OpenAI

# Dynamic Programming: Common Sequences

Dynamic Programming
1. Longest Common Subsequence Problem
The Longest Common Subsequence (LCS) problem is finding the longest subsequence present in given two sequences in the same order, i.e., find the longest sequence which can be obtained from the first original sequence by deleting some items and from the second original sequence by deleting other items.
The problem differs from the problem of finding the longest common substring. Unlike substrings, subsequences are not required to occupy consecutive positions within the original string.
For example, consider the two following sequences, X and Y: X: ABCBDAB
Y: BDCABA
The length of the LCS is 4
LCS are BDAB, BCAB, and BCBA
2. Longest Common Substring Problem
The longest common substring problem is the problem of finding the longest string (or strings) that is a substring (or are substrings) of two strings.
The problem differs from the problem of finding the Longest Common Subsequence (LCS). Unlike subsequences, substrings are required to occupy consecutive positions within the original string.
For example, the longest common substring of strings ABABC, BABCA is the string BABC having length 4. Other common substrings are ABC, A, AB, B, BA, BC, and C.
3. Matrix Chain Multiplication using Dynamic Programming (10x100),(100x5),(5x500) minimum cost is 5000
4. 0–1 Knapsack problem
Input: N = 3, W = 4, profit[] = {1, 2, 3}, weight[] = {4, 5, 1} Output: 3
5. Subset Sum Problem – Dynamic Programming Solution
Given a set of positive integers and an integer k, check if there is any non-empty subset that sums to k.

For example,
Input:
A = { 7, 3, 2, 5, 8 } k = 14
Output: Subset with the given sum exists
Subset { 7, 2, 5 } sums to 14
6. Rod Cutting Problem
Given a rod of lengthnand a list of rod prices of lengthi, where1 <= i <= n, find the optimal way to cut the rod into smaller rods to maximize profit.
For example, consider the following rod lengths and values:
Input:
length[] = [1, 2, 3, 4, 5, 6, 7, 8] price[] = [1, 5, 8, 9, 10, 17, 17, 20]
Rod length: 4
Best: Cut the rod into two pieces of length 2 each to gain revenue of 5 + 5 = 10
7. Create optimal cost to construct a binary search tree
If the keys are 10, 20, 30, 40, 50, 60, 70 Answer: 26 is the minimum cost 8. Find the maximum sum of a subsequence with no adjacent elements.
Given an integer array, find the maximum sum of subsequence where the subsequence contains no

element at adjacent positions.
Please note that the problem specifically targets subsequences that need not be contiguous, i.e., subsequences are not required to occupy consecutive positions within the original sequences.
For example,
Input: { 1, 2, 9, 4, 5, 0, 4, 11, 6 } Output: The maximum sum is 26

The maximum sum is formed by subsequence { 1, 9, 5, 11 }
DIVIDE and CONQUER
1. Inversion count of an array
Given an array, find the total number of inversions of it. If (i < j) and (A[i] > A[j]), then pair (i, j) is called an inversion of an array A. We need to count all such pairs in the array.
For example,
Input: A[] = [1, 9, 6, 4, 5]
Output: The inversion count is 5
There are 5 inversions in the array: (9, 6), (9, 4), (9, 5), (6, 4), (6, 5)
2. Find the first or last occurrence of a given number in a sorted array.
Given a sorted integer array, find the index of a given number's first or last occurrence. If the element is not present in the array, report that as well.
For example,
Input:
nums = [2, 5, 5, 5, 6, 6, 8, 9, 9, 9] target = 5
Output:
The first occurrence of element 5 is located at index 1 The last occurrence of element 5 is located at index 3
Input:
nums = [2, 5, 5, 5, 6, 6, 8, 9, 9, 9] target = 4
Output:
Element not found in the array

3. Find the smallest missing element from a sorted array
Given a sorted array of non-negative distinct integers, find the smallest missing non-negative element in it.
For example,
Input: nums[] = [0, 1, 2, 6, 9, 11, 15]
Output: The smallest missing element is 3 Input: nums[] = [1, 2, 3, 4, 6, 9, 11, 15]
Output: The smallest missing element is 0 Input: nums[] = [0, 1, 2, 3, 4, 5, 6]
Output: The smallest missing element is 7
4. Find the number of 1's in a sorted binary array
Given a sorted binary array, efficiently count the total number of 1's in it. For example,
Input: nums[] = [0, 0, 0, 0, 1, 1, 1]
Output: The total number of 1's present is 3
Input: nums[] = [0, 0, 1, 1, 1, 1, 1] Output: The total number of 1's present is 5
5. Find pairs with difference `k` in an array | Constant Space Solution
Given an unsorted integer array, find all pairs with a given difference k in it without using any extra space.
For example,
Input:

arr = [1, 5, 2, 2, 2, 5, 5, 4] k= 3
Output:
(2, 5) and (1, 4)

6. Find `k` closest elements to a given value in an array
Given a sorted integer array, find the k closest elements to target in the array where k and target are given positive integers.
The target may or may not be present in the input array. If target is less than or equal to the first element in the input array, return first k elements. Similarly, if target is more than or equal to the last element in the input array, return the last k elements. The returned elements should be in the same order as present in the input array.
For example,
Input: [10, 12, 15, 17, 18, 20, 25], k = 4, target = 16
Output: [12, 15, 17, 18]
Input: [2, 3, 4, 5, 6, 7], k = 3, target = 1
Output: [2, 3, 4]
Input: [2, 3, 4, 5, 6, 7], k = 2, target = 8
Output: [6, 7]
7. Longest Common Prefix (LCP) Problem
Write an efficient algorithm to find the longest common prefix (LCP) between a given set of strings.
For example,
Input: technique, technician, technology, technical
Output: The longest common prefix is techn
Input: techie delight, tech, techie, technology, technical Output: The longest common prefix is tech

8. Find the frequency of each element in a sorted array containing duplicates.
Given a sorted array containing duplicates, efficiently find each element's frequency without traversing the whole array.
For example,
Input: [2, 2, 2, 4, 4, 4, 5, 5, 6, 8, 8, 9] Output: {2: 3, 4: 3, 5: 2, 6: 1, 8: 2, 9: 1}
Explanation:
2 and 4 occurs thrice 5 and 8 occurs twice 6 and 9 occurs once
9. Maximum Subarray Sum using Divide and Conquer
Given an integer array, find the maximum sum among all subarrays possible.
The problem differs from the problem of finding the maximum subsequence sum. Unlike subsequences, subarrays are required to occupy consecutive positions within the original array.
For example,
Input: nums[] = [2, -4, 1, 9, -6, 7, -3]
Output: The maximum sum of the subarray is 11 (Marked in Green)
GREEDY Algorithms
1. Graph Coloring Problem

Graph coloring (also called vertex coloring) is a way of coloring a graph's vertices such that no two adjacent vertices share the same color. Implement a greedy algorithm for graph coloring and minimize the total number of colors used.
For example, consider the following graph:
We can color it in many ways by using the minimum of 3 colors.
2. Huffman Coding Compression Algorithm
Huffman coding (also known as Huffman Encoding) is an algorithm for doing data compression, and it forms the basic idea behind file compression. Implement the variable- length encoding, uniquely decodable codes, prefix rules, and Huffman Tree construction.
3. Kruskal's and Prim's Algorithm for finding Minimum Spanning Tree

Given a connected and weighted undirected graph, construct a minimum spanning tree out of it using Kruskal's and Prim's Algorithm.

these are my lab exam problems

you have to write complete java codes and explanation for all 18 problems

answer 1 problem per response

write every problem with its number and heading

i will ask "next" for next problem

Understood, let's tackle the first problem:

**Dynamic Programming**
**1. Longest Common Subsequence Problem**

In the Longest Common Subsequence (LCS) problem, the task is to find the longest subsequence present in two given sequences in the same order. Subsequences are not required to occupy consecutive positions within the original string.

**Java Code:**

```java
public class LongestCommonSubsequence {

    public static int longestCommonSubsequence(String X, String Y) {
        int m = X.length();
        int n = Y.length();

        int[][] dp = new int[m + 1][n + 1];

        // Building the dp table bottom-up
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (X.charAt(i - 1) == Y.charAt(j - 1))
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                else
                    dp[i][j] = Math.max(dp[i - 1][j], dp[i][j - 1]);
            }
        }

        return dp[m][n];
    }

    public static void main(String[] args) {
        String X = "ABCBDAB";
        String Y = "BDCABA";
        System.out.println("Length of Longest Common Subsequence: " +
longestCommonSubsequence(X, Y));
```

```
        }
    }
```

**Explanation:**

- We create a 2D array `dp` to store the lengths of longest common subsequences for different prefixes of the two strings.
- We iterate through the strings `X` and `Y`, comparing characters.
- If characters match, we increment the count of the LCS by 1 and consider the next characters.
- If characters don't match, we take the maximum LCS length from the previous characters.
- Finally, we return the length of the LCS.

next

**Dynamic Programming**
**2. Longest Common Substring Problem**

The Longest Common Substring problem involves finding the longest string that is a substring of two given strings. Unlike subsequences, substrings are required to occupy consecutive positions within the original string.

**Java Code:**

```java
public class LongestCommonSubstring {

    public static int longestCommonSubstring(String X, String Y) {
        int m = X.length();
        int n = Y.length();
        int[][] dp = new int[m + 1][n + 1];
        int maxLength = 0;

        // Building the dp table bottom-up
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; j++) {
                if (X.charAt(i - 1) == Y.charAt(j - 1)) {
                    dp[i][j] = 1 + dp[i - 1][j - 1];
                    maxLength = Math.max(maxLength, dp[i][j]);
                } else {
                    dp[i][j] = 0; // Resetting length for non-matching characters
                }
            }
        }

        return maxLength;
    }

    public static void main(String[] args) {
        String X = "ABABC";
        String Y = "BABCA";
        System.out.println("Length of Longest Common Substring: " +
```

```
longestCommonSubstring(X, Y));
    }
}
```

**Explanation:**

- Similar to the LCS problem, we use a 2D array `dp` to store the lengths of longest common substrings for different prefixes of the two strings.
- We iterate through the strings `X` and `Y`, comparing characters.
- If characters match, we increment the count of the current common substring length by 1 and update the maximum length found so far.
- If characters don't match, we reset the length for non-matching characters to 0.
- Finally, we return the maximum length of the common substring found.

next

**Dynamic Programming**
**3. Matrix Chain Multiplication using Dynamic Programming**

The Matrix Chain Multiplication problem involves finding the most efficient way to multiply a given sequence of matrices. The goal is to minimize the number of multiplications needed.

**Java Code:**

```java
public class MatrixChainMultiplication {

    public static int matrixChainOrder(int[] p) {
        int n = p.length;
        int[][] dp = new int[n][n];

        // Initializing the dp table
        for (int i = 1; i < n; i++) {
            dp[i][i] = 0;
        }

        // Building the dp table bottom-up
        for (int L = 2; L < n; L++) {
            for (int i = 1; i < n - L + 1; i++) {
                int j = i + L - 1;
                dp[i][j] = Integer.MAX_VALUE;
                for (int k = i; k <= j - 1; k++) {
                    int q = dp[i][k] + dp[k + 1][j] + p[i - 1] * p[k] * p[j];
                    if (q < dp[i][j]) {
                        dp[i][j] = q;
                    }
                }
            }
        }
        return dp[1][n - 1];
    }
```

```
    public static void main(String[] args) {
        int[] dimensions = {10, 100, 5, 500};
        System.out.println("Minimum cost of matrix chain multiplication: " +
matrixChainOrder(dimensions));
    }
}
```

**Explanation:**

- We use dynamic programming to solve this problem by building a 2D array `dp` to store the minimum number of multiplications needed to compute the matrix chain product for different subchains of matrices.
- We iterate over different lengths of subchains, starting from 2 matrices up to all matrices.
- For each subchain length, we iterate over all possible positions to split the subchain and calculate the cost of multiplying the two resulting subchains plus the cost of multiplying the matrices in the subchain.
- We update the minimum cost found so far in the `dp` table.
- Finally, we return the minimum cost of multiplying the entire matrix chain.

next

**Dynamic Programming**
**4. 0–1 Knapsack problem**

The 0-1 Knapsack problem involves maximizing the value of items selected for a knapsack while keeping the total weight within the knapsack's capacity. Each item can only be selected once (0-1 property).

**Java Code:**

```java
public class Knapsack {

    public static int knapsack(int W, int[] wt, int[] val, int n) {
        int[][] dp = new int[n + 1][W + 1];

        // Building the dp table bottom-up
        for (int i = 0; i <= n; i++) {
            for (int w = 0; w <= W; w++) {
                if (i == 0 || w == 0) {
                    dp[i][w] = 0;
                } else if (wt[i - 1] <= w) {
                    dp[i][w] = Math.max(val[i - 1] + dp[i - 1][w - wt[i - 1]], dp[i - 1]
[w]);
                } else {
                    dp[i][w] = dp[i - 1][w];
                }
            }
        }
        return dp[n][W];
    }
```

```
    public static void main(String[] args) {
        int[] val = {1, 2, 3};
        int[] wt = {4, 5, 1};
        int W = 4;
        int n = val.length;
        System.out.println("Maximum value in Knapsack: " + knapsack(W, wt, val, n));
    }
}
```

**Explanation:**

- We use dynamic programming to solve this problem by building a 2D array `dp` to store the maximum value that can be obtained for different combinations of items and knapsack capacities.
- We iterate over the items and knapsack capacities, considering each item and each possible weight capacity.
- If the weight of the current item is less than or equal to the current knapsack capacity, we choose the maximum value between including the current item and excluding it.
- If the weight of the current item exceeds the knapsack capacity, we exclude the item and carry forward the previous maximum value.
- Finally, we return the maximum value that can be obtained within the given knapsack capacity.

next

**Dynamic Programming**
**5. Subset Sum Problem – Dynamic Programming Solution**

The Subset Sum problem involves determining whether there exists a subset of a given set of positive integers that sums up to a given target integer.

**Java Code:**

```java
import java.util.Arrays;

public class SubsetSum {

    public static boolean subsetSum(int[] nums, int k) {
        int n = nums.length;
        boolean[][] dp = new boolean[n + 1][k + 1];

        // Base case initialization
        for (int i = 0; i <= n; i++) {
            dp[i][0] = true;
        }

        // Building the dp table bottom-up
        for (int i = 1; i <= n; i++) {
            for (int j = 1; j <= k; j++) {
                if (nums[i - 1] <= j) {
                    dp[i][j] = dp[i - 1][j] || dp[i - 1][j - nums[i - 1]];
                } else {
```

```
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }

        return dp[n][k];
    }

    public static void main(String[] args) {
        int[] A = {7, 3, 2, 5, 8};
        int k = 14;
        if (subsetSum(A, k)) {
            System.out.println("Subset with the given sum exists");
            System.out.println("Subset: " + Arrays.toString(getSubset(A, k)));
        } else {
            System.out.println("No subset with the given sum exists");
        }
    }

    public static int[] getSubset(int[] nums, int k) {
        int n = nums.length;
        StringBuilder subset = new StringBuilder();
        int i = n;
        while (i > 0 && k > 0) {
            if (nums[i - 1] <= k && dp[i][k] && !dp[i - 1][k]) {
                subset.insert(0, nums[i - 1] + " ");
                k -= nums[i - 1];
            }
            i--;
        }
        String[] subsetArray = subset.toString().split(" ");
        int[] result = new int[subsetArray.length];
        for (int j = 0; j < subsetArray.length; j++) {
            result[j] = Integer.parseInt(subsetArray[j]);
        }
        return result;
    }
}
```

**Explanation:**

- We use dynamic programming to solve this problem by building a 2D array `dp` to store whether there exists a subset of the given set that sums up to a target integer for different sizes of the set and different target integers.
- We initialize the base case where the sum is zero (empty subset) as true.
- We iterate over the set of numbers and the possible target sums, updating the dp table based on whether including the current number leads to a subset summing up to the target sum.
- Finally, we return whether there exists a subset that sums up to the given target integer.

next

**Dynamic Programming**
**6. Rod Cutting Problem**

The Rod Cutting problem involves finding the optimal way to cut a rod of length $n$ into smaller rods to maximize profit, given the prices of rods of different lengths.

**Java Code:**

```java
public class RodCutting {

    public static int rodCutting(int[] lengths, int[] prices, int n) {
        int[] dp = new int[n + 1];

        // Building the dp table bottom-up
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < i; j++) {
                dp[i] = Math.max(dp[i], prices[j] + dp[i - (j + 1)]);
            }
        }

        return dp[n];
    }

    public static void main(String[] args) {
        int[] lengths = {1, 2, 3, 4, 5, 6, 7, 8};
        int[] prices = {1, 5, 8, 9, 10, 17, 17, 20};
        int rodLength = 4;
        System.out.println("Maximum profit from rod cutting: " + rodCutting(lengths, prices,
rodLength));
    }
}
```

**Explanation:**

- We use dynamic programming to solve this problem by building a 1D array `dp` to store the maximum profit that can be obtained for different lengths of the rod.
- We iterate over the rod lengths from 1 to $n$, considering each length and finding the maximum profit that can be obtained by cutting the rod at different positions.
- For each rod length, we iterate over all possible cuts and calculate the profit by adding the price of the current cut piece and the maximum profit from the remaining rod length.
- Finally, we return the maximum profit that can be obtained by cutting the rod of length $n$.

next

**Dynamic Programming**
**7. Create optimal cost to construct a binary search tree**

The problem involves creating an optimal cost to construct a binary search tree from a given set of keys. The keys are given, and the goal is to minimize the total cost of constructing the binary search tree.

**Java Code:**

```java
public class OptimalBinarySearchTree {

    public static int optimalBST(int[] keys, int[] freq) {
        int n = keys.length;
        int[][] dp = new int[n][n];

        // Building the dp table bottom-up
        for (int len = 1; len <= n; len++) {
            for (int i = 0; i <= n - len; i++) {
                int j = i + len - 1;
                dp[i][j] = Integer.MAX_VALUE;
                int sum = sum(freq, i, j);

                for (int k = i; k <= j; k++) {
                    int cost = sum + (k - 1 < i ? 0 : dp[i][k - 1]) + (k + 1 > j ? 0 : dp[k
+ 1][j]);
                    dp[i][j] = Math.min(dp[i][j], cost);
                }
            }
        }
        return dp[0][n - 1];
    }

    private static int sum(int[] freq, int i, int j) {
        int sum = 0;
        for (int k = i; k <= j; k++) {
            sum += freq[k];
        }
        return sum;
    }

    public static void main(String[] args) {
        int[] keys = {10, 20, 30, 40, 50, 60, 70};
        int[] freq = {4, 2, 6, 3, 1, 5, 7};
        System.out.println("Minimum cost to construct optimal BST: " + optimalBST(keys,
freq));
    }
}
```

**Explanation:**

- We use dynamic programming to solve this problem by building a 2D array `dp` to store the minimum cost of constructing an optimal binary search tree for different ranges of keys.
- We iterate over different lengths of the ranges (subtrees), starting from 1 key up to all keys.
- For each range length, we iterate over all possible roots within that range and calculate the cost of constructing the subtree with that root.
- The cost consists of the sum of frequencies of keys in the range, plus the costs of left and right subtrees (if they exist).
- We update the minimum cost found so far in the `dp` table.
- Finally, we return the minimum cost of constructing the optimal binary search tree from the entire set of keys.

next

**Dynamic Programming**

**8. Find the maximum sum of a subsequence with no adjacent elements**

The problem involves finding the maximum sum of a subsequence where no two elements are adjacent. Subsequences are not required to occupy consecutive positions within the original sequence.

**Java Code:**

```java
public class MaxSumNoAdjacent {

    public static int maxSumNoAdjacent(int[] nums) {
        int n = nums.length;
        if (n == 0) return 0;
        if (n == 1) return nums[0];

        int[] dp = new int[n];
        dp[0] = nums[0];
        dp[1] = Math.max(nums[0], nums[1]);

        for (int i = 2; i < n; i++) {
            dp[i] = Math.max(dp[i - 1], nums[i] + dp[i - 2]);
        }

        return dp[n - 1];
    }

    public static void main(String[] args) {
        int[] nums = {1, 2, 9, 4, 5, 0, 4, 11, 6};
        System.out.println("The maximum sum is: " + maxSumNoAdjacent(nums));
    }
}
```

**Explanation:**

- We use dynamic programming to solve this problem by building a 1D array `dp` to store the maximum sum of subsequences ending at each position.
- We initialize the first two elements of `dp` array with the maximum of the first element and the second element.
- Then, we iterate from the third element onwards and compute the maximum sum of subsequences ending at the current position by considering whether to include the current element or not.
- The maximum sum at each position is the maximum of the sum at the previous position (`dp[i - 1]`) and the sum two positions back plus the current element (`nums[i] + dp[i - 2]`).
- Finally, we return the maximum sum of subsequences.

next

**Divide and Conquer**

**1. Inversion count of an array**

The problem involves finding the total number of inversions in an array. An inversion occurs when a pair of elements (i, j) where i < j, have A[i] > A[j].

**Java Code:**

```java
public class InversionCount {

    public static int inversionCount(int[] arr) {
        return mergeSort(arr, 0, arr.length - 1);
    }

    public static int mergeSort(int[] arr, int left, int right) {
        int count = 0;
        if (left < right) {
            int mid = (left + right) / 2;
            count += mergeSort(arr, left, mid);
            count += mergeSort(arr, mid + 1, right);
            count += merge(arr, left, mid, right);
        }
        return count;
    }

    public static int merge(int[] arr, int left, int mid, int right) {
        int[] temp = new int[right - left + 1];
        int i = left, j = mid + 1, k = 0;
        int count = 0;

        while (i <= mid && j <= right) {
            if (arr[i] <= arr[j]) {
                temp[k++] = arr[i++];
            } else {
                temp[k++] = arr[j++];
                count += mid - i + 1; // Counting inversions
            }
        }

        while (i <= mid) {
            temp[k++] = arr[i++];
        }
        while (j <= right) {
            temp[k++] = arr[j++];
        }

        for (i = left; i <= right; i++) {
            arr[i] = temp[i - left];
        }

        return count;
    }

    public static void main(String[] args) {
        int[] arr = {1, 9, 6, 4, 5};
        System.out.println("The inversion count is: " + inversionCount(arr));
    }
}
```

**Explanation:**

- We use the merge sort algorithm to count inversions in the array.

- During the merge step of merge sort, while merging two sorted halves of the array, we count the inversions by checking if the element in the left half is greater than the element in the right half.
- If it is, then there are `mid - i + 1` inversions because all elements from `i` to `mid` are greater than `arr[j]`.
- We accumulate the count of inversions during the merge step.
- Finally, we return the total count of inversions in the array.

next

**Divide and Conquer**
**2. Find the first or last occurrence of a given number in a sorted array**

This problem involves finding the index of the first or last occurrence of a given number in a sorted array. If the element is not present in the array, it reports that as well.

**Java Code:**

```java
public class FirstLastOccurrence {

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

        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) {
                result = mid;
                right = mid - 1; // Move to the left side to find the first occurrence
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
        }

        return result;
    }

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

        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == target) {
                result = mid;
                left = mid + 1; // Move to the right side to find the last occurrence
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                right = mid - 1;
            }
```

```
        }

        return result;
    }

    public static void main(String[] args) {
        int[] nums = {2, 5, 5, 5, 6, 6, 8, 9, 9, 9};
        int target = 5;
        int firstIndex = firstOccurrence(nums, target);
        int lastIndex = lastOccurrence(nums, target);

        if (firstIndex != -1) {
            System.out.println("The first occurrence of element " + target + " is located at index " + firstIndex);
            System.out.println("The last occurrence of element " + target + " is located at index " + lastIndex);
        } else {
            System.out.println("Element not found in the array");
        }
    }
}
```

**Explanation:**

- We use binary search to find the first and last occurrences of the target element.
- For finding the first occurrence, we modify the binary search to move towards the left side when the target element is found.
- For finding the last occurrence, we modify the binary search to move towards the right side when the target element is found.
- We return the index of the first or last occurrence accordingly.
- If the element is not found, we return -1.

next

**Divide and Conquer**
**3. Find the smallest missing element from a sorted array**

This problem involves finding the smallest missing non-negative element in a sorted array of distinct integers.

**Java Code:**

```java
public class SmallestMissingElement {

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

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

```
            // If mid equals its value in the array, missing element is on the right side
            if (nums[mid] == mid) {
                left = mid + 1;
            } else {
                right = mid - 1; // Otherwise, missing element is on the left side
            }
        }

        return left;
    }

    public static void main(String[] args) {
        int[] nums1 = {0, 1, 2, 6, 9, 11, 15};
        int[] nums2 = {1, 2, 3, 4, 6, 9, 11, 15};
        int[] nums3 = {0, 1, 2, 3, 4, 5, 6};

        System.out.println("The smallest missing element is: " +
smallestMissingElement(nums1));
        System.out.println("The smallest missing element is: " +
smallestMissingElement(nums2));
        System.out.println("The smallest missing element is: " +
smallestMissingElement(nums3));
    }
}
```

**Explanation:**

- We use binary search to find the smallest missing element.
- If the middle element is equal to its index in the array, the missing element must be on the right side, so we update the left pointer.
- If the middle element is not equal to its index, the missing element must be on the left side, so we update the right pointer.
- We continue this process until we find the smallest missing element, which will be the left pointer.

next

**Divide and Conquer**
**4. Find the number of 1's in a sorted binary array**

This problem involves efficiently counting the total number of 1's in a sorted binary array.

**Java Code:**

```java
public class CountOnes {

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

        // If the last element is 0, no 1's are present
        if (nums[right] == 0) return 0;
```

```
        // Binary search to find the index of first 1
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[mid] == 1 && (mid == 0 || nums[mid - 1] == 0)) {
                return nums.length - mid;
            } else if (nums[mid] == 1) {
                right = mid - 1; // Move to the left side to find the first 1
            } else {
                left = mid + 1;
            }
        }

        return 0; // Unreachable code
    }

    public static void main(String[] args) {
        int[] nums1 = {0, 0, 0, 0, 1, 1, 1};
        int[] nums2 = {0, 0, 1, 1, 1, 1, 1};

        System.out.println("The total number of 1's is: " + countOnes(nums1));
        System.out.println("The total number of 1's is: " + countOnes(nums2));
    }
}
```

**Explanation:**

- We use binary search to efficiently find the index of the first occurrence of 1 in the array.
- Once we find the first 1, the total number of 1's in the array can be calculated by subtracting the index of the first 1 from the length of the array.
- If the last element of the array is 0, it means there are no 1's in the array, so we return 0.
- If the element at the mid index is 1 and the element before it (if exists) is 0, then mid is the index of the first 1. We return the count of 1's from mid to the end of the array.
- If the element at mid is 1 but the element before it is also 1, we move to the left side to find the first 1.
- Otherwise, we move to the right side of the array.

next

**Divide and Conquer**
**5. Find pairs with difference `k` in an array | Constant Space Solution**

This problem involves finding all pairs with a given difference `k` in an unsorted integer array without using any extra space.

**Java Code:**

```java
import java.util.Arrays;

public class PairWithDifferenceK {
```

```
    public static void findPairsWithDifferenceK(int[] arr, int k) {
        Arrays.sort(arr); // Sort the array in non-decreasing order
        int left = 0;
        int right = 1;

        while (right < arr.length) {
            int diff = arr[right] - arr[left];
            if (diff == k) {
                System.out.println("(" + arr[left] + ", " + arr[right] + ")");
                left++;
                right++;
            } else if (diff < k) {
                right++;
            } else {
                left++;
            }
            // Avoid counting the same pair multiple times
            if (left == right) {
                right++;
            }
        }
    }

    public static void main(String[] args) {
        int[] arr = {1, 5, 2, 2, 2, 5, 5, 4};
        int k = 3;
        System.out.println("Pairs with difference " + k + " are:");
        findPairsWithDifferenceK(arr, k);
    }
}
```

**Explanation:**

- We first sort the array in non-decreasing order to easily find pairs with the given difference `k`.
- Then, we use two pointers, `left` and `right`, initially pointing to the first and second elements of the sorted array, respectively.
- We move the `right` pointer forward until the difference between the elements at `left` and `right` is equal to `k`.
- If the difference is less than `k`, we increment the `right` pointer.
- If the difference is greater than `k`, we increment the `left` pointer.
- We avoid counting the same pair multiple times by ensuring that the `left` and `right` pointers are not pointing to the same element before moving both forward.
- We continue this process until the `right` pointer reaches the end of the array.

next

**Divide and Conquer**

**6. Find `k` closest elements to a given value in an array**

Given a sorted integer array, this problem involves finding the `k` closest elements to a given target value in the array. If the target value is less than or equal to the first element in the array, it returns the first `k` elements. Similarly, if the target value is greater than or equal to the last element in the array, it returns the last `k` elements. The returned elements should be in the same order as they appear in the input array.

**Java Code:**

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

public class KClosestElements {

    public static List<Integer> findClosestElements(int[] arr, int k, int target) {
        List<Integer> result = new ArrayList<>();

        int left = 0;
        int right = arr.length - 1;

        // Binary search to find the closest element to the target
        while (left < right) {
            int mid = left + (right - left) / 2;
            if (arr[mid] == target) {
                left = mid;
                break;
            } else if (arr[mid] < target) {
                left = mid + 1;
            } else {
                right = mid;
            }
        }

        // Initialize pointers based on the position of the closest element found
        int l = left - 1;
        int r = left;

        // Find k closest elements using two pointers approach
        while (k > 0) {
            if (l < 0 || (r < arr.length && Math.abs(arr[l] - target) > Math.abs(arr[r] -
target))) {
                result.add(arr[r]);
                r++;
            } else {
                result.add(arr[l]);
                l--;
            }
            k--;
        }

        return result;
    }

    public static void main(String[] args) {
        int[] arr = {10, 12, 15, 17, 18, 20, 25};
        int k1 = 4;
        int target1 = 16;
        System.out.println("Closest elements to target " + target1 + ": " +
findClosestElements(arr, k1, target1));

        int k2 = 3;
        int target2 = 1;
        System.out.println("Closest elements to target " + target2 + ": " +
findClosestElements(arr, k2, target2));

        int k3 = 2;
        int target3 = 8;
        System.out.println("Closest elements to target " + target3 + ": " +
findClosestElements(arr, k3, target3));
```

```
        }
    }
```

**Explanation:**

- We first perform a binary search to find the closest element to the target value in the array.
- Once we find the closest element, we initialize two pointers, `l` and `r`, to the left and right of the closest element.
- We use a two-pointer approach to find `k` closest elements by comparing the absolute differences between elements and the target value.
- We add the closest elements to a list and return the list as the result.

next

**Divide and Conquer**
**7. Longest Common Prefix (LCP) Problem**

This problem involves finding the longest common prefix (LCP) among a given set of strings.

**Java Code:**

```java
public class LongestCommonPrefix {

    public static String longestCommonPrefix(String[] strs) {
        if (strs == null || strs.length == 0) {
            return "";
        }

        // Find the minimum length string in the array
        int minLen = Integer.MAX_VALUE;
        for (String str : strs) {
            minLen = Math.min(minLen, str.length());
        }

        // Perform binary search to find the longest common prefix
        int low = 0;
        int high = minLen - 1;

        while (low <= high) {
            int mid = low + (high - low) / 2;
            if (isCommonPrefix(strs, mid)) {
                low = mid + 1;
            } else {
                high = mid - 1;
            }
        }

        return strs[0].substring(0, (low + high) / 2 + 1);
    }

    private static boolean isCommonPrefix(String[] strs, int len) {
        String prefix = strs[0].substring(0, len + 1);
```

```java
        for (int i = 1; i < strs.length; i++) {
            if (!strs[i].startsWith(prefix)) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) {
        String[] strs1 = {"technique", "technician", "technology", "technical"};
        String[] strs2 = {"techie delight", "tech", "techie", "technology", "technical"};

        System.out.println("The longest common prefix is: " + longestCommonPrefix(strs1));
        System.out.println("The longest common prefix is: " + longestCommonPrefix(strs2));
    }
}
```

**Explanation:**

- We first find the minimum length string in the array, as the common prefix cannot be longer than this length.
- Then, we perform binary search on the range [0, minLen - 1] to find the longest common prefix.
- In each iteration of the binary search, we check if the prefix of length `mid` is common among all strings using the `isCommonPrefix` method.
- If it is, we move the lower bound `low` to `mid + 1` to search for a longer common prefix.
- If it's not, we move the upper bound `high` to `mid - 1` to search for a shorter common prefix.
- We continue this process until `low` is greater than `high`, at which point we return the longest common prefix found.

next

**Divide and Conquer**
**8. Find the frequency of each element in a sorted array containing duplicates**

This problem involves efficiently finding the frequency of each element in a sorted array containing duplicates.

**Java Code:**

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

public class FrequencyOfElements {

    public static Map<Integer, Integer> findFrequency(int[] nums) {
        Map<Integer, Integer> frequencyMap = new HashMap<>();

        int count = 1;
        for (int i = 1; i < nums.length; i++) {
            if (nums[i] == nums[i - 1]) {
```

```
                count++;
            } else {
                frequencyMap.put(nums[i - 1], count);
                count = 1;
            }
        }
        frequencyMap.put(nums[nums.length - 1], count);

        return frequencyMap;
    }

    public static void main(String[] args) {
        int[] nums = {2, 2, 2, 4, 4, 4, 5, 5, 6, 8, 8, 9};

        Map<Integer, Integer> frequencyMap = findFrequency(nums);
        System.out.println("Frequency of elements:");
        for (Map.Entry<Integer, Integer> entry : frequencyMap.entrySet()) {
            System.out.println(entry.getKey() + ": " + entry.getValue());
        }
    }
}
```

**Explanation:**

- We iterate through the sorted array and maintain a count for each element.
- When encountering a new element, we update the count and store the frequency of the previous element in a map.
- Finally, we add the frequency of the last element to the map and return it.

next

**Divide and Conquer**
**9. Maximum Subarray Sum using Divide and Conquer**

Given an integer array, this problem involves finding the maximum sum among all subarrays possible.

**Java Code:**

```java
public class MaximumSubarraySum {

    public static int maxSubarraySum(int[] nums) {
        return maxSubarraySum(nums, 0, nums.length - 1);
    }

    public static int maxSubarraySum(int[] nums, int left, int right) {
        if (left == right) {
            return nums[left];
        }

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

        // Maximum subarray sum in the left subarray
        int leftSum = maxSubarraySum(nums, left, mid);
```

```
        // Maximum subarray sum in the right subarray
        int rightSum = maxSubarraySum(nums, mid + 1, right);

        // Maximum subarray sum crossing the midpoint
        int crossSum = maxCrossingSum(nums, left, mid, right);

        // Return the maximum of the three sums
        return Math.max(Math.max(leftSum, rightSum), crossSum);
    }

    public 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];
            leftSum = Math.max(leftSum, sum);
        }

        int rightSum = Integer.MIN_VALUE;
        sum = 0;
        for (int i = mid + 1; i <= right; i++) {
            sum += nums[i];
            rightSum = Math.max(rightSum, sum);
        }

        return leftSum + rightSum;
    }

    public static void main(String[] args) {
        int[] nums = {2, -4, 1, 9, -6, 7, -3};
        System.out.println("The maximum subarray sum is: " + maxSubarraySum(nums));
    }
}
```

**Explanation:**

- The `maxSubarraySum` method recursively divides the array into two halves and computes the maximum subarray sum for each half.
- It also calculates the maximum subarray sum crossing the midpoint using the `maxCrossingSum` method.
- The `maxCrossingSum` method computes the maximum sum of a subarray that crosses the midpoint.
- Finally, it returns the maximum of the three sums computed: the maximum subarray sum in the left half, the maximum subarray sum in the right half, and the maximum subarray sum crossing the midpoint.

next greedy algorithms 2 problems left

Sure! Let's move on to the Greedy Algorithms section. Which problem would you like to start with: Graph Coloring Problem or Huffman Coding Compression Algorithm?

yes

Let's start with the **Graph Coloring Problem**.

### Graph Coloring Problem

Graph coloring (also called vertex coloring) is a way of coloring a graph's vertices such that no two adjacent vertices share the same color. Implementing a greedy algorithm for graph coloring aims to minimize the total number of colors used.

### Java Code:

```java
import java.util.*;

public class GraphColoring {

    static class Graph {
        int V;
        LinkedList<Integer>[] adj;

        Graph(int v) {
            V = v;
            adj = new LinkedList[v];
            for (int i = 0; i < v; ++i)
                adj[i] = new LinkedList();
        }

        void addEdge(int v, int w) {
            adj[v].add(w);
            adj[w].add(v);
        }

        void greedyColoring() {
            int[] result = new int[V];
            Arrays.fill(result, -1);

            result[0] = 0;

            boolean[] available = new boolean[V];
            Arrays.fill(available, true);

            for (int u = 1; u < V; u++) {
                Iterator<Integer> it = adj[u].iterator();
                while (it.hasNext()) {
                    int i = it.next();
                    if (result[i] != -1)
                        available[result[i]] = false;
                }

                int cr;
                for (cr = 0; cr < V; cr++) {
                    if (available[cr])
                        break;
                }

                result[u] = cr;
```

```java
                Arrays.fill(available, true);
            }

            for (int u = 0; u < V; u++)
                System.out.println("Vertex " + u + " ---> Color " + result[u]);
        }
    }

    public static void main(String args[]) {
        Graph g1 = new Graph(5);
        g1.addEdge(0, 1);
        g1.addEdge(0, 2);
        g1.addEdge(1, 2);
        g1.addEdge(1, 3);
        g1.addEdge(2, 3);
        g1.addEdge(3, 4);
        System.out.println("Coloring of graph 1 ");
        g1.greedyColoring();

        Graph g2 = new Graph(5);
        g2.addEdge(0, 1);
        g2.addEdge(0, 2);
        g2.addEdge(1, 2);
        g2.addEdge(1, 4);
        g2.addEdge(2, 4);
        g2.addEdge(4, 3);
        System.out.println("Coloring of graph 2 ");
        g2.greedyColoring();
    }
}
```

**Explanation:**

- The `Graph` class represents an undirected graph using adjacency lists.
- The `addEdge` method is used to add edges to the graph.
- The `greedyColoring` method implements the greedy algorithm for graph coloring.
- It iterates through each vertex of the graph and assigns the lowest available color to it, such that no adjacent vertices have the same color.
- Finally, it prints the color assigned to each vertex.

next

Let's proceed with the **Huffman Coding Compression Algorithm**.

**Huffman Coding Compression Algorithm**

Huffman coding (also known as Huffman Encoding) is an algorithm for data compression. It forms the basic idea behind file compression. Implementing the Huffman Coding algorithm involves variable-length encoding, uniquely decodable codes, prefix rules, and Huffman Tree construction.

**Java Code:**

java

```java
import java.util.*;

public class HuffmanCoding {

    static class HuffmanNode {
        int data;
        char c;
        HuffmanNode left, right;
    }

    static class MyComparator implements Comparator<HuffmanNode> {
        public int compare(HuffmanNode x, HuffmanNode y) {
            return x.data - y.data;
        }
    }

    public static void printCode(HuffmanNode root, String s) {
        if (root.left == null && root.right == null && Character.isLetter(root.c)) {
            System.out.println(root.c + ":" + s);
            return;
        }
        printCode(root.left, s + "0");
        printCode(root.right, s + "1");
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int n = 6;
        char[] charArray = {'a', 'b', 'c', 'd', 'e', 'f'};
        int[] charFreq = {5, 9, 12, 13, 16, 45};

        PriorityQueue<HuffmanNode> q = new PriorityQueue<HuffmanNode>(n, new
MyComparator());

        for (int i = 0; i < n; i++) {
            HuffmanNode hn = new HuffmanNode();

            hn.c = charArray[i];
            hn.data = charFreq[i];

            hn.left = null;
            hn.right = null;

            q.add(hn);
        }

        HuffmanNode root = null;

        while (q.size() > 1) {
            HuffmanNode x = q.peek();
            q.poll();

            HuffmanNode y = q.peek();
            q.poll();

            HuffmanNode f = new HuffmanNode();

            f.data = x.data + y.data;
            f.c = '-';
            f.left = x;
            f.right = y;
            root = f;
            q.add(f);
        }

        System.out.println("Huffman Codes are : ");
        printCode(root, "");
    }
}
```

**Explanation:**

- `HuffmanNode` class represents a node in the Huffman Tree.
- `MyComparator` class is used to compare `HuffmanNode` objects based on their frequency values.
- The `printCode` method recursively traverses the Huffman Tree and prints the Huffman codes for each character.
- In the `main` method, we initialize the character array and their corresponding frequencies.
- We construct a priority queue of `HuffmanNode` objects sorted by their frequencies.
- We then build the Huffman Tree by repeatedly merging the two nodes with the lowest frequencies until only one node remains.
- Finally, we print the Huffman codes for each character by traversing the Huffman Tree.