COMPETITIVE PROGRAMMING

Two Pointers Technique

COMPETITIVE PROGRAMMING

Introduction to Two Pointers

- The Two Pointers technique involves using two pointers to traverse an array or a list.
- These pointers can move in the same direction (usually used for searching or merging), or they can move in opposite directions (commonly used for finding pairs or subarrays with a certain property).

Basic Operations

- Initialization: Initialize two pointers at the start or end of the sequence, or at different positions within the sequence.
- Moving Pointers: Move the pointers towards each other or in the same direction based on the problem requirements.
- Comparing Pointers: Compare elements pointed to by the pointers and update the pointers accordingly.

COMPETITIVE PROGRAMMING

Example: Finding a Pair with a Given Sum

- Problem: Given a sorted array A (sorted in ascending order), having N integers, find if there exists any pair of elements (A[i], A[j]) such that their sum is equal to X.
- Solution:
  - We take two pointers, one representing the first element and other representing the last element of the array, and then we add the values kept at both the pointers.
  - If their sum is smaller than X then we shift the left pointer to right or if their sum is greater than X then we shift the right pointer to left, in order to get closer to the sum.
  - We keep moving the pointers until we get the sum as X.

COMPETITIVE PROGRAMMING

Example:Finding a Pair with a Given Sum

represents first pointer to the beginning of the array

represents second pointer to the end of the array

Return false if no pair with the target sum is found

```
bool findPairWithSum(int array[], int n, int targetSum) {
    int leftPtr = 0;
    int rightPtr = n - 1;

    while (leftPtr < rightPtr) {
        int currentSum = array[leftPtr] + array[rightPtr];

        if (currentSum == targetSum) {
            cout << "Pair found: " << array[leftPtr] << ", " << array[rightPtr] << endl;
            return true;
        } else if (currentSum < targetSum) {
            leftPtr++;
        } else {
            rightPtr--;
        }
    }

    cout << "No pair found with the given sum." << endl;
    return false;
}
```

Continue until the pointers meet or cross each other

Check if the current pair has the target sum

If sum of elements at current pointers is less, we move towards higher values

If sum of elements at current pointers is more, we move towards lower values

<COMPETITIVE PROGRAMMING

Sliding Window Technique

- The Sliding Window technique is used for efficiently processing arrays or sequences by maintaining a "window" of elements as you traverse the sequence.
- This window size can be fixed or variable.
- Can help optimize various algorithms of brute force approach from $O(n^2)$ or $O(n^3)$ to $O(n)$.
- The basic idea behind the technique is to transform two nested loops into a single loop.

COMPETITIVE PROGRAMMING

Example of Sliding Window Technique

Let's say that if you have an array like below:

[a, b, c, d, e, f, g, h

A sliding window of size 3 would run over it like below:

[a, b, c]
[b, c, d]
[c, d, e]
[d, e, f]
[e, f, g]
[f, g, h]

Identify Problems that can be solved using Sliding Window Technique

Below are some fundamental clues to identify such kind of problem:

- The problem will be based on an array, list or string type of data structure.
- It will ask to find sub-range in that array or string will have to give longest, shortest, or target values.
- Its concept is mainly based on ideas like the longest sequence or shortest sequence of something that satisfies a given condition perfectly.

COMPETITIVE PROGRAMMING

Basic Steps to Solve Sliding Window Problem

- Take hashmap or dictionary to count specific array input and uphold on increasing the window towards right using an outer loop.
- Take one inside a loop to reduce the window side by sliding towards the right. This loop will be very short.
- Store the current maximum or minimum window size or count based on the problem statement.

Example 1

Maximum Sum Subarray of Fixed Size

Problem: Given an array of integers and a fixed window size k, find the maximum sum of any contiguous subarray of size k.

Input: [3, 5, 2, 1, 7], k=2

Output: 8

Explanation: Here the subarray [1, 7] is the sum of the maximum sum.

COMPETITIVE PROGRAMMING

Maximum Sum Subarray of Fixed Size

```
int maxSumSubarray(const vector<int> arr, int k) {
    int maxSum = 0;
    int currentSum = 0;
    // Calculate the sum of the first window of size k
    for (int i = 0; i < k; i++) {
        currentSum += arr[i];
    }
    maxSum = currentSum;
    // Slide the window through the array
    for (int i = k; i < arr.size(); i++) {
    // Add the next element and remove the first element of the window
        currentSum += arr[i] - arr[i - k];
        maxSum = max(maxSum, currentSum);
    }
    return maxSum;
}
```

COMPETITIVE PROGRAMMING

Example 2:

Permutation in String

Problem: Given two strings s1 and s2, return true if s2 contains a permutation of s1, or false otherwise.

In other words, return true if one of s1's permutations is the substring of s2.

Input: s1 = "ab", s2 = "eidbaooo"

Output: true

Explanation: s2 contains one permutation of s1 ("ba").

COMPETITIVE PROGRAMMING

Permutation in String

```cpp
bool checkInclusion(string s1, string s2) {
    if (s1.empty() || s2.empty() || s1.length() > s2.length()) {
        return false;
    }
    vector<int> numbers(26, 0);
    vector<int> numbers2(26, 0);
    // Initialize the count of characters in s1
    for (char c : s1) {
        numbers[c - 'a']++;
    }
    for (int index = 0; index < s2.length(); index++) {
        // Increment the count of the current character in s2
        numbers2[s2[index] - 'a']++;
        if (index >= s1.length() - 1) {
            // Check if the counts match
            if (numbers == numbers2) {
                return true;
            }
            // Decrement the count of the character that is no longer in the sliding window
            numbers2[s2[index - s1.length() + 1] - 'a']--;
        }
    }
    return false;
}
```

COMPETITIVE PROGRAMMING

Example 3

Problem: Find the longest substring of a string containing ‘k’ distinct characters

Input: s = 'abcbdbdbbdcdabd‘

k=2

Output: bdbdbbd

```
string longestSubstringWithKDistinct(string s, int k) {
    int left = 0;
    int maxLength = 0;
    int maxLeft = 0;
    unordered_map<char, int> charFrequency;
    for (int right = 0; right < s.length(); right++) {
        char currentChar = s[right];
        charFrequency[currentChar]++;
        // Shrink the window if there are more than 'k' distinct characters
        while (charFrequency.size() > k) {
            char leftChar = s[left];
            charFrequency[leftChar]--;
            if (charFrequency[leftChar] == 0) {
                charFrequency.erase(leftChar);
            }
            left++;
        }
        // Update the maximum length and starting index of the substring
        if (right - left + 1 > maxLength) {
            maxLength = right - left + 1;
            maxLeft = left;
        }
    }
    // Extract and return the longest substring
    return s.substr(maxLeft, maxLength);
}
```

COMPETITIVE PROGRAMMING

Example 4

Problem: Find duplicates within a range ‘k’ in an array

Input: nums = [5, 6, 8, 2, 4, 6, 9]

k=2

Output: False

```
bool containsNearbyDuplicate(vector<int> nums, int k) {
    unordered_map<int, int> numIndices; // Stores the indices of numbers.
    for (int i = 0; i < nums.size(); i++) {
        // Check if the current number has been seen before and if it's within range 'k'.
        if (numIndices.find(nums[i]) != numIndices.end() && i - numIndices[nums[i]] <= k) {
            return true;
        } else {
            numIndices[nums[i]] = i; // Update the index of the current number.
        }
    }
    return false; // No duplicates within the specified range 'k'.
}
```

Some other Problems on Sliding Window

There are many problem statements of Sliding window techniques.

- Find minimum Sum SubArray of size k
- Length of the longest substring that doesn't contain any vowels
- Count negative elements present in every k-length subarray
- Minimum Size Subarray Sum
- Longest Repeating Character Replacement
- Count distinct absolute values in a sorted array
- Substrings of Size Three with Distinct Characters

Some other Problems on Sliding Window

- Maximum Erasure Value
- Maximum Number of Occurrences of a Substring
- Frequency of the Most Frequent Element
- Find Two Non-overlapping Sub-arrays Each With Target Sum
- Longest Subarray of 1's After Deleting One Element
- Minimum Operations to Reduce X to Zero
- Maximum Length of Repeated Subarray

COMPETITIVE PROGRAMMING

Thank You