Sliding Window Technique

The sliding window technique is a useful algorithmic pattern for solving problems that involve contiguous subarrays or substrings. The idea is to maintain a "window" that can expand and contract as needed to meet certain conditions, allowing for efficient traversal of the data structure (like an array or string).

Key Concepts:

1. **Window Size**: The window can be of fixed or variable size. In fixed-size problems, the window size remains constant, while in variable-size problems, it adjusts based on certain conditions.
2. **Two Pointers**: Typically, two pointers (or indices) are used to represent the current window's start and end. The left pointer usually represents the beginning of the window, and the right pointer represents the end.
3. **Expand and Contract**: You expand the window by moving the right pointer to include more elements, and you contract it by moving the left pointer to exclude elements when certain conditions are met.

Example Problem: Longest Substring Without Repeating Characters

Problem Statement: Given a string, find the length of the longest substring without repeating characters.

Example:

- Input: "abcabcbb"
- Output: 3 (The answer is "abc", with the length of 3.)

Sliding Window Approach in Java:

Here’s how you can implement the sliding window technique for the above problem in Java:

```java
import java.util.HashMap;

public class LongestSubstringWithoutRepeating {
    public static int lengthOfLongestSubstring(String s) {
        HashMap<Character, Integer> charIndexMap = new HashMap<>();
        int maxLength = 0;
        int left = 0; // Left pointer for the sliding window

        for (int right = 0; right < s.length(); right++) {
            char currentChar = s.charAt(right);

        // If the character is already in the map and its index is greater than or equal to the left pointer
        if (charIndexMap.containsKey(currentChar) && charIndexMap.get(currentChar) >= left) {
          // Move the left pointer to the right of the last occurrence of the current character
          left = charIndexMap.get(currentChar) + 1;
        }

        // Update the last index of the current character
        charIndexMap.put(currentChar, right);

        // Calculate the maximum length of the substring found
        maxLength = Math.max(maxLength, right - left + 1);
      }

      return maxLength;
    }

    public static void main(String[] args) {
      String input = "abcabcbb";
      int result = lengthOfLongestSubstring(input);
      System.out.println("Length of the longest substring without repeating characters: " + result);
    }
}
```

Explanation of the Code:

1. **HashMap**: We use a **HashMap** to store the last index of each character we encounter.
2. **Two Pointers**:
    - **left** pointer represents the start of the current window.
    - **right** pointer iterates through the string.
3. **Check for Repeats**:

    - If the character at the **right** pointer has been seen before and is within the current window (i.e., its index is greater than or equal to **left**), we move the **left** pointer to the right of the last occurrence of that character.

4. **Update the Maximum Length**: We calculate the length of the current substring by **right - left + 1** and update **maxLength** if this is larger than the previous maximum.
5. **Output**: Finally, the method returns the maximum length found.

Complexity:

- **Time Complexity**: O(n), where n is the length of the string. Each character is processed at most twice (once by the **right** pointer and once by the **left** pointer).
- **Space Complexity**: O(min(n, m)), where m is the size of the character set. The space is used by the HashMap to store the indices of characters.

This sliding window approach is efficient and effectively handles the problem of finding the longest substring without repeating characters.

Sliding window programs

Problem 1: Maximum Sum of a Subarray of Size K

Problem Statement: Given an array of integers and a number **K**, find the maximum sum of a subarray of size **K**.

```
public class MaxSumSubarray {

  public static int maxSum(int[] arr, int k) {

    int maxSum = 0;

    int windowSum = 0;


    // Calculate the sum of the first window

    for (int i = 0; i < k; i++) {

      windowSum += arr[i];

    }

    maxSum = windowSum;


    // Slide the window

    for (int i = k; i < arr.length; i++) {

      windowSum += arr[i] - arr[i - k];

      maxSum = Math.max(maxSum, windowSum);
```

```
        }

        return maxSum;
    }

    public static void main(String[] args) {
        int[] arr = {2, 1, 5, 1, 3, 2};
        int k = 3;
        System.out.println("Maximum sum of a subarray of size " + k + " is: " + maxSum(arr, k));
    }
}
```

Problem 2: Longest Substring Without Repeating Characters

Problem Statement: Given a string, find the length of the longest substring without repeating characters.

```
import java.util.HashSet;

public class LongestSubstring {
    public static int lengthOfLongestSubstring(String s) {
        HashSet<Character> set = new HashSet<>();
        int left = 0, maxLength = 0;

        for (int right = 0; right < s.length(); right++) {
            while (set.contains(s.charAt(right))) {
                set.remove(s.charAt(left));
                left++;
            }
            set.add(s.charAt(right));
            maxLength = Math.max(maxLength, right - left + 1);
        }
        return maxLength;
    }

    public static void main(String[] args) {
        String s = "abcabcbb";
        System.out.println("Length of the longest substring without repeating characters: " +
lengthOfLongestSubstring(s));
    }
}
```

Problem 3: Minimum Window Substring

Problem Statement: Given two strings **s** and **t**, return the minimum window substring of **s** such that every character in **t** (including duplicates) is included in the window. If there is no such substring, return the empty string.

```
import java.util.HashMap;

public class MinWindowSubstring {
    public static String minWindow(String s, String t) {
        if (t.length() > s.length()) return "";

        HashMap<Character, Integer> tCount = new HashMap<>();
        for (char c : t.toCharArray()) {
            tCount.put(c, tCount.getOrDefault(c, 0) + 1);
        }

        int required = tCount.size();
        int left = 0, right = 0, formed = 0;
        HashMap<Character, Integer> windowCounts = new HashMap<>();
        int[] ans = {-1, 0, 0}; // length, left, right
        while (right < s.length()) {
            char c = s.charAt(right);
            windowCounts.put(c, windowCounts.getOrDefault(c, 0) + 1);

            if (tCount.containsKey(c) && windowCounts.get(c).intValue() == tCount.get(c).intValue()) {
                formed++;
            }

            while (left <= right && formed == required) {
                c = s.charAt(left);

                if (ans[0] == -1 || right - left + 1 < ans[0]) {
                    ans[0] = right - left + 1;
                    ans[1] = left;
                    ans[2] = right;
                }

                windowCounts.put(c, windowCounts.get(c) - 1);
                if (tCount.containsKey(c) && windowCounts.get(c).intValue() < tCount.get(c).intValue()) {
                    formed--;
                }
                left++;
            }
            right++;
        }

        return ans[0] == -1 ? "" : s.substring(ans[1], ans[2] + 1);
    }

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

        String t = "ABC";

        System.out.println("Minimum window substring: " + minWindow(s, t));

    }

}
```

give me an example of dynamic programming in java

Dynamic Programming (DP) is a powerful technique used to solve problems that can be broken down into overlapping subproblems. It is particularly useful for optimization problems where the solution can be constructed from solutions to smaller subproblems.

Example Problem: Fibonacci Sequence

Problem Statement: Calculate the nth Fibonacci number. The Fibonacci sequence is defined as follows:

- F(0) = 0
- F(1) = 1
- F(n) = F(n-1) + F(n-2) for n > 1

Dynamic Programming Approach

We can solve this problem using dynamic programming in two ways: top-down (memoization) and bottom-up (tabulation). Below, I'll provide both implementations in Java.

1. Top-Down Approach (Memoization)

In this approach, we use recursion to solve the problem and store the results of subproblems in an array (or HashMap) to avoid redundant calculations.

java

VerifyOpen In EditorEditCopy code

```
import java.util.HashMap;

public class FibonacciTopDown {
    private HashMap<Integer, Integer> memo = new HashMap<>();

    public int fib(int n) {
        // Base cases
        if (n == 0) return 0;
        if (n == 1) return 1;

        // Check if the result is already in the memo
        if (memo.containsKey(n)) {
            return memo.get(n);
        }

        // Calculate the Fibonacci number and store it in the memo
        int result = fib(n - 1) + fib(n - 2);
        memo.put(n, result);
        return result;
    }

    public static void main(String[] args) {
        FibonacciTopDown fibonacci = new FibonacciTopDown();
        int n = 10; // Example input
        System.out.println("Fibonacci number at position " + n + " is: " + fibonacci.fib(n));
    }
}
```

2. Bottom-Up Approach (Tabulation)

In this approach, we iteratively build up the solution using an array to store the Fibonacci numbers from the bottom up.

java

VerifyOpen In EditorEditCopy code

```
public class FibonacciBottomUp {
    public int fib(int n) {
        // Base cases
        if (n == 0) return 0;
        if (n == 1) return 1;

```

```
   // Create an array to store Fibonacci numbers
   int[] fib = new int[n + 1];
   fib[0] = 0;
    fib[1] = 1;

    // Build the Fibonacci sequence iteratively
    for (int i = 2; i <= n; i++) {
      fib[i] = fib[i - 1] + fib[i - 2];
    }

    return fib[n];
  }

  public static void main(String[] args) {
    FibonacciBottomUp fibonacci = new FibonacciBottomUp();
    int n = 10; // Example input
    System.out.println("Fibonacci number at position " + n + " is: " + fibonacci.fib(n));
  }
}
```

Explanation of the Code:

1. **Top-Down Approach**:
    - We use a **HashMap** to store previously calculated Fibonacci numbers.
    - The function checks if the result for **n** is already computed; if not, it calculates it recursively and stores it in the **memo**.
2. **Bottom-Up Approach**:
    - We create an array **fib** to store the Fibonacci numbers up to **n**.
    - We initialize the base cases and then use a loop to compute the Fibonacci numbers iteratively.

Complexity:

- **Time Complexity**: O(n) for both approaches, as each Fibonacci number is computed once.
- **Space Complexity**:
    - Top-Down: O(n) due to the recursion stack and the memoization storage.

- Bottom-Up: O(n) due to the array used to store Fibonacci numbers.

Conclusion

Both approaches effectively demonstrate the dynamic programming technique for solving the Fibonacci sequence problem. The choice between top-down and bottom-up depends on the specific problem and personal preference.

give me an example of two pointer

The two-pointer technique is a common algorithmic approach used to solve problems involving arrays or strings, particularly when dealing with pairs or subarrays. The idea is to use two pointers to traverse the data structure, often from opposite ends or at different speeds, to find a solution efficiently.

Example Problem: Valid Palindrome

Problem Statement: Given a string, determine if it is a palindrome, considering only alphanumeric characters and ignoring case. A palindrome reads the same forward and backward.

Example:

- Input: **"A man, a plan, a canal: Panama"**
- Output: **true** (It is a palindrome)

Two-Pointer Approach in Java

Here’s how you can implement the two-pointer technique for the valid palindrome problem in Java:

java

VerifyOpen In EditorEditCopy code

```java
public class ValidPalindrome {
  public boolean isPalindrome(String s) {
    int left = 0; // Start pointer
    int right = s.length() - 1; // End pointer

    while (left < right) {
      // Move left pointer to the next alphanumeric character
      while (left < right && !isAlphanumeric(s.charAt(left))) {
        left++;
      }
```

```java
            // Move right pointer to the previous alphanumeric character
            while (left < right && !isAlphanumeric(s.charAt(right))) {
                right--;
            }
            // Compare characters at left and right pointers
            if (Character.toLowerCase(s.charAt(left)) != Character.toLowerCase(s.charAt(right))) {
                return false; // Not a palindrome
            }
            left++;
            right--;
        }
        return true; // It is a palindrome
    }

    // Helper method to check if a character is alphanumeric
    private boolean isAlphanumeric(char c) {
        return Character.isLetterOrDigit(c);
    }

    public static void main(String[] args) {
        ValidPalindrome vp = new ValidPalindrome();
        String input = "A man, a plan, a canal: Panama";
        boolean result = vp.isPalindrome(input);
        System.out.println("Is the input a valid palindrome? " + result);
    }
}
```

Explanation of the Code:

1. **Initialization**: Two pointers, **left** and **right**, are initialized to the start and end of the string, respectively.
2. **While Loop**: The loop continues until the **left** pointer is less than the **right** pointer.
3. **Skip Non-Alphanumeric Characters**:

- The inner while loops move the **left** and **right** pointers to the next valid alphanumeric character.
- The **isAlphanumeric** helper method checks if a character is a letter or a digit.

4. **Character Comparison**:
    - After finding valid characters at both pointers, we convert them to lowercase and compare.
    - If they do not match, the function returns **false**.
5. **Pointer Movement**: If the characters match, both pointers are moved inward (increment **left** and decrement **right**).
6. **Return Result**: If the loop completes without mismatches, the function returns **true**, indicating the string is a valid palindrome.

Complexity:

- **Time Complexity**: O(n), where n is the length of the string. Each character is processed at most twice.
- **Space Complexity**: O(1), since we are using only a fixed amount of extra space (for pointers and the helper method).