1717. Maximum Score From Removing Substrings String Medium Stack Greedy

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.
- number of points obtained through these operations.

transform it to "cabae" and 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

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

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

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.
- 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. 1 if x < y:

1 stk1 = []

1 stk2 = []

10

11

while stk1:

else:

Example Walkthrough

We iterate over s = "babab":

We push the first 'b' onto stk1.

Now we only have 'b' in stk1.

Step 3: Second Pass with stack stk2

any "ba" removals or additional points gained.

if value\_a < value\_b:</pre>

stack\_ab, stack\_ba = [], []

answer = 0

else:

else:

char = stack\_ab.pop()

if char != 'b':

while stack\_ab:

else:

return answer

the first pass, no additional points were gained in the second pass.

def maximum\_gain(self, string: str, value\_a: int, value\_b: int) -> int:

return self.maximum\_gain(string[::-1], value\_b, value\_a)

# then call the function again with the new parameters

if stack\_ab and stack\_ab[-1] == 'a':

if stack\_ba and stack\_ba[-1] == 'a':

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

// Method to calculate the maximum gain by removing pairs of 'ab' or 'ba'.

stack\_ab.pop()

stack\_ba.append(char)

stack\_ba.pop()

public int maximumGain(String s, int x, int y) {

answer += value\_a

stack\_ab.append(char)

# If value\_a is less than value\_b, reverse the string and swap the values

# Initialize the answer counter and two stacks to keep track of characters

# we found an 'ab' pair and we can add value\_a to the answer

# Otherwise, just add the character to the first stack

# Iterate over the remaining elements in stack\_ab to handle 'ba' pairs

# If the character is not 'b', simply add it to the second stack

# we found a 'ba' pair and can add value\_b to the answer

Step 1: Check Substring Value Priority

c = stk1.pop()

stk2.append(c)

stk2.pop()

stk2.append(c)

ans += y

if stk2 and stk2[-1] == 'a':

1f c != 'b':

else:

for c in s:

if c != 'b':

stk1.append(c)

substring, hence we pop from stk2, and y points are accumulated.

maximum number of points that can be gained and is returned.

Remember that removing "ab" gives us x points and removing "ba" gives us y points.

**Solution Approach** 

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.

else: if stk1 and stk1[-1] == 'a': stk1.pop() ans += xelse: 10 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"

4. Output the total points: Finally, the ans variable, which was used to accumulate points across both passes, now holds the

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.

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.

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".

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.

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

 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).

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

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

With both passes completed, the total points accumulated are 6. As there are no "ba" substrings in the modified "babab" string after

Using the described twin-stacks approach, we efficiently calculated the maximum points that could be gained from transforming

# would signify a "ba" substring.

Step 4: Output the Total Points

Python Solution

class Solution:

10

17

18

19

20

21

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

43

44

45

45

46

47

48

49

51

50 }

C++ Solution

2 public:

5

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

41

42

43

44

45

46

47

48

49

50

51

52

53

54

55

56

58

8

9

10

11

12

13

14

17

18

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

41

42

43

44

45

46

47

48

49

50

51

53

57 };

1 class Solution {

if (x < y) {

int totalGain = 0;

for (char c : s) {

else {

if (c != 'b')

} else {

while (!stkAb.empty()) {

stkAb.pop();

if (c != 'b')

} else {

return totalGain;

return maximumGain(s, y, x);

let totalGain: number = 0;

for (const c of s) {

if (c !== 'b') {

} else {

stackAb.push(c);

// Variable to store the total gain

const stackAb: string[] = []; // Stack to manage "ab" pairs

const stackBa: string[] = []; // Stack to manage "ba" pairs

// If the current character is not 'b', push it onto stackAb

// Iterate through the characters of the string

else {

char c = stkAb.top();

stkBa.push(c);

// add y to totalGain

stkBa.pop();

totalGain += y;

stkBa.push(c);

// Return the calculated total gain

stkAb.push(c);

// add x to totalGain

stkAb.pop();

totalGain += x;

stkAb.push(c);

"babab", which in this specific case was 6 points.

# If the character is 'b' and there's an 'a' on top of the first stack,

# If the character is 'b' and there's an 'a' on top of the second stack,

11 12 # Iterate over each character in the string to handle 'ab' pairs for char in string: 13 # If the character is not 'b', simply add it to the first stack 14 if char != 'b': 15 stack\_ab.append(char) 16

38 answer += value\_b else: 39 40 # Otherwise, just add the character to the second stack stack\_ba.append(char) 41 42

Java Solution

public class Solution {

```
// 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);
 8
 9
            int totalGain = 0; // Initialize total gain to 0.
10
           Deque<Character> stack1 = new ArrayDeque<>(); // Use a stack for checking 'ab' pairs.
11
           Deque<Character> stack2 = new ArrayDeque<>(); // Use a secondary stack for checking 'ba' pairs after 'ab'.
12
13
14
           // Process the string for 'ab' pairs first, which has a higher or equal gain compared to 'ba'.
15
            for (char charInString : s.toCharArray()) {
               // If the current character is not 'b', push onto the stack.
16
17
                if (charInString != 'b') {
                    stack1.push(charInString);
19
               } else {
20
                    // If the top of stack is 'a', we found an 'ab' pair; pop 'a' and add the gain of 'ab' to totalGain.
21
                    if (!stack1.isEmpty() && stack1.peek() == 'a') {
22
                        stack1.pop();
23
                        totalGain += x;
24
                    } else {
25
                        // Else, push 'b' onto the stack to check for future pairs.
26
                        stack1.push(charInString);
27
28
29
30
31
           // Process the remaining characters in stack1 for 'ba' pairs.
32
           while (!stack1.isEmpty()) {
33
                char currentChar = stack1.pop();
                if (currentChar != 'b') {
34
35
                    stack2.push(currentChar);
36
                } else {
37
                    // 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') {
38
39
                        stack2.pop();
40
                        totalGain += y;
                    } else {
41
                        // Else, push 'b' onto stack2 to continue checking.
                        stack2.push(currentChar);
43
44
```

return totalGain; // Return the total gain calculated.

int maximumGain(string s, int x, int y) {

reverse(s.begin(), s.end());

return maximumGain(s, y, x);

// Variable to store the total gain

stack<char> stkAb; // Stack to manage "ab" pairs

stack<char> stkBa; // Stack to manage "ba" pairs

// Iterate through the characters of the string

if (!stkAb.empty() && stkAb.top() == 'a') {

// Otherwise, push 'b' to stkAb

// While stkAb is not empty, we will manage "ba" pairs

if (!stkBa.empty() && stkBa.top() == 'a') {

// Otherwise, push 'b' to stkBa

// Function to calculate the maximum gain by removing "ab" or "ba" substrings

// 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 the current character is 'a' or it's not 'b', push it onto stkAb

// If the top of stkAb is 'a', we found "ab" so we pop 'a' and

// If the current character is 'a' or it's not 'b', push it onto stkBa

// If the top of stkBa is 'a', we found "ba" so we pop 'a' and

#### 1 // 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('');

Typescript Solution

```
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;
 52 }
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 0(1) operations. However, since
these operations occur within the loop that iterates over the string, they do not contribute additional complexity beyond what is
```

// If the top of stackAb is 'a', we found "ab": pop 'a' and add x to totalGain

# 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 0(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.