sorting
Greedy (Sort by Start)	O(N log N)	O(1)	Alternative greedy strategy	Requires sorting
Dynamic Programming	O(N^2)	O(N)	Demonstrates DP approach	Less efficient for large inputs

Conclusion:
- Use the Greedy Approach (Sort by End Time) for the most efficient and simple solution.
- Use the Greedy Approach (Sort by Start Time) as an alternative greedy strategy.
- Use the Dynamic Programming approach to understand the problem from a DP perspective, but it is less efficient for large inputs.

Meeting Rooms (Leetcode Premium)

The Meeting Rooms problem involves determining if a person can attend all meetings without any overlaps. This problem can be solved using sorting and a linear scan.

Problem Definition:
Given a list of meeting time intervals, determine if a person can attend all meetings.

Example:
Input: `intervals = [[0, 30], [5, 10], [15, 20]]`
Output: `false` (The person cannot attend all meetings due to overlapping intervals.)

Approach 1: Sorting and Linear Scan
This approach sorts the intervals by their start times and checks for overlaps between consecutive intervals.

Steps:
1. Sort the intervals by their start times.
2. Iterate through the sorted intervals and check if the current interval overlaps with the next interval.
3. If any overlap is found, return `false`.
4. If no overlaps are found, return `true`.

Code:

```
import java.util.*;
```

```java
class Solution {
    public boolean canAttendMeetings(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return true;
        }

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

        for (int i = 0; i < intervals.length - 1; i++) {
            if (intervals[i][1] > intervals[i + 1][0]) {
                // Overlapping intervals found
                return false;
            }
        }

        return true;
    }
}
```

Explanation:
- Time Complexity: O(N log N), where N is the number of intervals (due to sorting).
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and works well for most cases.

Approach 2: Brute Force
This approach checks every pair of intervals to see if they overlap.

Steps:
1. Iterate through all pairs of intervals.
2. Check if any two intervals overlap.
3. If any overlap is found, return `false`.
4. If no overlaps are found, return `true`.

Code:

```java
class Solution {
    public boolean canAttendMeetings(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return true;
        }
        for (int i = 0; i < intervals.length; i++) {
            for (int j = i + 1; j < intervals.length; j++) {
                if (overlap(intervals[i], intervals[j])) {
                    return false;
                }
            }
        }

        return true;
    }

    private boolean overlap(int[] interval1, int[] interval2) {
        return interval1[0] < interval2[1] && interval2[0] < interval1[1];
    }
}
```

Explanation:
- Time Complexity: O(N^2), where N is the number of intervals (due to nested loops).
- Space Complexity: O(1) (no extra space is used).
- This approach is less efficient but straightforward.

Approach 3: Using a Min-Heap (Priority Queue)
This approach uses a min-heap to track the end times of meetings and checks for overlaps.

Steps:
1. Sort the intervals by their start times.
2. Use a min-heap to store the end times of meetings.
3. Iterate through the sorted intervals:
   - If the current interval's start time is greater than or equal to the earliest end time in the heap, remove the earliest end time.
   - Add the current interval's end time to the heap.
4. If the size of the heap exceeds 1 at any point, return `false`.
5. Otherwise, return `true`.

Code:

```
import java.util.*;

class Solution {
    public boolean canAttendMeetings(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
```

```java
            return true;
        }

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

        // Min-heap to store end times
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int[] interval : intervals) {
            if (!minHeap.isEmpty() && interval[0] >= minHeap.peek()) {
                // No overlap, remove the earliest end time
                minHeap.poll();
            }
            minHeap.add(interval[1]);
        }

        // If the heap size is 1, no overlaps exist
        return minHeap.size() <= 1;
    }
}
```

Explanation:
- Time Complexity: O(N log N), where N is the number of intervals (due to sorting and heap operations).
- Space Complexity: O(N) for the min-heap.
- This approach is efficient and demonstrates the use of a min-heap for interval problems.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Sorting + Linear Scan	O(N log N)	O(1)	Efficient and simple	Requires sorting
Brute Force	O(N^2)	O(1)	Straightforward	Less efficient for large inputs
Min-Heap	O(N log N)	O(N)	Demonstrates heap usage	Requires extra space

Conclusion:
- Use the Sorting + Linear Scan approach for the most efficient and simple solution.
- Use the Brute Force approach for a straightforward but less efficient solution.
- Use the Min-Heap approach to understand how heaps can be used for interval problems.

Meeting Rooms II (Leetcode Premium)

The Meeting Rooms II problem involves determining the minimum number of meeting rooms required to accommodate all meetings without overlapping. This problem can be solved using sorting and a min-heap (priority queue) or using a chronological ordering approach.

Problem Definition:
Given a list of meeting time intervals, find the minimum number of meeting rooms required.

Example:
Input: `intervals = [[0, 30], [5, 10], [15, 20]]`
Output: `2` (Two meeting rooms are needed: one for `[0, 30]` and another for `[5, 10]` and `[15, 20]`.)

Approach 1: Sorting and Min-Heap (Priority Queue)
This approach sorts the intervals by their start times and uses a min-heap to track the end times of meetings.

Steps:
1. Sort the intervals by their start times.
2. Use a min-heap to store the end times of meetings.
3. Iterate through the sorted intervals:
   - If the current interval's start time is greater than or equal to the earliest end time in the heap, remove the earliest end time (reuse the room).
   - Add the current interval's end time to the heap.
4. The size of the heap at the end represents the minimum number of rooms required.

Code:

```java
import java.util.*;

class Solution {
    public int minMeetingRooms(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return 0;
        }

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

        // Min-heap to store end times
        PriorityQueue<Integer> minHeap = new PriorityQueue<>();

        for (int[] interval : intervals) {
            if (!minHeap.isEmpty() && interval[0] >= minHeap.peek()) {
                // Reuse the room by removing the earliest end time
                minHeap.poll();
            }
            minHeap.add(interval[1]);
        }

        // The size of the heap is the minimum number of rooms required
        return minHeap.size();
    }
}
```

Explanation:
- Time Complexity: O(N log N), where N is the number of intervals (due to sorting and heap operations).
- Space Complexity: O(N) for the min-heap.
- This approach is efficient and widely used for this problem.

Approach 2: Chronological Ordering (Two Pointers)
This approach separates the start and end times into two separate arrays, sorts them, and uses two pointers to determine the number of rooms required.

Steps:
1. Separate the start times and end times into two arrays.
2. Sort both arrays.
3. Use two pointers to iterate through the start and end times:
   - If the current start time is less than the current end time, increment the room count.
   - Otherwise, move the end pointer to the next end time.
4. The room count at the end represents the minimum number of rooms required.

Code:

```java
import java.util.*;

class Solution {
    public int minMeetingRooms(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return 0;
        }
        int n = intervals.length;
        int[] startTimes = new int[n];
        int[] endTimes = new int[n];

        // Separate start and end times
        for (int i = 0; i < n; i++) {
            startTimes[i] = intervals[i][0];
            endTimes[i] = intervals[i][1];
        }

        // Sort start and end times
        Arrays.sort(startTimes);
        Arrays.sort(endTimes);

        int rooms = 0;
        int endPointer = 0;

        for (int startPointer = 0; startPointer < n; startPointer++) {
            if (startTimes[startPointer] < endTimes[endPointer]) {
                // Need a new room
                rooms++;
            } else {
                // Reuse a room
                endPointer++;
            }
        }

        return rooms;
    }
}
```

Explanation:
- Time Complexity: O(N log N), where N is the number of intervals (due to sorting).
- Space Complexity: O(N) for the start and end time arrays.
- This approach is efficient and avoids using a heap.

Approach 3: Brute Force (Time Line Approach)
This approach creates a timeline of all start and end times and calculates the maximum number of overlapping intervals at any point.

Steps:
1. Create a list of all start and end times, marking them as `+1` (start) or `-1` (end).

2. Sort the timeline.
3. Iterate through the timeline, keeping track of the current number of overlapping intervals.
4. The maximum value of overlapping intervals is the minimum number of rooms required.

Code:

```java
import java.util.*;

class Solution {
    public int minMeetingRooms(int[][] intervals) {
        if (intervals == null || intervals.length == 0) {
            return 0;
        }

        List<int[]> timeline = new ArrayList<>();

        // Add start and end times to the timeline
        for (int[] interval : intervals) {
            timeline.add(new int[]{interval[0], 1}); // Start time: +1
            timeline.add(new int[]{interval[1], -1}); // End time: -1
        }

        // Sort the timeline
        timeline.sort((a, b) -> {
            if (a[0] == b[0]) {
                return Integer.compare(a[1], b[1]); // Process end times
first
            }
            return Integer.compare(a[0], b[0]);
        });

        int rooms = 0;
        int maxRooms = 0;

        for (int[] event : timeline) {
            rooms += event[1]; // Update the number of rooms in use
            maxRooms = Math.max(maxRooms, rooms);
        }

        return maxRooms;
    }
}
```

Explanation:
- Time Complexity: O(N log N), where N is the number of intervals (due to sorting).
- Space Complexity: O(N) for the timeline.
- This approach provides a clear visualization of overlapping intervals.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Sorting + Min-Heap	O(N log N)	O(N)	Efficient and widely used	Requires extra space for heap
Chronological Ordering	O(N log N)	O(N)	Efficient, avoids heap	Requires extra space for arrays
Brute Force (Timeline)	O(N log N)	O(N)	Clear visualization of overlaps	Requires extra space for timeline

Conclusion:
- Use the Sorting + Min-Heap approach for an efficient and widely accepted solution.
- Use the Chronological Ordering approach for an alternative efficient solution without a heap.
- Use the Brute Force (Timeline) approach to visualize overlapping intervals clearly.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

(Linked List)

Reverse a Linked List

Reversing a linked list is a classic problem that tests your understanding of pointers and linked list manipulation. Below, I'll explain different approaches to reverse a linked list in Java, along with their explanations.

Problem Definition:
Given the head of a singly linked list, reverse the list and return the new head.

Example:
Input: `1 -> 2 -> 3 -> 4 -> 5`
Output: `5 -> 4 -> 3 -> 2 -> 1`

Approach 1: Iterative Approach
This approach uses three pointers (`prev`, `curr`, and `next`) to reverse the linked list iteratively.

Steps:

1. Initialize `prev` as `null`, `curr` as the head of the list, and `next` as `null`.
2. Traverse the list:
   - Store the next node of `curr` in `next`.
   - Reverse the link of `curr` to point to `prev`.
   - Move `prev` and `curr` one step forward.
3. When `curr` becomes `null`, `prev` will be the new head of the reversed list.

Code:

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

class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        ListNode next = null;

        while (curr != null) {
            next = curr.next; // Store the next node
            curr.next = prev; // Reverse the link
            prev = curr;      // Move prev one step forward
            curr = next;      // Move curr one step forward
        }

        return prev; // prev is the new head of the reversed list
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of nodes in the list.
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and uses constant space.

Approach 2: Recursive Approach
This approach uses recursion to reverse the linked list.

Steps:

1. Base case: If the list is empty or has only one node, return the head.
2. Recursively reverse the rest of the list.
3. Reverse the link between the current node and the next node.
4. Return the new head of the reversed list.

Code:

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

class Solution {
    public ListNode reverseList(ListNode head) {
        // Base case: empty list or single node
        if (head == null || head.next == null) {
            return head;
        }

        // Recursively reverse the rest of the list
        ListNode newHead = reverseList(head.next);

        // Reverse the link between head and head.next
        head.next.next = head;
        head.next = null;

        return newHead;
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of nodes in the list.
- Space Complexity: O(N) for the recursion stack.
- This approach is elegant but uses extra space for the recursion stack.

Approach 3: Using a Stack
This approach uses a stack to reverse the linked list.

Steps:
1. Push all nodes of the linked list onto a stack.

2. Pop nodes from the stack and reconstruct the reversed linked list.

Code:

```java
import java.util.Stack;

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

class Solution {
    public ListNode reverseList(ListNode head) {
        if (head == null) {
            return null;
        }

        Stack<ListNode> stack = new Stack<>();
        ListNode curr = head;

        // Push all nodes onto the stack
        while (curr != null) {
            stack.push(curr);
            curr = curr.next;
        }

        // Pop nodes from the stack to construct the reversed list
        ListNode newHead = stack.pop();
        curr = newHead;

        while (!stack.isEmpty()) {
            curr.next = stack.pop();
            curr = curr.next;
        }

        curr.next = null; // Set the next of the last node to null

        return newHead;
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of nodes in the list.
- Space Complexity: O(N) for the stack.
- This approach is straightforward but uses extra space for the stack.

Approach 4: Using a New Linked List
This approach creates a new linked list by iterating through the original list and adding nodes to the front of the new list.

Steps:
1. Initialize a new linked list with `null` as its head.
2. Traverse the original list:
   - Store the next node of the current node.
   - Add the current node to the front of the new list.
   - Move to the next node in the original list.
3. Return the head of the new list.

Code:

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

class Solution {
    public ListNode reverseList(ListNode head) {
        ListNode newHead = null;
        ListNode curr = head;

        while (curr != null) {
            ListNode next = curr.next; // Store the next node
            curr.next = newHead;      // Add the current node to the front
of the new list
            newHead = curr;           // Update the head of the new list
            curr = next;              // Move to the next node in the
original list
        }

        return newHead;
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of nodes in the list.
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and uses constant space.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Iterative	O(N)	O(1)	Efficient, constant space	Requires careful pointer manipulation
Recursive	O(N)	O(N)	Elegant and simple	Uses extra space for recursion stack
Stack	O(N)	O(N)	Straightforward	Uses extra space for stack
New Linked List	O(N)	O(1)	Efficient, constant space	Creates a new list

Conclusion:
- Use the Iterative Approach for an efficient and space-optimized solution.
- Use the Recursive Approach for a simple and elegant solution, but be mindful of the recursion stack.
- Use the Stack Approach for a straightforward solution, but it uses extra space.
- Use the New Linked List Approach for an alternative iterative solution with constant space.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Detect Cycle in a Linked List
Detecting a cycle in a linked list is a common problem that tests your understanding of pointers and linked list traversal. Below, I'll explain different approaches to detect a cycle in a linked list in Java, along with their explanations.

Problem Definition:
Given the head of a singly linked list, determine if the linked list has a cycle. A cycle occurs if a node's `next` pointer points to a previous node in the list.

Example:
Input: `1 -> 2 -> 3 -> 4 -> 2` (cycle back to node `2`)

Output: `true`

Approach 1: Floyd's Cycle Detection Algorithm (Two Pointers)
This approach uses two pointers, a slow pointer (`slow`) and a fast pointer (`fast`), to detect a cycle.

Steps:
1. Initialize `slow` and `fast` pointers to the head of the list.
2. Move `slow` one step at a time and `fast` two steps at a time.
3. If `fast` meets `slow`, a cycle exists.
4. If `fast` reaches the end of the list (`null`), no cycle exists.

Code:

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

class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) {
            return false;
        }

        ListNode slow = head;
        ListNode fast = head;

        while (fast != null && fast.next != null) {
            slow = slow.next;          // Move slow one step
            fast = fast.next.next;     // Move fast two steps

            if (slow == fast) {
                return true; // Cycle detected
            }
        }

        return false; // No cycle detected
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of nodes in the list.
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and uses constant space.

Approach 2: Using a Hash Set
This approach uses a hash set to track visited nodes.

Steps:
1. Initialize a hash set to store visited nodes.
2. Traverse the list:
   - If the current node is already in the set, a cycle exists.
   - Otherwise, add the node to the set and move to the next node.
3. If the end of the list is reached (`null`), no cycle exists.

Code:

```java
import java.util.HashSet;
import java.util.Set;

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

class Solution {
    public boolean hasCycle(ListNode head) {
        Set<ListNode> visited = new HashSet<>();
        ListNode curr = head;

        while (curr != null) {
            if (visited.contains(curr)) {
                return true; // Cycle detected
            }
            visited.add(curr);
            curr = curr.next;
        }

        return false; // No cycle detected
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of nodes in the list.
- Space Complexity: O(N) for the hash set.
- This approach is simple but uses extra space for the hash set.

Approach 3: Marking Visited Nodes (Modifying the List)
This approach modifies the linked list by marking visited nodes.

Steps:
1. Traverse the list:
   - If the current node's `next` pointer points to a marked node, a cycle exists.
   - Otherwise, mark the current node and move to the next node.
2. If the end of the list is reached (`null`), no cycle exists.

Code:

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

class Solution {
    public boolean hasCycle(ListNode head) {
        ListNode curr = head;

        while (curr != null) {
            if (curr.next == null) {
                return false; // No cycle detected
            }
            if (curr.next == head) {
                return true; // Cycle detected
            }
            ListNode next = curr.next;
            curr.next = head; // Mark the current node
            curr = next;      // Move to the next node
        }
        return false; // No cycle detected
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of nodes in the list.
- Space Complexity: O(1) (no extra space is used).
- This approach modifies the original list, which may not be acceptable in some cases.

Approach 4: Using a Slow Pointer and Reversing the List
This approach uses a slow pointer and reverses the list to detect a cycle.

Steps:
1. Use a slow pointer to traverse the list.
2. Reverse the list as you traverse.
3. If the slow pointer meets the head of the reversed list, a cycle exists.
4. Restore the original list after detection.

Code:

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

class Solution {
    public boolean hasCycle(ListNode head) {
        if (head == null || head.next == null) {
            return false;
        }

        ListNode slow = head;
        ListNode prev = null;

        while (slow != null) {
            if (slow.next == head) {
                return true; // Cycle detected
            }
            ListNode next = slow.next;
            slow.next = prev; // Reverse the link
            prev = slow;
            slow = next;
        }

        return false; // No cycle detected
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of nodes in the list.
- Space Complexity: O(1) (no extra space is used).
- This approach modifies the original list, which may not be acceptable in some cases.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Floyd's Algorithm	O(N)	O(1)	Efficient, constant space	Requires careful pointer manipulation
Hash Set	O(N)	O(N)	Simple and intuitive	Uses extra space for hash set
Marking Visited Nodes	O(N)	O(1)	Constant space	Modifies the original list
Reversing the List	O(N)	O(1)	Constant space	Modifies the original list

Conclusion:
- Use Floyd's Cycle Detection Algorithm for an efficient and space-optimized solution.
- Use the Hash Set Approach for a simple and intuitive solution, but it uses extra space.
- Use the Marking Visited Nodes or Reversing the List approach if modifying the original list is acceptable.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Merge Two Sorted Lists

The Merge Two Sorted Lists problem involves merging two sorted linked lists into a single sorted linked list. This problem can be solved using an iterative approach or a recursive approach.

Problem Definition:
Given the heads of two sorted linked lists, merge them into one sorted linked list and return the head of the merged list.

Example:
Input: `list1 = [1, 2, 4]`, `list2 = [1, 3, 4]`
Output: `[1, 1, 2, 3, 4, 4]`

Approach 1: Iterative Approach
This approach uses a dummy node and a pointer to iteratively merge the two lists.

Steps:
1. Create a dummy node to serve as the head of the merged list.
2. Use a pointer (`curr`) to traverse and build the merged list.
3. Compare the values of the current nodes of both lists and append the smaller value to the merged list.
4. Move the pointer of the list from which the value was taken.
5. Append the remaining nodes of the non-empty list to the merged list.
6. Return the next node of the dummy node as the head of the merged list.

Code:

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

class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        // Create a dummy node to serve as the head of the merged list
        ListNode dummy = new ListNode(-1);
        ListNode curr = dummy;

        // Traverse both lists and merge them
        while (list1 != null && list2 != null) {
            if (list1.val <= list2.val) {
                curr.next = list1;
                list1 = list1.next;
            } else {
                curr.next = list2;
                list2 = list2.next;
            }
            curr = curr.next;
        }

        // Append the remaining nodes of the non-empty list
        if (list1 != null) {
            curr.next = list1;
        } else {
            curr.next = list2;
        }

        // Return the head of the merged list
        return dummy.next;
    }
}
```

Explanation:
- Time Complexity: O(N + M), where N and M are the lengths of the two lists.
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and uses constant space.

Approach 2: Recursive Approach
This approach uses recursion to merge the two lists.

Steps:
1. Base case: If one of the lists is `null`, return the other list.
2. Compare the values of the current nodes of both lists.
3. Recursively merge the remaining lists and set the `next` pointer of the smaller node.
4. Return the smaller node as the current head of the merged list.

Code:

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

class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        // Base case: if one of the lists is null, return the other list
        if (list1 == null) {
            return list2;
        }
        if (list2 == null) {
            return list1;
        }

        // Compare the values of the current nodes
        if (list1.val <= list2.val) {
            list1.next = mergeTwoLists(list1.next, list2);
            return list1;
        } else {
            list2.next = mergeTwoLists(list1, list2.next);
            return list2;
        }
    }
}
```

Explanation:
- Time Complexity: O(N + M), where N and M are the lengths of the two lists.
- Space Complexity: O(N + M) for the recursion stack.
- This approach is elegant but uses extra space for the recursion stack.

Approach 3: In-Place Iterative Approach (Without Dummy Node)
This approach merges the two lists in place without using a dummy node.

Steps:
1. Initialize a pointer (`curr`) to track the current node of the merged list.
2. Compare the values of the current nodes of both lists and link the smaller node to `curr`.
3. Move the pointer of the list from which the value was taken.
4. Move `curr` to the next node.
5. Append the remaining nodes of the non-empty list to the merged list.
6. Return the head of the merged list.

Code:

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int val) {
```

```java
        this.val = val;
    }
}

class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        if (list1 == null) {
            return list2;
        }
        if (list2 == null) {
            return list1;
        }

        // Determine the head of the merged list
        ListNode head;
        if (list1.val <= list2.val) {
            head = list1;
            list1 = list1.next;
        } else {
            head = list2;
            list2 = list2.next;
        }

        ListNode curr = head;

        // Traverse both lists and merge them
        while (list1 != null && list2 != null) {
            if (list1.val <= list2.val) {
                curr.next = list1;
                list1 = list1.next;
            } else {
                curr.next = list2;
                list2 = list2.next;
            }
            curr = curr.next;
        }

        // Append the remaining nodes of the non-empty list
        if (list1 != null) {
            curr.next = list1;
        } else {
            curr.next = list2;
        }
        return head;
    }
}
```

Explanation:
- Time Complexity: O(N + M), where N and M are the lengths of the two lists.
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and avoids using a dummy node.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Iterative (Dummy Node)	O(N + M)	O(1)	Efficient, constant space	Uses a dummy node
Recursive	O(N + M)	O(N + M)	Elegant and simple	Uses extra space for recursion stack
In-Place Iterative	O(N + M)	O(1)	Efficient, no dummy node	Requires careful pointer manipulation

Conclusion:
- Use the Iterative Approach (Dummy Node) for an efficient and straightforward solution.
- Use the Recursive Approach for an elegant solution, but be mindful of the recursion stack.
- Use the In-Place Iterative Approach for an efficient solution without a dummy node.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Merge K Sorted Lists

The Merge K Sorted Lists problem involves merging `k` sorted linked lists into a single sorted linked list. This problem can be solved using a min-heap (priority queue), divide and conquer, or sequential merging.

Problem Definition:
Given an array of `k` sorted linked lists, merge them into one sorted linked list and return the head of the merged list.

Example:
Input: `lists = [[1, 4, 5], [1, 3, 4], [2, 6]]`
Output: `[1, 1, 2, 3, 4, 4, 5, 6]`

Approach 1: Min-Heap (Priority Queue)
This approach uses a min-heap to efficiently merge the `k` sorted lists.

Steps:
1. Create a min-heap and insert the first node of each list into the heap.
2. Initialize a dummy node to serve as the head of the merged list.
3. While the heap is not empty:
   - Remove the smallest node from the heap and add it to the merged list.
   - If the removed node has a next node, insert the next node into the heap.
4. Return the next node of the dummy node as the head of the merged list.

Code:

```java
import java.util.*;

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

class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0) {
            return null;
        }

        // Create a min-heap and add the first node of each list
        PriorityQueue<ListNode> minHeap = new PriorityQueue<>((a, b) ->
a.val - b.val);
        for (ListNode list : lists) {
            if (list != null) {
                minHeap.add(list);
            }
        }

        // Create a dummy node to serve as the head of the merged list
        ListNode dummy = new ListNode(-1);
        ListNode curr = dummy;

        // Merge the lists
        while (!minHeap.isEmpty()) {
            ListNode smallest = minHeap.poll(); // Remove the smallest node
            curr.next = smallest;              // Add it to the merged list
            curr = curr.next;                  // Move the pointer

            // If the smallest node has a next node, add it to the heap
            if (smallest.next != null) {
                minHeap.add(smallest.next);
            }
        }

        return dummy.next; // Return the head of the merged list
    }
}
```

Explanation:
- Time Complexity: O(N log K), where N is the total number of nodes and K is the number of lists.
- Space Complexity: O(K) for the min-heap.
- This approach is efficient and widely used for merging `k` sorted lists.

Approach 2: Divide and Conquer (Merge Pairs)
This approach merges the lists in pairs using a divide-and-conquer strategy.

Steps:
1. Divide the `k` lists into pairs and merge each pair into a single list.
2. Repeat the process until only one list remains.
3. Return the final merged list.

Code:

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

class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0) {
            return null;
        }

        int interval = 1;
        int k = lists.length;

        // Merge lists in pairs until only one list remains
        while (interval < k) {
            for (int i = 0; i < k - interval; i += 2 * interval) {
                lists[i] = mergeTwoLists(lists[i], lists[i + interval]);
            }
            interval *= 2;
        }

        return lists[0]; // Return the final merged list
    }

    private ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        ListNode dummy = new ListNode(-1);
        ListNode curr = dummy;

        while (list1 != null && list2 != null) {
            if (list1.val <= list2.val) {
                curr.next = list1;
                list1 = list1.next;
            } else {
                curr.next = list2;
                list2 = list2.next;
            }
            curr = curr.next;
        }

        // Append the remaining nodes of the non-empty list
        if (list1 != null) {
            curr.next = list1;
        } else {
            curr.next = list2;
        }

        return dummy.next;
    }
}
```

Explanation:
- Time Complexity: O(N log K), where N is the total number of nodes and K is the number of lists.
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and avoids using extra space.

Approach 3: Sequential Merging
This approach merges the lists sequentially, one by one.

Steps:
1. Initialize the merged list as the first list.
2. Iterate through the remaining lists and merge each list with the current merged list.
3. Return the final merged list.

Code:

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

class Solution {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0) {
            return null;
        }

        ListNode mergedList = lists[0];

        // Merge each list with the current merged list
        for (int i = 1; i < lists.length; i++) {
            mergedList = mergeTwoLists(mergedList, lists[i]);
        }

        return mergedList;
    }

    private ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        ListNode dummy = new ListNode(-1);
        ListNode curr = dummy;

        while (list1 != null && list2 != null) {
            if (list1.val <= list2.val) {
                curr.next = list1;
                list1 = list1.next;
            } else {
                curr.next = list2;
                list2 = list2.next;
            }
            curr = curr.next;
        }

        // Append the remaining nodes of the non-empty list
        if (list1 != null) {
            curr.next = list1;
        } else {
            curr.next = list2;
        }

        return dummy.next;
    }
}
```

Explanation:
- Time Complexity: O(K * N), where N is the total number of nodes and K is the number of lists.
- Space Complexity: O(1) (no extra space is used).
- This approach is simple but less efficient for large `K`.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Min-Heap	O(N log K)	O(K)	Efficient and widely used	Uses extra space for heap
Divide and Conquer	O(N log K)	O(1)	Efficient, no extra space	Requires careful implementation
Sequential Merging	O(K * N)	O(1)	Simple and straightforward	Less efficient for large K

Conclusion:
- Use the Min-Heap Approach for an efficient and widely accepted solution.
- Use the Divide and Conquer Approach for an efficient solution without extra space.

- Use the Sequential Merging Approach for a simple solution, but it is less efficient for large `K`.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Remove Nth Node From End Of List

The Remove Nth Node From End Of List problem involves removing the `n-th` node from the end of a singly linked list. This problem can be solved using a two-pass approach, a one-pass approach with two pointers, or a dummy node approach.

Problem Definition:
Given the head of a singly linked list, remove the `n-th` node from the end of the list and return the head of the modified list.

Example:
Input: `head = [1, 2, 3, 4, 5]`, `n = 2`
Output: `[1, 2, 3, 5]` (Remove the second node from the end, which is `4`.)

Approach 1: Two-Pass Approach
This approach involves two passes through the linked list:
1. First pass: Calculate the length of the list.
2. Second pass: Remove the `(length - n + 1)`-th node from the beginning.

Steps:
1. Traverse the list to calculate its length.
2. Calculate the position of the node to remove: `position = length - n`.
3. Traverse the list again and remove the node at the calculated position.

Code:

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

class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        // Calculate the length of the list
        int length = 0;
        ListNode curr = head;
        while (curr != null) {
            length++;
            curr = curr.next;
        }

        // Calculate the position of the node to remove
        int position = length - n;

        // Handle the case where the head needs to be removed
        if (position == 0) {
            return head.next;
        }

        // Traverse the list and remove the node at the calculated position
        curr = head;
        for (int i = 1; i < position; i++) {
            curr = curr.next;
        }
        curr.next = curr.next.next;

        return head;
    }
}
```

Explanation:
- Time Complexity: O(L), where L is the length of the list (two passes).
- Space Complexity: O(1) (no extra space is used).
- This approach is simple and straightforward but requires two passes.

Approach 2: One-Pass Approach with Two Pointers
This approach uses two pointers (`fast` and `slow`) to remove the `n-th` node from the end in a single pass.

Steps:
1. Initialize two pointers, `fast` and `slow`, to the head of the list.
2. Move the `fast` pointer `n` steps ahead.
3. Move both pointers simultaneously until `fast` reaches the end of the list.
4. Remove the node pointed to by `slow`.

Code:

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

class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(-1); // Dummy node to handle edge
cases
        dummy.next = head;
        ListNode fast = dummy;
        ListNode slow = dummy;

        // Move fast pointer n steps ahead
        for (int i = 0; i <= n; i++) {
            fast = fast.next;
        }

        // Move both pointers until fast reaches the end
        while (fast != null) {
            fast = fast.next;
            slow = slow.next;
        }

        // Remove the nth node from the end
        slow.next = slow.next.next;

        return dummy.next;
    }
}
```

Explanation:
- Time Complexity: O(L), where L is the length of the list (single pass).
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and requires only one pass.

Approach 3: Dummy Node and Two Pointers
This approach uses a dummy node to handle edge cases (e.g., removing the head) and two pointers to remove the `n-th` node from the end.

Steps:
1. Create a dummy node and set its `next` pointer to the head of the list.
2. Initialize two pointers, `fast` and `slow`, to the dummy node.
3. Move the `fast` pointer `n` steps ahead.
4. Move both pointers simultaneously until `fast` reaches the end of the list.
5. Remove the node pointed to by `slow`.

Code:

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

class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(-1); // Dummy node to handle edge
cases
        dummy.next = head;
        ListNode fast = dummy;
        ListNode slow = dummy;

        // Move fast pointer n steps ahead
        for (int i = 0; i < n; i++) {
            fast = fast.next;
        }

        // Move both pointers until fast reaches the end
        while (fast.next != null) {
            fast = fast.next;
            slow = slow.next;
        }

        // Remove the nth node from the end
        slow.next = slow.next.next;

        return dummy.next;
    }
}
```

Explanation:
- Time Complexity: O(L), where L is the length of the list (single pass).
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and handles edge cases gracefully.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Two-Pass	O(L)	O(1)	Simple and straightforward	Requires two passes
One-Pass (Two Pointers)	O(L)	O(1)	Efficient, single pass	Requires careful pointer manipulation
Dummy Node + Two Pointers	O(L)	O(1)	Handles edge cases gracefully	Requires a dummy node

Conclusion:
- Use the Two-Pass Approach for simplicity and readability.
- Use the One-Pass Approach with Two Pointers for an efficient solution with a single pass.
- Use the Dummy Node and Two Pointers Approach for handling edge cases gracefully.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Reorder List

The Reorder List problem involves reordering a singly linked list in a specific way: the first node is followed by the last node, the second node is followed by the second-to-last node, and so on. This problem can be solved using a combination of finding the middle of the list, reversing the second half, and merging the two halves.

Problem Definition:
Given the head of a singly linked list, reorder the list such that:
- The first node is followed by the last node.
- The second node is followed by the second-to-last node.
- And so on.

Example:
Input: `head = [1, 2, 3, 4]`
Output: `[1, 4, 2, 3]`

Approach 1: Find Middle, Reverse Second Half, and Merge
This approach involves three steps:
1. Find the middle of the list.

2. Reverse the second half of the list.
3. Merge the first half and the reversed second half.

Steps:
1. Use the slow and fast pointer technique to find the middle of the list.
2. Reverse the second half of the list.
3. Merge the first half and the reversed second half by alternating nodes.

Code:

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

class Solution {
    public void reorderList(ListNode head) {
        if (head == null || head.next == null) {
            return;
        }

        // Step 1: Find the middle of the list
        ListNode slow = head;
        ListNode fast = head;
        while (fast != null && fast.next != null) {
            slow = slow.next;
            fast = fast.next.next;
        }

        // Step 2: Reverse the second half of the list
        ListNode secondHalf = reverseList(slow.next);
        slow.next = null; // Split the list into two halves

        // Step 3: Merge the two halves
        ListNode firstHalf = head;
        while (secondHalf != null) {
            ListNode temp1 = firstHalf.next;
            ListNode temp2 = secondHalf.next;

            firstHalf.next = secondHalf;
            secondHalf.next = temp1;

            firstHalf = temp1;
            secondHalf = temp2;
        }
    }

    private ListNode reverseList(ListNode head) {
        ListNode prev = null;
        ListNode curr = head;
        while (curr != null) {
            ListNode next = curr.next;
            curr.next = prev;
            prev = curr;
            curr = next;
        }
        return prev;
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of nodes in the list.
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and uses constant space.

Approach 2: Using a Stack
This approach uses a stack to store the second half of the list and then merges the first half with the reversed second half.

Steps:
1. Traverse the list to calculate its length.
2. Push the second half of the list onto a stack.
3. Merge the first half of the list with the nodes popped from the stack.

Code:

```
import java.util.Stack;

class ListNode {
    int val;
    ListNode next;
    ListNode(int val) {
```

```java
        this.val = val;
    }
}

class Solution {
    public void reorderList(ListNode head) {
        if (head == null || head.next == null) {
            return;
        }

        // Step 1: Calculate the length of the list
        int length = 0;
        ListNode curr = head;
        while (curr != null) {
            length++;
            curr = curr.next;
        }

        // Step 2: Push the second half of the list onto a stack
        Stack<ListNode> stack = new Stack<>();
        curr = head;
        for (int i = 0; i < length / 2; i++) {
            curr = curr.next;
        }
        ListNode secondHalf = curr.next;
        curr.next = null; // Split the list into two halves

        while (secondHalf != null) {
            stack.push(secondHalf);
            secondHalf = secondHalf.next;
        }

        // Step 3: Merge the first half with the reversed second half
        curr = head;
        while (!stack.isEmpty()) {
            ListNode node = stack.pop();
            node.next = curr.next;
            curr.next = node;
            curr = curr.next.next;
        }
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of nodes in the list.
- Space Complexity: O(N) for the stack.
- This approach is simple but uses extra space for the stack.

Approach 3: Recursive Approach
This approach uses recursion to reorder the list.

Steps:
1. Use recursion to reach the end of the list.
2. During backtracking, reorder the list by linking the current node to the node from the end.

Code:

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

class Solution {
    private ListNode reorderHelper(ListNode head, int length) {
        if (length == 1) {
            return head;
        }
        if (length == 2) {
            return head.next;
        }

        ListNode tail = reorderHelper(head.next, length - 2);
        ListNode temp = tail.next;
        tail.next = temp.next;
        temp.next = head.next;
        head.next = temp;

        return tail;
    }

    public void reorderList(ListNode head) {
        if (head == null || head.next == null) {
            return;
        }

        // Calculate the length of the list
        int length = 0;
        ListNode curr = head;
        while (curr != null) {
            length++;
            curr = curr.next;
        }

        reorderHelper(head, length);
    }
}
```

Explanation:
- Time Complexity: O(N), where N is the number of nodes in the list.
- Space Complexity: O(N) for the recursion stack.
- This approach is elegant but uses extra space for the recursion stack.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Find Middle + Reverse + Merge	O(N)	O(1)	Efficient, constant space	Requires careful pointer manipulation
Using a Stack	O(N)	O(N)	Simple and straightforward	Uses extra space for stack
Recursive	O(N)	O(N)	Elegant and clean	Uses extra space for recursion stack

Conclusion:
- Use the Find Middle, Reverse Second Half, and Merge Approach for an efficient and space-optimized solution.
- Use the Stack Approach for a simple and straightforward solution, but it uses extra space.
- Use the Recursive Approach for an elegant solution, but it uses extra space for the recursion stack.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

(Matrix)

Set Matrix Zeroes

The Set Matrix Zeroes problem involves modifying a matrix such that if an element is `0`, its entire row and column are set to `0`. This problem can be solved using additional space, in-place modification with markers, or optimized in-place modification.

Problem Definition:
Given an `m x n` matrix, if an element is `0`, set its entire row and column to `0`. Do this in-place.

Example:
Input:

```
[
  [1, 1, 1],
  [1, 0, 1],
  [1, 1, 1]
]
```

Output:

```
[
  [1, 0, 1],
  [0, 0, 0],
  [1, 0, 1]
]
```

Approach 1: Using Additional Space
This approach uses additional arrays to track which rows and columns need to be set to `0`.

Steps:
1. Create two arrays, `rows` and `cols`, to track which rows and columns contain `0`.
2. Traverse the matrix and mark the rows and columns that need to be set to `0`.
3. Traverse the matrix again and set the elements to `0` based on the `rows` and `cols` arrays.

Code:

```java
class Solution {
    public void setZeroes(int[][] matrix) {
        int m = matrix.length;
        int n = matrix[0].length;

        boolean[] rows = new boolean[m];
        boolean[] cols = new boolean[n];

        // Step 1: Mark rows and columns that need to be set to 0
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (matrix[i][j] == 0) {
                    rows[i] = true;
                    cols[j] = true;
                }
            }
        }

        // Step 2: Set rows and columns to 0
        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (rows[i] || cols[j]) {
                    matrix[i][j] = 0;
                }
            }
        }
    }
}
```

Explanation:
- Time Complexity: O(m * n), where `m` is the number of rows and `n` is the number of columns.
- Space Complexity: O(m + n) for the `rows` and `cols` arrays.
- This approach is simple but uses extra space.

Approach 2: In-Place Modification with Markers
This approach uses the first row and first column of the matrix to track which rows and columns need to be set to `0`.

Steps:
1. Use the first row and first column as markers to track `0`s.
2. Traverse the matrix and mark the first row and first column if a `0` is found.
3. Traverse the matrix again and set the elements to `0` based on the markers in the first row and first column.
4. Handle the first row and first column separately to avoid overwriting the markers.

Code:

```
class Solution {
    public void setZeroes(int[][] matrix) {
        int m = matrix.length;
        int n = matrix[0].length;

        boolean firstRowHasZero = false;
        boolean firstColHasZero = false;

        // Step 1: Check if the first row and first column need to be set
to 0
        for (int j = 0; j < n; j++) {
            if (matrix[0][j] == 0) {
                firstRowHasZero = true;
                break;
            }
        }
        for (int i = 0; i < m; i++) {
            if (matrix[i][0] == 0) {
                firstColHasZero = true;
                break;
            }
        }

        // Step 2: Use the first row and first column as markers
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (matrix[i][j] == 0) {
                    matrix[i][0] = 0; // Mark the first column
                    matrix[0][j] = 0; // Mark the first row
                }
            }
        }

        // Step 3: Set rows and columns to 0 based on the markers
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                    matrix[i][j] = 0;
                }
            }
        }

        // Step 4: Handle the first row and first column
        if (firstRowHasZero) {
            for (int j = 0; j < n; j++) {
                matrix[0][j] = 0;
            }
        }
        if (firstColHasZero) {
            for (int i = 0; i < m; i++) {
                matrix[i][0] = 0;
            }
        }
    }
}
```

Explanation:
- Time Complexity: O(m * n), where `m` is the number of rows and `n` is the number of columns.
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and uses constant space.

Approach 3: Optimized In-Place Modification
This approach is similar to Approach 2 but optimizes the handling of the first row and first column.

Steps:
1. Use the first row and first column as markers to track `0`s.
2. Traverse the matrix and mark the first row and first column if a `0` is found.
3. Traverse the matrix again and set the elements to `0` based on the markers in the first row and first column.
4. Handle the first row and first column separately to avoid overwriting the markers.

Code:

```
class Solution {
    public void setZeroes(int[][] matrix) {
        int m = matrix.length;
        int n = matrix[0].length;

        boolean firstRowHasZero = false;
        boolean firstColHasZero = false;

        // Step 1: Check if the first row and first column need to be set
to 0
        for (int j = 0; j < n; j++) {
            if (matrix[0][j] == 0) {
                firstRowHasZero = true;
                break;
            }
        }
        for (int i = 0; i < m; i++) {
            if (matrix[i][0] == 0) {
                firstColHasZero = true;
                break;
            }
        }

        // Step 2: Use the first row and first column as markers
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (matrix[i][j] == 0) {
                    matrix[i][0] = 0; // Mark the first column
                    matrix[0][j] = 0; // Mark the first row
                }
            }
        }

        // Step 3: Set rows and columns to 0 based on the markers
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                    matrix[i][j] = 0;
                }
            }
        }

        // Step 4: Handle the first row and first column
        if (firstRowHasZero) {
            for (int j = 0; j < n; j++) {
                matrix[0][j] = 0;
            }
        }
        if (firstColHasZero) {
            for (int i = 0; i < m; i++) {
                matrix[i][0] = 0;
            }
        }
    }
}
```

Explanation:
- Time Complexity: O(m * n), where `m` is the number of rows and `n` is the number of columns.
- Space Complexity: O(1) (no extra space is used).
- This approach is efficient and uses constant space.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Additional Space	O(m * n)	O(m + n)	Simple and straightforward	Uses extra space
In-Place with Markers	O(m * n)	O(1)	Efficient, constant space	Requires careful handling of markers
Optimized In-Place	O(m * n)	O(1)	Efficient, constant space	Requires careful handling of markers

Conclusion:
- Use the Additional Space Approach for simplicity and readability.
- Use the In-Place Modification with Markers Approach for an efficient and space-optimized solution.
- Use the Optimized In-Place Modification Approach for an efficient solution with careful handling of markers.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Spiral Matrix

The Spiral Matrix problem involves traversing a 2D matrix in a spiral order and returning the elements in that order. This problem can be solved using layer-by-layer traversal or simulation of the spiral order.

Problem Definition:
Given an `m x n` matrix, return all elements of the matrix in spiral order.

Example:
Input:

```
[
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
]
```

Output: `[1, 2, 3, 6, 9, 8, 7, 4, 5]`

Approach 1: Layer-by-Layer Traversal
This approach treats the matrix as a series of layers and traverses each layer in a spiral order.

Steps:
1. Define the boundaries of the matrix: `topRow`, `bottomRow`, `leftCol`, and `rightCol`.
2. Traverse the top row from left to right.
3. Traverse the right column from top to bottom.
4. If the bottom row is still within bounds, traverse it from right to left.
5. If the left column is still within bounds, traverse it from bottom to top.
6. Move the boundaries inward and repeat the process until all layers are traversed.

Code:

```java
import java.util.ArrayList;
import java.util.List;

class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> result = new ArrayList<>();
        if (matrix == null || matrix.length == 0) {
            return result;
        }

        int topRow = 0, bottomRow = matrix.length - 1;
        int leftCol = 0, rightCol = matrix[0].length - 1;

        while (topRow <= bottomRow && leftCol <= rightCol) {
            // Traverse from left to right along the top row
            for (int j = leftCol; j <= rightCol; j++) {
                result.add(matrix[topRow][j]);
            }
            topRow++;

            // Traverse from top to bottom along the right column
            for (int i = topRow; i <= bottomRow; i++) {
                result.add(matrix[i][rightCol]);
            }
            rightCol--;

            // Traverse from right to left along the bottom row (if still
within bounds)
            if (topRow <= bottomRow) {
                for (int j = rightCol; j >= leftCol; j--) {
                    result.add(matrix[bottomRow][j]);
                }
                bottomRow--;
            }

            // Traverse from bottom to top along the left column (if still
within bounds)
            if (leftCol <= rightCol) {
                for (int i = bottomRow; i >= topRow; i--) {
                    result.add(matrix[i][leftCol]);
                }
                leftCol++;
            }
        }

        return result;
    }
}
```

Explanation:
- Time Complexity: O(m * n), where `m` is the number of rows and `n` is the number of columns.
- Space Complexity: O(1) (excluding the output list).
- This approach is efficient and uses constant space.

Approach 2: Simulation of Spiral Order
This approach simulates the spiral order by maintaining a direction and updating the boundaries as we traverse the matrix.

Steps:
1. Define the boundaries of the matrix: `topRow`, `bottomRow`, `leftCol`, and `rightCol`.
2. Use a direction variable to track the current direction of traversal.
3. Traverse the matrix in the current direction and update the boundaries as needed.
4. Repeat until all elements are traversed.

Code:

```
import java.util.ArrayList;
import java.util.List;
```

```java
class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> result = new ArrayList<>();
        if (matrix == null || matrix.length == 0) {
            return result;
        }

        int topRow = 0, bottomRow = matrix.length - 1;
        int leftCol = 0, rightCol = matrix[0].length - 1;

        while (topRow <= bottomRow && leftCol <= rightCol) {
            // Traverse from left to right along the top row
            for (int j = leftCol; j <= rightCol; j++) {
                result.add(matrix[topRow][j]);
            }
            topRow++;

            // Traverse from top to bottom along the right column
            for (int i = topRow; i <= bottomRow; i++) {
                result.add(matrix[i][rightCol]);
            }
            rightCol--;

            // Traverse from right to left along the bottom row (if still
within bounds)
            if (topRow <= bottomRow) {
                for (int j = rightCol; j >= leftCol; j--) {
                    result.add(matrix[bottomRow][j]);
                }
                bottomRow--;
            }

            // Traverse from bottom to top along the left column (if still
within bounds)
            if (leftCol <= rightCol) {
                for (int i = bottomRow; i >= topRow; i--) {
                    result.add(matrix[i][leftCol]);
                }
                leftCol++;
            }
        }

        return result;
    }
}
```

Explanation:
- Time Complexity: O(m * n), where `m` is the number of rows and `n` is the number of columns.
- Space Complexity: O(1) (excluding the output list).
- This approach is efficient and uses constant space.

Approach 3: Using Direction Vectors
This approach uses direction vectors to simulate the spiral order.

Steps:
1. Define direction vectors for right, down, left, and up.
2. Use a visited matrix to track visited elements.
3. Traverse the matrix in the current direction and change direction when the boundary or a visited element is encountered.

Code:

```
import java.util.ArrayList;
import java.util.List;

class Solution {
    public List<Integer> spiralOrder(int[][] matrix) {
        List<Integer> result = new ArrayList<>();
        if (matrix == null || matrix.length == 0) {
            return result;
        }

        int m = matrix.length, n = matrix[0].length;
        boolean[][] visited = new boolean[m][n];
        int[] dr = {0, 1, 0, -1}; // Direction vectors for right, down,
left, up
        int[] dc = {1, 0, -1, 0};
        int direction = 0; // Start with right direction
        int row = 0, col = 0;

        for (int i = 0; i < m * n; i++) {
            result.add(matrix[row][col]);
            visited[row][col] = true;

            int nextRow = row + dr[direction];
            int nextCol = col + dc[direction];

            // Change direction if out of bounds or visited
            if (nextRow < 0 || nextRow >= m || nextCol < 0 || nextCol >= n
|| visited[nextRow][nextCol]) {
                direction = (direction + 1) % 4; // Change direction
                nextRow = row + dr[direction];
                nextCol = col + dc[direction];
            }

            row = nextRow;
            col = nextCol;
        }

        return result;
    }
}
```

Explanation:
- Time Complexity: O(m * n), where `m` is the number of rows and `n` is the number of columns.
- Space Complexity: O(m * n) for the `visited` matrix.
- This approach is intuitive but uses extra space for the `visited` matrix.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Layer-by-Layer	O(m * n)	O(1)	Efficient, constant space	Requires careful boundary handling
Simulation	O(m * n)	O(1)	Efficient, constant space	Requires careful boundary handling
Direction Vectors	O(m * n)	O(m * n)	Intuitive and clean	Uses extra space for `visited` matrix

Conclusion:
- Use the Layer-by-Layer Traversal or Simulation Approach for an efficient and space-optimized solution.
- Use the Direction Vectors Approach for an intuitive solution, but it uses extra space for the `visited` matrix.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Rotate Image

The Rotate Image problem involves rotating an `n x n` 2D matrix (image) by 90 degrees clockwise. This problem can be solved using transpose and reverse, layer-by-layer rotation, or cyclic replacement.

Problem Definition:
Given an `n x n` 2D matrix, rotate it by 90 degrees clockwise in-place.

Example:
Input:

```
[
  [1, 2, 3],
  [4, 5, 6],
  [7, 8, 9]
]
```

Output:

```
[
  [7, 4, 1],
  [8, 5, 2],
  [9, 6, 3]
]
```

Approach 1: Transpose and Reverse
This approach involves two steps:
1. Transpose the matrix (swap rows and columns).
2. Reverse each row of the transposed matrix.

Steps:
1. Transpose the matrix: Swap `matrix[i][j]` with `matrix[j][i]`.
2. Reverse each row of the transposed matrix.

Code:

```
class Solution {
    public void rotate(int[][] matrix) {
        int n = matrix.length;

        // Step 1: Transpose the matrix
        for (int i = 0; i < n; i++) {
            for (int j = i; j < n; j++) {
                int temp = matrix[i][j];
                matrix[i][j] = matrix[j][i];
                matrix[j][i] = temp;
            }
        }

        // Step 2: Reverse each row
        for (int i = 0; i < n; i++) {
            for (int j = 0; j < n / 2; j++) {
                int temp = matrix[i][j];
                matrix[i][j] = matrix[i][n - 1 - j];
                matrix[i][n - 1 - j] = temp;
            }
        }
    }
}
```

Explanation:
- Time Complexity: O(n^2), where `n` is the size of the matrix.
- Space Complexity: O(1) (in-place rotation).
- This approach is efficient and uses constant space.

Approach 2: Layer-by-Layer Rotation
This approach rotates the matrix layer by layer, starting from the outermost layer and moving inward.

Steps:
1. Rotate the elements of each layer in groups of four.
2. Move to the next inner layer and repeat the process.

Code:

```
class Solution {
    public void rotate(int[][] matrix) {
        int n = matrix.length;

        for (int layer = 0; layer < n / 2; layer++) {
            int first = layer;
            int last = n - 1 - layer;
            for (int i = first; i < last; i++) {
                int offset = i - first;

                // Save the top element
                int top = matrix[first][i];

                // Move left to top
                matrix[first][i] = matrix[last - offset][first];

                // Move bottom to left
                matrix[last - offset][first] = matrix[last][last - offset];

                // Move right to bottom
                matrix[last][last - offset] = matrix[i][last];

                // Move top to right
                matrix[i][last] = top;
            }
        }
    }
}
```

Explanation:
- Time Complexity: O(n^2), where `n` is the size of the matrix.
- Space Complexity: O(1) (in-place rotation).
- This approach is efficient and uses constant space.

Approach 3: Cyclic Replacement
This approach rotates the matrix by cyclically replacing elements in groups of four.

Steps:
1. For each element at position `(i, j)`, calculate its new position after rotation.
2. Perform the cyclic replacement for all elements in the matrix.

Code:

```
class Solution {
    public void rotate(int[][] matrix) {
        int n = matrix.length;

        for (int i = 0; i < (n + 1) / 2; i++) {
            for (int j = 0; j < n / 2; j++) {
                int temp = matrix[n - 1 - j][i];
                matrix[n - 1 - j][i] = matrix[n - 1 - i][n - 1 - j];
                matrix[n - 1 - i][n - 1 - j] = matrix[j][n - 1 - i];
                matrix[j][n - 1 - i] = matrix[i][j];
                matrix[i][j] = temp;
            }
        }
    }
}
```

Explanation:
- Time Complexity: O(n^2), where `n` is the size of the matrix.
- Space Complexity: O(1) (in-place rotation).
- This approach is efficient and uses constant space.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Transpose and Reverse	O(n^2)	O(1)	Simple and efficient	Requires two passes
Layer-by-Layer	O(n^2)	O(1)	Efficient, intuitive	Requires careful indexing
Cyclic Replacement	O(n^2)	O(1)	Efficient, in-place rotation	Requires careful indexing

Conclusion:
- Use the Transpose and Reverse Approach for a simple and efficient solution.
- Use the Layer-by-Layer Rotation Approach for an intuitive and efficient solution.
- Use the Cyclic Replacement Approach for an efficient in-place rotation.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Word Search

The Word Search problem involves determining whether a given word exists in a 2D grid of characters. The word can be constructed from letters of sequentially adjacent cells, where "adjacent" cells are those horizontally or vertically neighboring. This problem can be solved using backtracking or depth-first search (DFS).

Problem Definition:

Given a 2D board of characters and a word, determine if the word exists in the grid. The word can be constructed from letters of sequentially adjacent cells, where "adjacent" cells are those horizontally or vertically neighboring. The same letter cell may not be used more than once.

Example:
Input:

board = [
  ['A', 'B', 'C', 'E'],
  ['S', 'F', 'C', 'S'],
  ['A', 'D', 'E', 'E']
]
word = "ABCCED"

Output: `true` (The word "ABCCED" exists in the grid.)

Approach 1: Backtracking (DFS)
This approach uses backtracking to explore all possible paths in the grid to find the word.

Steps:
1. Iterate through each cell in the grid.
2. For each cell, start a DFS to check if the word can be formed starting from that cell.
3. During DFS, mark the current cell as visited to avoid reuse.
4. If the word is found, return `true`; otherwise, backtrack and try other paths.

Code:

```
class Solution {
    public boolean exist(char[][] board, String word) {
        if (board == null || board.length == 0 || word == null ||
word.length() == 0) {
            return false;
        }

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

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (dfs(board, word, i, j, 0)) {
                    return true;
                }
            }
        }

        return false;
    }

    private boolean dfs(char[][] board, String word, int i, int j, int
index) {
        if (index == word.length()) {
            return true; // Word found
        }

        if (i < 0 || i >= board.length || j < 0 || j >= board[0].length ||
board[i][j] != word.charAt(index)) {
            return false; // Out of bounds or mismatch
        }

        char temp = board[i][j];
        board[i][j] = '#'; // Mark the cell as visited

        // Explore all four directions
        boolean found = dfs(board, word, i + 1, j, index + 1) ||
                        dfs(board, word, i - 1, j, index + 1) ||
                        dfs(board, word, i, j + 1, index + 1) ||
                        dfs(board, word, i, j - 1, index + 1);

        board[i][j] = temp; // Backtrack (unmark the cell)
        return found;
    }
}
```

Explanation:
- Time Complexity: O(m * n * 4^L), where `m` is the number of rows, `n` is the number of columns, and `L` is the length of the word.
- Space Complexity: O(L) for the recursion stack.
- This approach is efficient and uses backtracking to explore all possible paths.

Approach 2: Optimized Backtracking with Early Exit
This approach is similar to Approach 1 but includes optimizations to reduce unnecessary computations.

Steps:
1. Iterate through each cell in the grid.
2. For each cell, start a DFS to check if the word can be formed starting from that cell.

3. During DFS, mark the current cell as visited to avoid reuse.
4. If the word is found, return `true`; otherwise, backtrack and try other paths.
5. Use early exit to stop further exploration if the word cannot be formed from the current path.

Code:

```
class Solution {
    public boolean exist(char[][] board, String word) {
        if (board == null || board.length == 0 || word == null ||
word.length() == 0) {
            return false;
        }

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

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (dfs(board, word, i, j, 0)) {
                    return true;
                }
            }
        }

        return false;
    }

    private boolean dfs(char[][] board, String word, int i, int j, int
index) {
        if (index == word.length()) {
            return true; // Word found
        }

        if (i < 0 || i >= board.length || j < 0 || j >= board[0].length ||
board[i][j] != word.charAt(index)) {
            return false; // Out of bounds or mismatch
        }

        char temp = board[i][j];
        board[i][j] = '#'; // Mark the cell as visited

        // Explore all four directions
        boolean found = dfs(board, word, i + 1, j, index + 1) ||
                        dfs(board, word, i - 1, j, index + 1) ||
                        dfs(board, word, i, j + 1, index + 1) ||
                        dfs(board, word, i, j - 1, index + 1);

        board[i][j] = temp; // Backtrack (unmark the cell)
        return found;
    }
}
```

Explanation:
- Time Complexity: O(m * n * 4^L), where `m` is the number of rows, `n` is the number of columns, and `L` is the length of the word.
- Space Complexity: O(L) for the recursion stack.
- This approach is efficient and uses backtracking with early exit to reduce unnecessary computations.

Approach 3: Using a Visited Matrix
This approach uses a separate `visited` matrix to track visited cells during DFS.

Steps:
1. Create a `visited` matrix to track visited cells.
2. Iterate through each cell in the grid.
3. For each cell, start a DFS to check if the word can be formed starting from that cell.
4. During DFS, mark the current cell as visited in the `visited` matrix.
5. If the word is found, return `true`; otherwise, backtrack and try other paths.

Code:

```
class Solution {
    public boolean exist(char[][] board, String word) {
        if (board == null || board.length == 0 || word == null ||
word.length() == 0) {
            return false;
        }

        int m = board.length;
        int n = board[0].length;
        boolean[][] visited = new boolean[m][n];

        for (int i = 0; i < m; i++) {
            for (int j = 0; j < n; j++) {
                if (dfs(board, word, i, j, 0, visited)) {
                    return true;
                }
            }
        }

        return false;
    }

    private boolean dfs(char[][] board, String word, int i, int j, int
index, boolean[][] visited) {
        if (index == word.length()) {
            return true; // Word found
        }

        if (i < 0 || i >= board.length || j < 0 || j >= board[0].length ||
visited[i][j] || board[i][j] != word.charAt(index)) {
            return false; // Out of bounds, already visited, or mismatch
        }

        visited[i][j] = true; // Mark the cell as visited

        // Explore all four directions
        boolean found = dfs(board, word, i + 1, j, index + 1, visited) ||
                        dfs(board, word, i - 1, j, index + 1, visited) ||
                        dfs(board, word, i, j + 1, index + 1, visited) ||
                        dfs(board, word, i, j - 1, index + 1, visited);

        visited[i][j] = false; // Backtrack (unmark the cell)
        return found;
    }
}
```

Explanation:
- Time Complexity: O(m * n * 4^L), where `m` is the number of rows, `n` is the number of columns, and `L` is the length of the word.
- Space Complexity: O(m * n) for the `visited` matrix.
- This approach is intuitive but uses extra space for the `visited` matrix.

Comparison of Approaches:
| Approach | Time Complexity | Space Complexity | Pros | Cons |
||--||-|-|

Backtracking (DFS)	O(m * n * 4^L)	O(L)	Efficient, uses backtracking	Requires careful implementation
Optimized Backtracking	O(m * n * 4^L)	O(L)	Efficient, early exit	Requires careful implementation
Visited Matrix	O(m * n * 4^L)	O(m * n)	Intuitive and clean	Uses extra space for `visited` matrix

Conclusion:
- Use the Backtracking (DFS) Approach for an efficient and space-optimized solution.
- Use the Optimized Backtracking Approach for an efficient solution with early exit.
- Use the Visited Matrix Approach for an intuitive solution, but it uses extra space for the `visited` matrix.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

(String)

Longest Substring Without Repeating Characters

The Longest Substring Without Repeating Characters problem involves finding the length of the longest substring in a given string that does not contain any repeating characters. This problem can be solved using a sliding window approach, a hash map, or a boolean array.

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

Example:
Input: `s = "abcabcbb"`
Output: `3` (The longest substring without repeating characters is "abc".)

Approach 1: Sliding Window with Two Pointers
This approach uses two pointers (`left` and `right`) to represent the current window and a hash set to track unique characters.

Steps:
1. Initialize two pointers, `left` and `right`, to represent the current window.
2. Use a hash set to track unique characters in the current window.
3. Move the `right` pointer to expand the window and add characters to the set.
4. If a duplicate character is found, move the `left` pointer to shrink the window and remove characters from the set.
5. Track the maximum length of the window.

Code:

```java
import java.util.HashSet;
import java.util.Set;

class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length();
        Set<Character> set = new HashSet<>();
        int left = 0, right = 0;
        int maxLength = 0;

        while (right < n) {
            if (!set.contains(s.charAt(right))) {
                set.add(s.charAt(right));
                maxLength = Math.max(maxLength, right - left + 1);
                right++;
            } else {
                set.remove(s.charAt(left));
                left++;
            }
        }

        return maxLength;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the string.
- Space Complexity: O(min(N, M)), where `M` is the size of the character set (e.g., 26 for lowercase English letters).
- This approach is efficient and uses a sliding window to track unique characters.

Approach 2: Sliding Window with Hash Map
This approach uses a hash map to store the last index of each character and two pointers to represent the current window.

Steps:
1. Initialize two pointers, `left` and `right`, to represent the current window.
2. Use a hash map to store the last index of each character.
3. Move the `right` pointer to expand the window and update the last index of the character in the map.

4. If a duplicate character is found, move the `left` pointer to the right of the last occurrence of the character.
5. Track the maximum length of the window.

Code:

```java
import java.util.HashMap;
import java.util.Map;

class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length();
        Map<Character, Integer> map = new HashMap<>();
        int left = 0, right = 0;
        int maxLength = 0;

        while (right < n) {
            char currentChar = s.charAt(right);
            if (map.containsKey(currentChar)) {
                left = Math.max(left, map.get(currentChar) + 1);
            }
            map.put(currentChar, right);
            maxLength = Math.max(maxLength, right - left + 1);
            right++;
        }

        return maxLength;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the string.
- Space Complexity: O(min(N, M)), where `M` is the size of the character set.
- This approach is efficient and uses a hash map to track the last index of each character.

Approach 3: Sliding Window with Boolean Array
This approach uses a boolean array to track the presence of characters and two pointers to represent the current window.

Steps:
1. Initialize two pointers, `left` and `right`, to represent the current window.
2. Use a boolean array to track the presence of characters in the current window.
3. Move the `right` pointer to expand the window and mark the character as present in the array.

4. If a duplicate character is found, move the `left` pointer to shrink the window and mark the character as absent in the array.
5. Track the maximum length of the window.

Code:

```java
class Solution {
    public int lengthOfLongestSubstring(String s) {
        int n = s.length();
        boolean[] visited = new boolean[256]; // Assuming ASCII characters
        int left = 0, right = 0;
        int maxLength = 0;

        while (right < n) {
            char currentChar = s.charAt(right);
            if (!visited[currentChar]) {
                visited[currentChar] = true;
                maxLength = Math.max(maxLength, right - left + 1);
                right++;
            } else {
                visited[s.charAt(left)] = false;
                left++;
            }
        }

        return maxLength;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the string.
- Space Complexity: O(1) (constant space for the boolean array).
- This approach is efficient and uses a boolean array to track the presence of characters.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Sliding Window (Set)	O(N)	O(min(N, M))	Simple and efficient	Uses extra space for set
Sliding Window (Map)	O(N)	O(min(N, M))	Efficient, tracks last index	Uses extra space for map

| Sliding Window (Array) | O(N) | O(1) | Efficient, constant space | Limited to fixed character set |

Conclusion:
- Use the Sliding Window with Set Approach for a simple and efficient solution.
- Use the Sliding Window with Map Approach for an efficient solution that tracks the last index of each character.
- Use the Sliding Window with Array Approach for an efficient solution with constant space.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

<u>Longest Repeating Character Replacement</u>

The Longest Repeating Character Replacement problem involves finding the length of the longest substring that can be formed by replacing at most `k` characters in the string with any other character. This problem can be solved using a sliding window approach with a frequency map.

Problem Definition:
Given a string `s` and an integer `k`, find the length of the longest substring that can be formed by replacing at most `k` characters in the string with any other character.

Example:
Input: `s = "AABABBA"`, `k = 1`
Output: `4` (Replace the 'B' at index 3 to get "AAAABBA", and the longest substring is "AAAA".)

Approach 1: Sliding Window with Frequency Map
This approach uses a sliding window to maintain a window of characters and a frequency map to track the count of each character in the window.

Steps:
1. Initialize two pointers, `left` and `right`, to represent the current window.
2. Use a frequency map to track the count of each character in the window.
3. Move the `right` pointer to expand the window and update the frequency map.
4. Calculate the maximum count of any character in the current window.
5. If the number of characters to replace (`windowSize - maxCount`) exceeds `k`, move the `left` pointer to shrink the window.
6. Track the maximum length of the window.

Code:

```
class Solution {
    public int characterReplacement(String s, int k) {
        int n = s.length();
        int[] freq = new int[26]; // Frequency map for characters
        int left = 0, right = 0;
        int maxCount = 0; // Maximum count of any character in the current
window
        int maxLength = 0;

        while (right < n) {
            char currentChar = s.charAt(right);
            freq[currentChar - 'A']++; // Update frequency map
            maxCount = Math.max(maxCount, freq[currentChar - 'A']); //
Update maxCount

            // If the number of characters to replace exceeds k, shrink the
window
            if (right - left + 1 - maxCount > k) {
                freq[s.charAt(left) - 'A']--; // Decrease frequency of the
left character
                left++; // Move the left pointer
            }

            maxLength = Math.max(maxLength, right - left + 1); // Update
maxLength
            right++; // Move the right pointer
        }

        return maxLength;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the string.
- Space Complexity: O(1) (constant space for the frequency map).
- This approach is efficient and uses a sliding window with a frequency map to track character counts.

Approach 2: Optimized Sliding Window
This approach is similar to Approach 1 but optimizes the calculation of the maximum count of any character in the current window.

Steps:
1. Initialize two pointers, `left` and `right`, to represent the current window.

2. Use a frequency map to track the count of each character in the window.
3. Move the `right` pointer to expand the window and update the frequency map.
4. Track the maximum count of any character in the current window.
5. If the number of characters to replace (`windowSize - maxCount`) exceeds `k`, move the `left` pointer to shrink the window.
6. Track the maximum length of the window.

Code:

```java
class Solution {
    public int characterReplacement(String s, int k) {
        int n = s.length();
        int[] freq = new int[26]; // Frequency map for characters
        int left = 0, right = 0;
        int maxCount = 0; // Maximum count of any character in the current
window
        int maxLength = 0;

        while (right < n) {
            char currentChar = s.charAt(right);
            freq[currentChar - 'A']++; // Update frequency map
            maxCount = Math.max(maxCount, freq[currentChar - 'A']); //
Update maxCount

            // If the number of characters to replace exceeds k, shrink the
window
            if (right - left + 1 - maxCount > k) {
                freq[s.charAt(left) - 'A']--; // Decrease frequency of the
left character
                left++; // Move the left pointer
            }

            maxLength = Math.max(maxLength, right - left + 1); // Update
maxLength
            right++; // Move the right pointer
        }

        return maxLength;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the string.

- Space Complexity: O(1) (constant space for the frequency map).
- This approach is efficient and uses a sliding window with a frequency map to track character counts.

Approach 3: Binary Search with Sliding Window
This approach uses binary search to find the maximum length of the substring that can be formed by replacing at most `k` characters.

Steps:
1. Use binary search to search for the maximum length of the substring.
2. For each candidate length, check if a substring of that length can be formed by replacing at most `k` characters.
3. Use a sliding window to check if the substring can be formed.

Code:

```java
class Solution {
    public int characterReplacement(String s, int k) {
        int left = 1, right = s.length();
        int maxLength = 0;

        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (canFormSubstring(s, mid, k)) {
                maxLength = mid;
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return maxLength;
    }

    private boolean canFormSubstring(String s, int length, int k) {
        int[] freq = new int[26]; // Frequency map for characters
        int maxCount = 0;

        for (int i = 0; i < s.length(); i++) {
            freq[s.charAt(i) - 'A']++; // Update frequency map
            if (i >= length) {
                freq[s.charAt(i - length) - 'A']--; // Decrease frequency
of the left character
```

```
            }
            maxCount = Math.max(maxCount, freq[s.charAt(i) - 'A']); // Update maxCount

            // Check if the substring can be formed
            if (i >= length - 1 && length - maxCount <= k) {
                return true;
            }
        }

        return false;
    }
}
```

Explanation:
- Time Complexity: O(N log N), where `N` is the length of the string.
- Space Complexity: O(1) (constant space for the frequency map).
- This approach uses binary search to find the maximum length of the substring.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Sliding Window (Map)	O(N)	O(1)	Efficient, uses sliding window	Requires careful implementation
Optimized Sliding Window	O(N)	O(1)	Efficient, optimized	Requires careful implementation
Binary Search + Sliding Window	O(N log N)	O(1)	Uses binary search	Slower than sliding window

Conclusion:
- Use the Sliding Window with Frequency Map Approach for an efficient and straightforward solution.
- Use the Optimized Sliding Window Approach for an efficient solution with optimized calculations.
- Use the Binary Search with Sliding Window Approach for a solution that uses binary search, but it is slower than the sliding window approach.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Minimum Window Substring

The Minimum Window Substring problem involves finding the smallest substring in a given string that contains all the characters of another string. This problem can be solved using a sliding window approach with two pointers and a frequency map.

Problem Definition:
Given two strings `s` and `t`, find the minimum window in `s` that contains all the characters of `t`. If there is no such window, return an empty string.

Example:
Input: `s = "ADOBECODEBANC"`, `t = "ABC"`
Output: `"BANC"` (The smallest substring in `s` that contains all characters of `t` is "BANC".)

Approach 1: Sliding Window with Two Pointers and Frequency Map
This approach uses two pointers (`left` and `right`) to represent the current window and a frequency map to track the characters of `t`.

Steps:
1. Create a frequency map for the characters of `t`.
2. Initialize two pointers, `left` and `right`, to represent the current window.
3. Move the `right` pointer to expand the window and update the frequency map.
4. When all characters of `t` are found in the window, move the `left` pointer to shrink the window and find the minimum window.
5. Track the minimum window that contains all characters of `t`.

Code:

```java
import java.util.HashMap;
import java.util.Map;

class Solution {
    public String minWindow(String s, String t) {
        if (s == null || t == null || s.length() == 0 || t.length() == 0) {
            return "";
        }

        // Create a frequency map for characters in t
        Map<Character, Integer> freqMap = new HashMap<>();
        for (char c : t.toCharArray()) {
            freqMap.put(c, freqMap.getOrDefault(c, 0) + 1);
        }

        int left = 0, right = 0; // Pointers for the sliding window
        int minLength = Integer.MAX_VALUE; // Length of the minimum window
        int minLeft = 0; // Starting index of the minimum window
        int required = freqMap.size(); // Number of unique characters in t
        int formed = 0; // Number of unique characters in the current
window

        Map<Character, Integer> windowMap = new HashMap<>(); // Frequency
map for the current window

        while (right < s.length()) {
            char currentChar = s.charAt(right);
            windowMap.put(currentChar, windowMap.getOrDefault(currentChar,
0) + 1);

            // If the current character is in t and its count matches,
increment formed
            if (freqMap.containsKey(currentChar) &&
windowMap.get(currentChar).intValue() ==
freqMap.get(currentChar).intValue()) {
                formed++;
            }

            // Try to shrink the window from the left
            while (left <= right && formed == required) {
                char leftChar = s.charAt(left);

                // Update the minimum window
                if (right - left + 1 < minLength) {
                    minLength = right - left + 1;
                    minLeft = left;
                }

                // Remove the left character from the window map
                windowMap.put(leftChar, windowMap.get(leftChar) - 1);

                // If the left character is in t and its count is less than
required, decrement formed
                if (freqMap.containsKey(leftChar) &&
windowMap.get(leftChar).intValue() < freqMap.get(leftChar).intValue()) {
                    formed--;
                }

                left++; // Move the left pointer
            }
            right++; // Move the right pointer
        }

        return minLength == Integer.MAX_VALUE ? "" : s.substring(minLeft, 
minLeft + minLength);
    }
}
```

Explanation:
- Time Complexity: O(|S| + |T|), where `|S|` is the length of `s` and `|T|` is the length of `t`.
- Space Complexity: O(|S| + |T|) for the frequency maps.
- This approach is efficient and uses a sliding window with two pointers and frequency maps.

Approach 2: Optimized Sliding Window with Array
This approach uses an array instead of a hash map to track the frequency of characters, which improves performance.

Steps:
1. Create a frequency array for the characters of `t`.
2. Initialize two pointers, `left` and `right`, to represent the current window.
3. Move the `right` pointer to expand the window and update the frequency array.
4. When all characters of `t` are found in the window, move the `left` pointer to shrink the window and find the minimum window.
5. Track the minimum window that contains all characters of `t`.

Code:

```
class Solution {
    public String minWindow(String s, String t) {
        if (s == null || t == null || s.length() == 0 || t.length() == 0) {
            return "";
        }

        // Create a frequency array for characters in t
        int[] freqMap = new int[128]; // Assuming ASCII characters
        for (char c : t.toCharArray()) {
            freqMap[c]++;
        }

        int left = 0, right = 0; // Pointers for the sliding window
        int minLength = Integer.MAX_VALUE; // Length of the minimum window
        int minLeft = 0; // Starting index of the minimum window
        int required = t.length(); // Number of characters in t
        int formed = 0; // Number of characters in the current window

        while (right < s.length()) {
            char currentChar = s.charAt(right);
            if (freqMap[currentChar] > 0) {
                formed++;
            }
            freqMap[currentChar]--; // Update frequency array

            // Try to shrink the window from the left
            while (left <= right && formed == required) {
                char leftChar = s.charAt(left);

                // Update the minimum window
                if (right - left + 1 < minLength) {
                    minLength = right - left + 1;
                    minLeft = left;
                }

                // Remove the left character from the window
                freqMap[leftChar]++;
                if (freqMap[leftChar] > 0) {
                    formed--;
                }

                left++; // Move the left pointer
            }

            right++; // Move the right pointer
        }

        return minLength == Integer.MAX_VALUE ? "" : s.substring(minLeft,
minLeft + minLength);
    }
}
```

Explanation:
- Time Complexity: O(|S| + |T|), where `|S|` is the length of `s` and `|T|` is the length of `t`.
- Space Complexity: O(1) (constant space for the frequency array).
- This approach is efficient and uses a sliding window with an array for better performance.

Approach 3: Sliding Window with Two Pointers and Counters

This approach uses two pointers and counters to track the characters of `t` in the window.

Steps:
1. Create a frequency map for the characters of `t`.
2. Initialize two pointers, `left` and `right`, to represent the current window.
3. Move the `right` pointer to expand the window and update the frequency map.
4. When all characters of `t` are found in the window, move the `left` pointer to shrink the window and find the minimum window.
5. Track the minimum window that contains all characters of `t`.

Code:

```java
import java.util.HashMap;
import java.util.Map;

class Solution {
    public String minWindow(String s, String t) {
        if (s == null || t == null || s.length() == 0 || t.length() == 0) {
            return "";
        }

        // Create a frequency map for characters in t
        Map<Character, Integer> freqMap = new HashMap<>();
        for (char c : t.toCharArray()) {
            freqMap.put(c, freqMap.getOrDefault(c, 0) + 1);
        }

        int left = 0, right = 0; // Pointers for the sliding window
        int minLength = Integer.MAX_VALUE; // Length of the minimum window
        int minLeft = 0; // Starting index of the minimum window
        int required = freqMap.size(); // Number of unique characters in t
        int formed = 0; // Number of unique characters in the current
window

        Map<Character, Integer> windowMap = new HashMap<>(); // Frequency
map for the current window

        while (right < s.length()) {
            char currentChar = s.charAt(right);
            windowMap.put(currentChar, windowMap.getOrDefault(currentChar,
0) + 1);
```

```
            // If the current character is in t and its count matches,
increment formed
            if (freqMap.containsKey(currentChar) &&
windowMap.get(currentChar).intValue() ==
freqMap.get(currentChar).intValue()) {
                formed++;
            }

            // Try to shrink the window from the left
            while (left <= right && formed == required) {
                char leftChar = s.charAt(left);

                // Update the minimum window
                if (right - left + 1 < minLength) {
                    minLength = right - left + 1;
                    minLeft = left;
                }

                // Remove the left character from the window map
                windowMap.put(leftChar, windowMap.get(leftChar) - 1);

                // If the left character is in t and its count is less than
required, decrement formed
                if (freqMap.containsKey(leftChar) &&
windowMap.get(leftChar).intValue() < freqMap.get(leftChar).intValue()) {
                    formed--;
                }

                left++; // Move the left pointer
            }

            right++; // Move the right pointer
        }

        return minLength == Integer.MAX_VALUE ? "" : s.substring(minLeft,
minLeft + minLength);
    }
}
```

Explanation:
- Time Complexity: O(|S| + |T|), where `|S|` is the length of `s` and `|T|` is the length of `t`.
- Space Complexity: O(|S| + |T|) for the frequency maps.

- This approach is efficient and uses a sliding window with two pointers and frequency maps.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Sliding Window (Map)	O(|S| + |T|)	O(|S| + |T|)	Efficient, uses sliding window	Requires careful implementation
Optimized Sliding Window (Array)	O(|S| + |T|)	O(1)	Efficient, uses array for better performance	Limited to fixed character set
Sliding Window (Counters)	O(|S| + |T|)	O(|S| + |T|)	Efficient, uses counters	Requires careful implementation

Conclusion:
- Use the Sliding Window with Frequency Map Approach for an efficient and straightforward solution.
- Use the Optimized Sliding Window with Array Approach for an efficient solution with better performance.
- Use the Sliding Window with Counters Approach for an efficient solution that uses counters.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Valid Anagram

The Valid Anagram problem involves determining whether two strings are anagrams of each other. Two strings are anagrams if they contain the same characters in the same frequency. This problem can be solved using sorting, hash maps, or frequency arrays.

Problem Definition:
Given two strings `s` and `t`, determine if `t` is an anagram of `s`.

Example:
Input: `s = "anagram"`, `t = "nagaram"`
Output: `true` (Both strings contain the same characters in the same frequency.)

Approach 1: Sorting
This approach sorts both strings and checks if they are equal.

Steps:
1. Sort both strings.
2. Compare the sorted strings.
3. If they are equal, return `true`; otherwise, return `false`.

Code:

```java
import java.util.Arrays;

class Solution {
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) {
            return false;
        }

        // Convert strings to char arrays and sort them
        char[] sArray = s.toCharArray();
        char[] tArray = t.toCharArray();
        Arrays.sort(sArray);
        Arrays.sort(tArray);

        // Compare the sorted arrays
        return Arrays.equals(sArray, tArray);
    }
}
```

Explanation:
- Time Complexity: O(N log N), where `N` is the length of the strings (due to sorting).
- Space Complexity: O(N) for the character arrays.
- This approach is simple but less efficient due to sorting.

Approach 2: Hash Map
This approach uses a hash map to track the frequency of characters in both strings.

Steps:
1. Create a hash map to store the frequency of characters in `s`.
2. Traverse `t` and decrement the frequency of characters in the hash map.
3. If all frequencies in the hash map are zero, return `true`; otherwise, return `false`.

Code:

```java
import java.util.HashMap;
import java.util.Map;

class Solution {
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) {
            return false;
        }

        // Create a frequency map for characters in s
        Map<Character, Integer> freqMap = new HashMap<>();
        for (char c : s.toCharArray()) {
            freqMap.put(c, freqMap.getOrDefault(c, 0) + 1);
        }

        // Traverse t and update the frequency map
        for (char c : t.toCharArray()) {
            if (!freqMap.containsKey(c)) {
                return false;
            }
            freqMap.put(c, freqMap.get(c) - 1);
            if (freqMap.get(c) == 0) {
                freqMap.remove(c);
            }
        }

        // If the map is empty, t is an anagram of s
        return freqMap.isEmpty();
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the strings.
- Space Complexity: O(N) for the hash map.
- This approach is efficient and uses a hash map to track character frequencies.

Approach 3: Frequency Array
This approach uses a frequency array to track the frequency of characters in both strings.

Steps:
1. Create a frequency array of size 26 (for lowercase English letters).
2. Traverse `s` and increment the frequency of characters in the array.
3. Traverse `t` and decrement the frequency of characters in the array.
4. If all frequencies in the array are zero, return `true`; otherwise, return `false`.

Code:

```
class Solution {
```

```java
    public boolean isAnagram(String s, String t) {
        if (s.length() != t.length()) {
            return false;
        }

        // Create a frequency array for characters in s
        int[] freqArray = new int[26];
        for (char c : s.toCharArray()) {
            freqArray[c - 'a']++;
        }

        // Traverse t and update the frequency array
        for (char c : t.toCharArray()) {
            freqArray[c - 'a']--;
        }

        // Check if all frequencies are zero
        for (int count : freqArray) {
            if (count != 0) {
                return false;
            }
        }

        return true;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the strings.
- Space Complexity: O(1) (constant space for the frequency array).
- This approach is efficient and uses a frequency array to track character frequencies.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Sorting	O(N log N)	O(N)	Simple and straightforward	Less efficient due to sorting
Hash Map	O(N)	O(N)	Efficient, uses hash map	Uses extra space for hash map
Frequency Array	O(N)	O(1)	Efficient, constant space	Limited to fixed character set

Conclusion:
- Use the Sorting Approach for simplicity and readability.
- Use the Hash Map Approach for an efficient solution that works for any character set.
- Use the Frequency Array Approach for an efficient solution with constant space, but it is limited to lowercase English letters.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Group Anagrams

The Group Anagrams problem involves grouping anagrams together from a list of strings. Anagrams are words that contain the same characters in the same frequency but in a different order. This problem can be solved using sorting, hash maps, or frequency arrays.

Problem Definition:
Given a list of strings, group the anagrams together.

Example:
Input: `["eat", "tea", "tan", "ate", "nat", "bat"]`
Output:

```
[
  ["eat", "tea", "ate"],
  ["tan", "nat"],
  ["bat"]
]
```

Approach 1: Sorting and Hash Map
This approach sorts each string and uses the sorted string as a key in a hash map to group anagrams.

Steps:
1. Create a hash map where the key is the sorted version of a string, and the value is a list of strings that are anagrams of the key.
2. Traverse the list of strings:
   - Sort each string and use it as a key in the hash map.
   - Add the original string to the list corresponding to the key.
3. Return the values of the hash map as the grouped anagrams.

Code:

```java
import java.util.*;
```

```
class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        if (strs == null || strs.length == 0) {
            return new ArrayList<>();
        }

        // Create a hash map to group anagrams
        Map<String, List<String>> map = new HashMap<>();

        for (String s : strs) {
            // Sort the string to use as a key
            char[] charArray = s.toCharArray();
            Arrays.sort(charArray);
            String sorted = new String(charArray);

            // Add the original string to the list corresponding to the
sorted key
            if (!map.containsKey(sorted)) {
                map.put(sorted, new ArrayList<>());
            }
            map.get(sorted).add(s);
        }

        // Return the grouped anagrams
        return new ArrayList<>(map.values());
    }
}
```

Explanation:
- Time Complexity: O(N * K log K), where `N` is the number of strings and `K` is the maximum length of a string (due to sorting).
- Space Complexity: O(N * K) for the hash map.
- This approach is simple and uses sorting to group anagrams.

Approach 2: Frequency Array and Hash Map
This approach uses a frequency array to represent the count of characters in each string and uses the frequency array as a key in a hash map to group anagrams.

Steps:
1. Create a hash map where the key is a frequency array (or its string representation), and the value is a list of strings that are anagrams of the key.
2. Traverse the list of strings:

  - Create a frequency array for each string.
  - Use the frequency array as a key in the hash map.
  - Add the original string to the list corresponding to the key.
3. Return the values of the hash map as the grouped anagrams.

Code:

```java
import java.util.*;

class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        if (strs == null || strs.length == 0) {
            return new ArrayList<>();
        }

        // Create a hash map to group anagrams
        Map<String, List<String>> map = new HashMap<>();

        for (String s : strs) {
            // Create a frequency array for the string
            int[] freqArray = new int[26];
            for (char c : s.toCharArray()) {
                freqArray[c - 'a']++;
            }

            // Convert the frequency array to a string key
            String key = Arrays.toString(freqArray);

            // Add the original string to the list corresponding to the key
            if (!map.containsKey(key)) {
                map.put(key, new ArrayList<>());
            }
            map.get(key).add(s);
        }

        // Return the grouped anagrams
        return new ArrayList<>(map.values());
    }
}
```

Explanation:
- Time Complexity: O(N * K), where `N` is the number of strings and `K` is the maximum length of a string.

- Space Complexity: O(N * K) for the hash map.
- This approach is efficient and uses a frequency array to group anagrams.

Approach 3: Prime Number Multiplication and Hash Map
This approach assigns a unique prime number to each character and uses the product of prime numbers as a key in a hash map to group anagrams.

Steps:
1. Assign a unique prime number to each lowercase English letter.
2. Create a hash map where the key is the product of prime numbers corresponding to the characters in a string, and the value is a list of strings that are anagrams of the key.
3. Traverse the list of strings:
   - Calculate the product of prime numbers for each string.
   - Use the product as a key in the hash map.
   - Add the original string to the list corresponding to the key.
4. Return the values of the hash map as the grouped anagrams.

Code:

```java
import java.util.*;

class Solution {
    public List<List<String>> groupAnagrams(String[] strs) {
        if (strs == null || strs.length == 0) {
            return new ArrayList<>();
        }

        // Assign a unique prime number to each lowercase English letter
        int[] primes = {
            2, 3, 5, 7, 11, 13, 17, 19, 23, 29,
            31, 37, 41, 43, 47, 53, 59, 61, 67,
            71, 73, 79, 83, 89, 97, 101
        };

        // Create a hash map to group anagrams
        Map<Long, List<String>> map = new HashMap<>();

        for (String s : strs) {
            // Calculate the product of prime numbers for the string
            long product = 1;
            for (char c : s.toCharArray()) {
                product *= primes[c - 'a'];
            }
            // Add the original string to the list corresponding to the
product key
            if (!map.containsKey(product)) {
                map.put(product, new ArrayList<>());
            }
            map.get(product).add(s);
        }

        // Return the grouped anagrams
        return new ArrayList<>(map.values());
    }
}
```

Explanation:
- Time Complexity: O(N * K), where `N` is the number of strings and `K` is the maximum length of a string.
- Space Complexity: O(N * K) for the hash map.
- This approach is efficient and uses prime number multiplication to group anagrams.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Sorting + Hash Map	O(N * K log K)	O(N * K)	Simple and straightforward	Less efficient due to sorting
Frequency Array + Hash Map	O(N * K)	O(N * K)	Efficient, uses frequency array	Limited to lowercase English letters
Prime Number Multiplication + Hash Map	O(N * K)	O(N * K)	Efficient, uses prime numbers	Limited to lowercase English letters

Conclusion:
- Use the Sorting + Hash Map Approach for simplicity and readability.
- Use the Frequency Array + Hash Map Approach for an efficient solution that works for lowercase English letters.
- Use the Prime Number Multiplication + Hash Map Approach for an efficient solution that uses prime numbers.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Valid Parentheses

The Valid Parentheses problem involves determining whether a given string of parentheses is valid. A string is valid if all parentheses are properly closed in the correct order. This problem can be solved using a stack, two pointers, or recursion.

Problem Definition:
Given a string `s` containing only the characters `'('`, `')'`, `'{'`, `'}'`, `'['`, and `']'`, determine if the input string is valid.

Example:
Input: `s = "()[]{}"`
Output: `true` (The string is valid because all parentheses are properly closed.)

Approach 1: Stack
This approach uses a stack to keep track of opening parentheses and ensures that they are properly closed.

Steps:
1. Initialize an empty stack.
2. Traverse the string:
   - If the current character is an opening parenthesis (`(`, `{`, `[`), push it onto the stack.
   - If the current character is a closing parenthesis (`)`, `}`, `]`), check if it matches the top element of the stack.
      - If it matches, pop the top element from the stack.
      - If it doesn't match, the string is invalid.
3. After traversing the string, if the stack is empty, the string is valid; otherwise, it is invalid.

Code:

```java
import java.util.Stack;

class Solution {
    public boolean isValid(String s) {
        Stack<Character> stack = new Stack<>();

        for (char c : s.toCharArray()) {
            if (c == '(' || c == '{' || c == '[') {
                stack.push(c);
            } else {
                if (stack.isEmpty()) {
                    return false;
                }
                char top = stack.pop();
                if ((c == ')' && top != '(') ||
                    (c == '}' && top != '{') ||
                    (c == ']' && top != '[')) {
                    return false;
                }
            }
        }

        return stack.isEmpty();
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the string.
- Space Complexity: O(N) for the stack.
- This approach is efficient and uses a stack to track opening parentheses.

Approach 2: Two Pointers (Without Stack)
This approach uses two pointers to simulate the stack behavior without explicitly using a stack.

Steps:
1. Initialize a pointer `i` to track the position in the string.
2. Traverse the string:
   - If the current character is an opening parenthesis (`(`, `{`, `[`), move the pointer forward.
   - If the current character is a closing parenthesis (`)`, `}`, `]`), check if it matches the previous opening parenthesis.
     - If it matches, move the pointer backward.
     - If it doesn't match, the string is invalid.
3. After traversing the string, if the pointer is at the start of the string, the string is valid; otherwise, it is invalid.

Code:

```
class Solution {
    public boolean isValid(String s) {
        char[] stack = new char[s.length()];
        int i = 0;

        for (char c : s.toCharArray()) {
            if (c == '(' || c == '{' || c == '[') {
                stack[i++] = c;
            } else {
                if (i == 0) {
                    return false;
                }
                char top = stack[--i];
                if ((c == ')' && top != '(') ||
                    (c == '}' && top != '{') ||
                    (c == ']' && top != '[')) {
                    return false;
                }
            }
        }

        return i == 0;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the string.
- Space Complexity: O(N) for the simulated stack.
- This approach is efficient and simulates the stack behavior using an array.

Approach 3: Recursion
This approach uses recursion to validate the parentheses string.

Steps:
1. Define a recursive function that takes the string and the current index as parameters.
2. If the current character is an opening parenthesis (`(`, `{`, `[`), recursively check the next character.
3. If the current character is a closing parenthesis (`)`, `}`, `]`), check if it matches the previous opening parenthesis.
   - If it matches, continue recursion.
   - If it doesn't match, the string is invalid.
4. After traversing the string, if the recursion reaches the end of the string, the string is valid; otherwise, it is invalid.

Code:

```
class Solution {
    private int index = 0;

    public boolean isValid(String s) {
        return isValidHelper(s) && index == s.length();
    }
    private boolean isValidHelper(String s) {
        while (index < s.length()) {
            char c = s.charAt(index);
            if (c == '(' || c == '{' || c == '[') {
                index++;
                if (!isValidHelper(s)) {
                    return false;
                }
            } else {
                if (index == 0) {
                    return false;
                }
                char top = s.charAt(index - 1);
                if ((c == ')' && top != '(') ||
                    (c == '}' && top != '{') ||
                    (c == ']' && top != '[')) {
                    return false;
                }
                index++;
                return true;
            }
        }
        return true;
    }
}
```

<u>Explanation</u>:
- Time Complexity: O(N), where `N` is the length of the string.
- Space Complexity: O(N) for the recursion stack.
- This approach is intuitive but uses extra space for the recursion stack.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Stack	O(N)	O(N)	Efficient, uses stack	Requires extra space for stack
Two Pointers	O(N)	O(N)	Efficient, simulates stack	Requires extra space for array
Recursion	O(N)	O(N)	Intuitive and clean	Uses extra space for recursion stack

Conclusion:
- Use the Stack Approach for an efficient and straightforward solution.
- Use the Two Pointers Approach for an efficient solution that simulates the stack behavior.
- Use the Recursion Approach for an intuitive solution, but it uses extra space for the recursion stack.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Valid Palindrome

The Valid Palindrome problem involves determining whether a given string is a palindrome after removing non-alphanumeric characters and ignoring cases. A palindrome is a string that reads the same forward and backward. This problem can be solved using two pointers, reverse and compare, or recursion.

Problem Definition:
Given a string `s`, determine if it is a palindrome after removing all non-alphanumeric characters and ignoring cases.

Example:
Input: `s = "A man, a plan, a canal: Panama"`
Output: `true` (The string is a palindrome after removing non-alphanumeric characters and ignoring cases.)

Approach 1: Two Pointers
This approach uses two pointers to compare characters from the beginning and end of the string.

Steps:
1. Initialize two pointers, `left` and `right`, at the start and end of the string, respectively.
2. Move the pointers toward each other while skipping non-alphanumeric characters.
3. Compare the characters at the `left` and `right` pointers.
   - If they match, move both pointers inward.
   - If they don't match, the string is not a palindrome.
4. If the pointers meet or cross, the string is a palindrome.

Code:

```java
class Solution {
    public boolean isPalindrome(String s) {
        int left = 0, right = s.length() - 1;
        while (left < right) {
            // Skip non-alphanumeric characters from the left
            while (left < right &&
!Character.isLetterOrDigit(s.charAt(left))) {
                left++;
            }
            // Skip non-alphanumeric characters from the right
            while (left < right &&
!Character.isLetterOrDigit(s.charAt(right))) {
                right--;
            }

            // Compare characters (case-insensitive)
            if (Character.toLowerCase(s.charAt(left)) !=
Character.toLowerCase(s.charAt(right))) {
                return false;
            }

            left++;
            right--;
        }

        return true;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the string.
- Space Complexity: O(1) (constant space).
- This approach is efficient and uses two pointers to compare characters.

Approach 2: Reverse and Compare
This approach creates a cleaned version of the string (removing non-alphanumeric characters and ignoring cases) and compares it with its reverse.

Steps:
1. Create a cleaned version of the string by removing non-alphanumeric characters and converting it to lowercase.
2. Compare the cleaned string with its reverse.
3. If they match, the string is a palindrome; otherwise, it is not.

Code:

```java
class Solution {
    public boolean isPalindrome(String s) {
        // Remove non-alphanumeric characters and convert to lowercase
        String cleaned = s.replaceAll("[^a-zA-Z0-9]", "").toLowerCase();

        // Compare the cleaned string with its reverse
        String reversed = new StringBuilder(cleaned).reverse().toString();
        return cleaned.equals(reversed);
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the string.
- Space Complexity: O(N) for the cleaned and reversed strings.
- This approach is simple but uses extra space for the cleaned and reversed strings.

Approach 3: Recursion
This approach uses recursion to compare characters from the beginning and end of the string.

Steps:
1. Define a recursive function that takes the string and the current indices as parameters.
2. Skip non-alphanumeric characters and compare the characters at the current indices.
   - If they match, continue recursion.
   - If they don't match, the string is not a palindrome.
3. If the indices meet or cross, the string is a palindrome.

Code:

```java
class Solution {
    public boolean isPalindrome(String s) {
        return isPalindromeHelper(s, 0, s.length() - 1);
    }

    private boolean isPalindromeHelper(String s, int left, int right) {
        // Base case: indices meet or cross
        if (left >= right) {
            return true;
        }

        // Skip non-alphanumeric characters from the left
        while (left < right && !Character.isLetterOrDigit(s.charAt(left)))
{
            left++;
        }
        // Skip non-alphanumeric characters from the right
        while (left < right && !Character.isLetterOrDigit(s.charAt(right)))
{
            right--;
        }

        // Compare characters (case-insensitive)
        if (Character.toLowerCase(s.charAt(left)) !=
Character.toLowerCase(s.charAt(right))) {
            return false;
        }

        // Recur for the next pair of characters
        return isPalindromeHelper(s, left + 1, right - 1);
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the string.
- Space Complexity: O(N) for the recursion stack.
- This approach is intuitive but uses extra space for the recursion stack.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Two Pointers	O(N)	O(1)	Efficient, constant space	Requires careful implementation
Reverse and Compare	O(N)	O(N)	Simple and straightforward	Uses extra space for cleaned and reversed strings
Recursion	O(N)	O(N)	Intuitive and clean	Uses extra space for recursion stack

Conclusion:
- Use the Two Pointers Approach for an efficient and space-optimized solution.
- Use the Reverse and Compare Approach for a simple and straightforward solution.
- Use the Recursion Approach for an intuitive solution, but it uses extra space for the recursion stack.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Longest Palindromic Substring

The Longest Palindromic Substring problem involves finding the longest substring in a given string that is a palindrome. A palindrome is a string that reads the same forward and backward. This problem can be solved using dynamic programming, expand around center, or Manacher's algorithm.

Problem Definition:
Given a string `s`, find the longest substring that is a palindrome.

Example:
Input: `s = "babad"`
Output: `"bab"` or `"aba"` (Both are valid longest palindromic substrings.)

Approach 1: Dynamic Programming
This approach uses a 2D dynamic programming table to store whether a substring is a palindrome.

Steps:
1. Create a 2D boolean array `dp` where `dp[i][j]` is `true` if the substring `s[i...j]` is a palindrome.
2. Initialize the diagonal of the table (`dp[i][i]`) to `true` because a single character is always a palindrome.
3. Check for palindromes of length 2 (`dp[i][i+1]`).
4. For substrings longer than 2, use the recurrence relation:
   - `dp[i][j] = (s[i] == s[j]) && dp[i+1][j-1]`.
5. Track the longest palindromic substring.

Code:

```
class Solution {
    public String longestPalindrome(String s) {
        if (s == null || s.length() == 0) {
            return "";
        }

        int n = s.length();
        boolean[][] dp = new boolean[n][n];
        int start = 0, maxLength = 1;

        // Every single character is a palindrome
        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
        }

        // Check for palindromes of length 2
        for (int i = 0; i < n - 1; i++) {
            if (s.charAt(i) == s.charAt(i + 1)) {
                dp[i][i + 1] = true;
                start = i;
                maxLength = 2;
            }
        }

        // Check for palindromes longer than 2
        for (int length = 3; length <= n; length++) {
            for (int i = 0; i < n - length + 1; i++) {
                int j = i + length - 1;
                if (s.charAt(i) == s.charAt(j) && dp[i + 1][j - 1]) {
                    dp[i][j] = true;
                    if (length > maxLength) {
                        start = i;
                        maxLength = length;
                    }
                }
            }
        }

        return s.substring(start, start + maxLength);
    }
}
```

Explanation:
- Time Complexity: O(N^2), where `N` is the length of the string.
- Space Complexity: O(N^2) for the `dp` table.
- This approach is efficient and uses dynamic programming to track palindromic substrings.

Approach 2: Expand Around Center
This approach expands around each character (and between characters) to find the longest palindromic substring.

Steps:
1. Iterate through each character in the string.
2. For each character, expand around it to find the longest palindrome with odd length.

3. For each pair of characters, expand around the center to find the longest palindrome with even length.
4. Track the longest palindromic substring.

Code:

```java
class Solution {
    public String longestPalindrome(String s) {
        if (s == null || s.length() == 0) {
            return "";
        }

        int start = 0, end = 0;

        for (int i = 0; i < s.length(); i++) {
            // Expand around the center for odd-length palindromes
            int len1 = expandAroundCenter(s, i, i);
            // Expand around the center for even-length palindromes
            int len2 = expandAroundCenter(s, i, i + 1);
            // Find the maximum length
            int len = Math.max(len1, len2);
            // Update the start and end indices of the longest palindrome
            if (len > end - start) {
                start = i - (len - 1) / 2;
                end = i + len / 2;
            }
        }

        return s.substring(start, end + 1);
    }

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

Explanation:
- Time Complexity: O(N^2), where `N` is the length of the string.

- Space Complexity: O(1) (constant space).
- This approach is efficient and uses constant space.

Approach 3: Manacher's Algorithm
This approach uses Manacher's algorithm to find the longest palindromic substring in linear time.

Steps:
1. Transform the string by inserting special characters (e.g., `#`) to handle even-length palindromes.
2. Use an array `p` to store the length of the palindrome centered at each character.
3. Iterate through the transformed string and update the `p` array using the properties of palindromes.
4. Track the longest palindromic substring.

Code:

```
class Solution {
    public String longestPalindrome(String s) {
        if (s == null || s.length() == 0) {
            return "";
        }

        // Transform the string to handle even-length palindromes
        String transformed = "#";
        for (char c : s.toCharArray()) {
            transformed += c + "#";
        }

        int n = transformed.length();
        int[] p = new int[n];
        int center = 0, right = 0;
        int maxCenter = 0, maxLength = 0;

        for (int i = 0; i < n; i++) {
            // Mirror index for the current center
            int mirror = 2 * center - i;

            // If the current index is within the right boundary, use the
mirror value
            if (i < right) {
                p[i] = Math.min(right - i, p[mirror]);
            }
            // Expand around the current center
            int leftBoundary = i - (1 + p[i]);
            int rightBoundary = i + (1 + p[i]);
            while (leftBoundary >= 0 && rightBoundary < n &&
transformed.charAt(leftBoundary) == transformed.charAt(rightBoundary)) {
                p[i]++;
                leftBoundary--;
                rightBoundary++;
            }

            // Update the center and right boundary if the current
palindrome expands beyond the right boundary
            if (i + p[i] > right) {
                center = i;
                right = i + p[i];
            }

            // Track the maximum palindrome
            if (p[i] > maxLength) {
                maxLength = p[i];
                maxCenter = i;
            }
        }

        // Extract the longest palindromic substring
        int start = (maxCenter - maxLength) / 2;
        int end = start + maxLength;
        return s.substring(start, end);
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the length of the string.
- Space Complexity: O(N) for the `p` array.
- This approach is highly efficient and uses Manacher's algorithm to find the longest palindromic substring in linear time.

Comparison of Approaches:
| Approach | Time Complexity | Space Complexity | Pros | Cons
|
||--||-|-|

Dynamic Programming	O(N^2)	O(N^2)	Efficient, uses DP	Uses extra space for DP table
Expand Around Center	O(N^2)	O(1)	Efficient, constant space	Requires careful implementation
Manacher's Algorithm	O(N)	O(N)	Highly efficient, linear time	More complex to implement

Conclusion:
- Use the Dynamic Programming Approach for an efficient and straightforward solution.
- Use the Expand Around Center Approach for an efficient solution with constant space.
- Use Manacher's Algorithm for a highly efficient solution with linear time complexity.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Palindromic Substrings

The Palindromic Substrings problem involves counting the number of palindromic substrings in a given string. A palindrome is a string that reads the same forward and backward. This problem can be solved using dynamic programming, expand around center, or Manacher's algorithm.

Problem Definition:
Given a string `s`, count the number of palindromic substrings in it.

Example:
Input: `s = "abc"`
Output: `3` (The palindromic substrings are `"a"`, `"b"`, and `"c"`.)

Input: `s = "aaa"`
Output: `6` (The palindromic substrings are `"a"`, `"a"`, `"a"`, `"aa"`, `"aa"`, and `"aaa"`.)

Approach 1: Dynamic Programming
This approach uses a 2D dynamic programming table to store whether a substring is a palindrome.

Steps:
1. Create a 2D boolean array `dp` where `dp[i][j]` is `true` if the substring `s[i...j]` is a palindrome.
2. Initialize the diagonal of the table (`dp[i][i]`) to `true` because a single character is always a palindrome.
3. Check for palindromes of length 2 (`dp[i][i+1]`).
4. For substrings longer than 2, use the recurrence relation:
   - `dp[i][j] = (s[i] == s[j]) && dp[i+1][j-1]`.
5. Count the number of `true` values in the `dp` table.

Code:

```
class Solution {
    public int countSubstrings(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }

        int n = s.length();
        boolean[][] dp = new boolean[n][n];
        int count = 0;

        // Every single character is a palindrome
        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
            count++;
        }

        // Check for palindromes of length 2
        for (int i = 0; i < n - 1; i++) {
            if (s.charAt(i) == s.charAt(i + 1)) {
                dp[i][i + 1] = true;
                count++;
            }
        }

        // Check for palindromes longer than 2
        for (int length = 3; length <= n; length++) {
            for (int i = 0; i < n - length + 1; i++) {
                int j = i + length - 1;
                if (s.charAt(i) == s.charAt(j) && dp[i + 1][j - 1]) {
                    dp[i][j] = true;
                    count++;
                }
            }
        }

        return count;
    }
}
```

Explanation:

- Time Complexity: O(N^2), where `N` is the length of the string.
- Space Complexity: O(N^2) for the `dp` table.
- This approach is efficient and uses dynamic programming to count palindromic substrings.

Approach 2: Expand Around Center
This approach expands around each character (and between characters) to count all palindromic substrings.

Steps:
1. Iterate through each character in the string.
2. For each character, expand around it to count all odd-length palindromic substrings.
3. For each pair of characters, expand around the center to count all even-length palindromic substrings.
4. Return the total count of palindromic substrings.

Code:

```java
class Solution {
    public int countSubstrings(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }

        int count = 0;

        for (int i = 0; i < s.length(); i++) {
            // Expand around the center for odd-length palindromes
            count += expandAroundCenter(s, i, i);
            // Expand around the center for even-length palindromes
            count += expandAroundCenter(s, i, i + 1);
        }

        return count;
    }

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

Explanation:
- Time Complexity: O(N^2), where `N` is the length of the string.
- Space Complexity: O(1) (constant space).
- This approach is efficient and uses constant space.

Approach 3: Manacher's Algorithm
This approach uses Manacher's algorithm to count all palindromic substrings in linear time.

Steps:
1. Transform the string by inserting special characters (e.g., `#`) to handle even-length palindromes.
2. Use an array `p` to store the length of the palindrome centered at each character.
3. Iterate through the transformed string and update the `p` array using the properties of palindromes.
4. Count the number of palindromic substrings based on the `p` array.

Code:

```java
class Solution {
    public int countSubstrings(String s) {
        if (s == null || s.length() == 0) {
            return 0;
        }

        // Transform the string to handle even-length palindromes
        String transformed = "#";
        for (char c : s.toCharArray()) {
            transformed += c + "#";
        }

        int n = transformed.length();
        int[] p = new int[n];
        int center = 0, right = 0;
        int count = 0;

        for (int i = 0; i < n; i++) {
            // Mirror index for the current center
            int mirror = 2 * center - i;
            // If the current index is within the right boundary, use the
mirror value
            if (i < right) {
                p[i] = Math.min(right - i, p[mirror]);
            }

            // Expand around the current center
            int leftBoundary = i - (1 + p[i]);
            int rightBoundary = i + (1 + p[i]);
            while (leftBoundary >= 0 && rightBoundary < n &&
transformed.charAt(leftBoundary) == transformed.charAt(rightBoundary)) {
                p[i]++;
                leftBoundary--;
                rightBoundary++;
            }

            // Update the center and right boundary if the current
palindrome expands beyond the right boundary
            if (i + p[i] > right) {
                center = i;
                right = i + p[i];
            }

            // Count the number of palindromic substrings
            count += (p[i] + 1) / 2;
        }

        return count;
    }
}
```

<u>Explanation</u>:
- Time Complexity: O(N), where `N` is the length of the string.
- Space Complexity: O(N) for the `p` array.
- This approach is highly efficient and uses Manacher's algorithm to count palindromic substrings in linear time.

Comparison of Approaches:
| Approach | Time Complexity | Space Complexity | Pros | Cons
|
||--||-|-|

Dynamic Programming	O(N^2)	O(N^2)	Efficient, uses DP	Uses extra space for DP table
Expand Around Center	O(N^2)	O(1)	Efficient, constant space	Requires careful implementation
Manacher's Algorithm	O(N)	O(N)	Highly efficient, linear time	More complex to implement

Conclusion:
- Use the Dynamic Programming Approach for an efficient and straightforward solution.
- Use the Expand Around Center Approach for an efficient solution with constant space.
- Use Manacher's Algorithm for a highly efficient solution with linear time complexity.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Encode and Decode Strings (Leetcode Premium)

The Encode and Decode Strings problem involves encoding a list of strings into a single string and then decoding it back into the original list of strings. This problem can be solved using length-prefix encoding, delimiter-based encoding, or escape character encoding.

Problem Definition:
Design an algorithm to encode a list of strings into a single string and decode it back into the original list of strings.

Example:
Input: `["lint", "code", "love", "you"]`
Encoded Output: `"4#lint4#code4#love3#you"`
Decoded Output: `["lint", "code", "love", "you"]`

Approach 1: Length-Prefix Encoding
This approach encodes each string with its length followed by a delimiter (e.g., `#`) and then decodes by reading the length and extracting the string.

Steps:
1. Encoding:
   - For each string, write its length followed by a delimiter (e.g., `#`) and then the string itself.
   - Concatenate all encoded strings into a single string.
2. Decoding:
   - Iterate through the encoded string, read the length, and extract the string based on the length.
   - Repeat until the entire encoded string is processed.

Code:

```java
import java.util.*;

public class Codec {

    // Encodes a list of strings to a single string.
    public String encode(List<String> strs) {
        StringBuilder encoded = new StringBuilder();
        for (String s : strs) {
            encoded.append(s.length()).append("#").append(s);
        }
        return encoded.toString();
    }

    // Decodes a single string to a list of strings.
    public List<String> decode(String s) {
        List<String> decoded = new ArrayList<>();
        int i = 0;
        while (i < s.length()) {
            // Find the delimiter
            int delimiterIndex = s.indexOf('#', i);
            // Extract the length
            int length = Integer.parseInt(s.substring(i, delimiterIndex));
            // Extract the string
            String str = s.substring(delimiterIndex + 1, delimiterIndex + 1
+ length);
            decoded.add(str);
            // Move the index
            i = delimiterIndex + 1 + length;
        }
        return decoded;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the total number of characters in all strings.
- Space Complexity: O(N) for the encoded string.
- This approach is efficient and uses length-prefix encoding to handle strings of varying lengths.

Approach 2: Delimiter-Based Encoding
This approach uses a special delimiter (e.g., `#`) to separate strings and escape the delimiter if it appears in the strings.

Steps:
1. Encoding:
   - Join the strings with a special delimiter (e.g., `#`).
   - Escape the delimiter if it appears in any string.
2. Decoding:
   - Split the encoded string by the delimiter and handle escaped delimiters.

Code:

```
import java.util.*;

public class Codec {

    // Encodes a list of strings to a single string.
    public String encode(List<String> strs) {
        StringBuilder encoded = new StringBuilder();
        for (String s : strs) {
            // Escape the delimiter
            String escaped = s.replace("#", "##");
            encoded.append(escaped).append("#");
        }
        return encoded.toString();
    }

    // Decodes a single string to a list of strings.
    public List<String> decode(String s) {
        List<String> decoded = new ArrayList<>();
        StringBuilder current = new StringBuilder();
        int i = 0;
        while (i < s.length()) {
            if (s.charAt(i) == '#') {
                // Check if it's an escaped delimiter
                if (i + 1 < s.length() && s.charAt(i + 1) == '#') {
                    current.append('#');
                    i += 2;
                } else {
                    // End of a string
                    decoded.add(current.toString());
                    current = new StringBuilder();
                    i++;
                }
            } else {
                current.append(s.charAt(i));
                i++;
            }
        }
        return decoded;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the total number of characters in all strings.
- Space Complexity: O(N) for the encoded string.
- This approach is efficient and uses a delimiter to separate strings.

Approach 3: Escape Character Encoding
This approach uses an escape character (e.g., `\`) to handle special characters and delimiters.

Steps:
1. Encoding:
   - Use an escape character (e.g., `\`) to escape special characters and delimiters.
   - Join the strings with a delimiter (e.g., `#`).
2. Decoding:
   - Split the encoded string by the delimiter and handle escaped characters.

Code:

```
import java.util.*;

public class Codec {

    // Encodes a list of strings to a single string.
    public String encode(List<String> strs) {
        StringBuilder encoded = new StringBuilder();
        for (String s : strs) {
            // Escape special characters and the delimiter
            String escaped = s.replace("\\", "\\\\").replace("#", "\\#");
            encoded.append(escaped).append("#");
        }
        return encoded.toString();
    }

    // Decodes a single string to a list of strings.
    public List<String> decode(String s) {
        List<String> decoded = new ArrayList<>();
        StringBuilder current = new StringBuilder();
        int i = 0;
        while (i < s.length()) {
            if (s.charAt(i) == '\\') {
                // Handle escaped characters
                if (i + 1 < s.length()) {
                    current.append(s.charAt(i + 1));
                    i += 2;
                }
            } else if (s.charAt(i) == '#') {
                // End of a string
                decoded.add(current.toString());
                current = new StringBuilder();
                i++;
            } else {
                current.append(s.charAt(i));
                i++;
            }
        }
        return decoded;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the total number of characters in all strings.
- Space Complexity: O(N) for the encoded string.
- This approach is efficient and uses escape characters to handle special characters and delimiters.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Length-Prefix Encoding	O(N)	O(N)	Efficient, handles varying lengths	Requires length calculation
Delimiter-Based Encoding	O(N)	O(N)	Simple and straightforward	Requires escaping delimiters
Escape Character Encoding	O(N)	O(N)	Handles special characters	Requires escaping characters

Conclusion:
- Use the Length-Prefix Encoding Approach for an efficient and straightforward solution.
- Use the Delimiter-Based Encoding Approach for a simple solution with delimiter handling.

- Use the Escape Character Encoding Approach for handling special characters and delimiters.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

(Tree)

Maximum Depth of Binary Tree

The Maximum Depth of Binary Tree problem involves finding the maximum depth (or height) of a binary tree. The depth of a binary tree is the number of nodes along the longest path from the root node down to the farthest leaf node. This problem can be solved using recursion, BFS (level-order traversal), or DFS (iterative).

Problem Definition:
Given the root of a binary tree, find its maximum depth.

Example:
Input:

```
    3
   / \
  9  20
    /  \
   15   7
```

Output: `3` (The longest path is `3 -> 20 -> 15` or `3 -> 20 -> 7`.)

Approach 1: Recursion (Depth-First Search - DFS)
This approach uses recursion to calculate the maximum depth of the left and right subtrees and returns the maximum of the two.

Steps:
1. If the root is `null`, return `0` (base case).
2. Recursively calculate the depth of the left subtree.
3. Recursively calculate the depth of the right subtree.
4. Return the maximum depth of the left and right subtrees plus `1` (for the current node).

Code:

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }
        int leftDepth = maxDepth(root.left);
        int rightDepth = maxDepth(root.right);
        return Math.max(leftDepth, rightDepth) + 1;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(H), where `H` is the height of the tree (recursion stack space).
- This approach is simple and uses recursion to traverse the tree.

Approach 2: Breadth-First Search (BFS - Level-Order Traversal)
This approach uses BFS to traverse the tree level by level and counts the number of levels.

Steps:
1. If the root is `null`, return `0`.
2. Use a queue to perform level-order traversal.
3. Initialize a counter for the depth.
4. For each level, increment the depth counter and process all nodes at that level.
5. Return the depth counter.

Code:

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
```

```java
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }

        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);
        int depth = 0;

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            for (int i = 0; i < levelSize; i++) {
                TreeNode node = queue.poll();
                if (node.left != null) {
                    queue.add(node.left);
                }
                if (node.right != null) {
                    queue.add(node.right);
                }
            }
            depth++;
        }

        return depth;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(N) for the queue in the worst case (when the tree is a complete binary tree).
- This approach uses BFS to traverse the tree level by level.

Approach 3: Iterative Depth-First Search (DFS) with Stack
This approach uses an iterative DFS with a stack to simulate the recursion stack and track the depth of each node.

Steps:
1. If the root is `null`, return `0`.
2. Use a stack to perform DFS and store each node along with its depth.
3. Initialize a variable to track the maximum depth.
4. Traverse the tree using the stack and update the maximum depth.
5. Return the maximum depth.

Code:

```java
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    public int maxDepth(TreeNode root) {
        if (root == null) {
            return 0;
        }

        Stack<Pair<TreeNode, Integer>> stack = new Stack<>();
        stack.push(new Pair<>(root, 1));
        int maxDepth = 0;

        while (!stack.isEmpty()) {
            Pair<TreeNode, Integer> current = stack.pop();
            TreeNode node = current.getKey();
            int depth = current.getValue();
            maxDepth = Math.max(maxDepth, depth);

            if (node.left != null) {
                stack.push(new Pair<>(node.left, depth + 1));
            }
            if (node.right != null) {
                stack.push(new Pair<>(node.right, depth + 1));
            }
        }

        return maxDepth;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(H), where `H` is the height of the tree (stack space).
- This approach uses an iterative DFS with a stack to simulate recursion.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Recursion (DFS)	O(N)	O(H)	Simple and intuitive	Uses recursion stack space
BFS (Level-Order)	O(N)	O(N)	Traverses level by level	Uses extra space for queue
Iterative DFS (Stack)	O(N)	O(H)	Simulates recursion	Uses extra space for stack

Conclusion:
- Use the Recursion (DFS) Approach for a simple and intuitive solution.
- Use the BFS (Level-Order) Approach for a solution that traverses the tree level by level.
- Use the Iterative DFS (Stack) Approach for a solution that simulates recursion without using the call stack.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Same Tree

The Same Tree problem involves determining whether two binary trees are identical. Two trees are considered the same if they are structurally identical and the nodes have the same values. This problem can be solved using recursion, iterative DFS (using a stack), or iterative BFS (using a queue).

Problem Definition:
Given the roots of two binary trees `p` and `q`, determine if they are the same.

Example:

Input:

```
Tree p:      Tree q:
    1            1
   / \          / \
  2   3        2   3
```

Output: `true` (Both trees are identical.)

Approach 1: Recursion (Depth-First Search - DFS)
This approach uses recursion to compare the structure and values of the two trees.

Steps:
1. If both trees are `null`, they are the same (base case).
2. If one tree is `null` and the other is not, they are not the same.
3. Compare the values of the current nodes.
4. Recursively check the left and right subtrees.

Code:

```java
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        // Base case: both trees are null
        if (p == null && q == null) {
            return true;
        }
        // One tree is null, the other is not
        if (p == null || q == null) {
            return false;
        }
        // Compare node values
        if (p.val != q.val) {
            return false;
        }
        // Recursively check left and right subtrees
        return isSameTree(p.left, q.left) && isSameTree(p.right, q.right);
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(H), where `H` is the height of the tree (recursion stack space).
- This approach is simple and uses recursion to compare the trees.

Approach 2: Iterative DFS (Using a Stack)
This approach uses an iterative DFS with a stack to compare the structure and values of the two trees.

Steps:
1. Use a stack to perform DFS and store pairs of nodes from both trees.
2. Pop a pair of nodes from the stack and compare their values.
3. Push the left and right children of the nodes onto the stack if they exist.
4. If the stack is empty and no mismatches are found, the trees are the same.

Code:

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        Stack<Pair<TreeNode, TreeNode>> stack = new Stack<>();
        stack.push(new Pair<>(p, q));

        while (!stack.isEmpty()) {
            Pair<TreeNode, TreeNode> current = stack.pop();
            TreeNode nodeP = current.getKey();
            TreeNode nodeQ = current.getValue();

            // Both nodes are null
            if (nodeP == null && nodeQ == null) {
                continue;
            }
            // One node is null, the other is not
            if (nodeP == null || nodeQ == null) {
                return false;
            }
            // Compare node values
            if (nodeP.val != nodeQ.val) {
                return false;
            }

            // Push left and right children onto the stack
            stack.push(new Pair<>(nodeP.left, nodeQ.left));
            stack.push(new Pair<>(nodeP.right, nodeQ.right));
        }

        return true;
    }
}
```

<u>Explanation</u>:
- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(H), where `H` is the height of the tree (stack space).
- This approach uses an iterative DFS with a stack to compare the trees.

Approach 3: Iterative BFS (Using a Queue)
This approach uses an iterative BFS with a queue to compare the structure and values of the two trees.

Steps:
1. Use a queue to perform BFS and store pairs of nodes from both trees.
2. Dequeue a pair of nodes and compare their values.
3. Enqueue the left and right children of the nodes if they exist.
4. If the queue is empty and no mismatches are found, the trees are the same.

Code:

```java
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        Queue<Pair<TreeNode, TreeNode>> queue = new LinkedList<>();
        queue.add(new Pair<>(p, q));

        while (!queue.isEmpty()) {
            Pair<TreeNode, TreeNode> current = queue.poll();
            TreeNode nodeP = current.getKey();
            TreeNode nodeQ = current.getValue();

            // Both nodes are null
            if (nodeP == null && nodeQ == null) {
                continue;
            }
            // One node is null, the other is not
            if (nodeP == null || nodeQ == null) {
                return false;
            }
            // Compare node values
            if (nodeP.val != nodeQ.val) {
                return false;
            }

            // Enqueue left and right children
            queue.add(new Pair<>(nodeP.left, nodeQ.left));
            queue.add(new Pair<>(nodeP.right, nodeQ.right));
        }

        return true;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(N) for the queue in the worst case (when the tree is a complete binary tree).
- This approach uses an iterative BFS with a queue to compare the trees.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Recursion (DFS)	O(N)	O(H)	Simple and intuitive	Uses recursion stack space
Iterative DFS (Stack)	O(N)	O(H)	Simulates recursion	Uses extra space for stack
Iterative BFS (Queue)	O(N)	O(N)	Traverses level by level	Uses extra space for queue

Conclusion:
- Use the Recursion (DFS) Approach for a simple and intuitive solution.
- Use the Iterative DFS (Stack) Approach for a solution that simulates recursion without using the call stack.
- Use the Iterative BFS (Queue) Approach for a solution that traverses the tree level by level.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Invert/Flip Binary Tree

The Invert/Flip Binary Tree problem involves flipping a binary tree such that the left and right children of every node are swapped. This problem can be solved using recursion, iterative DFS (using a stack), or iterative BFS (using a queue).

Problem Definition:
Given the root of a binary tree, invert the tree and return its root.

Example:
Input:

```
    4
  /   \
  2     7
 / \   / \
1   3 6   9
```

Output:

```
    4
  /  \
 7    2
/ \  / \
9  6 3  1
```

Approach 1: Recursion (Depth-First Search - DFS)
This approach uses recursion to swap the left and right children of each node.

Steps:
1. If the root is `null`, return `null` (base case).
2. Swap the left and right children of the current node.
3. Recursively invert the left and right subtrees.
4. Return the root of the inverted tree.

Code:

```java
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    public TreeNode invertTree(TreeNode root) {
        if (root == null) {
            return null;
        }

        // Swap the left and right children
        TreeNode temp = root.left;
        root.left = root.right;
        root.right = temp;

        // Recursively invert the left and right subtrees
        invertTree(root.left);
        invertTree(root.right);
        return root;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(H), where `H` is the height of the tree (recursion stack space).
- This approach is simple and uses recursion to invert the tree.

Approach 2: Iterative DFS (Using a Stack)
This approach uses an iterative DFS with a stack to swap the left and right children of each node.

Steps:
1. If the root is `null`, return `null`.
2. Use a stack to perform DFS and store nodes.
3. Swap the left and right children of each node.
4. Push the children of the current node onto the stack.
5. Return the root of the inverted tree.

Code:

```java
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    public TreeNode invertTree(TreeNode root) {
        if (root == null) {
            return null;
        }

        Stack<TreeNode> stack = new Stack<>();
        stack.push(root);

        while (!stack.isEmpty()) {
            TreeNode node = stack.pop();

            // Swap the left and right children
            TreeNode temp = node.left;
            node.left = node.right;
            node.right = temp;

            // Push the children onto the stack
            if (node.left != null) {
                stack.push(node.left);
            }
            if (node.right != null) {
                stack.push(node.right);
            }
        }

        return root;
    }
}
```

<u>Explanation</u>:
- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(H), where `H` is the height of the tree (stack space).
- This approach uses an iterative DFS with a stack to invert the tree.

Approach 3: Iterative BFS (Using a Queue)
This approach uses an iterative BFS with a queue to swap the left and right children of each node.

Steps:
1. If the root is `null`, return `null`.
2. Use a queue to perform BFS and store nodes.
3. Swap the left and right children of each node.
4. Enqueue the children of the current node.
5. Return the root of the inverted tree.

Code:

```java
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    public TreeNode invertTree(TreeNode root) {
        if (root == null) {
            return null;
        }

        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);

        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();

            // Swap the left and right children
            TreeNode temp = node.left;
            node.left = node.right;
            node.right = temp;

            // Enqueue the children
            if (node.left != null) {
                queue.add(node.left);
            }
            if (node.right != null) {
                queue.add(node.right);
            }
        }

        return root;
    }
}
```

Explanation:

- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(N) for the queue in the worst case (when the tree is a complete binary tree).
- This approach uses an iterative BFS with a queue to invert the tree.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Recursion (DFS)	O(N)	O(H)	Simple and intuitive	Uses recursion stack space
Iterative DFS (Stack)	O(N)	O(H)	Simulates recursion	Uses extra space for stack
Iterative BFS (Queue)	O(N)	O(N)	Traverses level by level	Uses extra space for queue

Conclusion:
- Use the Recursion (DFS) Approach for a simple and intuitive solution.
- Use the Iterative DFS (Stack) Approach for a solution that simulates recursion without using the call stack.
- Use the Iterative BFS (Queue) Approach for a solution that traverses the tree level by level.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Binary Tree Maximum Path Sum

The Binary Tree Maximum Path Sum problem involves finding the maximum path sum in a binary tree. A path is defined as any sequence of nodes from some starting node to any node in the tree along the parent-child connections. The path must contain at least one node. This problem can be solved using recursion or dynamic programming.

Problem Definition:
Given the root of a binary tree, find the maximum path sum.

Example:
Input:

```
   -10
   / \
  9   20
     / \
    15   7
```

Output: `42` (The path with the maximum sum is `15 -> 20 -> 7`.)

Approach 1: Recursion (Post-Order Traversal)
This approach uses recursion to calculate the maximum path sum for each node and updates the global maximum path sum.

Steps:
1. Define a recursive function that returns the maximum path sum starting from the current node.
2. For each node, calculate the maximum path sum that includes the left child, the right child, or just the node itself.
3. Update the global maximum path sum with the maximum path sum that includes the current node as the root of the path.
4. Return the maximum path sum that can be extended to the parent node.

Code:

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    private int maxSum = Integer.MIN_VALUE;

    public int maxPathSum(TreeNode root) {
        maxGain(root);
        return maxSum;
    }

    private int maxGain(TreeNode node) {
        if (node == null) {
            return 0;
        }

        // Calculate the maximum gain from the left and right subtrees
        int leftGain = Math.max(maxGain(node.left), 0); // Ignore negative
gains
```

```
        int rightGain = Math.max(maxGain(node.right), 0); // Ignore
negative gains

        // Update the global maximum path sum
        int currentPathSum = node.val + leftGain + rightGain;
        maxSum = Math.max(maxSum, currentPathSum);

        // Return the maximum gain that can be extended to the parent node
        return node.val + Math.max(leftGain, rightGain);
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(H), where `H` is the height of the tree (recursion stack space).
- This approach uses recursion to calculate the maximum path sum for each node.

Approach 2: Iterative Post-Order Traversal (Using a Stack)
This approach uses an iterative post-order traversal with a stack to calculate the maximum path sum.

Steps:
1. Use a stack to perform post-order traversal and store nodes.
2. Track the maximum path sum for each node.
3. Update the global maximum path sum with the maximum path sum that includes the current node as the root of the path.
4. Return the global maximum path sum.

Code:

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}
```

```
class Solution {
    public int maxPathSum(TreeNode root) {
        if (root == null) {
            return 0;
        }

        int maxSum = Integer.MIN_VALUE;
        Stack<TreeNode> stack = new Stack<>();
        Map<TreeNode, Integer> maxGainMap = new HashMap<>();
        stack.push(root);

        while (!stack.isEmpty()) {
            TreeNode node = stack.peek();

            if (node.left != null && !maxGainMap.containsKey(node.left)) {
                stack.push(node.left);
            } else if (node.right != null &&
!maxGainMap.containsKey(node.right)) {
                stack.push(node.right);
            } else {
                stack.pop();

                // Calculate the maximum gain from the left and right
subtrees
                int leftGain = Math.max(maxGainMap.getOrDefault(node.left,
0), 0); // Ignore negative gains
                int rightGain =
Math.max(maxGainMap.getOrDefault(node.right, 0), 0); // Ignore negative
gains

                // Update the global maximum path sum
                int currentPathSum = node.val + leftGain + rightGain;
                maxSum = Math.max(maxSum, currentPathSum);

                // Store the maximum gain for the current node
                maxGainMap.put(node, node.val + Math.max(leftGain,
rightGain));
            }
        }

        return maxSum;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(N) for the stack and the map.
- This approach uses an iterative post-order traversal with a stack to calculate the maximum path sum.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
Recursion (Post-Order)	O(N)	O(H)	Simple and intuitive	Uses recursion stack space
Iterative Post-Order	O(N)	O(N)	Simulates recursion	Uses extra space for stack and map

Conclusion:
- Use the Recursion (Post-Order) Approach for a simple and intuitive solution.
- Use the Iterative Post-Order Approach for a solution that simulates recursion without using the call stack.
- Both approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Binary Tree Level Order Traversal

The Binary Tree Level Order Traversal problem involves traversing a binary tree level by level and returning the nodes at each level as a list of lists. This problem can be solved using BFS (level-order traversal), DFS (pre-order traversal with level tracking), or iterative BFS (using a queue).

Problem Definition:
Given the root of a binary tree, return the level order traversal of its nodes' values. (i.e., from left to right, level by level).

Example:
Input:

```
   3
  / \
 9  20
   /  \
  15   7
```

Output:

```
[
  [3],
  [9, 20],
  [15, 7]
]
```

Approach 1: BFS (Level-Order Traversal) with Queue
This approach uses a queue to perform BFS and traverse the tree level by level.

Steps:
1. If the root is `null`, return an empty list.
2. Use a queue to perform BFS and store nodes.
3. For each level, process all nodes in the queue and add their children to the queue.
4. Add the nodes' values at each level to the result list.

Code:

```java
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) {
            return result;
        }

        Queue<TreeNode> queue = new LinkedList<>();
        queue.add(root);

        while (!queue.isEmpty()) {
            int levelSize = queue.size();
            List<Integer> currentLevel = new ArrayList<>();

            for (int i = 0; i < levelSize; i++) {
                TreeNode node = queue.poll();
                currentLevel.add(node.val);

                if (node.left != null) {
                    queue.add(node.left);
                }
                if (node.right != null) {
                    queue.add(node.right);
                }
            }

            result.add(currentLevel);
        }

        return result;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(N) for the queue in the worst case (when the tree is a complete binary tree).
- This approach uses BFS to traverse the tree level by level.

Approach 2: DFS (Pre-Order Traversal) with Level Tracking
This approach uses DFS with level tracking to traverse the tree and store nodes at each level.

Steps:
1. If the root is `null`, return an empty list.
2. Use a list to store nodes at each level.
3. Perform DFS and pass the current level to the recursive function.
4. Add the nodes' values to the corresponding level in the result list.

Code:

```
import java.util.*;
```

```java
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) {
            return result;
        }

        dfs(root, 0, result);
        return result;
    }

    private void dfs(TreeNode node, int level, List<List<Integer>> result)
{
        if (node == null) {
            return;
        }

        // Add a new list for the current level if it doesn't exist
        if (result.size() <= level) {
            result.add(new ArrayList<>());
        }

        // Add the current node's value to the corresponding level
        result.get(level).add(node.val);

        // Recursively traverse the left and right subtrees
        dfs(node.left, level + 1, result);
        dfs(node.right, level + 1, result);
    }
}
```

Explanation:

- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(H), where `H` is the height of the tree (recursion stack space).
- This approach uses DFS with level tracking to traverse the tree.

Approach 3: Iterative BFS (Using Two Queues)
This approach uses two queues to perform BFS and traverse the tree level by level.

Steps:
1. If the root is `null`, return an empty list.
2. Use two queues to alternate between levels.
3. For each level, process all nodes in the current queue and add their children to the next queue.
4. Add the nodes' values at each level to the result list.

Code:

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int val) {
        this.val = val;
    }
}

class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> result = new ArrayList<>();
        if (root == null) {
            return result;
        }

        Queue<TreeNode> currentQueue = new LinkedList<>();
        Queue<TreeNode> nextQueue = new LinkedList<>();
        currentQueue.add(root);

        while (!currentQueue.isEmpty()) {
            List<Integer> currentLevel = new ArrayList<>();

            while (!currentQueue.isEmpty()) {
                TreeNode node = currentQueue.poll();
                currentLevel.add(node.val);

                if (node.left != null) {
                    nextQueue.add(node.left);
                }
                if (node.right != null) {
                    nextQueue.add(node.right);
                }
            }

            result.add(currentLevel);
            // Swap the queues
            Queue<TreeNode> temp = currentQueue;
            currentQueue = nextQueue;
            nextQueue = temp;
        }

        return result;
    }
}
```

Explanation:
- Time Complexity: O(N), where `N` is the number of nodes in the tree (each node is visited once).
- Space Complexity: O(N) for the queues in the worst case (when the tree is a complete binary tree).
- This approach uses two queues to alternate between levels.

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Pros	Cons
BFS (Queue)	O(N)	O(N)	Simple and straightforward	Uses extra space for queue
DFS (Level Tracking)	O(N)	O(H)	Uses recursion	Requires careful level tracking
Iterative BFS (Two Queues)	O(N)	O(N)	Alternates between levels	Uses extra space for two queues

Conclusion:
- Use the BFS (Queue) Approach for a simple and straightforward solution.
- Use the DFS (Level Tracking) Approach for a solution that uses recursion and level tracking.
- Use the Iterative BFS (Two Queues) Approach for a solution that alternates between levels using two queues.
- All approaches have their trade-offs, so the choice depends on the specific use case and constraints.

Serialize and Deserialize Binary Tree

Serialization and deserialization of a binary tree are common tasks in computer science, especially when you need to store or transmit the structure of a tree. Below are different approaches to serialize and deserialize a binary tree in Java.

1. Preorder Traversal with Null Markers

This approach uses a preorder traversal (Root -> Left -> Right) and marks null nodes with a special character (e.g., `#`).

Serialization:
- Traverse the tree in preorder.
- Append each node's value to a string.
- If a node is null, append a special marker (e.g., `#`).

Deserialization:
- Split the serialized string into tokens.
- Reconstruct the tree using preorder traversal.

```java
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class Codec {
    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        buildString(root, sb);
        return sb.toString();
    }

    private void buildString(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append("#").append(",");
        } else {
            sb.append(node.val).append(",");
            buildString(node.left, sb);
            buildString(node.right, sb);
        }
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        Queue<String> nodes = new
LinkedList<>(Arrays.asList(data.split(",")));
        return buildTree(nodes);
    }

    private TreeNode buildTree(Queue<String> nodes) {
        String val = nodes.poll();
        if (val.equals("#")) return null;
        TreeNode node = new TreeNode(Integer.parseInt(val));
        node.left = buildTree(nodes);
        node.right = buildTree(nodes);
        return node;
    }

    public static void main(String[] args) {
        Codec codec = new Codec();
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.right.left = new TreeNode(4);
        root.right.right = new TreeNode(5);

        String serialized = codec.serialize(root);
        System.out.println("Serialized: " + serialized);
```

```java
        TreeNode deserialized = codec.deserialize(serialized);
        System.out.println("Deserialized: " +
codec.serialize(deserialized));
    }
}
```

2. Level Order Traversal (BFS)

This approach uses a level-order traversal (BFS) to serialize and deserialize the tree.

Serialization:
- Use a queue to perform a level-order traversal.
- Append each node's value to a string.
- If a node is null, append a special marker (e.g., `#`).

Deserialization:
- Split the serialized string into tokens.
- Reconstruct the tree using a queue.

```java
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class Codec {

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        if (root == null) return "";
        Queue<TreeNode> queue = new LinkedList<>();
        StringBuilder sb = new StringBuilder();
        queue.offer(root);
        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();
            if (node == null) {
                sb.append("#").append(",");
            } else {
                sb.append(node.val).append(",");
                queue.offer(node.left);
                queue.offer(node.right);
            }
        }
        return sb.toString();
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        if (data.isEmpty()) return null;
        Queue<String> nodes = new
LinkedList<>(Arrays.asList(data.split(",")));
        Queue<TreeNode> queue = new LinkedList<>();
        TreeNode root = new TreeNode(Integer.parseInt(nodes.poll()));
        queue.offer(root);
        while (!queue.isEmpty()) {
            TreeNode node = queue.poll();
            String leftVal = nodes.poll();
            if (!leftVal.equals("#")) {
                node.left = new TreeNode(Integer.parseInt(leftVal));
                queue.offer(node.left);
            }
            String rightVal = nodes.poll();
            if (!rightVal.equals("#")) {
                node.right = new TreeNode(Integer.parseInt(rightVal));
                queue.offer(node.right);
            }
        }
        return root;
    }

    public static void main(String[] args) {
        Codec codec = new Codec();
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.right.left = new TreeNode(4);
        root.right.right = new TreeNode(5);

        String serialized = codec.serialize(root);
        System.out.println("Serialized: " + serialized);

        TreeNode deserialized = codec.deserialize(serialized);
        System.out.println("Deserialized: " +
codec.serialize(deserialized));
    }
}
```

3. Postorder Traversal with Null Markers

This approach uses a postorder traversal (Left -> Right -> Root) and marks null nodes with a special character (e.g., `#`).

Serialization:
- Traverse the tree in postorder.
- Append each node's value to a string.
- If a node is null, append a special marker (e.g., `#`).

Deserialization:
- Split the serialized string into tokens.
- Reconstruct the tree using postorder traversal.

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class Codec {

    // Encodes a tree to a single string.
    public String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        buildString(root, sb);
        return sb.toString();
    }
    private void buildString(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append("#").append(",");
        } else {
            buildString(node.left, sb);
            buildString(node.right, sb);
            sb.append(node.val).append(",");
        }
    }

    // Decodes your encoded data to tree.
    public TreeNode deserialize(String data) {
        Stack<String> nodes = new Stack<>();
        nodes.addAll(Arrays.asList(data.split(",")));
        return buildTree(nodes);
    }

    private TreeNode buildTree(Stack<String> nodes) {
        String val = nodes.pop();
        if (val.equals("#")) return null;
        TreeNode node = new TreeNode(Integer.parseInt(val));
        node.right = buildTree(nodes);
        node.left = buildTree(nodes);
        return node;
    }

    public static void main(String[] args) {
        Codec codec = new Codec();
        TreeNode root = new TreeNode(1);
        root.left = new TreeNode(2);
        root.right = new TreeNode(3);
        root.right.left = new TreeNode(4);
        root.right.right = new TreeNode(5);

        String serialized = codec.serialize(root);
        System.out.println("Serialized: " + serialized);

        TreeNode deserialized = codec.deserialize(serialized);
        System.out.println("Deserialized: " +
codec.serialize(deserialized));
    }
}
```

Explanation:

1. Preorder Traversal with Null Markers:
   - Simple and easy to implement.
   - Uses a recursive approach to serialize and deserialize the tree.
   - The serialized string is compact.

2. Level Order Traversal (BFS):
   - Uses an iterative approach with a queue.
   - Suitable for trees that are not too deep.
   - The serialized string is longer but easier to reconstruct.

3. Postorder Traversal with Null Markers:
   - Similar to preorder but processes children before the root.
   - Uses a stack for deserialization.
   - The serialized string is compact.

Each approach has its own advantages and trade-offs, so the choice depends on the specific requirements of your application.

Subtree of Another Tree

The problem "Subtree of Another Tree" involves determining whether a given tree `t` is a subtree of another tree `s`. A subtree of `s` is a tree that consists of a node in `s` and all of its descendants. Below are different approaches to solve this problem in Java.

1. Recursive Approach (Brute Force)

Explanation:
- Traverse the main tree `s` recursively.
- For each node in `s`, check if the subtree rooted at that node is identical to `t`.
- If a match is found, return `true`; otherwise, continue searching.

Time Complexity:
- Worst Case: `O(m * n)`, where `m` is the number of nodes in `s` and `n` is the number of nodes in `t`.
- This is because, in the worst case, we compare `t` with every subtree of `s`.

Code:

```java
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class SubtreeOfAnotherTree {

    // Main function to check if t is a subtree of s
    public boolean isSubtree(TreeNode s, TreeNode t) {
        if (s == null) {
            return t == null; // If s is null, t must also be null to be a
subtree
        }
        // Check if the current subtree of s matches t
        if (isSameTree(s, t)) {
            return true;
        }
        // Recursively check the left and right subtrees of s
        return isSubtree(s.left, t) || isSubtree(s.right, t);
    }

    // Helper function to check if two trees are identical
    private boolean isSameTree(TreeNode s, TreeNode t) {
        if (s == null && t == null) {
            return true; // Both trees are null, so they are identical
        }
        if (s == null || t == null) {
            return false; // One tree is null, the other is not
        }
        // Check if the current nodes' values are equal and recursively
check left and right subtrees
        return s.val == t.val && isSameTree(s.left, t.left) &&
isSameTree(s.right, t.right);
    }

    public static void main(String[] args) {
        SubtreeOfAnotherTree solution = new SubtreeOfAnotherTree();

        // Example 1
        TreeNode s = new TreeNode(3);
        s.left = new TreeNode(4);
        s.right = new TreeNode(5);
        s.left.left = new TreeNode(1);
        s.left.right = new TreeNode(2);

        TreeNode t = new TreeNode(4);
        t.left = new TreeNode(1);
        t.right = new TreeNode(2);

        System.out.println(solution.isSubtree(s, t)); // Output: true

        // Example 2
        s.left.right.left = new TreeNode(0); // Modify s to make t not a
subtree
        System.out.println(solution.isSubtree(s, t)); // Output: false
    }
}
```

2. Serialization Approach (Using Preorder Traversal)

Explanation:
- Serialize both trees `s` and `t` into strings using preorder traversal.
- Include `null` markers (e.g., `#`) to represent null nodes.
- Check if the serialized string of `t` is a substring of the serialized string of `s`.

Time Complexity:
- Serialization: `O(m + n)`, where `m` and `n` are the number of nodes in `s` and `t`, respectively.
- Substring Check: `O(m + n)` (using KMP or other efficient substring search algorithms).

Code:

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
```

```java
}

public class SubtreeOfAnotherTree {

    // Main function to check if t is a subtree of s
    public boolean isSubtree(TreeNode s, TreeNode t) {
        String tree1 = serialize(s);
        String tree2 = serialize(t);
        return tree1.contains(tree2);
    }

    // Helper function to serialize a tree using preorder traversal
    private String serialize(TreeNode root) {
        StringBuilder sb = new StringBuilder();
        buildString(root, sb);
        return sb.toString();
    }

    private void buildString(TreeNode node, StringBuilder sb) {
        if (node == null) {
            sb.append("#").append(","); // Use '#' to represent null nodes
        } else {
            sb.append(node.val).append(",");
            buildString(node.left, sb);
            buildString(node.right, sb);
        }
    }

    public static void main(String[] args) {
        SubtreeOfAnotherTree solution = new SubtreeOfAnotherTree();

        // Example 1
        TreeNode s = new TreeNode(3);
        s.left = new TreeNode(4);
        s.right = new TreeNode(5);
        s.left.left = new TreeNode(1);
        s.left.right = new TreeNode(2);

        TreeNode t = new TreeNode(4);
        t.left = new TreeNode(1);
        t.right = new TreeNode(2);

        System.out.println(solution.isSubtree(s, t)); // Output: true
        // Example 2
        s.left.right.left = new TreeNode(0); // Modify s to make t not a
subtree
        System.out.println(solution.isSubtree(s, t)); // Output: false
    }
}
```

3. Optimized Recursive Approach (Using Hashing)

Explanation:
- Use a hashing mechanism to represent each subtree uniquely.
- Compute the hash of `t` and compare it with the hash of every subtree of `s`.
- If a matching hash is found, verify the subtree structure.

Time Complexity:
- Average Case: `O(m + n)`, where `m` and `n` are the number of nodes in `s` and `t`, respectively.
- Worst Case: `O(m * n)` (if hash collisions occur frequently).

Code:

```
import java.util.*;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class SubtreeOfAnotherTree {

    // Main function to check if t is a subtree of s
    public boolean isSubtree(TreeNode s, TreeNode t) {
        String hashT = computeHash(t);
        return checkSubtree(s, t, hashT);
    }
    // Helper function to compute the hash of a tree
    private String computeHash(TreeNode root) {
        if (root == null) {
            return "#";
        }
        String leftHash = computeHash(root.left);
        String rightHash = computeHash(root.right);
        return root.val + "," + leftHash + "," + rightHash;
    }

    // Helper function to check if any subtree of s matches t
    private boolean checkSubtree(TreeNode s, TreeNode t, String hashT) {
        if (s == null) {
            return false;
        }
        String hashS = computeHash(s);
        if (hashS.equals(hashT)) {
            return true;
        }
        return checkSubtree(s.left, t, hashT) || checkSubtree(s.right, t,
hashT);
    }

    public static void main(String[] args) {
        SubtreeOfAnotherTree solution = new SubtreeOfAnotherTree();

        // Example 1
        TreeNode s = new TreeNode(3);
        s.left = new TreeNode(4);
        s.right = new TreeNode(5);
        s.left.left = new TreeNode(1);
        s.left.right = new TreeNode(2);

        TreeNode t = new TreeNode(4);
        t.left = new TreeNode(1);
        t.right = new TreeNode(2);

        System.out.println(solution.isSubtree(s, t)); // Output: true

        // Example 2
        s.left.right.left = new TreeNode(0); // Modify s to make t not a
subtree
```

```
        System.out.println(solution.isSubtree(s, t)); // Output: false
    }
}
```

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Notes
Recursive (Brute Force)	`O(m * n)`	`O(m)`	Simple but inefficient for large trees.
Serialization	`O(m + n)`	`O(m + n)`	Efficient but requires additional space for serialized strings.
Hashing	`O(m + n)`	`O(m + n)`	Efficient but may have hash collisions.

Choose the approach based on the constraints and requirements of your problem. For most cases, the Serialization Approach is a good balance between simplicity and efficiency.

Construct Binary Tree from Preorder and Inorder Traversal

Constructing a binary tree from its preorder and inorder traversals is a classic problem in computer science. Below are different approaches to solve this problem in Java, along with explanations.

1. Recursive Approach

Explanation:
- Preorder Traversal: The first element is always the root of the tree.
- Inorder Traversal: The elements to the left of the root in the inorder sequence form the left subtree, and the elements to the right form the right subtree.
- Use recursion to construct the left and right subtrees.

Steps:
1. Find the root from the preorder traversal.
2. Locate the root in the inorder traversal to determine the left and right subtrees.
3. Recursively build the left and right subtrees.

Time Complexity:
- Worst Case: `O(n^2)` (if the tree is skewed).
- Average Case: `O(n)` (if the tree is balanced).

Code:

```java
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class ConstructBinaryTree {

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        return buildTreeHelper(preorder, inorder, 0, 0, inorder.length -
1);
    }

    private TreeNode buildTreeHelper(int[] preorder, int[] inorder, int
preStart, int inStart, int inEnd) {
        if (preStart >= preorder.length || inStart > inEnd) {
            return null;
        }

        // The first element in preorder is the root
        TreeNode root = new TreeNode(preorder[preStart]);

        // Find the index of the root in inorder traversal
        int inIndex = 0;
        for (int i = inStart; i <= inEnd; i++) {
            if (inorder[i] == root.val) {
                inIndex = i;
                break;
            }
        }

        // Recursively build the left and right subtrees
        root.left = buildTreeHelper(preorder, inorder, preStart + 1,
inStart, inIndex - 1);
        root.right = buildTreeHelper(preorder, inorder, preStart + (inIndex
- inStart) + 1, inIndex + 1, inEnd);

        return root;
    }
    public static void main(String[] args) {
        ConstructBinaryTree solution = new ConstructBinaryTree();

        int[] preorder = {3, 9, 20, 15, 7};
        int[] inorder = {9, 3, 15, 20, 7};

        TreeNode root = solution.buildTree(preorder, inorder);

        // Print the tree (inorder traversal to verify)
        System.out.println("Inorder Traversal of Constructed Tree:");
        printInorder(root);
    }

    private static void printInorder(TreeNode root) {
        if (root == null) return;
        printInorder(root.left);
        System.out.print(root.val + " ");
        printInorder(root.right);
    }
}
```

2. Optimized Recursive Approach (Using HashMap)

Explanation:
- Use a `HashMap` to store the indices of the inorder traversal for quick lookup.
- This avoids the need to search for the root index in the inorder array repeatedly.

Time Complexity:
- Average Case: `O(n)` (since we avoid repeated searches in the inorder array).

Code:

```
import java.util.HashMap;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}
```

```java
public class ConstructBinaryTree {

    private HashMap<Integer, Integer> inorderMap;
    private int preIndex;

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        inorderMap = new HashMap<>();
        preIndex = 0;

        // Populate the HashMap with inorder indices
        for (int i = 0; i < inorder.length; i++) {
            inorderMap.put(inorder[i], i);
        }

        return buildTreeHelper(preorder, 0, inorder.length - 1);
    }

    private TreeNode buildTreeHelper(int[] preorder, int inStart, int
inEnd) {
        if (inStart > inEnd) {
            return null;
        }

        // The current element in preorder is the root
        TreeNode root = new TreeNode(preorder[preIndex]);
        preIndex++;

        // Find the index of the root in inorder traversal
        int inIndex = inorderMap.get(root.val);

        // Recursively build the left and right subtrees
        root.left = buildTreeHelper(preorder, inStart, inIndex - 1);
        root.right = buildTreeHelper(preorder, inIndex + 1, inEnd);

        return root;
    }

    public static void main(String[] args) {
        ConstructBinaryTree solution = new ConstructBinaryTree();

        int[] preorder = {3, 9, 20, 15, 7};
        int[] inorder = {9, 3, 15, 20, 7};
        TreeNode root = solution.buildTree(preorder, inorder);

        // Print the tree (inorder traversal to verify)
        System.out.println("Inorder Traversal of Constructed Tree:");
        printInorder(root);
    }

    private static void printInorder(TreeNode root) {
        if (root == null) return;
        printInorder(root.left);
        System.out.print(root.val + " ");
        printInorder(root.right);
    }
}
```

3. Iterative Approach (Using Stack)

Explanation:
- Use a stack to simulate the recursive process.
- Traverse the preorder array and construct the tree iteratively.

Steps:
1. Push the root node (first element of preorder) onto the stack.
2. Traverse the preorder array:
   - If the current element is not equal to the top of the stack, it is the left child of the top.
   - If the current element is equal to the top of the stack, pop elements from the stack until the current element is not equal to the top, and then assign it as the right child.

Time Complexity:
- Average Case: `O(n)`.

Code:

```
import java.util.Stack;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
```

```java
    TreeNode(int x) { val = x; }
}

public class ConstructBinaryTree {

    public TreeNode buildTree(int[] preorder, int[] inorder) {
        if (preorder.length == 0) return null;

        Stack<TreeNode> stack = new Stack<>();
        TreeNode root = new TreeNode(preorder[0]);
        stack.push(root);

        int inIndex = 0;

        for (int i = 1; i < preorder.length; i++) {
            TreeNode currentNode = stack.peek();
            if (currentNode.val != inorder[inIndex]) {
                // The current element is the left child of the top of the
stack
                currentNode.left = new TreeNode(preorder[i]);
                stack.push(currentNode.left);
            } else {
                // The current element is the right child of a node in the
stack
                while (!stack.isEmpty() && stack.peek().val ==
inorder[inIndex]) {
                    currentNode = stack.pop();
                    inIndex++;
                }
                currentNode.right = new TreeNode(preorder[i]);
                stack.push(currentNode.right);
            }
        }

        return root;
    }

    public static void main(String[] args) {
        ConstructBinaryTree solution = new ConstructBinaryTree();

        int[] preorder = {3, 9, 20, 15, 7};
        int[] inorder = {9, 3, 15, 20, 7};
        TreeNode root = solution.buildTree(preorder, inorder);

        // Print the tree (inorder traversal to verify)
        System.out.println("Inorder Traversal of Constructed Tree:");
        printInorder(root);
    }

    private static void printInorder(TreeNode root) {
        if (root == null) return;
        printInorder(root.left);
        System.out.print(root.val + " ");
        printInorder(root.right);
    }
}
```

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Notes
Recursive	`O(n^2)`	`O(n)`	Simple but inefficient for skewed trees.
Optimized Recursive	`O(n)`	`O(n)`	Uses a HashMap for efficient lookups.
Iterative	`O(n)`	`O(n)`	Simulates recursion using a stack.

Choose the approach based on the constraints and requirements of your problem. The Optimized Recursive Approach is generally the most efficient and easy to implement.

Validate Binary Search Tree

Validating a Binary Search Tree (BST) involves checking whether a given binary tree satisfies the BST property:
- For every node, all nodes in its left subtree must have values less than the node's value.
- For every node, all nodes in its right subtree must have values greater than the node's value.

Below are different approaches to solve this problem in Java, along with explanations.

1. Recursive Approach (Using Valid Range)

Explanation:

- Traverse the tree recursively.
- For each node, define a valid range `(min, max)` that its value must lie within.
- Initially, the root can have any value, so the range is `(Integer.MIN_VALUE, Integer.MAX_VALUE)`.
- For the left subtree, update the range to `(min, current node's value)`.
- For the right subtree, update the range to `(current node's value, max)`.

Time Complexity:
- Average Case: `O(n)`, where `n` is the number of nodes in the tree.
- Worst Case: `O(n)` (if the tree is skewed).

Code:

```java
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class ValidateBST {

    public boolean isValidBST(TreeNode root) {
        return isValidBSTHelper(root, Long.MIN_VALUE, Long.MAX_VALUE);
    }

    private boolean isValidBSTHelper(TreeNode node, long min, long max) {
        if (node == null) {
            return true; // An empty tree is a valid BST
        }
        // Check if the current node's value is within the valid range
        if (node.val <= min || node.val >= max) {
            return false;
        }
        // Recursively check the left and right subtrees
        return isValidBSTHelper(node.left, min, node.val) &&
isValidBSTHelper(node.right, node.val, max);
    }

    public static void main(String[] args) {
        ValidateBST solution = new ValidateBST();
        // Example 1: Valid BST
        TreeNode root1 = new TreeNode(2);
        root1.left = new TreeNode(1);
        root1.right = new TreeNode(3);
        System.out.println(solution.isValidBST(root1)); // Output: true

        // Example 2: Invalid BST
        TreeNode root2 = new TreeNode(5);
        root2.left = new TreeNode(1);
        root2.right = new TreeNode(4);
        root2.right.left = new TreeNode(3);
        root2.right.right = new TreeNode(6);
        System.out.println(solution.isValidBST(root2)); // Output: false
    }
}
```

2. Inorder Traversal Approach

Explanation:
- Perform an inorder traversal of the tree.
- In a valid BST, the inorder traversal should produce a strictly increasing sequence of values.
- Keep track of the previously visited node's value and ensure the current node's value is greater than the previous value.

Time Complexity:
- Average Case: `O(n)`.
- Worst Case: `O(n)`.

Code:

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class ValidateBST {

    private TreeNode prev;
    public boolean isValidBST(TreeNode root) {
        prev = null;
        return inorderTraversal(root);
    }

    private boolean inorderTraversal(TreeNode node) {
        if (node == null) {
            return true; // An empty tree is a valid BST
        }
        // Traverse the left subtree
        if (!inorderTraversal(node.left)) {
            return false;
        }
        // Check if the current node's value is greater than the previous
node's value
        if (prev != null && node.val <= prev.val) {
            return false;
        }
        prev = node; // Update the previous node
        // Traverse the right subtree
        return inorderTraversal(node.right);
    }

    public static void main(String[] args) {
        ValidateBST solution = new ValidateBST();

        // Example 1: Valid BST
        TreeNode root1 = new TreeNode(2);
        root1.left = new TreeNode(1);
        root1.right = new TreeNode(3);
        System.out.println(solution.isValidBST(root1)); // Output: true

        // Example 2: Invalid BST
        TreeNode root2 = new TreeNode(5);
        root2.left = new TreeNode(1);
        root2.right = new TreeNode(4);
        root2.right.left = new TreeNode(3);
        root2.right.right = new TreeNode(6);
        System.out.println(solution.isValidBST(root2)); // Output: false
    }
}
```

3. Iterative Inorder Traversal Approach

Explanation:
- Use an iterative approach (using a stack) to perform an inorder traversal.
- Keep track of the previously visited node's value and ensure the current node's value is greater than the previous value.

Time Complexity:
- Average Case: `O(n)`.
- Worst Case: `O(n)`.

Code:

```java
import java.util.Stack;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class ValidateBST {

    public boolean isValidBST(TreeNode root) {
        Stack<TreeNode> stack = new Stack<>();
        TreeNode prev = null;

        while (root != null || !stack.isEmpty()) {
            // Traverse to the leftmost node
            while (root != null) {
                stack.push(root);
                root = root.left;
            }
            // Process the current node
            root = stack.pop();
            // Check if the current node's value is greater than the
previous node's value
            if (prev != null && root.val <= prev.val) {
                return false;
            }
            prev = root; // Update the previous node
            // Move to the right subtree
            root = root.right;
        }

        return true;
    }

    public static void main(String[] args) {
        ValidateBST solution = new ValidateBST();

        // Example 1: Valid BST
        TreeNode root1 = new TreeNode(2);
        root1.left = new TreeNode(1);
        root1.right = new TreeNode(3);
        System.out.println(solution.isValidBST(root1)); // Output: true

        // Example 2: Invalid BST
        TreeNode root2 = new TreeNode(5);
        root2.left = new TreeNode(1);
        root2.right = new TreeNode(4);
        root2.right.left = new TreeNode(3);
        root2.right.right = new TreeNode(6);
        System.out.println(solution.isValidBST(root2)); // Output: false
    }
}
```

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Notes
Recursive (Valid Range)	`O(n)`	`O(n)`	Simple and intuitive.
Inorder Traversal	`O(n)`	`O(n)`	Leverages the property of BST that inorder traversal is sorted.
Iterative Inorder	`O(n)`	`O(n)`	Avoids recursion and uses a stack for traversal.

Choose the approach based on the constraints and requirements of your problem. The Recursive Approach is the most intuitive, while the Iterative Inorder Approach is useful for avoiding recursion stack overflow in deep trees.

Kth Smallest Element in a BST

Finding the kth smallest element in a Binary Search Tree (BST) is a common problem that leverages the properties of BSTs. Below are different approaches to solve this problem in Java, along with explanations.

1. Inorder Traversal (Recursive)

Explanation:
- Perform an inorder traversal of the BST, which produces the nodes in ascending order.
- Keep a counter to track the number of nodes visited.
- When the counter equals `k`, return the current node's value.

Time Complexity:
- Average Case: `O(n)`, where `n` is the number of nodes in the tree.
- Worst Case: `O(n)` (if the tree is skewed).

Code:

```java
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class KthSmallestElement {

    private int count = 0;
    private int result = 0;

    public int kthSmallest(TreeNode root, int k) {
        inorderTraversal(root, k);
        return result;
    }

    private void inorderTraversal(TreeNode node, int k) {
        if (node == null) {
            return;
        }
        // Traverse the left subtree
        inorderTraversal(node.left, k);
        // Process the current node
        count++;
        if (count == k) {
            result = node.val;
            return;
        }
        // Traverse the right subtree
        inorderTraversal(node.right, k);
    }

    public static void main(String[] args) {
        KthSmallestElement solution = new KthSmallestElement();

        // Example BST
        TreeNode root = new TreeNode(5);
        root.left = new TreeNode(3);
        root.right = new TreeNode(6);
        root.left.left = new TreeNode(2);
        root.left.right = new TreeNode(4);
        root.left.left.left = new TreeNode(1);

        int k = 3;
        System.out.println("The " + k + "th smallest element is: " +
solution.kthSmallest(root, k)); // Output: 3
    }
}
```

2. Inorder Traversal (Iterative)

Explanation:
- Use an iterative approach with a stack to perform the inorder traversal.
- Keep a counter to track the number of nodes visited.
- When the counter equals `k`, return the current node's value.

Time Complexity:
- Average Case: `O(n)`.
- Worst Case: `O(n)`.

Code:

```java
import java.util.Stack;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class KthSmallestElement {

    public int kthSmallest(TreeNode root, int k) {
        Stack<TreeNode> stack = new Stack<>();
        TreeNode current = root;
        int count = 0;

        while (current != null || !stack.isEmpty()) {
            // Traverse to the leftmost node
            while (current != null) {
                stack.push(current);
                current = current.left;
            }
            // Process the current node
            current = stack.pop();
            count++;
            if (count == k) {
                return current.val;
            }
            // Move to the right subtree
            current = current.right;
        }

        return -1; // If k is invalid
    }

    public static void main(String[] args) {
        KthSmallestElement solution = new KthSmallestElement();

        // Example BST
        TreeNode root = new TreeNode(5);
        root.left = new TreeNode(3);
        root.right = new TreeNode(6);
        root.left.left = new TreeNode(2);
        root.left.right = new TreeNode(4);
        root.left.left.left = new TreeNode(1);

        int k = 3;
        System.out.println("The " + k + "th smallest element is: " +
solution.kthSmallest(root, k)); // Output: 3
    }
}
```

3. Augmented Tree (With Size Field)

Explanation:
- Augment the BST by storing the size of the left subtree in each node.
- Use this information to determine the kth smallest element efficiently.

Steps:
1. Build the augmented BST.
2. Use the size field to determine the kth smallest element.

Time Complexity:
- Build Augmented Tree: `O(n)`.
- Query: `O(h)`, where `h` is the height of the tree.

Code:

```
class TreeNode {
    int val;
    int size; // Size of the left subtree
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; size = 1; }
}

public class KthSmallestElement {

    public int kthSmallest(TreeNode root, int k) {
        while (root != null) {
            int leftSize = (root.left != null) ? root.left.size : 0;
            if (k <= leftSize) {
                root = root.left;
            } else if (k == leftSize + 1) {
                return root.val;
            } else {
                k -= (leftSize + 1);
                root = root.right;
            }
        }
        return -1; // If k is invalid
    }

    public static void main(String[] args) {
        KthSmallestElement solution = new KthSmallestElement();

        // Example Augmented BST
        TreeNode root = new TreeNode(5);
        root.left = new TreeNode(3);
        root.right = new TreeNode(6);
        root.left.left = new TreeNode(2);
        root.left.right = new TreeNode(4);
        root.left.left.left = new TreeNode(1);

        // Update sizes (for demonstration purposes)
        root.left.left.size = 2;
        root.left.size = 4;
        root.size = 5;

        int k = 3;
        System.out.println("The " + k + "th smallest element is: " +
solution.kthSmallest(root, k)); // Output: 3
    }
}
```

4. Morris Traversal (Inorder with O(1) Space)

Explanation:
- Use Morris Traversal to perform an inorder traversal with constant space.
- Keep a counter to track the number of nodes visited.
- When the counter equals `k`, return the current node's value.

Time Complexity:
- Average Case: `O(n)`.
- Worst Case: `O(n)`.

Code:

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class KthSmallestElement {

    public int kthSmallest(TreeNode root, int k) {
        TreeNode current = root;
        int count = 0;

        while (current != null) {
            if (current.left == null) {
                // Process the current node
                count++;
                if (count == k) {
                    return current.val;
                }
                current = current.right;
            } else {
                // Find the inorder predecessor
                TreeNode predecessor = current.left;
                while (predecessor.right != null && predecessor.right !=
current) {
                    predecessor = predecessor.right;
                }
                if (predecessor.right == null) {
                    predecessor.right = current;
                    current = current.left;
                } else {
                    predecessor.right = null;
                    // Process the current node
                    count++;
                    if (count == k) {
                        return current.val;
                    }
                    current = current.right;
                }
            }
        }

        return -1; // If k is invalid
    }

    public static void main(String[] args) {
        KthSmallestElement solution = new KthSmallestElement();

        // Example BST
        TreeNode root = new TreeNode(5);
        root.left = new TreeNode(3);
        root.right = new TreeNode(6);
        root.left.left = new TreeNode(2);
        root.left.right = new TreeNode(4);
        root.left.left.left = new TreeNode(1);

        int k = 3;
        System.out.println("The " + k + "th smallest element is: " +
solution.kthSmallest(root, k)); // Output: 3
    }
}
```

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Notes
Recursive Inorder	`O(n)`	`O(n)`	Simple and intuitive.
Iterative Inorder	`O(n)`	`O(n)`	Avoids recursion and uses a stack.
Augmented Tree	`O(n)` (build)	`O(n)`	Efficient for multiple queries.
Morris Traversal	`O(n)`	`O(1)`	Uses constant space but modifies the tree temporarily.

Choose the approach based on the constraints and requirements of your problem. The Iterative Inorder Approach is a good balance between simplicity and efficiency, while Morris Traversal is useful for space-constrained environments.

Lowest Common Ancestor of BST

The Lowest Common Ancestor (LCA) of two nodes in a Binary Search Tree (BST) is the deepest node that has both nodes as descendants. Below are different approaches to solve this problem in Java, along with explanations.

1. Recursive Approach

Explanation:
- Use the properties of a BST:
  - If both nodes are smaller than the current node, the LCA lies in the left subtree.
  - If both nodes are larger than the current node, the LCA lies in the right subtree.
  - Otherwise, the current node is the LCA.

Time Complexity:
- Average Case: `O(h)`, where `h` is the height of the tree.
- Worst Case: `O(n)` (if the tree is skewed).

Code:

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class LowestCommonAncestorBST {

    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
TreeNode q) {
        if (root == null) {
            return null;
        }
        // If both p and q are smaller than root, LCA lies in the left
subtree
```

```
        if (p.val < root.val && q.val < root.val) {
            return lowestCommonAncestor(root.left, p, q);
        }
        // If both p and q are larger than root, LCA lies in the right
subtree
        if (p.val > root.val && q.val > root.val) {
            return lowestCommonAncestor(root.right, p, q);
        }
        // Otherwise, root is the LCA
        return root;
    }

    public static void main(String[] args) {
        LowestCommonAncestorBST solution = new LowestCommonAncestorBST();

        // Example BST
        TreeNode root = new TreeNode(6);
        root.left = new TreeNode(2);
        root.right = new TreeNode(8);
        root.left.left = new TreeNode(0);
        root.left.right = new TreeNode(4);
        root.right.left = new TreeNode(7);
        root.right.right = new TreeNode(9);
        root.left.right.left = new TreeNode(3);
        root.left.right.right = new TreeNode(5);

        TreeNode p = root.left; // Node with value 2
        TreeNode q = root.left.right.right; // Node with value 5

        TreeNode lca = solution.lowestCommonAncestor(root, p, q);
        System.out.println("LCA of " + p.val + " and " + q.val + " is: " +
lca.val); // Output: 2
    }
}
```

2. Iterative Approach

Explanation:
- Use a loop to traverse the tree iteratively.
- Apply the same logic as the recursive approach:
  - If both nodes are smaller than the current node, move to the left subtree.
  - If both nodes are larger than the current node, move to the right subtree.
  - Otherwise, the current node is the LCA.

Time Complexity:
- Average Case: `O(h)`.
- Worst Case: `O(n)`.

Code:

```
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class LowestCommonAncestorBST {

    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
TreeNode q) {
        while (root != null) {
            // If both p and q are smaller than root, move to the left
subtree
            if (p.val < root.val && q.val < root.val) {
                root = root.left;
            }
            // If both p and q are larger than root, move to the right
subtree
            else if (p.val > root.val && q.val > root.val) {
                root = root.right;
            }
            // Otherwise, root is the LCA
            else {
                break;
            }
        }
        return root;
    }

    public static void main(String[] args) {
        LowestCommonAncestorBST solution = new LowestCommonAncestorBST();

        // Example BST
        TreeNode root = new TreeNode(6);
        root.left = new TreeNode(2);
        root.right = new TreeNode(8);
        root.left.left = new TreeNode(0);
        root.left.right = new TreeNode(4);
        root.right.left = new TreeNode(7);
        root.right.right = new TreeNode(9);
        root.left.right.left = new TreeNode(3);
        root.left.right.right = new TreeNode(5);

        TreeNode p = root.left; // Node with value 2
        TreeNode q = root.left.right.right; // Node with value 5

        TreeNode lca = solution.lowestCommonAncestor(root, p, q);
        System.out.println("LCA of " + p.val + " and " + q.val + " is: " +
lca.val); // Output: 2
    }
}
```

3. Path Comparison Approach

Explanation:
- Find the paths from the root to both nodes `p` and `q`.
- Compare the paths to find the last common node, which is the LCA.

Time Complexity:
- Average Case: `O(h)`.
- Worst Case: `O(n)`.

Code:

```
import java.util.List;

class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class LowestCommonAncestorBST {

    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p,
TreeNode q) {
        List<TreeNode> pathP = findPath(root, p);
        List<TreeNode> pathQ = findPath(root, q);

        // Find the last common node in the paths
        TreeNode lca = null;
        for (int i = 0; i < Math.min(pathP.size(), pathQ.size()); i++) {
            if (pathP.get(i) == pathQ.get(i)) {
                lca = pathP.get(i);
            } else {
                break;
            }
        }
        return lca;
    }

    private List<TreeNode> findPath(TreeNode root, TreeNode target) {
        List<TreeNode> path = new ArrayList<>();
        while (root != null) {
            path.add(root);
            if (target.val < root.val) {
                root = root.left;
            } else if (target.val > root.val) {
                root = root.right;
            } else {
                break;
            }
        }
        return path;
    }

    public static void main(String[] args) {
        LowestCommonAncestorBST solution = new LowestCommonAncestorBST();

        // Example BST
        TreeNode root = new TreeNode(6);
        root.left = new TreeNode(2);
        root.right = new TreeNode(8);
        root.left.left = new TreeNode(0);
        root.left.right = new TreeNode(4);
        root.right.left = new TreeNode(7);
        root.right.right = new TreeNode(9);
        root.left.right.left = new TreeNode(3);
        root.left.right.right = new TreeNode(5);

        TreeNode p = root.left; // Node with value 2
        TreeNode q = root.left.right.right; // Node with value 5

        TreeNode lca = solution.lowestCommonAncestor(root, p, q);
        System.out.println("LCA of " + p.val + " and " + q.val + " is: " + lca.val); // Output: 2
    }
}
```

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Notes
Recursive	`O(h)`	`O(h)`	Simple and intuitive.
Iterative	`O(h)`	`O(1)`	Avoids recursion and uses constant space.
Path Comparison	`O(h)`	`O(h)`	Finds paths explicitly, useful for understanding the tree structure.

Choose the approach based on the constraints and requirements of your problem. The Iterative Approach is the most efficient in terms of space, while the Recursive Approach is the simplest to implement.

Implement Trie (Prefix Tree)

A Trie (Prefix Tree) is a tree-like data structure used to store a dynamic set of strings, where the keys are usually strings. It is particularly useful for tasks like autocomplete, spell checking, and IP routing. Below are different approaches to implement a Trie in Java, along with explanations.

1. Basic Trie Implementation

Explanation:
- Each node in the Trie contains:
  - A boolean flag to indicate if the node represents the end of a word.
  - An array or map of child nodes (one for each character).
- Insertion: Traverse the Trie, adding nodes as necessary.
- Search: Traverse the Trie to check if a word exists.
- Prefix Search: Traverse the Trie to check if a prefix exists.

Time Complexity:
- Insertion: `O(m)`, where `m` is the length of the word.
- Search: `O(m)`.
- Prefix Search: `O(m)`.

Code:

```java
class TrieNode {
    TrieNode[] children;
    boolean isEndOfWord;

    public TrieNode() {
        children = new TrieNode[26]; // Assuming lowercase English letters
        isEndOfWord = false;
    }
}

public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Insert a word into the Trie
    public void insert(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                current.children[index] = new TrieNode();
            }
            current = current.children[index];
        }
        current.isEndOfWord = true;
    }

    // Search for a word in the Trie
    public boolean search(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                return false;
            }
            current = current.children[index];
        }
        return current.isEndOfWord;
    }

    // Check if a prefix exists in the Trie
    public boolean startsWith(String prefix) {
        TrieNode current = root;
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                return false;
            }
            current = current.children[index];
        }
        return true;
    }

    public static void main(String[] args) {
        Trie trie = new Trie();
        trie.insert("apple");
        System.out.println(trie.search("apple"));   // Output: true
        System.out.println(trie.search("app"));     // Output: false
        System.out.println(trie.startsWith("app")); // Output: true
        trie.insert("app");
        System.out.println(trie.search("app"));     // Output: true
    }
}
```

2. Trie with HashMap for Children

Explanation:
- Use a `HashMap` to store child nodes instead of an array.
- This approach is more flexible and can handle any character set (not just lowercase English letters).

Time Complexity:
- Insertion: `O(m)`.
- Search: `O(m)`.
- Prefix Search: `O(m)`.

Code:

```java
import java.util.HashMap;
import java.util.Map;

class TrieNode {
    Map<Character, TrieNode> children;
    boolean isEndOfWord;

    public TrieNode() {
        children = new HashMap<>();
        isEndOfWord = false;
    }
}

public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Insert a word into the Trie
    public void insert(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            current.children.putIfAbsent(c, new TrieNode());
            current = current.children.get(c);
        }
        current.isEndOfWord = true;
    }

    // Search for a word in the Trie
    public boolean search(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            if (!current.children.containsKey(c)) {
                return false;
            }
            current = current.children.get(c);
        }
        return current.isEndOfWord;
    }

    // Check if a prefix exists in the Trie
    public boolean startsWith(String prefix) {
        TrieNode current = root;
        for (char c : prefix.toCharArray()) {
            if (!current.children.containsKey(c)) {
                return false;
            }
            current = current.children.get(c);
        }
        return true;
    }

    public static void main(String[] args) {
        Trie trie = new Trie();
        trie.insert("apple");
        System.out.println(trie.search("apple"));   // Output: true
        System.out.println(trie.search("app"));     // Output: false
        System.out.println(trie.startsWith("app")); // Output: true
        trie.insert("app");
        System.out.println(trie.search("app"));     // Output: true
    }
}
```

3. Trie with Count for Prefix Frequency

Explanation:
- Extend the Trie to store the count of words with a given prefix.
- Useful for applications like autocomplete, where you need to know how many words start with a given prefix.

Time Complexity:
- Insertion: `O(m)`.
- Search: `O(m)`.
- Prefix Search: `O(m)`.

Code:

```java
class TrieNode {
    TrieNode[] children;
    boolean isEndOfWord;
    int prefixCount; // Number of words with this prefix

    public TrieNode() {
        children = new TrieNode[26]; // Assuming lowercase English letters
        isEndOfWord = false;
        prefixCount = 0;
    }
}

public class Trie {
    private TrieNode root;

    public Trie() {
        root = new TrieNode();
    }

    // Insert a word into the Trie
    public void insert(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                current.children[index] = new TrieNode();
            }
            current = current.children[index];
            current.prefixCount++; // Increment prefix count
        }
        current.isEndOfWord = true;
    }

    // Search for a word in the Trie
    public boolean search(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                return false;
            }
            current = current.children[index];
        }
        return current.isEndOfWord;
    }

    // Check if a prefix exists in the Trie
    public boolean startsWith(String prefix) {
        TrieNode current = root;
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                return false;
            }
            current = current.children[index];
        }
        return true;
    }

    // Get the number of words with a given prefix
    public int countPrefix(String prefix) {
        TrieNode current = root;
        for (char c : prefix.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                return 0;
            }
            current = current.children[index];
        }
        return current.prefixCount;
    }

    public static void main(String[] args) {
        Trie trie = new Trie();
        trie.insert("apple");
        trie.insert("app");
        trie.insert("apricot");
        System.out.println(trie.countPrefix("ap")); // Output: 3
        System.out.println(trie.countPrefix("app")); // Output: 2
    }
}
```

Comparison of Approaches:

| Approach | Time Complexity | Space Complexity | Notes
|
||--||--|

Basic Trie	`O(m)`	`O(m)`	Simple and efficient for lowercase English letters.
Trie with HashMap	`O(m)`	`O(m)`	Flexible and supports any character set.
Trie with Prefix Count	`O(m)`	`O(m)`	Useful for applications requiring prefix frequency.

Choose the approach based on the constraints and requirements of your problem. The Basic Trie is the most efficient for fixed character sets, while the Trie with HashMap is more flexible for general use cases. The Trie with Prefix Count is ideal for applications like autocomplete.

Add and Search Word

The Add and Search Word problem involves designing a data structure that supports adding new words and searching for words, including support for wildcard characters (e.g., `.`). Below are different approaches to solve this problem in Java, along with explanations.

1. Trie with Recursive Search

Explanation:
- Use a Trie data structure to store words.
- For the `addWord` operation, insert the word into the Trie.
- For the `search` operation, use a recursive approach to handle wildcard characters (`.`).
  - If the current character is a wildcard, recursively check all possible child nodes.

Time Complexity:
- Add Word: `O(m)`, where `m` is the length of the word.
- Search Word: `O(m)` for exact matches, `O(26^m)` for wildcard matches in the worst case.

Code:

```java
class TrieNode {
    TrieNode[] children;
    boolean isEndOfWord;

    public TrieNode() {
        children = new TrieNode[26]; // Assuming lowercase English letters
        isEndOfWord = false;
    }
}

public class WordDictionary {
    private TrieNode root;

    public WordDictionary() {
        root = new TrieNode();
    }

    // Add a word into the Trie
    public void addWord(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                current.children[index] = new TrieNode();
            }
            current = current.children[index];
        }
        current.isEndOfWord = true;
    }

    // Search for a word in the Trie
    public boolean search(String word) {
        return searchHelper(word, 0, root);
    }

    private boolean searchHelper(String word, int index, TrieNode node) {
        if (index == word.length()) {
            return node.isEndOfWord;
        }
        char c = word.charAt(index);
        if (c == '.') {
            // Wildcard: Check all possible child nodes
            for (TrieNode child : node.children) {
                if (child != null && searchHelper(word, index + 1, child))
{
                    return true;
                }
            }
            return false;
        } else {
            // Exact match: Check the specific child node
            int childIndex = c - 'a';
            if (node.children[childIndex] == null) {
                return false;
            }
            return searchHelper(word, index + 1,
node.children[childIndex]);
        }
    }

    public static void main(String[] args) {
        WordDictionary dict = new WordDictionary();
        dict.addWord("bad");
        dict.addWord("dad");
        dict.addWord("mad");

        System.out.println(dict.search("pad")); // Output: false
        System.out.println(dict.search("bad")); // Output: true
        System.out.println(dict.search(".ad")); // Output: true
        System.out.println(dict.search("b..")); // Output: true
    }
}
```

2. Trie with Iterative Search

Explanation:
- Use a Trie data structure to store words.
- For the `addWord` operation, insert the word into the Trie.
- For the `search` operation, use an iterative approach with a queue to handle wildcard characters (`.`).
  - Use a queue to keep track of nodes to process at each level.

Time Complexity:
- Add Word: `O(m)`.
- Search Word: `O(m)` for exact matches, `O(26^m)` for wildcard matches in the worst case.

Code:

```
import java.util.Queue;

class TrieNode {
    TrieNode[] children;
    boolean isEndOfWord;

    public TrieNode() {
        children = new TrieNode[26]; // Assuming lowercase English letters
```

```java
        isEndOfWord = false;
    }
}

public class WordDictionary {
    private TrieNode root;

    public WordDictionary() {
        root = new TrieNode();
    }

    // Add a word into the Trie
    public void addWord(String word) {
        TrieNode current = root;
        for (char c : word.toCharArray()) {
            int index = c - 'a';
            if (current.children[index] == null) {
                current.children[index] = new TrieNode();
            }
            current = current.children[index];
        }
        current.isEndOfWord = true;
    }

    // Search for a word in the Trie
    public boolean search(String word) {
        Queue<TrieNode> queue = new LinkedList<>();
        queue.offer(root);

        for (int i = 0; i < word.length(); i++) {
            char c = word.charAt(i);
            int size = queue.size();
            for (int j = 0; j < size; j++) {
                TrieNode node = queue.poll();
                if (c == '.') {
                    // Wildcard: Add all child nodes to the queue
                    for (TrieNode child : node.children) {
                        if (child != null) {
                            queue.offer(child);
                        }
                    }
                } else {
                    // Exact match: Add the specific child node to the
queue
                    int index = c - 'a';
                    if (node.children[index] != null) {
                        queue.offer(node.children[index]);
                    }
                }
            }
            if (queue.isEmpty()) {
                return false;
            }
        }

        // Check if any node in the queue is the end of a word
        while (!queue.isEmpty()) {
            if (queue.poll().isEndOfWord) {
                return true;
            }
        }
        return false;
    }

    public static void main(String[] args) {
        WordDictionary dict = new WordDictionary();
        dict.addWord("bad");
        dict.addWord("dad");
        dict.addWord("mad");

        System.out.println(dict.search("pad")); // Output: false
        System.out.println(dict.search("bad")); // Output: true
        System.out.println(dict.search(".ad")); // Output: true
        System.out.println(dict.search("b..")); // Output: true
    }
}
```

3. HashMap-Based Approach

Explanation:
- Use a `HashMap` to store words grouped by their lengths.
- For the `addWord` operation, add the word to the appropriate group in the HashMap.
- For the `search` operation, check only the words with the same length as the search word.
  - Use a helper function to compare the search word with each candidate word, handling wildcard characters (`.`).

Time Complexity:
- Add Word: `O(1)`.
- Search Word: `O(n * m)`, where `n` is the number of words with the same length and `m` is the length of the word.

Code:

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Map;
import java.util.Set;

public class WordDictionary {
    private Map<Integer, Set<String>> map;

    public WordDictionary() {
        map = new HashMap<>();
    }

    // Add a word into the HashMap
    public void addWord(String word) {
        int length = word.length();
        map.putIfAbsent(length, new HashSet<>());
        map.get(length).add(word);
    }

    // Search for a word in the HashMap
    public boolean search(String word) {
        int length = word.length();
        if (!map.containsKey(length)) {
            return false;
        }
        for (String candidate : map.get(length)) {
            if (isMatch(candidate, word)) {
                return true;
            }
        }
        return false;
    }

    // Helper function to compare two words with wildcard support
    private boolean isMatch(String candidate, String word) {
        for (int i = 0; i < word.length(); i++) {
```

```java
            char c = word.charAt(i);
            if (c != '.' && c != candidate.charAt(i)) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        WordDictionary dict = new WordDictionary();
        dict.addWord("bad");
        dict.addWord("dad");
        dict.addWord("mad");

        System.out.println(dict.search("pad")); // Output: false
        System.out.println(dict.search("bad")); // Output: true
        System.out.println(dict.search(".ad")); // Output: true
        System.out.println(dict.search("b..")); // Output: true
    }
}
```

Comparison of Approaches:

Approach	Time Complexity (Add)	Time Complexity (Search)	Space Complexity	Notes
Trie with Recursive Search	`O(m)`	`O(m)` (exact), `O(26^m)` (wildcard)	`O(m)`	Efficient for exact matches, but slower for wildcard matches.
Trie with Iterative Search	`O(m)`	`O(m)` (exact), `O(26^m)` (wildcard)	`O(m)`	Avoids recursion, but still slow for wildcard matches.
HashMap-Based Approach	`O(1)`	`O(n * m)`	`O(n * m)`	Simple and fast for adding words, but slower for searching.

Choose the approach based on the constraints and requirements of your problem. The Trie with Recursive Search is the most efficient for exact matches, while the HashMap-Based Approach is simpler and faster for adding words.

<u>Word Search II</u>

The Word Search II problem involves finding all words from a given list that exist in a 2D board of characters. Words can be constructed from adjacent cells (horizontally or vertically neighboring). Below are different approaches to solve this problem in Java, along with explanations.

1. Backtracking with Trie

Explanation:
- Use a Trie to store all words from the list for efficient prefix matching.
- Perform a backtracking search on the board:
  - Start from each cell and explore all possible paths.
  - Use the Trie to check if the current path forms a valid prefix or word.
  - If a word is found, add it to the result.

Time Complexity:
- Building Trie: `O(m * k)`, where `m` is the number of words and `k` is the average length of a word.
- Searching Board: `O(n * n * 4^l)`, where `n` is the size of the board and `l` is the maximum length of a word.

Code:

```java
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

class TrieNode {
    TrieNode[] children;
    String word;

    public TrieNode() {
        children = new TrieNode[26]; // Assuming lowercase English letters
        word = null;
    }
}

public class WordSearchII {
    private static final int[][] DIRECTIONS = {{-1, 0}, {1, 0}, {0, -1},
{0, 1}};

    public List<String> findWords(char[][] board, String[] words) {
        // Build the Trie
        TrieNode root = buildTrie(words);
        // Perform backtracking search
        Set<String> result = new HashSet<>();
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                backtrack(board, i, j, root, result);
            }
        }
        return new ArrayList<>(result);
    }

    private TrieNode buildTrie(String[] words) {
        TrieNode root = new TrieNode();
        for (String word : words) {
            TrieNode current = root;
            for (char c : word.toCharArray()) {
                int index = c - 'a';
                if (current.children[index] == null) {
                    current.children[index] = new TrieNode();
                }
                current = current.children[index];
            }
            current.word = word; // Store the word at the end node
        }
        return root;
    }

    private void backtrack(char[][] board, int i, int j, TrieNode node,
Set<String> result) {
        char c = board[i][j];
        if (c == '#' || node.children[c - 'a'] == null) {
            return; // Invalid path or prefix
        }

        node = node.children[c - 'a'];
        if (node.word != null) {
            result.add(node.word); // Found a word
            node.word = null; // Avoid duplicate results
        }

        board[i][j] = '#'; // Mark the cell as visited
        for (int[] dir : DIRECTIONS) {
            int x = i + dir[0];
            int y = j + dir[1];
            if (x >= 0 && x < board.length && y >= 0 && y <
board[0].length) {
                backtrack(board, x, y, node, result);
            }
        }
        board[i][j] = c; // Restore the cell
    }

    public static void main(String[] args) {
        WordSearchII solution = new WordSearchII();
        char[][] board = {
            {'o', 'a', 'a', 'n'},
            {'e', 't', 'a', 'e'},
            {'i', 'h', 'k', 'r'},
            {'i', 'f', 'l', 'v'}
        };
        String[] words = {"oath", "pea", "eat", "rain"};
        List<String> result = solution.findWords(board, words);
        System.out.println(result); // Output: ["oath", "eat"]
    }
}
```

2. Backtracking with HashSet

Explanation:
- Use a HashSet to store all words from the list.
- Perform a backtracking search on the board:
  - Start from each cell and explore all possible paths.
  - Use the HashSet to check if the current path forms a valid word.
  - If a word is found, add it to the result.

Time Complexity:
- Building HashSet: `O(m * k)`.
- Searching Board: `O(n * n * 4^l)`.

Code:

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
```

```java
public class WordSearchII {
    private static final int[][] DIRECTIONS = {{-1, 0}, {1, 0}, {0, -1},
{0, 1}};

    public List<String> findWords(char[][] board, String[] words) {
        // Build the HashSet
        Set<String> wordSet = new HashSet<>();
        for (String word : words) {
            wordSet.add(word);
        }

        // Perform backtracking search
        Set<String> result = new HashSet<>();
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                backtrack(board, i, j, new StringBuilder(), wordSet,
result);
            }
        }
        return new ArrayList<>(result);
    }

    private void backtrack(char[][] board, int i, int j, StringBuilder
path, Set<String> wordSet, Set<String> result) {
        if (i < 0 || i >= board.length || j < 0 || j >= board[0].length ||
board[i][j] == '#') {
            return; // Invalid cell or already visited
        }

        char c = board[i][j];
        path.append(c);
        String currentWord = path.toString();

        if (wordSet.contains(currentWord)) {
            result.add(currentWord); // Found a word
        }

        board[i][j] = '#'; // Mark the cell as visited
        for (int[] dir : DIRECTIONS) {
            int x = i + dir[0];
            int y = j + dir[1];
            backtrack(board, x, y, path, wordSet, result);
        }
        board[i][j] = c; // Restore the cell
        path.deleteCharAt(path.length() - 1); // Backtrack
    }

    public static void main(String[] args) {
        WordSearchII solution = new WordSearchII();
        char[][] board = {
            {'o', 'a', 'a', 'n'},
            {'e', 't', 'a', 'e'},
            {'i', 'h', 'k', 'r'},
            {'i', 'f', 'l', 'v'}
        };
        String[] words = {"oath", "pea", "eat", "rain"};
        List<String> result = solution.findWords(board, words);
        System.out.println(result); // Output: ["oath", "eat"]
    }
}
```

3. Optimized Backtracking with Trie and Pruning

Explanation:
- Use a Trie to store all words from the list.
- Perform a backtracking search on the board:
  - Start from each cell and explore all possible paths.
  - Use the Trie to check if the current path forms a valid prefix or word.
  - If a word is found, add it to the result and remove it from the Trie to avoid duplicates.
  - Prune the Trie by removing nodes that are no longer needed.

Time Complexity:
- Building Trie: `O(m * k)`.
- Searching Board: `O(n * n * 4^l)`.

Code:

```
import java.util.ArrayList;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

class TrieNode {
    TrieNode[] children;
    String word;
    public TrieNode() {
        children = new TrieNode[26]; // Assuming lowercase English letters
        word = null;
    }
}

public class WordSearchII {
    private static final int[][] DIRECTIONS = {{-1, 0}, {1, 0}, {0, -1},
{0, 1}};

    public List<String> findWords(char[][] board, String[] words) {
        // Build the Trie
        TrieNode root = buildTrie(words);

        // Perform backtracking search
        Set<String> result = new HashSet<>();
        for (int i = 0; i < board.length; i++) {
            for (int j = 0; j < board[0].length; j++) {
                backtrack(board, i, j, root, result);
            }
        }
        return new ArrayList<>(result);
    }

    private TrieNode buildTrie(String[] words) {
        TrieNode root = new TrieNode();
        for (String word : words) {
            TrieNode current = root;
            for (char c : word.toCharArray()) {
                int index = c - 'a';
                if (current.children[index] == null) {
                    current.children[index] = new TrieNode();
                }
                current = current.children[index];
            }
            current.word = word; // Store the word at the end node
        }
        return root;
    }

    private void backtrack(char[][] board, int i, int j, TrieNode node,
Set<String> result) {
```

```java
        char c = board[i][j];
        if (c == '#' || node.children[c - 'a'] == null) {
            return; // Invalid path or prefix
        }

        node = node.children[c - 'a'];
        if (node.word != null) {
            result.add(node.word); // Found a word
            node.word = null; // Avoid duplicate results
        }

        board[i][j] = '#'; // Mark the cell as visited
        for (int[] dir : DIRECTIONS) {
            int x = i + dir[0];
            int y = j + dir[1];
            if (x >= 0 && x < board.length && y >= 0 && y <
board[0].length) {
                backtrack(board, x, y, node, result);
            }
        }
        board[i][j] = c; // Restore the cell
    }

    public static void main(String[] args) {
        WordSearchII solution = new WordSearchII();
        char[][] board = {
            {'o', 'a', 'a', 'n'},
            {'e', 't', 'a', 'e'},
            {'i', 'h', 'k', 'r'},
            {'i', 'f', 'l', 'v'}
        };
        String[] words = {"oath", "pea", "eat", "rain"};
        List<String> result = solution.findWords(board, words);
        System.out.println(result); // Output: ["oath", "eat"]
    }
}
```

Comparison of Approaches:

| Approach | Time Complexity (Build) | Time Complexity (Search) | Space Complexity | Notes |
||-|--||--|

Backtracking with Trie	`O(m * k)`	`O(n * n * 4^l)`	`O(m * k)`	Efficient for prefix matching and avoids unnecessary searches.
Backtracking with HashSet	`O(m * k)`	`O(n * n * 4^l)`	`O(m * k)`	Simpler but less efficient for prefix matching.
Optimized Backtracking with Trie	`O(m * k)`	`O(n * n * 4^l)`	`O(m * k)`	Prunes the Trie to avoid redundant searches and improve performance.

Choose the approach based on the constraints and requirements of your problem. The Backtracking with Trie approach is the most efficient for large datasets, while the Backtracking with HashSet approach is simpler to implement. The Optimized Backtracking with Trie approach provides additional performance improvements by pruning the Trie.

(Heap)

Merge K Sorted Lists

The Merge K Sorted Lists problem involves merging `k` sorted linked lists into a single sorted linked list. Below are different approaches to solve this problem in Java, along with explanations.

1. Brute Force Approach

Explanation:
- Traverse all the linked lists and collect the values into an array.
- Sort the array.
- Create a new linked list from the sorted array.

Time Complexity:
- Collecting Values: `O(N)`, where `N` is the total number of nodes.
- Sorting: `O(N log N)`.
- Creating New List: `O(N)`.
- Overall: `O(N log N)`.

Space Complexity:
- `O(N)` for storing the values in an array.

Code:

```
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
```

```java
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

public class MergeKSortedLists {
    public ListNode mergeKLists(ListNode[] lists) {
        List<Integer> values = new ArrayList<>();

        // Collect all values from the linked lists
        for (ListNode list : lists) {
            while (list != null) {
                values.add(list.val);
                list = list.next;
            }
        }

        // Sort the values
        Collections.sort(values);

        // Create a new linked list from the sorted values
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;
        for (int val : values) {
            current.next = new ListNode(val);
            current = current.next;
        }

        return dummy.next;
    }

    public static void main(String[] args) {
        MergeKSortedLists solution = new MergeKSortedLists();

        // Example input
        ListNode list1 = new ListNode(1);
        list1.next = new ListNode(4);
        list1.next.next = new ListNode(5);

        ListNode list2 = new ListNode(1);
        list2.next = new ListNode(3);
        list2.next.next = new ListNode(4);

        ListNode list3 = new ListNode(2);
        list3.next = new ListNode(6);

        ListNode[] lists = {list1, list2, list3};

        ListNode result = solution.mergeKLists(lists);

        // Print the merged list
        while (result != null) {
            System.out.print(result.val + " ");
            result = result.next;
        }
        // Output: 1 1 2 3 4 4 5 6
    }
}
```

2. Merge Lists One by One

Explanation:
- Merge the lists one by one using the Merge Two Sorted Lists approach.
- Start with the first list and merge it with the second list, then merge the result with the third list, and so on.

Time Complexity:
- Merging Two Lists: `O(N)`, where `N` is the total number of nodes.
- Total Merges: `k-1` merges.
- Overall: `O(kN)`.

Space Complexity:
- `O(1)` (no extra space is used).

Code:

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

public class MergeKSortedLists {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0) {
            return null;
        }

        ListNode result = lists[0];
        for (int i = 1; i < lists.length; i++) {
            result = mergeTwoLists(result, lists[i]);
        }
        return result;
    }

    private ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;

        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                current.next = l1;
                l1 = l1.next;
            } else {
                current.next = l2;
                l2 = l2.next;
            }
            current = current.next;
        }

        if (l1 != null) {
            current.next = l1;
        } else {
            current.next = l2;
        }

        return dummy.next;
    }

    public static void main(String[] args) {
        MergeKSortedLists solution = new MergeKSortedLists();

        // Example input
        ListNode list1 = new ListNode(1);
        list1.next = new ListNode(4);
        list1.next.next = new ListNode(5);
        ListNode list2 = new ListNode(1);
        list2.next = new ListNode(3);
        list2.next.next = new ListNode(4);

        ListNode list3 = new ListNode(2);
        list3.next = new ListNode(6);

        ListNode[] lists = {list1, list2, list3};

        ListNode result = solution.mergeKLists(lists);

        // Print the merged list
        while (result != null) {
            System.out.print(result.val + " ");
            result = result.next;
        }
        // Output: 1 1 2 3 4 4 5 6
    }
}
```

3. Divide and Conquer (Merge Pairs)

Explanation:
- Use a Divide and Conquer approach to merge pairs of lists iteratively.
- In each step, merge pairs of lists and repeat until only one list remains.

Time Complexity:
- Merging Pairs: `O(N log k)`, where `k` is the number of lists and `N` is the total number of nodes.

Space Complexity:
- `O(1)` (no extra space is used).

Code:

```
class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}
```

```
public class MergeKSortedLists {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0) {
            return null;
        }

        int interval = 1;
        while (interval < lists.length) {
            for (int i = 0; i + interval < lists.length; i += interval * 2)
{
                lists[i] = mergeTwoLists(lists[i], lists[i + interval]);
            }
            interval *= 2;
        }
        return lists[0];
    }

    private ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        ListNode dummy = new ListNode(0);
        ListNode current = dummy;

        while (l1 != null && l2 != null) {
            if (l1.val < l2.val) {
                current.next = l1;
                l1 = l1.next;
            } else {
                current.next = l2;
                l2 = l2.next;
            }
            current = current.next;
        }

        if (l1 != null) {
            current.next = l1;
        } else {
            current.next = l2;
        }

        return dummy.next;
    }

    public static void main(String[] args) {
        MergeKSortedLists solution = new MergeKSortedLists();
        // Example input
        ListNode list1 = new ListNode(1);
        list1.next = new ListNode(4);
        list1.next.next = new ListNode(5);

        ListNode list2 = new ListNode(1);
        list2.next = new ListNode(3);
        list2.next.next = new ListNode(4);

        ListNode list3 = new ListNode(2);
        list3.next = new ListNode(6);

        ListNode[] lists = {list1, list2, list3};

        ListNode result = solution.mergeKLists(lists);

        // Print the merged list
        while (result != null) {
            System.out.print(result.val + " ");
            result = result.next;
        }
        // Output: 1 1 2 3 4 4 5 6
    }
}
```

4. Priority Queue (Min-Heap)

Explanation:
- Use a Min-Heap (Priority Queue) to efficiently merge the lists.
- Insert the first node of each list into the heap.
- Repeatedly extract the smallest node from the heap and add it to the result list.
- If the extracted node has a next node, insert it into the heap.

Time Complexity:
- Building Heap: `O(k log k)`.
- Extracting Nodes: `O(N log k)`.
- Overall: `O(N log k)`.

Space Complexity:
- `O(k)` for the heap.

Code:

```java
import java.util.PriorityQueue;

class ListNode {
    int val;
    ListNode next;
    ListNode(int x) { val = x; }
}

public class MergeKSortedLists {
    public ListNode mergeKLists(ListNode[] lists) {
        if (lists == null || lists.length == 0) {
            return null;
        }

        // Min-Heap to store the smallest nodes
        PriorityQueue<ListNode> heap = new PriorityQueue<>((a, b) -> a.val
- b.val);

        // Add the first node of each list to the heap
        for (ListNode list : lists) {
            if (list != null) {
                heap.offer(list);
            }
        }

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

        // Extract the smallest node from the heap and add it to the result
        while (!heap.isEmpty()) {
            ListNode smallest = heap.poll();
            current.next = smallest;
            current = current.next;

            // Add the next node of the extracted node to the heap
            if (smallest.next != null) {
                heap.offer(smallest.next);
            }
        }

        return dummy.next;
    }
    public static void main(String[] args) {
        MergeKSortedLists solution = new MergeKSortedLists();

        // Example input
        ListNode list1 = new ListNode(1);
        list1.next = new ListNode(4);
        list1.next.next = new ListNode(5);

        ListNode list2 = new ListNode(1);
        list2.next = new ListNode(3);
        list2.next.next = new ListNode(4);

        ListNode list3 = new ListNode(2);
        list3.next = new ListNode(6);

        ListNode[] lists = {list1, list2, list3};

        ListNode result = solution.mergeKLists(lists);

        // Print the merged list
        while (result != null) {
            System.out.print(result.val + " ");
            result = result.next;
        }
        // Output: 1 1 2 3 4 4 5 6
    }
}
```

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Notes
Brute Force	`O(N log N)`	`O(N)`	Simple but inefficient for large datasets.
Merge One by One	`O(kN)`	`O(1)`	Easy to implement but slow for large `k`.
Divide and Conquer	`O(N log k)`	`O(1)`	Efficient and avoids extra space.
Priority Queue	`O(N log k)`	`O(k)`	Efficient and easy to implement.

Choose the approach based on the constraints and requirements of your problem. The Priority Queue approach is the most efficient and widely used for this problem.

Top K Frequent Elements

The Top K Frequent Elements problem involves finding the `k` most frequent elements in an array. Below are different approaches to solve this problem in Java, along with explanations.

1. HashMap with Sorting

Explanation:
- Use a `HashMap` to count the frequency of each element.
- Sort the elements based on their frequency.
- Return the top `k` elements.

Time Complexity:
- Counting Frequencies: `O(n)`.
- Sorting: `O(n log n)`.
- Overall: `O(n log n)`.

Space Complexity:
- `O(n)` for the HashMap and the list of elements.

Code:

```java
import java.util.*;

public class TopKFrequentElements {
    public int[] topKFrequent(int[] nums, int k) {
        // Count the frequency of each element
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        }

        // Create a list of unique elements
        List<Integer> uniqueElements = new
ArrayList<>(frequencyMap.keySet());

        // Sort the elements based on frequency in descending order
        uniqueElements.sort((a, b) -> frequencyMap.get(b) -
frequencyMap.get(a));

        // Get the top k elements
        int[] result = new int[k];
        for (int i = 0; i < k; i++) {
            result[i] = uniqueElements.get(i);
        }

        return result;
    }

    public static void main(String[] args) {
        TopKFrequentElements solution = new TopKFrequentElements();
        int[] nums = {1, 1, 1, 2, 2, 3};
        int k = 2;
        int[] result = solution.topKFrequent(nums, k);
        System.out.println(Arrays.toString(result)); // Output: [1, 2]
    }
}
```

2. Priority Queue (Min-Heap)

Explanation:
- Use a `HashMap` to count the frequency of each element.
- Use a Min-Heap (Priority Queue) to keep track of the top `k` frequent elements.
- Iterate through the frequency map and maintain the heap size as `k`.

Time Complexity:
- Counting Frequencies: `O(n)`.
- Building Heap: `O(n log k)`.
- Overall: `O(n log k)`.

Space Complexity:
- `O(n)` for the HashMap and the heap.

Code:

```java
import java.util.*;

public class TopKFrequentElements {
    public int[] topKFrequent(int[] nums, int k) {
        // Count the frequency of each element
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        }

        // Use a Min-Heap to keep track of the top k frequent elements
        PriorityQueue<Integer> heap = new PriorityQueue<>(
            (a, b) -> frequencyMap.get(a) - frequencyMap.get(b)
        );

        for (int num : frequencyMap.keySet()) {
            heap.offer(num);
            if (heap.size() > k) {
                heap.poll();
            }
        }

        // Extract the top k elements from the heap
        int[] result = new int[k];
        for (int i = k - 1; i >= 0; i--) {
            result[i] = heap.poll();
        }

        return result;
    }

    public static void main(String[] args) {
        TopKFrequentElements solution = new TopKFrequentElements();
        int[] nums = {1, 1, 1, 2, 2, 3};
        int k = 2;
        int[] result = solution.topKFrequent(nums, k);
        System.out.println(Arrays.toString(result)); // Output: [1, 2]
    }
}
```

3. Bucket Sort

Explanation:
- Use a `HashMap` to count the frequency of each element.
- Use an array of lists (buckets) to group elements by their frequency.
- Iterate through the buckets in descending order to get the top `k` elements.

Time Complexity:

- Counting Frequencies: `O(n)`.
- Bucket Sort: `O(n)`.
- Overall: `O(n)`.

Space Complexity:
- `O(n)` for the HashMap and the buckets.

Code:

```java
import java.util.*;

public class TopKFrequentElements {
    public int[] topKFrequent(int[] nums, int k) {
        // Count the frequency of each element
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        }

        // Create buckets to group elements by frequency
        List<Integer>[] buckets = new List[nums.length + 1];
        for (int num : frequencyMap.keySet()) {
            int frequency = frequencyMap.get(num);
            if (buckets[frequency] == null) {
                buckets[frequency] = new ArrayList<>();
            }
            buckets[frequency].add(num);
        }

        // Extract the top k elements from the buckets
        int[] result = new int[k];
        int index = 0;
        for (int i = buckets.length - 1; i >= 0 && index < k; i--) {
            if (buckets[i] != null) {
                for (int num : buckets[i]) {
                    result[index++] = num;
                    if (index == k) {
                        break;
                    }
                }
            }
        }
        return result;
    }

    public static void main(String[] args) {
        TopKFrequentElements solution = new TopKFrequentElements();
        int[] nums = {1, 1, 1, 2, 2, 3};
        int k = 2;
        int[] result = solution.topKFrequent(nums, k);
        System.out.println(Arrays.toString(result)); // Output: [1, 2]
    }
}
```

4. Quickselect Algorithm

Explanation:
- Use a `HashMap` to count the frequency of each element.
- Use the Quickselect algorithm to find the `k`th most frequent element.
- Partition the elements based on their frequency and return the top `k` elements.

Time Complexity:
- Counting Frequencies: `O(n)`.
- Quickselect: `O(n)` on average.
- Overall: `O(n)` on average.

Space Complexity:
- `O(n)` for the HashMap and the list of unique elements.

Code:

```
import java.util.*;

public class TopKFrequentElements {
    public int[] topKFrequent(int[] nums, int k) {
        // Count the frequency of each element
        Map<Integer, Integer> frequencyMap = new HashMap<>();
        for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) + 1);
        }

        // Create a list of unique elements
        List<Integer> uniqueElements = new
ArrayList<>(frequencyMap.keySet());
```

```java
        // Use Quickselect to find the kth most frequent element
        int left = 0, right = uniqueElements.size() - 1;
        while (left <= right) {
            int pivotIndex = partition(uniqueElements, left, right,
frequencyMap);
            if (pivotIndex == k - 1) {
                break;
            } else if (pivotIndex < k - 1) {
                left = pivotIndex + 1;
            } else {
                right = pivotIndex - 1;
            }
        }

        // Extract the top k elements
        int[] result = new int[k];
        for (int i = 0; i < k; i++) {
            result[i] = uniqueElements.get(i);
        }

        return result;
    }

    private int partition(List<Integer> list, int left, int right,
Map<Integer, Integer> frequencyMap) {
        int pivotFrequency = frequencyMap.get(list.get(right));
        int i = left;
        for (int j = left; j < right; j++) {
            if (frequencyMap.get(list.get(j)) >= pivotFrequency) {
                Collections.swap(list, i, j);
                i++;
            }
        }
        Collections.swap(list, i, right);
        return i;
    }

    public static void main(String[] args) {
        TopKFrequentElements solution = new TopKFrequentElements();
        int[] nums = {1, 1, 1, 2, 2, 3};
        int k = 2;
        int[] result = solution.topKFrequent(nums, k);
        System.out.println(Arrays.toString(result)); // Output: [1, 2]
    }
}
```

Comparison of Approaches:

Approach	Time Complexity	Space Complexity	Notes
HashMap with Sorting	`O(n log n)`	`O(n)`	Simple but inefficient for large datasets.
Priority Queue	`O(n log k)`	`O(n)`	Efficient and widely used.
Bucket Sort	`O(n)`	`O(n)`	Optimal for large datasets with limited frequency range.
Quickselect	`O(n)` (avg)	`O(n)`	Efficient but has a worst-case time complexity of `O(n^2)`.

Choose the approach based on the constraints and requirements of your problem. The Bucket Sort approach is the most efficient for large datasets, while the Priority Queue approach is a good balance between simplicity and efficiency.

Find Median from Data Stream

The Find Median from Data Stream problem involves designing a data structure that supports two operations:
1. Adding a number to the data structure.
2. Finding the median of all numbers added so far.

The median is the middle value in an ordered list. If the list has an even number of elements, the median is the average of the two middle numbers.

Below are different approaches to solve this problem in Java, along with explanations.

1. Sorting Approach

Explanation:
- Maintain a list of all numbers.
- For the `addNum` operation, append the number to the list.
- For the `findMedian` operation, sort the list and find the median.

Time Complexity:

- Add Operation: `O(1)` (appending to a list).
- Find Median Operation: `O(n log n)` (sorting the list).

Space Complexity:
- `O(n)` for storing the list.

Code:

```java
import java.util.*;

public class MedianFinder {
    private List<Integer> numbers;

    public MedianFinder() {
        numbers = new ArrayList<>();
    }

    public void addNum(int num) {
        numbers.add(num);
    }

    public double findMedian() {
        Collections.sort(numbers);
        int size = numbers.size();
        if (size % 2 == 0) {
            return (numbers.get(size / 2 - 1) + numbers.get(size / 2)) /
2.0;
        } else {
            return numbers.get(size / 2);
        }
    }

    public static void main(String[] args) {
        MedianFinder medianFinder = new MedianFinder();
        medianFinder.addNum(1);
        medianFinder.addNum(2);
        System.out.println(medianFinder.findMedian()); // Output: 1.5
        medianFinder.addNum(3);
        System.out.println(medianFinder.findMedian()); // Output: 2.0
    }
}
```

2. Two Heaps (Min-Heap and Max-Heap)

Explanation:
- Use two heaps:
  - A Max-Heap to store the smaller half of the numbers.
  - A Min-Heap to store the larger half of the numbers.
- Maintain the size of the Max-Heap to be either equal to or one greater than the size of the Min-Heap.
- The median is:
  - The top of the Max-Heap if the total number of elements is odd.
  - The average of the tops of both heaps if the total number of elements is even.

Time Complexity:
- Add Operation: `O(log n)` (heap insertion).
- Find Median Operation: `O(1)` (accessing the top of the heaps).

Space Complexity:
- `O(n)` for storing the numbers in the heaps.

Code:

```java
import java.util.*;

public class MedianFinder {
    private PriorityQueue<Integer> maxHeap; // Stores the smaller half
    private PriorityQueue<Integer> minHeap; // Stores the larger half

    public MedianFinder() {
        maxHeap = new PriorityQueue<>((a, b) -> b - a); // Max-Heap
        minHeap = new PriorityQueue<>(); // Min-Heap
    }

    public void addNum(int num) {
        // Add to the appropriate heap
        if (maxHeap.isEmpty() || num <= maxHeap.peek()) {
            maxHeap.offer(num);
        } else {
            minHeap.offer(num);
        }

        // Balance the heaps
        if (maxHeap.size() > minHeap.size() + 1) {
            minHeap.offer(maxHeap.poll());
        } else if (minHeap.size() > maxHeap.size()) {
            maxHeap.offer(minHeap.poll());
        }
    }

    public double findMedian() {
        if (maxHeap.size() == minHeap.size()) {
            return (maxHeap.peek() + minHeap.peek()) / 2.0;
        } else {
            return maxHeap.peek();
        }
    }

    public static void main(String[] args) {
        MedianFinder medianFinder = new MedianFinder();
        medianFinder.addNum(1);
        medianFinder.addNum(2);
        System.out.println(medianFinder.findMedian()); // Output: 1.5
        medianFinder.addNum(3);
        System.out.println(medianFinder.findMedian()); // Output: 2.0
    }
}
```

3. Insertion Sort Approach

Explanation:
- Maintain a sorted list of numbers.
- For the `addNum` operation, insert the number into the correct position in the sorted list.
- For the `findMedian` operation, directly find the median from the sorted list.

Time Complexity:
- Add Operation: `O(n)` (inserting into a sorted list).
- Find Median Operation: `O(1)` (accessing the middle element).

Space Complexity:
- `O(n)` for storing the list.

Code:

```java
import java.util.*;

public class MedianFinder {
    private List<Integer> numbers;

    public MedianFinder() {
        numbers = new ArrayList<>();
    }

    public void addNum(int num) {
        // Find the correct position to insert the number
        int index = Collections.binarySearch(numbers, num);
        if (index < 0) {
            index = -(index + 1); // Convert to insertion point
        }
        numbers.add(index, num);
    }

    public double findMedian() {
        int size = numbers.size();
        if (size % 2 == 0) {
            return (numbers.get(size / 2 - 1) + numbers.get(size / 2)) /
2.0;
        } else {
            return numbers.get(size / 2);
        }
    }

    public static void main(String[] args) {
        MedianFinder medianFinder = new MedianFinder();
        medianFinder.addNum(1);
        medianFinder.addNum(2);
        System.out.println(medianFinder.findMedian()); // Output: 1.5
        medianFinder.addNum(3);
        System.out.println(medianFinder.findMedian()); // Output: 2.0
    }
}
```

4. Multiset Approach (Using TreeMap)

Explanation:
- Use a TreeMap (or a multiset-like structure) to maintain the numbers in sorted order.
- For the `addNum` operation, insert the number into the TreeMap.
- For the `findMedian` operation, use the TreeMap's properties to find the median.

Time Complexity:

- Add Operation: `O(log n)` (inserting into a TreeMap).
- Find Median Operation: `O(log n)` (finding the middle element).

Space Complexity:
- `O(n)` for storing the numbers in the TreeMap.

Code:

```java
import java.util.*;

public class MedianFinder {
    private TreeMap<Integer, Integer> treeMap;
    private int size;

    public MedianFinder() {
        treeMap = new TreeMap<>();
        size = 0;
    }

    public void addNum(int num) {
        treeMap.put(num, treeMap.getOrDefault(num, 0) + 1);
        size++;
    }

    public double findMedian() {
        int mid = size / 2;
        if (size % 2 == 0) {
            return (findKthElement(mid) + findKthElement(mid + 1)) / 2.0;
        } else {
            return findKthElement(mid + 1);
        }
    }

    private int findKthElement(int k) {
        int count = 0;
        for (Map.Entry<Integer, Integer> entry : treeMap.entrySet()) {
            count += entry.getValue();
            if (count >= k) {
                return entry.getKey();
            }
        }
        return -1; // Should not reach here
    }
    public static void main(String[] args) {
        MedianFinder medianFinder = new MedianFinder();
        medianFinder.addNum(1);
        medianFinder.addNum(2);
        System.out.println(medianFinder.findMedian()); // Output: 1.5
        medianFinder.addNum(3);
        System.out.println(medianFinder.findMedian()); // Output: 2.0
    }
}
```

Comparison of Approaches:

Approach	Add Operation	Find Median Operation	Space Complexity	Notes
Sorting Approach	`O(1)`	`O(n log n)`	`O(n)`	Simple but inefficient for large datasets.
Two Heaps	`O(log n)`	`O(1)`	`O(n)`	Efficient and widely used.
Insertion Sort	`O(n)`	`O(1)`	`O(n)`	Efficient for small datasets but slow for large datasets.
Multiset (TreeMap)	`O(log n)`	`O(log n)`	`O(n)`	Flexible but slower than the Two Heaps approach.

Choose the approach based on the constraints and requirements of your problem. The Two Heaps approach is the most efficient and widely used for this problem.