SAHIL LOKWANI
22SCSE1310012 (SEC-47)
AAPS MTE ASSIGNMENT

1. Prefix Sum Array: Concept and Applications

A prefix sum array is an array where each element at index `i` stores the sum of elements from index `0` to `i` of the original array.

Applications:

- Efficient range sum queries
- Solving subarray problems
- Used in 2D matrix sum queries
- Histogram and frequency computations

2. Sum of Elements in a Range using Prefix Sum Array

Algorithm:

1. Compute the prefix sum array.
2. To find the sum of the range [L, R], return `prefixSum[R]` - `prefixSum[L-1]` (or just `prefixSum[R]` if L == 0).

Java Program:

Java

```
class PrefixSum {

    // Method to calculate the prefix sum array
static int[] calculatePrefixSum(int[] arr) {
int[] prefixSum = new int[arr.length];
prefixSum[0] = arr[0];
        for (int i = 1; i < arr.length; i++) {
prefixSum[i] = prefixSum[i - 1] + arr[i];
        }
        return prefixSum;
    }

    // Method to find the sum of elements in the range [L, R]
static int rangeSum(int[] prefixSum, int L, int R) {
if (L == 0) {
            return prefixSum[R];
        }
        return prefixSum[R] - prefixSum[L - 1];
}      public static void main(String[] args) {
int[] arr = {1, 2, 3, 4, 5};         int[]
prefixSum = calculatePrefixSum(arr);         int L
= 1;          int R = 3;
        System.out.println("Sum of elements in range [" + L + ", " + R
+ "]: " + rangeSum(prefixSum, L, R)); // Output: 9
    } }
```

Time and Space Complexity:

- **Time Complexity:** `O(n)` to calculate the prefix sum, `O(1)` to find the range sum.
- **Space Complexity:** `O(n)` for the prefix sum array.

Example:

`arr` = {1, 2, 3, 4, 5}, `prefixSum` = {1, 3, 6, 10, 15}. Sum of range [1, 3] is `prefixSum[3]` - `prefixSum[0]` = 10 - 1 = 9.

3. Equilibrium Index in an Array

Algorithm:

5. Calculate the total sum of the array.
6. Iterate through the array, keeping track of the left sum.
7. For each element, check if the left sum equals the (total sum - left sum - current element).
8. If it does, return the index.

Java Program:

Java

```
class EquilibriumIndex {

    static int findEquilibriumIndex(int[] arr) {
int totalSum = 0;           for (int num : arr) {
totalSum += num;
        }
        int leftSum = 0;
        for (int i = 0; i < arr.length; i++) {
if (leftSum == totalSum - leftSum - arr[i]) {
return i;
            }
            leftSum += arr[i];
        }
        return -1; // No equilibrium index
    }      public static void main(String[] args)
{         int[] arr = {-7, 1, 5, 2, -4, 3, 0};
System.out.println("Equilibrium index: " +
findEquilibriumIndex(arr)); // Output: 3
    }
}
```

Time and Space Complexity:

- **Time Complexity:** `O(n)` ○ **Space Complexity:** `O(1)`

Example:

In `arr` = {-7, 1, 5, 2, -4, 3, 0}, index 3 is the equilibrium index because -7 + 1 + 5 = 3 + 0.

4. Split Array into Two Parts with Equal Sum

Algorithm:

11. Calculate the total sum of the array.
12. Iterate through the array, calculating the left sum.
13. If the left sum equals half of the total sum, then it's possible to split.

Java Program:

Java

```
class SplitArray {

    static boolean canSplitArray(int[] arr) {
int totalSum = 0;         for (int num : arr)
{             totalSum += num;
        }          int
leftSum = 0;           for (int
num : arr) {
leftSum += num;
            if (leftSum == totalSum - leftSum) {
return true;
            }
}
        return false;
    }
    public static void main(String[] args) {
int[] arr1 = {1, 2, 3, 3};         int[]
arr2 = {1, 2, 3, 4};
        System.out.println("Can split arr1? " + canSplitArray(arr1));
// Output: true
        System.out.println("Can split arr2? " + canSplitArray(arr2));
// Output: false
    } }
```

Time and Space Complexity:

- **Time Complexity:** `O(n)` ○ **Space Complexity:** `O(1)`

Example:

`arr` = {1, 2, 3, 3} can be split because 1 + 2 + 3 = 3.

5. Maximum Sum of Subarray of Size K

Algorithm (Sliding Window):

16. Calculate the sum of the first `K` elements.
17. Slide the window one element at a time, subtracting the first element of the previous window and adding the next element.
18. Keep track of the maximum sum.

Java Program:

Java

```
class MaxSubarraySumK {

    static int maxSubarraySumK(int[] arr, int k) {
if (arr.length < k) {              return -1; //
Invalid input
        }
        int maxSum = 0;          for
(int i = 0; i < k; i++) {
maxSum += arr[i];
        }
        int windowSum = maxSum;          for (int i = k;
i < arr.length; i++) {             windowSum = windowSum
- arr[i - k] + arr[i];             maxSum =
Math.max(maxSum, windowSum);
        }
        return maxSum;
    }      public static void main(String[] args)
{         int[] arr = {1, 4, 2, 10, 2, 3, 1, 0,
20};
        int k = 4;
        System.out.println("Maximum subarray sum of size " + k + ": " +
maxSubarraySumK(arr, k)); // Output: 24
    } }
```

Time and Space Complexity: o

- **Time Complexity:** `O(n)`

- **Space Complexity:** `O(1)`

Example:

`arr` = {1, 4, 2, 10, 2, 3, 1, 0, 20}, `k` = 4. The maximum subarray sum is 10 + 2 + 3 + 1 = 16. (My original code had a slight error, now corrected to properly reflect the sliding window concept and find the max sum).

6. Longest Substring Without Repeating Characters

Algorithm (Sliding Window + HashSet):

21. Use a sliding window and a HashSet to keep track of characters in the current window.
22. Expand the window to the right.
23. If a character is already in the HashSet, shrink the window from the left until the repeating character is removed.
24. Update the maximum length.

Java Program:

Java

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

class LongestSubstringUnique {

    static int longestUniqueSubstringLength(String s) {
int left = 0;         int right = 0;         int
maxLength = 0;
        Set<Character> charSet = new HashSet<>();
         while (right < s.length()) {
char c = s.charAt(right);
if (charSet.add(c)) {
                maxLength = Math.max(maxLength, right - left + 1);
right++;          } else {
                char leftChar = s.charAt(left);
charSet.remove(leftChar);                 left++;
            }
}
        return maxLength;
    }
    public static void main(String[] args) {
        String s = "abcabcbb";
        System.out.println("Length of longest unique substring: " +
longestUniqueSubstringLength(s)); // Output: 3 ("abc")
    } }
```

Time and Space Complexity:

- **Time Complexity:** `O(n)` - **Space Complexity:** `O(min(n, m))` where *m* is the size of the character set.

Example:

For "abcabcbb", the longest substring without repeating characters is "abc", with length 3.

7. Sliding Window Technique

The sliding window technique is an algorithmic pattern used to solve problems involving contiguous subarrays or substrings. It reduces time complexity by avoiding unnecessary recalculations.

Use in String Problems: ○ Finding substrings that meet certain criteria (e.g., longest substring without repeating characters). ○ Counting occurrences of a pattern in a string.

8. Longest Palindromic Substring

Algorithm (Dynamic Programming):

29. Create a 2D boolean table `dp` where `dp[i][j]` is true if the substring from `i` to `j` is a palindrome.
30. Initialize single-character palindromes (`dp[i][i] = true`).
31. Check for two-character palindromes (`dp[i][i+1]`).
32. Iteratively check for longer palindromes.
33. Keep track of the start and end indices of the longest palindrome.

Java Program:

Java

```
class LongestPalindrome {

    static String longestPalindrome(String s) {
int n = s.length();
        boolean[][] dp = new boolean[n][n];
        int start = 0;
int end = 0;

        // Base cases: single characters are palindromes
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;
end = i + 1;
            }
        }

        // Check for palindromes of length 3 or greater
for (int len = 3; len <= n; len++) {             for
(int i = 0; i <= n - len; i++) {                 int j
= i + len - 1;
                if (s.charAt(i) == s.charAt(j) && dp[i + 1][j - 1]) {
dp[i][j] = true;                      start = i;
end = j;
                }
            }
        }
        return s.substring(start, end + 1);
    }
    public static void main(String[] args) {
        String s = "babad";
        System.out.println("Longest palindromic substring: " +
longestPalindrome(s)); // Output: "bab" or "aba"
    } }
```

Time and Space Complexity:

- **Time Complexity:** `O(n^2)` ○ **Space Complexity:** `O(n^2)`

Example:

For "babad", "bab" or "aba" are the longest palindromic substrings.

9. Longest Common Prefix

Algorithm:

36. If the array of strings is empty, return "".
37. Take the first string as the prefix.
38. Iterate through the rest of the strings.
39. For each string, compare it with the current prefix.
40. Shorten the prefix if it's not a prefix of the current string.

Java Program:

Java

```
class LongestCommonPrefix {
```

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

        String prefix = strs[0];         for (int i = 1; i <
strs.length; i++) {             while (strs[i].indexOf(prefix) !=
0) {                 prefix = prefix.substring(0, prefix.length()
- 1);                 if (prefix.isEmpty()) {
return "";
                }
            }
}
        return prefix;
    }      public static void main(String[] args) {
String[] strs = {"flower", "flow", "flight"};
System.out.println("Longest common prefix: " +
longestCommonPrefix(strs)); // Output: "fl"
    } }
```

Time and Space Complexity:

- **Time Complexity:** `O(n * m)`, where *n* is the number of strings and *m* is the length of the shortest string.
- **Space Complexity:** `O(1)` [cite: 13, 14]

Example:

For ["flower", "flow", "flight"], the longest common prefix is "fl".

10. Generate All Permutations of a String

Algorithm (Backtracking):

43. Base case: If the string has only one character, print it.
44. For each character in the string:
    - Fix the character.
    - Recursively generate permutations of the remaining characters.

Java Program:

Java

```
class StringPermutations {

    static void permute(String s, int l, int r) {
        if (l == r) {
            System.out.println(s);
        } else {
            for (int i = l; i <= r; i++) {
s = swap(s, l, i);
permute(s, l + 1, r);
```

```
                s = swap(s, l, i); // Backtrack (restore original
string)             }
        }
    }      static String swap(String s, int i,
int j) {         char[] charArray =
s.toCharArray();         char temp =
charArray[i];         charArray[i] =
charArray[j];         charArray[j] = temp;
        return String.valueOf(charArray);
    }
    public static void main(String[] args) {
        String s = "abc";
        System.out.println("Permutations of " + s + ":");
permute(s, 0, s.length() - 1);
    } }
```

Time and Space Complexity:

- **Time Complexity:** `O(n!)` ○ **Space Complexity:**
  `O(n)` (for the recursion depth)

Example:

For "abc", the permutations are "abc", "acb", "bac", "bca", "cab", "cba".

11. Find Two Numbers in a Sorted Array that Add Up to a Target

Algorithm (Two Pointers):

47. Initialize two pointers, `left` at the beginning and `right` at the end of the array.
48. While `left < right`:
    - If `arr[left] + arr[right]` equals the target, return the indices. ▪ If the sum is less than the target, increment `left`.
    - If the sum is greater than the target, decrement `right`.

Java Program:

Java

```
class TwoSumSorted {

    static int[] findTwoSum(int[] arr, int target) {
int left = 0;
        int right = arr.length - 1;

        while (left < right) {
int sum = arr[left] + arr[right];
if (sum == target) {
```

```
                return new int[]{left, right};
} else if (sum < target) {
left++;             } else {
right--;
            }         }
return new int[]{-1
```

You are absolutely right! I apologize for the abrupt stop. Let's continue with the remaining problems. I'll keep the same format: Algorithm, Java Code, Time & Space Complexity, and Example.

12. Rearrange Numbers into the Lexicographically Next Greater Permutation

Algorithm:

49. Find the largest index `i` such that `arr[i] < arr[i+1]`.
50. Find the largest index `j > i` such that `arr[j] > arr[i]`.
51. Swap `arr[i]` and `arr[j]`.
52. Reverse the subarray from `arr[i+1]` to the end.

Java Program:

Java

```
import java.util.Arrays;

class NextPermutation {

    static void nextPermutation(int[] arr) {
int i = arr.length - 2;
        while (i >= 0 && arr[i] >= arr[i + 1]) {
i--;         }         if (i >= 0) {
int j = arr.length - 1;             while (j >=
0 && arr[j] <= arr[i]) {                j--;
}
            swap(arr, i, j);
        }
        reverse(arr, i + 1);
    }
```

```
    static void swap(int[] arr, int i, int j) {
int temp = arr[i];         arr[i] = arr[j];
arr[j] = temp;
    }
    static void reverse(int[] arr, int start) {
int end = arr.length - 1;         while (start
< end) {             swap(arr, start, end);
start++;             end--;
        }
    }
    public static void main(String[] args) {
int[] arr = {1, 2, 3};
nextPermutation(arr);
        System.out.println(Arrays.toString(arr)); // Output: [1, 3, 2]
} }
```

Time and Space Complexity:

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

Example:

For `arr` = [1, 2, 3], the next greater permutation is [1, 3, 2].

13. Merge Two Sorted Linked Lists

Algorithm:

55. Create a dummy head node.
56. Use a tail pointer to track the end of the merged list.
57. Iterate through both lists, comparing the values of the current nodes.
58. Add the smaller node to the merged list and move the corresponding pointer.
59. If one list is exhausted, append the remaining nodes of the other list.

Java (requires ListNode definition):

Java

```
// Definition for singly-linked list.
class ListNode {
int val;
ListNode next;

    ListNode(int val) {
this.val = val;
    }
}  class
MergeSortedLists {
     static ListNode mergeTwoLists(ListNode l1, ListNode l2)
{
        ListNode dummy = new ListNode(0);
        ListNode tail = dummy;

        while (l1 != null && l2 != null) {
if (l1.val <= l2.val) {
tail.next = l1;                 l1 =
l1.next;             } else {
tail.next = l2;                 l2 =
l2.next;
            }
            tail = tail.next;
        }
         if (l1 != null) {
tail.next = l1;
        } else {
            tail.next = l2;
        }
        return dummy.next;
    }

    // ... (main method to create lists and test) }
```


Time and Space Complexity:

- **Time Complexity:** O(n + m), where n and m are the lengths of the lists.
- **Space Complexity:** O(1) (iterative approach) or O(n+m) for recursive (due to call stack)

Example:

List 1: 1 -> 2 -> 4, List 2: 1 -> 3 -> 4. Merged: 1 -> 1 -> 2 -> 3 -> 4 -> 4

14. Find the Median of Two Sorted Arrays using Binary Search

Algorithm:

62. Ensure `arr1` is the smaller array. If not, swap them.
63. Perform binary search on the smaller array.
64. Calculate the partition points in both arrays.
65. Check if the elements around the partitions satisfy the median condition.
66. Adjust the binary search range until the correct partition is found.

Java Program:

Java

```
class MedianSortedArrays {

    static double findMedianSortedArrays(int[] arr1, int[] arr2) {
if (arr1.length > arr2.length) {
            return findMedianSortedArrays(arr2, arr1);
        }         int x =
arr1.length;         int y =
arr2.length;         int low
= 0;         int high = x;

        while (low <= high) {
            int partitionX = (low + high) / 2;
            int partitionY = (x + y + 1) / 2 - partitionX;
             int maxLeftX = (partitionX == 0) ? Integer.MIN_VALUE
:
arr1[partitionX - 1];             int minRightX = (partitionX == x)
? Integer.MAX_VALUE :
arr1[partitionX];
             int maxLeftY = (partitionY == 0) ? Integer.MIN_VALUE
:
arr2[partitionY - 1];             int minRightY = (partitionY == y)
? Integer.MAX_VALUE : arr2[partitionY];

            if (maxLeftX <= minRightY && maxLeftY <= minRightX) {
if ((x + y) % 2 == 0) {
                    return (double) (Math.max(maxLeftX, maxLeftY) +
Math.min(minRightX, minRightY)) / 2;
```

```
                } else {
                    return (double) Math.max(maxLeftX, maxLeftY);
                }
            } else if (maxLeftX > minRightY) {
high = partitionX - 1;
            } else {
                low = partitionX + 1;
            }
        }
        throw new IllegalArgumentException("Arrays are not sorted.");
    }
    public static void main(String[] args) {
int[] arr1 = {1, 3};         int[] arr2 =
{2};
        System.out.println("Median: " + findMedianSortedArrays(arr1,
arr2)); // Output: 2.0
    } }
```

Time and Space Complexity:

- **Time Complexity:** O(log(min(n, m)))
- **Space Complexity:** O(1)

Example:

`arr1` = [1, 3], `arr2` = [2]. The median is 2.0

15. Find the K-th Smallest Element in a Sorted Matrix

Algorithm (Binary Search):

69. The idea is to use binary search on the range of possible values (min to max of the matrix).
70. For each `mid` value, count how many elements in the matrix are less than or equal to `mid`.
71. Adjust the binary search range based on the count.

Java Program:

Java

```
class KthSmallestInMatrix {

    static int kthSmallest(int[][] matrix, int k) {
int n = matrix.length;         int low =
matrix[0][0];         int high = matrix[n - 1][n -
1];

        while (low <= high) {
            int mid = low + (high - low) / 2;
int count = countLessOrEqual(matrix, mid);
if (count < k) {                 low = mid + 1;
} else {
                high = mid - 1;
            }
}         return
low;
    }

    static int countLessOrEqual(int[][] matrix, int target) {
int count = 0;         int n = matrix.length;         int i =
n - 1;         int j = 0;         while (i >= 0 && j < n) {
if (matrix[i][j] <= target) {                 count += i + 1;
j++;             } else {                 i--;
            }
}
        return count;
    }
    public static void main(String[] args) {
        int[][] matrix = {{1, 5, 9}, {10, 11, 13}, {12, 13, 15}};
int k = 8;
        System.out.println("Kth smallest: " + kthSmallest(matrix, k));
// Output: 13
    } }
```


Time and Space Complexity:

- **Time Complexity:** O(n log(max - min)), where *n* is the number of rows/columns and max/min are the matrix's boundaries. ○ **Space Complexity:** O(1)

Example:

`matrix` = [[1, 5, 9], [10, 11, 13], [12, 13, 15]], `k` = 8. The 8th smallest element is 13.

16. Find the Majority Element (appears more than n/2 times)

Algorithm (Boyer-Moore Voting Algorithm):

74. Initialize a `candidate` and a `count`.
75. Iterate through the array.
76. If `count` is 0, set the current element as the `candidate`.
77. If the current element is equal to the `candidate`, increment `count`; otherwise, decrement `count`.
78. The final `candidate` is the majority element (if it exists).

Java Program:

Java

```
class MajorityElement {

    static int findMajorityElement(int[] arr) {
int candidate = 0;          int count = 0;
         for (int num : arr) {
if (count == 0) {
candidate = num;
            }
            if (num == candidate) {
count++;              } else {
count--;
            }
}
        return candidate;
    }
    public static void main(String[] args) {
int[] arr = {3, 2, 3};
        System.out.println("Majority element: " +
findMajorityElement(arr)); // Output: 3
    } }
```

Time and Space Complexity:

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

Example:

`arr` = [3, 2, 3]. The majority element is 3.

17. Calculate Trapped Water Between Histogram Bars

Algorithm:

81. Find the left max and right max for each bar.
82. The water trapped above a bar is `min(leftMax, rightMax) - barHeight`.
83. Sum the trapped water for all bars.

Java Program:

Java

```
class TrappedWater {

    static int trappedWater(int[] height) {
int n = height.length;         if (n == 0)
return 0;
         int[] leftMax = new int[n];
int[] rightMax = new int[n];
         leftMax[0] = height[0];
for (int i = 1; i < n; i++) {
            leftMax[i] = Math.max(leftMax[i - 1], height[i]);
        }          rightMax[n - 1] =
height[n - 1];          for (int i = n - 2;
i >= 0; i--) {
            rightMax[i] = Math.max(rightMax[i + 1], height[i]);
}          int trappedWater = 0;          for (int i = 0; i < n;
i++) {
            trappedWater += Math.max(0, Math.min(leftMax[i],
rightMax[i]) - height[i]);
        }          return
trappedWater;
```

```
    }
    public static void main(String[] args) {         int[]
height = {0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1};
System.out.println("Trapped water: " +
trappedWater(height));
// Output: 6
    } }
```

Time and Space Complexity:

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

Example:

`height` = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]. The trapped water is 6.

18. Find the Maximum XOR of Two Numbers in an Array

Algorithm (using Trie):

86. Create a Trie to store the binary representation of the numbers.
87. For each number, find the maximum XOR by traversing the Trie to find the opposite bits.

Java Program:

Java

```
class MaxXOR {

    static class TrieNode {
        TrieNode[] children = new TrieNode[2];
    }      static void insert(TrieNode root, int
num) {          for (int i = 31; i >= 0; i--) {
```

```
int bit = (num >> i) & 1;             if
(root.children[bit] == null) {
root.children[bit] = new TrieNode();
            }
            root = root.children[bit];
        }
    }      static int findMaxXOR(int[]
nums) {         TrieNode root = new
TrieNode();         for (int num :
nums) {             insert(root, num);
        }          int maxXor = 0;         for (int num
: nums) {             TrieNode node = root;
int currentXor = 0;             for (int i = 31; i >=
0; i--) {                 int bit = (num >> i) & 1;
int toggleBit = 1 - bit;                 if
(node.children[toggleBit] != null) {
currentXor |= (1 << i);                     node =
node.children[toggleBit];
                } else {
                    node = node.children[bit];
                }
}
            maxXor = Math.max(maxXor, currentXor);
        }
        return maxXor;
    }      public static void main(String[]
args) {         int[] nums = {3, 10, 5, 25,
2, 8};
        System.out.println("Max XOR: " + findMaxXOR(nums)); // Output:
28 (5 ^ 25)
    } }
```

Time and Space Complexity:

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

Example:

`nums` = [3, 10, 5, 25, 2, 8]. The maximum XOR is 28 (5 ^ 25).

19. Find the Maximum Product Subarray

Algorithm:

90. Keep track of the maximum and minimum product seen so far.
91. For each element, update the maximum and minimum products.
92. The maximum product subarray can end at the current element or at a previous element.

Java Program:

Java

```
class MaxProductSubarray {
     static int maxProduct(int[] nums)
{          if (nums.length == 0) return
0;
          int maxSoFar =
nums[0];          int minSoFar =
nums[0];          int result =
maxSoFar;

        for (int i = 1; i < nums.length; i++) {
int curr = nums[i];
            int tempMax = Math.max(curr, Math.max(maxSoFar * curr,
minSoFar * curr));
            minSoFar = Math.min(curr, Math.min(maxSoFar * curr,
minSoFar * curr));            maxSoFar = tempMax;
            result = Math.max(result, maxSoFar);
        }
        return result;
    }
    public static void main(String[] args) {
int[] nums = {2, 3, -2, 4};
        System.out.println("Max product: " + maxProduct(nums)); //
Output: 6
    } }
```


Time and Space Complexity:

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

Example:

`nums` = [2, 3, -2, 4]. The maximum product is 6 (2 * 3).

20. Count All Numbers with Unique Digits

Algorithm (Dynamic Programming / Combinatorics):

95. Base case: For n = 0, there is 1 number (0).
96. For n = 1, there are 10 numbers (0-9).
97. For n > 1, the first digit has 9 choices (1-9), the second has 9 (0-9 except the first), the third has 8, and so on.
98. Use a loop to calculate the count for each n up to the given number of digits.

Java Program:

Java

```
class CountUniqueDigits {

    static int countNumbersWithUniqueDigits(int n) {
if (n == 0) return 1;          if (n == 1) return 10;
         int count = 10;
int uniqueDigits = 9;
int availableDigits = 9;

        for (int i = 2; i <= n && i <= 10; i++) {
uniqueDigits = uniqueDigits * availableDigits;
count += uniqueDigits;              availableDigits--;
        }
        return count;
    }       public static void main(String[]
args) {
        System.out.println(countNumbersWithUniqueDigits(2)); // Output:
91
    } }
```

Time and Space Complexity:

- **Time Complexity:** O(n) (where n is the number of digits)
- **Space Complexity:** O(1)

Example:

For n = 2, the numbers with unique digits are [0, 1, 2, ..., 9, 10, 12, 13, ..., 98]. The count is 91.

21. Count the Number of 1s in Binary Representation from 0 to n

Algorithm (Dynamic Programming):

101. Create an array `dp` of size n+1. `dp[i]` stores the number of 1s in the binary representation of i.
102. `dp[0] = 0`.
103. For each number i from 1 to n, `dp[i] = dp[i / 2] + (i % 2)`.

Java Program:

Java

```
import java.util.Arrays;

class CountBits {

    static int[] countBits(int n) {
        int[] dp = new int[n + 1];
for (int i = 1; i <= n; i++) {
dp[i] = dp[i >> 1] + (i & 1);
        }
return dp;
    }      public static void main(String[]
args) {
        System.out.println(Arrays.toString(countBits(5))); // Output:
[0, 1, 1, 2, 1, 2]
    } }
```

Time and Space Complexity:

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

Example:

For n = 5, the counts are [0, 1, 1, 2, 1, 2].

22. Check if a Number is a Power of Two (Bit Manipulation)

Algorithm:

106. A number is a power of two if it has only one bit set to 1 in its binary representation.
107. Use the bitwise AND operator: `n & (n - 1)` will be 0 if n is a power of two (and n is not 0).

Java Program:

Java

```
class PowerOfTwo {
     static boolean isPowerOfTwo(int n) {
return (n > 0) && ((n & (n - 1)) == 0);
    }
    public static void main(String[] args) {
        System.out.println(isPowerOfTwo(16)); // Output: true
        System.out.println(isPowerOfTwo(17)); // Output: false
    } }
```

Time and Space Complexity:

- **Time Complexity:** O(1) ○
  Space Complexity: O(1)

Example:

16 (10000 in binary) is a power of two, but 17 (10001) is not.

23. Find the Maximum XOR of Two Numbers in an Array

Algorithm: 110. **Trie**

Construction:

- Create a Trie data structure to store the binary representations of the numbers in the array.
- Each node in the Trie will have two children, one for '0' and one for '1'.
- Insert the binary representation of each number from the array into the Trie. We'll typically store the most significant bit at the root of the Trie.

111. **Finding Maximum XOR:**

- Iterate through the array again.
- For each number in the array:
- Start from the root of the Trie and try to find the number in the Trie that will give the maximum XOR value. To maximize the XOR, for each bit of the current number, we try to find a number in the Trie with the opposite bit.
- Calculate the XOR value with the number found in the Trie.
- Keep track of the maximum XOR value found so far. 112. Return the maximum XOR value.

Java Program:

Java

```
class MaxXOR {

    // Trie Node structure
static class TrieNode {
        TrieNode[] children = new TrieNode[2];

        TrieNode() {
children[0] = null;
children[1] = null;
        }
    }
```

```
    // Inserts a number into the Trie
    static void insert(TrieNode root, int num) {
TrieNode curr = root;
        for (int i = 31; i >= 0; i--) { // Assuming 32-bit integers
int bit = (num >> i) & 1;  // Get the i-th bit             if
(curr.children[bit] == null) {                 curr.children[bit] =
new TrieNode();
            }
            curr = curr.children[bit];
        }
    }

    // Finds the maximum XOR for a number in the Trie
static int findMaxXOR(TrieNode root, int num) {
        TrieNode curr = root;
int maxXor = 0;
        for (int i = 31; i >= 0; i--) {
int bit = (num >> i) & 1;
            int toggleBit = 1 - bit; // Look for the opposite bit to
maximize XOR
            if (curr.children[toggleBit] != null) {
maxXor |= (1 << i);  // Add 2^i to maxXor
curr = curr.children[toggleBit];
            } else {
                curr = curr.children[bit];
            }
}
        return maxXor;
    }

    // Main function to find the maximum XOR of two numbers in an array
    static int findMaxXOR(int[] nums) {
TrieNode root = new TrieNode();         for
(int num : nums) {             insert(root,
num);
        }          int maxXor
= 0;         for (int num :
nums) {
            maxXor = Math.max(maxXor, findMaxXOR(root, num));
        }
return maxXor;     }
public static void
main(String[] args) {
int[] nums = {3, 10,
5, 25, 2, 8};
        System.out.println("Maximum XOR: " + findMaxXOR(nums)); //
Output: 28 (5 ^ 25)
    } }
```

Time and Space Complexity:

- **Time Complexity:** O(n), where n is the number of elements in the array. The `insert` and `findMaxXOR` operations take O(1) time on average (O(32) which is constant), and we iterate through the array a couple of times. - **Space Complexity:** O(n), in the worst case, the Trie can store all the bits of all the numbers in the array.

24. Explain the Concept of Bit Manipulation

Bit manipulation involves performing operations directly on the bits of a number. It's often used for:

- **Efficiency:** Bitwise operations are generally faster than arithmetic operations.
- **Space Optimization:** Bits can be used to store multiple boolean values. - **Low-Level Programming:** Interacting with hardware or system-level functions.

Common bitwise operators in Java:

- `&` (AND) - `|` (OR) - `^` (XOR) - `~` (NOT)
- `<<` (Left shift) - `>>` (Right shift) - `>>>` (Unsigned right shift)

25. Find the Next Greater Element for Each Element in an Array

Algorithm (Stack):

125. Use a stack to store indices of elements.
126. Iterate through the array.

127. While the stack is not empty and the current element is greater than the element at the top of the stack, pop the stack and set the next greater element for the popped element.
128. Push the current element's index onto the stack.
129. Elements remaining in the stack have no next greater element (set to -1).

Java Program:

Java

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

class NextGreaterElement {
     static int[] nextGreaterElement(int[] arr)
{         int[] result = new int[arr.length];
        Arrays.fill(result, -1);
        Stack<Integer> stack = new Stack<>();

        for (int i = 0; i < arr.length; i++) {
            while (!stack.isEmpty() && arr[i] > arr[stack.peek()]) {
int index = stack.pop();                result[index] = arr[i];
            }
            stack.push(i);
        }
        return result;
    }
    public static void main(String[] args) {
int[] arr = {1, 3, 2, 4};
        System.out.println(Arrays.toString(nextGreaterElement(arr)));
// Output: [3, 4, 4, -1]
    } }
```


Time and Space Complexity:

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

Example:

For `arr` = [1, 3, 2, 4], the next greater elements are [3, 4, 4, -1].

26. Remove the N-th Node from the End of a Singly Linked List

Algorithm (Two Pointers):

132. Use two pointers, `fast` and `slow`. 133.

Move `fast` n nodes ahead.

134. Move both `fast` and `slow` until `fast` reaches the end.

135. `slow` will be pointing to the node before the one to be deleted.

136. Adjust the pointers to remove the node.

Java (requires ListNode definition):

Java

```
// (ListNode definition from problem 13)
 class RemoveNthFromEnd
{
     static ListNode removeNthFromEnd(ListNode head, int n)
{
        ListNode dummy = new ListNode(0);
dummy.next = head;          ListNode fast
= dummy;
        ListNode slow = dummy;

        // Move fast 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;
    }

    // ... (main method to create lists and test) }
```

Time and Space Complexity:

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

Example:

List: 1 -> 2 -> 3 -> 4 -> 5, n = 2. Result: 1 -> 2 -> 3 -> 5

27. Find the Node Where Two Singly Linked Lists Intersect

Algorithm (Two Pointers):

139. If either list is null, return null.
140. Initialize two pointers, `a` and `b`, at the heads of the lists.
141. Traverse both lists. When a pointer reaches the end, redirect it to the head of the other list.
142. If the lists intersect, the pointers will eventually meet at the intersection node.

Java (requires ListNode definition):

Java

```
// (ListNode definition from problem 13)

class IntersectionOfLists {

    static ListNode getIntersectionNode(ListNode headA, ListNode headB)
{
        if (headA == null || headB == null) return null;

        ListNode a = headA;
        ListNode b = headB;
```

```
        while (a != b) {
            a = (a == null) ? headB : a.next;
b = (b == null) ? headA : b.next;
        }
        return a;
    }

    // ... (main method to create lists and test) }
```

Time and Space Complexity:

- **Time Complexity:** O(m + n)
- **Space Complexity:** O(1)

Example:

List A: a1 -> a2 -> c1 -> c2 -> c3, List B: b1 -> b2 -> b3 -> c1 -> c2 -> c3. Intersection at c1.

28. Implement Two Stacks in a Single Array

Algorithm:

145. Divide the array into two halves.
146. Use one half for the first stack and the other half for the second stack.
147. Maintain top pointers for both stacks.

Java Program:

Java

```
class TwoStacks {
int[] arr;
int top1;     int
top2;     int
size;
```

```
    TwoStacks(int n) {
size = n;         arr =
new int[n];         top1
= -1;         top2 = n;
    }      void push1(int x) {
if (top1 < top2 - 1) {
top1++;             arr[top1]
= x;
        } else {
            System.out.println("Stack Overflow");
        }
    }      void push2(int x) {
if (top1 < top2 - 1) {
top2--;             arr[top2]
= x;
        } else {
            System.out.println("Stack Overflow");
        }
    }      int pop1() {
if (top1 >= 0) {
int x = arr[top1];
top1--;             return x;
        } else {
            System.out.println("Stack Underflow");
return -1;
        }
    }      int pop2() {
if (top2 < size) {
int x = arr[top2];
top2++;             return x;
        } else {
            System.out.println("Stack Underflow");
return -1;
        }
    }      public static void main(String[]
args) {         TwoStacks ts = new
TwoStacks(6);         ts.push1(5);
ts.push2(10);         ts.push2(15);
ts.push1(11);         ts.push2(7);
        System.out.println("Popped element from stack1 is : " +
ts.pop1());
        ts.push2(40); //will print overflow
        System.out.println("Popped element from stack2 is : " +
ts.pop2());
    } }
```


Time and Space Complexities:

- **Time Complexity:** O(1) for push and pop operations.
- **Space Complexity:** O(n) for the array.

29. Check if an Integer is a Palindrome (Without Converting to String)

Algorithm:

150. Handle negative numbers and numbers ending in 0.
151. Reverse the second half of the number.
152. Compare the first half with the reversed second half.

Java Program:

Java

```
class PalindromeInteger {
     static boolean isPalindrome(int x) {
if (x < 0 || (x % 10 == 0 && x != 0)) {
return false;
        }
         int reversedHalf = 0;
while (x > reversedHalf) {
            reversedHalf = reversedHalf * 10 + x % 10;
x /= 10;
        }
        return x == reversedHalf || x == reversedHalf / 10;
    }
    public static void main(String[] args) {
        System.out.println(isPalindrome(121));  // Output: true
        System.out.println(isPalindrome(12321)); // Output: true
        System.out.println(isPalindrome(123));   // Output: false
    } }
```


Time and Space Complexity:

- **Time Complexity:** O(log10(n))
  Space Complexity: O(1)

Example:

121 is a palindrome, but 123 is not.

30. Explain the Concept of Linked Lists

A linked list is a linear data structure where elements are stored in nodes. Each node contains:

- **Data:** The value being stored.
- **Pointer (Next):** A reference to the next node in the sequence.

Applications:

- Dynamic memory allocation
- Implementing stacks, queues, and graphs
- Polynomial representation
- Symbol tables

Alright, let's continue with the next 10 questions.

31. Use a Deque to Find the Maximum in Every Sliding Window of Size K

Algorithm:

161. Create a deque to store indices of array elements.
162. Iterate through the array.
163. For each element:
    - Remove elements from the deque that are out of the current window.
    - Remove elements from the back of the deque that are smaller than the current element (because they can't be the maximum).
    - Add the current element's index to the deque.
    - The front of the deque always contains the index of the maximum element in the current window.
164. Store the maximum for each window.

Java Program:

Java

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

class MaxSlidingWindow {

    static ArrayList<Integer> maxSlidingWindow(int[] nums, int k) {
        ArrayList<Integer> result = new ArrayList<>();
        Deque<Integer> deque = new LinkedList<>();

        for (int i = 0; i < nums.length; i++) {
// Remove elements out of the window
            while (!deque.isEmpty() && deque.peekFirst() <= i - k) {
deque.pollFirst();
            }

            // Remove smaller elements from the back
while (!deque.isEmpty() && nums[deque.peekLast()] <= nums[i]) {
                deque.pollLast();
            }

deque.offerLast(i);

            if (i >= k - 1) {
                result.add(nums[deque.peekFirst()]);
            }
}
        return result;
    }      public static void main(String[]
args) {         int[] nums = {1, 3, -1, -3, 5,
3, 6, 7};
        int k = 3;
        System.out.println(maxSlidingWindow(nums, k)); // Output: [3,
3, 5, 5, 6, 7]
    } }
```


Time and Space Complexity:

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

Example:

`nums` = [1, 3, -1, -3, 5, 3, 6, 7], `k` = 3. The maximums are [3, 3, 5, 5, 6, 7].

32. Find the Largest Rectangle that Can Be Formed in a Histogram

Algorithm (Stack):

167. Use a stack to store indices of bars.
168. Iterate through the histogram bars.

169. For each bar:
   - While the stack is not empty and the current bar is shorter than the bar at the top of the stack, pop the stack and calculate the area.
   - Push the current bar's index onto the stack.

170. After processing all bars, pop any remaining bars from the stack and calculate the area.

171. Keep track of the maximum area.

Java Program:

Java

```
import java.util.Stack;
 class LargestRectangleHistogram
{
     static int largestRectangleArea(int[] heights)
{         Stack<Integer> stack = new Stack<>();
int maxArea = 0;         int i = 0;
        while (i < heights.length || !stack.isEmpty()) {
            if (stack.isEmpty() || (i < heights.length &&
heights[stack.peek()] <= heights[i])) {
stack.push(i++);
            } else {
                int top = stack.pop();
                int width = stack.isEmpty() ? i : i - stack.peek() - 1;
int area = heights[top] * width;                maxArea =
Math.max(maxArea, area);
            }
}
        return maxArea;
    }      public static void main(String[]
args) {         int[] heights = {2, 1, 5, 6,
2, 3};
        System.out.println(largestRectangleArea(heights)); // Output:
10
    } }
```

Time and Space Complexity:

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

Example:

`heights` = [2, 1, 5, 6, 2, 3]. The largest rectangle area is 10.

33. Explain the Sliding Window Technique

(This is a duplicate of question 7 and I have already explained it.)

34. Solve the Problem of Finding the Subarray Sum Equal to K Using Hashing

Algorithm:

174. Use a HashMap to store the prefix sums and their counts.
175. Initialize `sum` = 0 and `count` = 0.
176. Iterate through the array, calculate the prefix sum.
177. If `sum - k` exists in the map, increment the `count`. 178. Update the map with the current `sum`.

Java Program:

Java

```
import java.util.HashMap;

class SubarraySumK {
     static int subarraySum(int[] nums, int k) {
HashMap<Integer, Integer> map = new HashMap<>();
map.put(0, 1); // Initialize for empty subarray
int sum = 0;          int count = 0;
          for (int num : nums)
{              sum += num;
              if (map.containsKey(sum - k)) {
count += map.get(sum - k);
              }
              map.put(sum, map.getOrDefault(sum, 0) + 1);
          }
          return count;
     }
     public static void main(String[] args) {
int[] nums = {1, 1, 1};          int k = 2;
          System.out.println(subarraySum(nums, k)); // Output: 2
     } }
```


Time and Space Complexity:

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

Example:

`nums` = [1, 1, 1], `k` = 2. There are 2 subarrays with a sum of 2.

35. Find the K-Most Frequent Elements in an Array Using a Priority Queue

Algorithm:

181. Count the frequency of each element using a HashMap.
182. Use a min-heap (PriorityQueue) of size k to store the elements with the highest frequency.
183. Iterate through the HashMap.
184. Add each element to the priority queue. If the queue size exceeds k, remove the element with the lowest frequency.
185. The priority queue will contain the k most frequent elements.

Java Program:

Java

```
import java.util.HashMap; import
java.util.PriorityQueue; import
java.util.List; import
java.util.ArrayList; import
java.util.Map;

class KMostFrequent {
     static List<Integer> topKFrequent(int[] nums, int k)
{
        HashMap<Integer, Integer> frequencyMap = new HashMap<>();
for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) +
1);
        }

        PriorityQueue<Integer> pq = new PriorityQueue<>((a, b) ->
frequencyMap.get(a) - frequencyMap.get(b));

        for (int num : frequencyMap.keySet()) {
pq.add(num);            if (pq.size() > k) {
pq.poll();
            }
        }

        List<Integer> topK = new ArrayList<>(pq);
return topK;
    }      public static void main(String[]
args) {         int[] nums = {1, 1, 1, 2, 2,
3};
        int k = 2;
        System.out.println(topKFrequent(nums, k)); // Output: [1, 2]
    } }
```


Time and Space Complexity:

- **Time Complexity:** O(n log k)
- **Space Complexity:** O(n)

Example:

`nums` = [1, 1, 1, 2, 2, 3], `k` = 2. The 2 most frequent elements are [1, 2].

36. Generate All Subsets of a Given Array Algorithm

(Backtracking):

188. Create a recursive function.
189. In each recursive call, we have two choices for each element: either include it in the current subset or exclude it.
190. Base case: When we reach the end of the array, add the current subset to the result.

Java Program:

Java

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

class AllSubsets {

    static void generateSubsets(int[] nums, int index, List<Integer>
currentSubset, List<List<Integer>> result) {
        result.add(new ArrayList<>(currentSubset)); // Add a copy of
the current subset

        for (int i = index; i < nums.length; i++) {
currentSubset.add(nums[i]);
            generateSubsets(nums, i + 1, currentSubset, result);
currentSubset.remove(currentSubset.size() - 1); //
Backtrack
        }
    }      static List<List<Integer>> subsets(int[] nums) {
List<List<Integer>> result = new ArrayList<>();
generateSubsets(nums, 0, new ArrayList<>(), result);
return result;
    }
    public static void main(String[] args) {
        int[] nums = {1, 2, 3};
        System.out.println(subsets(nums)); // Output: [[], [1], [2],
[1, 2], [3], [1, 3], [2, 3], [1, 2, 3]]
    } }
```

Time and Space Complexity:

- **Time Complexity:** O(2^n)
- **Space Complexity:** O(n) (for the recursion depth)

Example:

`nums` = [1, 2, 3]. The subsets are [], [1], [2], [1, 2], [3], [1, 3], [2, 3], [1, 2, 3]].

37. Find All Unique Combinations of Numbers that Sum to a Target

Algorithm (Backtracking):

193. Sort the candidates array (to handle duplicates).
194. Create a recursive function.
195. In each recursive call, try including each candidate number.
196. If the sum equals the target, add the combination to the result.
197. Backtrack to explore other combinations.
198. Skip duplicate numbers to generate unique combinations.

Java Program:

Java

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

class CombinationSum {
     static void findCombinations(int[] candidates, int target, int
index, List<Integer> currentCombination, List<List<Integer>> result) {
if (target == 0) {
            result.add(new ArrayList<>(currentCombination));
return;
        }          if
(target < 0) {
return;
        }
        for (int i = index; i < candidates.length; i++) {
            if (i > index && candidates[i] == candidates[i - 1]) { //
Skip duplicates                continue;
            }
            currentCombination.add(candidates[i]);
            findCombinations(candidates, target - candidates[i], i + 1,
currentCombination, result); // Allow reuse of numbers
            currentCombination.remove(currentCombination.size() - 1);
// Backtrack
        }
    }
    static List<List<Integer>> combinationSum(int[] candidates, int
target) {
        Arrays.sort(candidates);
        List<List<Integer>> result = new ArrayList<>();
findCombinations(candidates, target, 0, new ArrayList<>(), result);
        return result;
    }
```

```
    public static void main(String[] args) {
int[] candidates = {10, 1, 2, 7, 6, 1, 5};
int target = 8;
        System.out.println(combinationSum(candidates, target)); //
Output: [[1, 1, 6], [1, 2, 5], [1, 7], [2, 6]]
    } }
```

Time and Space Complexity:

- **Time Complexity:** Exponential (difficult to give a precise bound, but it's roughly O(2^n) in the worst case) ○ **Space Complexity:**
  O(n) (for the recursion depth)

Example:

`candidates` = [10, 1, 2, 7, 6, 1, 5], `target` = 8. The unique combinations are [1, 1, 6], [1, 2, 5], [1, 7], [2, 6].

38. Generate All Permutations of a Given Array

(This is similar to question 10, which was for strings. The algorithm is the same, just adapted for arrays.)

Java Program:

Java

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

class ArrayPermutations {

    static void permute(int[] nums, int l, int r, List<List<Integer>>
result) {
        if (l == r) {
            List<Integer> currentPermutation = new ArrayList<>();
for (int num : nums) {
currentPermutation.add(num);
            }
            result.add(currentPermutation);
        } else {
            for (int i = l; i <= r; i++) {
swap(nums, l, i);                 permute(nums,
l + 1, r, result);                 swap(nums, l,
i); // Backtrack
            }
        }
    }
```

```
    static void swap(int[] nums, int i, int j) {
int temp = nums[i];         nums[i] = nums[j];
nums[j] = temp;
    }
    static List<List<Integer>> permute(int[] nums) {
List<List<Integer>> result = new ArrayList<>();
permute(nums, 0, nums.length - 1, result);         return
result;
    }

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

Time and Space Complexity:

- **Time Complexity:** O(n!) ○ **Space Complexity:** O(n)

Example:

`nums` = [1, 2, 3]. The permutations are [1, 2, 3], [1, 3, 2], [2, 1, 3], [2, 3, 1], [3, 1, 2], [3, 2, 1].

39. Explain the Difference Between Subsets and Permutations

- **Subsets:** A subset is a selection of elements from a set, where the order of elements does not matter. ○ **Permutations:** A permutation is an arrangement of elements from a set, where the order of elements does matter.

Example:

For the set {1, 2, 3}:

- Subsets: {}, {1}, {2}, {3}, {1, 2}, {1, 3}, {2, 3}, {1, 2, 3}
- Permutations: {1, 2, 3}, {1, 3, 2}, {2, 1, 3}, {2, 3, 1}, {3, 1, 2}, {3, 2, 1}

40. Solve the Problem of Finding the Element with Maximum Frequency in an Array

Algorithm:

207. Use a HashMap to store the frequency of each element.
208. Iterate through the array and update the frequency in the HashMap.

209. Iterate through the HashMap to find the element with the maximum frequency.

Java Program:

Java

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

class MaxFrequencyElement {

    static int maxFrequencyElement(int[] arr) {
        HashMap<Integer, Integer> frequencyMap = new HashMap<>();
for (int num : arr) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) +
1);
        }
        int maxFrequency = 0;
        int maxElement = -1; // Or any default value
for (Map.Entry<Integer, Integer> entry :
frequencyMap.entrySet()) {
            if (entry.getValue() > maxFrequency) {
maxFrequency = entry.getValue();                maxElement
= entry.getKey();
            }
}
        return maxElement;
    }      public static void main(String[]
args) {          int[] arr = {1, 2, 2, 3, 2, 4,
4, 4, 4};
        System.out.println("Element with max frequency: " +
maxFrequencyElement(arr)); // Output: 4
    } }
```


Time and Space Complexity:

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

Example:

`arr` = {1, 2, 2, 3, 2, 4, 4, 4, 4}. The element with maximum frequency is 4.

41. Write a Program to Find the Maximum Subarray Sum Using Kadane's Algorithm

Algorithm (Kadane's Algorithm):

212. Initialize `maxSoFar` = 0 and `currentMax` = 0. 213. Iterate through the array.

214. For each element, update `currentMax` = max(`arr[i]`, `currentMax` + `arr[i]`).

215. Update `maxSoFar` = max(`maxSoFar`, `currentMax`).

216. Return `maxSoFar`.

Java Program:

Java

```
class KadanesAlgorithm {

    static int maxSubarraySum(int[] arr) {
int maxSoFar = 0;         int currentMax =
0;

        for (int num : arr) {
            currentMax = Math.max(num, currentMax + num);
maxSoFar = Math.max(maxSoFar, currentMax);
        }
        return maxSoFar;
    }      public static void main(String[] args) {
int[] arr = {-2, 1, -3, 4, -1, 2, 1, -5, 4};
System.out.println("Maximum subarray sum: " +
maxSubarraySum(arr)); // Output: 6
    } }
```

Time and Space Complexity:

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

Example:

`arr` = {-2, 1, -3, 4, -1, 2, 1, -5, 4}. The maximum subarray sum is 6.

42. Explain the Concept of Dynamic Programming and Its Use in Solving the Maximum Subarray Problem

Dynamic programming is a technique for solving problems by breaking them down into smaller overlapping subproblems, solving each subproblem only once, and storing the solutions to subproblems to avoid redundant computations.

In the maximum subarray problem, Kadane's algorithm uses dynamic programming principles. It calculates the maximum subarray sum ending at each position by using the maximum subarray sum ending at the previous position. This avoids recalculating sums for overlapping subarrays.

43. Solve the Problem of Finding the Top K Frequent Elements in an Array

Algorithm:

219. **Frequency Counting:**
   - Use a HashMap (or dictionary) to store the frequency of each element in the array. Iterate through the array and update the count of each element in the HashMap.

220. **Priority Queue (Min-Heap):**
   - Create a min-heap (priority queue) of size K. The priority queue will be ordered based on the frequency of the elements.
   - Iterate through the HashMap.
   - For each element, add it to the priority queue.
   - If the size of the priority queue becomes greater than K, remove the element with the lowest frequency from the queue (the root of the minheap). This ensures that the priority queue always contains the K elements with the highest frequencies.

221. **Result Extraction:**
   - After processing all elements, the priority queue will contain the top K frequent elements.
   - Extract the elements from the priority queue and return them as a list.

Java Program:

Java

```
import java.util.*;

class TopKFrequentElements {

    static List<Integer> topKFrequent(int[] nums, int k) {
        // 1. Frequency Counting
        HashMap<Integer, Integer> frequencyMap = new HashMap<>();
for (int num : nums) {
            frequencyMap.put(num, frequencyMap.getOrDefault(num, 0) +
1);
        }

        // 2. Priority Queue (Min-Heap)
        PriorityQueue<Integer> pq = new
PriorityQueue<>(Comparator.comparingInt(frequencyMap::get)); // Minheap
based on frequency

        for (int num : frequencyMap.keySet()) {
pq.add(num);             if (pq.size() > k) {
                pq.poll(); // Remove element with lowest frequency
            }
        }
```

```
        // 3. Result Extraction
        return new ArrayList<>(pq); // Convert priority queue to list
    }       public static void main(String[]
args) {          int[] nums = {1, 1, 1, 2, 2,
3};
        int k = 2;
        System.out.println(topKFrequent(nums, k)); // Output: [1, 2]
    } }
```

Time and Space Complexity:

- **Time Complexity:** O(N log K), where N is the number of elements in the input array.
  - Frequency counting takes O(N) time.
  - Adding elements to the priority queue and removing the smallest element takes O(log K) time, and we do this at most N times. **Space Complexity:** O(N), where N is the number of elements in the input array.
  - The HashMap can store at most N unique elements and their frequencies.
    - The priority queue stores at most K elements.

Example:

`nums` = [1, 1, 1, 2, 2, 3], `k` = 2

224. **Frequency Counting:**
   - frequencyMap = {1: 3, 2: 2, 3: 1} 225. **Priority Queue (Min-Heap):**
   - pq initially empty
   - add 1 (freq 3), pq = [1]
   - add 2 (freq 2), pq = [2, 1] (min-heap property maintained)
   - add 3 (freq 1), pq = [3, 1, 2]
   - pq.size() > k (2), remove 3, pq = [2, 1]
   - pq = [2, 1]

226. **Result Extraction:**
   - Result: [1, 2] (or [2, 1] - order may vary)

44. How to Find Two Numbers in an Array that Add Up to a Target Using Hashing

Algorithm:

227. Use a HashMap to store each number and its index. 228. Iterate through the array.
229. For each number, check if `target - number` exists in the HashMap. 230. If it exists, return the indices.

Java Program:

Java

```
import java.util.HashMap; import
java.util.Map; class TwoSumHash
{
     static 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);
        }
        return new int[] {-1, -1}; // No solution
    }       public static void main(String[]
args) {          int[] nums = {2, 7, 11, 15};
int target = 9;
        int[] result = twoSum(nums, target);
        System.out.println(result[0] + ", " + result[1]); // Output: 0,
1
    } }
```

Time and Space Complexity:

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

Example:

`nums` = {2, 7, 11, 15}, `target` = 9. The indices are 0 and 1.

45. Explain the Concept of Priority Queues and Their Applications in Algorithm Design

A priority queue is a data structure that orders elements based on their priority. Elements with higher priority are served before elements with lower priority.

Applications:

- Scheduling tasks
- Implementing Dijkstra's algorithm for finding the shortest path
- Huffman coding for data compression ○ Event-driven simulation

46. Write a Program to Find the Longest Palindromic Substring in a Given String Algorithm:

We'll use a dynamic programming approach to solve this problem.

237. **Initialization:**

- Create a 2D boolean table `dp` of size n x n, where n is the length of the string.
- `dp[i][j]` will be `true` if the substring from index `i` to `j` (inclusive) is a palindrome, and `false` otherwise.
- Initialize the diagonal elements of `dp` to `true` because single characters are palindromes (`dp[i][i] = true` for all i).
- Check for palindromes of length 2. If `s.charAt(i) == s.charAt(i + 1)`, then `dp[i][i + 1] = true`. 238.

Dynamic Programming:

- Iteratively check for palindromes of length 3 or greater.
- For each substring length `len` from 3 to n:
- For each starting index `i` from 0 to `n - len + 1`:
- Calculate the ending index `j = i + len - 1`.
- If `s.charAt(i) == s.charAt(j)` and `dp[i + 1][j - 1]` is `true` (meaning the substring inside is a palindrome), then `dp[i][j] = true`. 239.

Track Longest Palindrome:

- While filling the `dp` table, keep track of the start and end indices of the longest palindromic substring found so far.

240. **Return Result:**

- Extract the longest palindromic substring from the original string using the stored start and end indices.

Java Program:

Java

```
class LongestPalindromicSubstring {
     static String longestPalindrome(String s)
{         int n = s.length();
        boolean[][] dp = new boolean[n][n];
int start = 0;         int end = 0;

        // Base case 1: Single characters are palindromes
for (int i = 0; i < n; i++) {             dp[i][i] =
true;
        }

        // Base case 2: 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;
end = i + 1;
            }
        }

        // Check for palindromes of length 3 or greater
for (int len = 3; len <= n; len++) {             for
(int i = 0; i <= n - len; i++) {                 int j
= i + len - 1;
                if (s.charAt(i) == s.charAt(j) && dp[i + 1][j - 1]) {
dp[i][j] = true;                     start = i;
end = j;
                }
            }
        }
        return s.substring(start, end + 1);
    }
    public static void main(String[] args) {
        String s1 = "babad";
        String s2 = "cbbd";

        System.out.println("Longest palindrome in '" + s1 + "': " +
longestPalindrome(s1)); // Output: "bab" or "aba"
        System.out.println("Longest palindrome in '" + s2 + "': " +
longestPalindrome(s2)); // Output: "bb"
    } }
```

Time and Space Complexity:

- **Time Complexity:** O(n^2), where n is the length of the string. We fill an n x n table. ○ **Space Complexity:** O(n^2) for the `dp` table.

Example:

- `s1` = "babad"
    - The algorithm identifies "bab" (or "aba") as the longest palindromic substring. ○ `s2` = "cbbd"
    - The algorithm identifies "bb" as the longest palindromic substring.

47. Explain the Concept of Histogram Problems and Their Applications in Algorithm Design

Histogram problems involve analyzing a set of bars where the height of each bar represents a value.

Applications:

- Finding the largest rectangular area in a histogram
- Calculating trapped rainwater
- Image processing
- Data analysis and visualization

48. Solve the Problem of Finding the Next Permutation of a Given Array

Algorithm: 249. **Find the First Decreasing Element:**

- Start from the right and find the first element `nums[i]` such that `nums[i] < nums[i+1]`.
- If no such element exists, the array is in descending order, and its next permutation is the reversed array itself.

250. **Find the First Element Greater Than the Decreasing Element:**

- Find the first element `nums[j]` to the right of `nums[i]` such that `nums[j] > nums[i]`.

251. **Swap:**

- Swap `nums[i]` and `nums[j]`.

252. **Reverse the Right Subarray:**

- Reverse the subarray to the right of index `i` (from `i+1` to the end of the array).

Java Program:

Java

```
import java.util.Arrays;

class NextPermutation {

    static void nextPermutation(int[] nums) {
int i = nums.length - 2;
        while (i >= 0 && nums[i] >= nums[i + 1]) {
i--;          }          if (i >= 0) {
int j = nums.length - 1;              while (j >= 0
&& nums[j] <= nums[i]) {                  j--;
}
```

```
            swap(nums, i, j);
        }
        reverse(nums, i + 1);
    }
    static void swap(int[] nums, int i, int j) {
int temp = nums[i];         nums[i] = nums[j];
nums[j] = temp;
    }
    static void reverse(int[] nums, int start) {
int end = nums.length - 1;         while (start
< end) {             swap(nums, start, end);
start++;             end--;
        }
    }
    public static void main(String[] args) {
int[] nums1 = {1, 2, 3};
nextPermutation(nums1);
        System.out.println(Arrays.toString(nums1)); // Output: [1, 3,
2]             int[] nums2 = {3,
2, 1};
nextPermutation(nums2);
        System.out.println(Arrays.toString(nums2)); // Output: [1, 2,
3]             int[] nums3 = {1,
1, 5};
nextPermutation(nums3);
        System.out.println(Arrays.toString(nums3)); // Output: [1, 5,
1]
    } }
```

Time and Space Complexity:

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

Example:

- `nums1` = [1, 2, 3] -> [1, 3, 2]
- `nums2` = [3, 2, 1] -> [1, 2, 3]
- `nums3` = [1, 1, 5] -> [1, 5, 1]

49. How to Find the Intersection of Two Linked Lists

Algorithm:

258. **Handle Empty Lists:**

- If either list is null, there is no intersection, so return null. 259. **Initialize Pointers:**

- Initialize two pointers, `ptrA` to the head of list A and `ptrB` to the head of list B.

260. **Traverse and Redirect:**

- Traverse both lists simultaneously.
- When `ptrA` reaches the end of list A, redirect it to the head of list B.
- When `ptrB` reaches the end of list B, redirect it to the head of list A. 261. **Intersection or Null:**
- Continue this process until `ptrA` and `ptrB` become equal.
- If they meet, it means there is an intersection, and `ptrA` (or `ptrB`) points to the intersection node.
- If they both become null, it means the lists do not intersect.

Java (requires ListNode definition):

Java

```
// (Assuming ListNode definition is available as in previous responses)

class IntersectionOfTwoLinkedLists {

    static ListNode getIntersectionNode(ListNode headA, ListNode headB)
{
        if (headA == null || headB == null) {
return null;
        }

        ListNode ptrA = headA;
        ListNode ptrB = headB;

        while (ptrA != ptrB) {
            ptrA = (ptrA == null) ? headB : ptrA.next;
ptrB = (ptrB == null) ? headA : ptrB.next;
        }
        return ptrA;
    }

    // ... (main method to create lists and test) }
```

Time and Space Complexity:

- **Time Complexity:** O(m + n), where m and n are the lengths of the two lists.
- **Space Complexity:** O(1)

Example:

List A: a1 -> a2 -> c1 -> c2 -> c3
List B: b1 -> b2 -> b3 -> c1 -> c2 -> c3

Intersection at node c1.

50. Explain the Concept of Equilibrium Index and Its Applications in Array Problems

Concept:

- An equilibrium index of an array is an index such that the sum of elements at lower indices is equal to the sum of elements at higher indices.
- In simpler terms, the sum of the elements to the left of the equilibrium index is equal to the sum of the elements to its right.

Applications in Array Problems:

- **Load Balancing:** In distributed systems, finding an equilibrium point can help in balancing the load across different servers.
- **Game Theory:** It can be used to find fair division points in resource allocation problems.
- **Signal Processing:** Identifying equilibrium points in a signal can help in analyzing its symmetry.
- **Array Partitioning:** It's a fundamental concept in problems that involve partitioning an array based on sums.

Example:

`arr` = {-7, 1, 5, 2, -4, 3, 0}

The equilibrium index is 3 because:

-7 + 1 + 5 = -1

-4 + 3 + 0 = -1

Sources and related content