Day 12

Prepared by Azan Imtiaz

Problem 1

Longest Substring Without Repeatation Characters

Description:

Given a string `s`, find the length of the **longest**

substring

without repeating characters.

Example 1:

```
Input: s = "abcabcbb"
Output: 3
Explanation: The answer is "abc", with the length of 3.
```

Example 2:

```
Input: s = "bbbbb"
Output: 1
Explanation: The answer is "b", with the length of 1.
```

Example 3:

```
Input: s = "pwwkew"
Output: 3
Explanation: The answer is "wke", with the length of 3.
Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.
```

Approach

The given code is a solution to find the length of the longest substring without repeating characters in a given string `s`. Here's how it works:

1. **Create a Map**: A `HashMap` called `charIndexMap` is created to store characters and their latest indices in the string.
2. **Initialize Variables**: Two pointers, `left` and `right`, are used to define the current window of non-repeating characters. `res` is used to keep track of the maximum length

found.

3. **Iterate Over String**: Convert the string `s` into a character array `s2`. Then iterate over `s2` using the `right` pointer.
4. **Check and Update Left Pointer**: For each character at `s2[right]`, check if it's already in the map and whether its index is within the current window (between `left` and `right`). If so, update the result `res` if needed and move the `left` pointer to one position right of the repeated character's last index.
5. **Update Map**: Put the current character and its index into the map.
6. **Return Result**: After the loop ends, return the maximum of `res` and the length of the last considered substring (`right-1-left+1`).

```
class Solution {
    public int lengthOfLongestSubstring(String s) {
        Map<Character, Integer> charIndexMap = new HashMap<>();
        int right;
        char[] s2 = s.toCharArray();
        int res = 0;
        int left = 0;

        for (right = 0; right < s2.length; right++) {
            int idx1 = charIndexMap.getOrDefault(s2[right], -1);
            if (idx1 != -1 && idx1 >= left) {
                res = Math.max(res, right - 1 - left + 1);
                left = idx1 + 1;
            }
            charIndexMap.put(s2[right], right);
        }

        return Math.max(res, right - 1 - left + 1);
    }
}
```

Dry Run Example

Let's dry run the code with an example string `s = "abcabcbb"`.

1. **Initialization**: `charIndexMap` is empty, `res = 0`, `left = 0`, `right = 0`.
2. **First Iteration** (`right = 0`):
   - `s2[0] = 'a'`, not in map, so no change to `left`.
   - Update map with `'a': 0`.
   - Move `right` to 1.
3. **Second Iteration** (`right = 1`):

- `s2[1] = 'b'`, not in map, so no change to `left`.
- Update map with `'b': 1`.
- Move `right` to 2.

4. **Third Iteration** (`right = 2`):
   - `s2[2] = 'c'`, not in map, so no change to `left`.
   - Update map with `'c': 2`.
   - Move `right` to 3.
5. **Fourth Iteration** (`right = 3`):
   - `s2[3] = 'a'`, in map at index 0, which is less than `left`, so no change to `left`.
   - Update map with `'a': 3`.
   - Move `right` to 4.
6. **Fifth Iteration** (`right = 4`):
   - `s2[4] = 'b'`, in map at index 1, which is less than `left`, so no change to `left`.
   - Update map with `'b': 4`.
   - Move `right` to 5.
7. **Sixth Iteration** (`right = 5`):
   - `s2[5] = 'c'`, in map at index 2, which is less than `left`, so no change to `left`.
   - Update map with `'c': 5`.
   - Move `right` to 6.
8. **Seventh Iteration** (`right = 6`):
   - `s2[6] = 'a'`, in map at index 3, which is greater or equal to `left`.
   - Update `res` to `max(res, right-1-left+1)` which is `max(0, 6-1-0+1) = 6`.
   - Update `left` to `index of 'a' + 1` which is `4`.
   - Update map with `'a': 6`.
   - Move `right` to 7.
9. **Eighth Iteration** (`right = 7`):
   - `s2[7] = 'b'`, in map at index 4, which is equal to `left`.
   - No need to update `res` as `right-1-left+1` would be less than current `res`.

- Update `left` to `index of 'b' + 1` which is `5`.
- Update map with `'b': 7`.
- Move `right` to 8.

10. **End of Loop**:
    - `right` is now equal to `s2.length`, so the loop ends.
    - Return `max(res, right-1-left+1)` which is `max(6, 8-1-5+1) = 6`.

The length of the longest substring without repeating characters is `6`, corresponding to the substring `"abcabc"`.

Problem 2

Longest Repeating Character With Replacement

Approach to Solve the Problem

The `characterReplacement` method finds the length of the longest substring in a string `s` where you can replace up to `k` characters to make all characters in the substring identical.

Here's how it works:

1. **Initialize Variables**: Set `maxCount` to track the count of the most frequent character in the current window. `maxLength` is for the length of the longest valid substring found so far. `start` marks the beginning of the current window. `count` is an array to keep track of the frequency of each letter.
2. **Iterate Over String**: Loop through the string with `end` as the end pointer of the current window.
3. **Update Count Array**: Increment the count of the character at the current `end` position.
4. **Find Max Count**: Update `maxCount` to reflect the highest frequency of a single character in the current window.

5. **Calculate Window Size**: Determine the size of the current window from `start` to `end`.
6. **Shrink Window if Needed**: If the number of characters to be replaced `(windowSize - maxCount)` exceeds `k`, decrement the count of the character at the `start` position and move `start` forward.
7. **Update Max Length**: Check if the current window size is the largest one found that satisfies the replacement condition and update `maxLength` accordingly.
8. **Return Result**: After iterating through the entire string, return `maxLength` as the result.

Code

```
class Solution {
    public int characterReplacement(String s, int k) {

        int maxCount = 0;
        int maxLength = 0;
        int start = 0;
        int[] count = new int[26];

        for (int end = 0; end < s.length(); end++) {
            count[s.charAt(end) - 'A']++;
            maxCount = Math.max(maxCount, count[s.charAt(end) - 'A']);
            int windowSize = end - start + 1;
            if ((windowSize - maxCount) > k) {
                count[s.charAt(start) - 'A']--;
                start++;
            }

            maxLength = Math.max(maxLength, end - start + 1);
        }

        return maxLength;
    }
}
```

Example and Dry Run

Let's dry run the solution with an example:

- **String**: "AABABBA"

- **k**: 1

Step-by-step Execution:

1. Start with an empty window. `maxCount = 0`, `maxLength = 0`, `start = 0`.
2. Add 'A', `count[A] = 1`. `maxCount = 1`, `windowSize = 1`. No need to shrink window.
3. Add another 'A', `count[A] = 2`. `maxCount = 2`, `windowSize = 2`. Still no need to shrink.
4. Add 'B', `count[B] = 1`. `maxCount = 2`, `windowSize = 3`. No shrink needed.
5. Add 'A', `count[A] = 3`. `maxCount = 3`, `windowSize = 4`. No shrink needed.
6. Add 'B', `count[B] = 2`. `maxCount = 3`, `windowSize = 5`. Now `(windowSize - maxCount) = 2` which is greater than `k`. Shrink by moving `start` and decrementing `count[A]`.
7. Add 'B', `count[B] = 3`. `maxCount = 3`, `windowSize = 5`. No shrink needed.
8. Add 'A', `count[A] = 3`. `maxCount = 3`, `windowSize = 5`. No shrink needed.

At the end of the iteration, `maxLength = 5`, which is the length of the longest substring where we can replace up to `k` characters to make all characters identical. In this case, the substring is "BABBA" after replacing one 'A' with 'B'.