1717. Maximum Score From Removing Substrings

Medium Stack Greedy String [Leetcode Link]

Problem Description

You have a string s and two integers, x and y. Your task is to perform operations that involve removing specific substrings from s in order to gain points. The two operations you can perform are:

1. Remove the substring "ab" to gain x points.
2. Remove the substring "ba" to gain y points.

These operations can be done any number of times, in any order, wherever the substrings occur in s. The goal is to maximize the number of points obtained through these operations.

For example, if s is "cabbbae", by removing "ab" you could transform it to "cbbbae" and gain x points, or by removing "ba" you could transform it to "cabae" and gain y points.

The question asks for the maximum points that can be gained after applying the above operations on string s.

Intuition

The solution hinges on the idea that "ab" and "ba" substrings do not overlap. Therefore, we can treat the operations independently and order them to maximize our points. The optimal strategy is to always remove the most valuable substring first, which depends on whether x is greater than y or vice versa.

We use a two-pass algorithm incorporating stack data structures to ensure we always remove the most valuable substring first.

1. First, determine which of the substrings "ab" or "ba" is more valuable by comparing x and y.
2. Knowing which substring is more valuable, perform a single pass on the string s (reversed if "ba" is more valuable), removing all instances of the most valuable substring first and accumulating points. This is done using stack stk1.
3. With the residual string where the most valuable substring has been removed, perform a second pass to remove the less valuable substring and accumulate points. This is accomplished using the stack stk2.
4. The total points are the sum of points from both passes.

This approach ensures that we always prioritize the operation with higher points and leave the other operation for the remaining string, thereby guaranteeing the maximum points that can be gained.

Solution Approach

The implemented solution adopts a twin stack approach for efficiently identifying and removing the substring with the higher point value ("ab" or "ba") and then the one with the lower point value. The code goes as follows:

1. **Checking whether to reverse the string:** The solution first checks which of the two substrings, "ab" or "ba", will gain more points ($x > y$ or $y > x$). If "ba" is more valuable ($y > x$), the string is reversed and the function is called recursively with the values of x and y swapped. This is because we can treat the reversed string the same way as we treat the non-reversed string just by swapping "ab" with "ba". It ensures that we always remove "ab" substrings first from the string when we start processing, thereby generalizing the approach.

```
if x < y:
    return self.maximumGain(s[::-1], y, x)
```

2. **First pass with stack stk1:** The first pass through the string s uses a stack stk1 to remove all occurrences of the substring "ab" (which is deemed more valuable following the first step). While iterating through the string, every time a 'b' is encountered and the top of the stack is an 'a', that signifies an "ab" substring, and so the 'a' is popped from stk1, and x points are accumulated.

```
stk1 = []
for c in s:
    if c != 'b':
        stk1.append(c)
    else:
        if stk1 and stk1[-1] == 'a':
            stk1.pop()
            ans += x
        else:
            stk1.append(c)
```

3. **Second pass with stack stk2:** After completing the first pass, we are left with a string in stk1 with all "ab" substrings removed. We then repeat the process with another stack stk2 during the second pass over this residual string to remove all occurrences of "ba". Now, while popping elements from stk1, every time we encounter 'b' and the top of stk2 is an 'a', we have a "ba" substring, hence we pop from stk2, and y points are accumulated.

```
stk2 = []
while stk1:
    c = stk1.pop()
    if c != 'b':
        stk2.append(c)
    else:
        if stk2 and stk2[-1] == 'a':
            stk2.pop()
            ans += y
        else:
            stk2.append(c)
```

4. **Output the total points:** Finally, the ans variable, which was used to accumulate points across both passes, now holds the maximum number of points that can be gained and is returned.

Using stacks allows for only traversing the string twice, making the algorithm time complexity O(n), where n is the length of the string s. The space complexity is also O(n) due to the additional stacks used to keep track of characters for potential removal.

Example Walkthrough

Let's take an example to walk through the solution approach. Suppose we are given the string s = "babab", with x = 3 and y = 4. Remember that removing "ab" gives us x points and removing "ba" gives us y points.

Step 1: Check Substring Value Priority

First, we compare x and y to determine which substring removal is more valuable. In this case, since $y > x$, the "ba" substring is more valuable. Therefore, we reverse the string and swap the values of x and y, making the problem equivalent to first removing "ab" from "babab" reversed (which is "babab" itself, as the string is symmetric), and then removing "ba".

Step 2: First Pass with stack stk1

We initiate the first pass with stk1 which starts as an empty stack. We want to remove all instances of "ab".

We iterate over s = "babab":

- We push the first 'b' onto stk1.
- Next, we encounter an 'a'. We cannot form "ab" yet, so 'a' also gets pushed.
- When we meet the second 'b', 'a' is on top of stk1, we have found "ab", so we remove 'a' and accumulate 3 points (former value of x).
- Now we only have 'b' in stk1.
- The process continues, and each time we see an "ab", we remove it and gain 3 points.

After the first pass, stk1 will have "bb". We have removed two "ab" substrings for a total of 6 points.

Step 3: Second Pass with stack stk2

Now it's time for the second pass with stk2 for residual "ba" substrings in the string leftover in stk1 ("bb").

During this pass, we pop elements from stk1 and push them onto stk2 unless we encounter a 'b' and 'a' is at the top of stk2, which would signify a "ba" substring.

Since during the first pass we removed all instances of "ab", stk1 only has 'b's at this stage, so, 'b's are just moved to stk2 without any "ba" removals or additional points gained.

Step 4: Output the Total Points

With both passes completed, the total points accumulated are 6. As there are no "ba" substrings in the modified "babab" string after the first pass, no additional points were gained in the second pass.

Using the described twin-stacks approach, we efficiently calculated the maximum points that could be gained from transforming "babab", which in this specific case was 6 points.

Python Solution

```
class Solution:
    def maximum_gain(self, string: str, value_a: int, value_b: int) -> int:
        # If value_a is less than value_b, reverse the string and swap the values
        # then call the function again with the new parameters
        if value_a < value_b:
            return self.maximum_gain(string[::-1], value_b, value_a)

        # Initialize the answer counter and two stacks to keep track of characters
        answer = 0
        stack_ab, stack_ba = [], []

        # Iterate over each character in the string to handle 'ab' pairs
        for char in string:
            # If the character is not 'b', simply add it to the first stack
            if char != 'b':
                stack_ab.append(char)
            else:
                # If the character is 'b' and there's an 'a' on top of the first stack,
                # we found an 'ab' pair and we can add value_a to the answer
                if stack_ab and stack_ab[-1] == 'a':
                    stack_ab.pop()
                    answer += value_a
                else:
                    # Otherwise, just add the character to the first stack
                    stack_ab.append(char)

        # Iterate over the remaining elements in stack_ab to handle 'ba' pairs
        while stack_ab:
            char = stack_ab.pop()
            # If the character is not 'b', simply add it to the second stack
            if char != 'b':
                stack_ba.append(char)
            else:
                # If the character is 'b' and there's an 'a' on top of the second stack,
                # we found a 'ba' pair and can add value_b to the answer
                if stack_ba and stack_ba[-1] == 'a':
                    stack_ba.pop()
                    answer += value_b
                else:
                    # Otherwise, just add the character to the second stack
                    stack_ba.append(char)

        # Return the total gain from forming 'ab' and 'ba' pairs
        return answer

```

Java Solution

```
public class Solution {

    // Method to calculate the maximum gain by removing pairs of 'ab' or 'ba'.
    public int maximumGain(String s, int x, int y) {
        // If the gain from 'ba' is higher, reverse the string and swap the gains to reuse the same logic.
        if (x < y) {
            return maximumGain(new StringBuilder(s).reverse().toString(), y, x);
        }

        int totalGain = 0; // Initialize total gain to 0.
        Deque<Character> stack1 = new ArrayDeque<>(); // Use a stack for checking 'ab' pairs.
        Deque<Character> stack2 = new ArrayDeque<>(); // Use a secondary stack for checking 'ba' pairs after 'ab'.

        // Process the string for 'ab' pairs first, which has a higher or equal gain compared to 'ba'.
        for (char charInString : s.toCharArray()) {
            // If the current character is not 'b', push onto the stack.
            if (charInString != 'b') {
                stack1.push(charInString);
            } else {
                // If the top of stack is 'a', we found an 'ab' pair; pop 'a' and add the gain of 'ab' to totalGain.
                if (!stack1.isEmpty() && stack1.peek() == 'a') {
                    stack1.pop();
                    totalGain += x;
                } else {
                    // Else, push 'b' onto the stack to check for future pairs.
                    stack1.push(charInString);
                }
            }
        }

        // Process the remaining characters in stack1 for 'ba' pairs.
        while (!stack1.isEmpty()) {
            char currentChar = stack1.pop();
            if (currentChar != 'b') {
                stack2.push(currentChar);
            } else {
                // If the top of stack2 is 'a', we found a 'ba' pair; pop 'a' and add the gain of 'ba' to totalGain.
                if (!stack2.isEmpty() && stack2.peek() == 'a') {
                    stack2.pop();
                    totalGain += y;
                } else {
                    // Else, push 'b' onto stack2 to continue checking.
                    stack2.push(currentChar);
                }
            }
        }

        return totalGain; // Return the total gain calculated.
    }
}

```

C++ Solution

```
class Solution {
public:
    // Function to calculate the maximum gain by removing "ab" or "ba" substrings
    int maximumGain(string s, int x, int y) {
        // If x is less than y, reverse the string and swap values of x and y to
        // prioritize removing "ba" before "ab" since we want to maximize the gain
        if (x < y) {
            reverse(s.begin(), s.end());
            return maximumGain(s, y, x);
        }
        // Variable to store the total gain
        int totalGain = 0;
        stack<char> stkAb; // Stack to manage "ab" pairs
        stack<char> stkBa; // Stack to manage "ba" pairs

        // Iterate through the characters of the string
        for (char c : s) {
            // If the current character is 'a' or it's not 'b', push it onto stkAb
            if (c != 'b')
                stkAb.push(c);
            else {
                // If the top of stkAb is 'a', we found "ab" so we pop 'a' and
                // add x to totalGain
                if (!stkAb.empty() && stkAb.top() == 'a') {
                    stkAb.pop();
                    totalGain += x;
                } else {
                    // Otherwise, push 'b' to stkAb
                    stkAb.push(c);
                }
            }
        }

        // While stkAb is not empty, we will manage "ba" pairs
        while (!stkAb.empty()) {
            char c = stkAb.top();
            stkAb.pop();
            // If the current character is 'a' or it's not 'b', push it onto stkBa
            if (c != 'b')
                stkBa.push(c);
            else {
                // If the top of stkBa is 'a', we found "ba" so we pop 'a' and
                // add y to totalGain
                if (!stkBa.empty() && stkBa.top() == 'a') {
                    stkBa.pop();
                    totalGain += y;
                } else {
                    // Otherwise, push 'b' to stkBa
                    stkBa.push(c);
                }
            }
        }

        // Return the calculated total gain
        return totalGain;
    }
};

```

Typescript Solution

```
// Function to calculate the maximum gain by removing "ab" or "ba" substrings
function maximumGain(s: string, x: number, y: number): number {
  // If x is less than y, reverse the string and swap values of x and y to
  // prioritize removing "ba" before "ab" since we want to maximize the gain
  if (x < y) {
    s = s.split('').reverse().join('');
    return maximumGain(s, y, x);
  }

  // Variable to store the total gain
  let totalGain: number = 0;
  const stackAb: string[] = []; // Stack to manage "ab" pairs
  const stackBa: string[] = []; // Stack to manage "ba" pairs

  // Iterate through the characters of the string
  for (const c of s) {
    // If the current character is not 'b', push it onto stackAb
    if (c !== 'b') {
      stackAb.push(c);
    } else {
      // If the top of stackAb is 'a', we found "ab": pop 'a' and add x to totalGain
      if (stackAb.length > 0 && stackAb[stackAb.length - 1] === 'a') {
        stackAb.pop();
        totalGain += x;
      } else {
        // Otherwise, push 'b' onto stackAb
        stackAb.push(c);
      }
    }
  }

  // While stackAb is not empty, we will manage "ba" pairs
  while (stackAb.length > 0) {
    const topChar = stackAb.pop(); // Top character from stackAb
    // If the current character is not 'b' or stackBa is empty, push it onto stackBa
    if (topChar !== 'b' || stackBa.length === 0) {
      stackBa.push(topChar);
    } else {
      // If the top of stackBa is 'a', we found "ba": pop 'a' and add y to totalGain
      if (stackBa.length > 0 && stackBa[stackBa.length - 1] === 'a') {
        stackBa.pop();
        totalGain += y;
      } else {
        // Otherwise, push 'b' onto stackBa
        stackBa.push(topChar);
      }
    }
  }

  // Return the calculated total gain
  return totalGain;
}

```

Time and Space Complexity

Time Complexity

The time complexity of the provided code is mainly determined by iterating through the string s twice, once in the main function and once in the recursive call (when $x < y$). This is because the string reversal s[::-1] is $O(n)$ and the iteration over the string s is also $O(n)$, where n is the length of the string.

Moreover, operations with stack stk1 and stack2, such as appending and popping elements, are $O(1)$ operations. However, since these operations occur within the loop that iterates over the string, they do not contribute additional complexity beyond what is already accounted for by the iteration.

Consequently, the overall time complexity is $O(n)$.

Space Complexity

The space complexity of the code is driven by the storage required for the stacks, stk1 and stk2. In the worst case, all the characters of the string could be stored in the stacks if they do not form the defined pairs that trigger a pop operation—namely, 'ab' and 'ba' pairs depending on the values of x and y. Therefore, the space complexity is proportional to the length of the string s, which is $O(n)$.

In summary, both the time complexity and space complexity of the code are $O(n)$, where n is the length of the input string s.