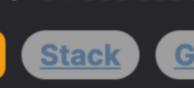
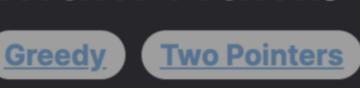
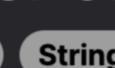
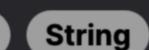
1963. Minimum Number of Swaps to Make the String Balanced

| | 4 |
|------|---|
| dium | |
| | ◥ |
| | |









Leetcode Link

Problem Description

In this problem, you are presented with a string s with an even length n. The string contains exactly n / 2 opening brackets '[' and n / 2 closing brackets ']'. A balanced string is defined as:

An empty string, or;

- A concatenation of two balanced strings AB, or;
- A string that contains another balanced string within it enclosed by a pair of brackets '[C]'. Your task is to determine the minimum number of swaps between any two indices needed to transform string s into a balanced

point in the future, a swap will be needed to pair that closing bracket with a previous opening bracket.

string. A swap means exchanging the characters at two specified indices in the string. You can perform any number of swaps.

The intuition behind the solution is to track the imbalance of brackets at any point in the string. When traversing the string, keep a

Intuition

counter that increases when encountering an opening bracket '[' and decreases when encountering a closing bracket ']', as long as there was a previously unpaired opening bracket (i.e., the counter is positive). If the counter is zero, and a closing bracket is found, then this is an excess closing bracket that can potentially be swapped to balance a previous excess opening bracket. Every time you encounter an excess closing bracket (signaled by the counter at zero before decrement), you know that at some

Once the entire string has been traversed, the counter will have the total count of unpaired opening brackets. Since each swap can

total count of unpaired opening brackets. The answer is the integer division of (ans + 1) >> 1, which is equivalent to math.ceil(ans / 2). This takes care of both even and

fix two unpaired brackets (by bringing an opening and a closing bracket together), the minimum number of swaps will be half of the

brackets, you would need at least 2 swaps to make the string balanced. **Solution Approach**

odd counts, as an odd count of unbalanced brackets will still require an extra swap. For example, if there are 3 unpaired opening

The implementation of the solution is straightforward, using a simple counter to keep track of bracket balance. No additional data structures are necessary, and the solution employs a greedy approach. The code iterates over the string character by character,

following these steps: 1. Initialize a variable ans to zero. This variable will track the number of unpaired opening brackets as the code iterates through the string.

- 2. Iterate through each character c in the string s: o If c is an opening bracket '[', increment ans. This is because each opening bracket could potentially require a closing bracket
- to balance it out.
- Else if c is a closing bracket ']' and ans is greater than zero, it means there's an unpaired opening bracket available, so decrement ans. This represents pairing the closing bracket with an earlier opening bracket that was unmatched. 3. After the loop, ans will represent the total count of excess opening brackets, which are unpaired. Since each swap can balance
 - two unpaired brackets, the minimum number of swaps required is (ans + 1) >> 1. This right shift operation is equivalent to a floor division by two and then a ceiling operation on the result (for odd numbers of ans), ensuring a correct count of swaps for balancing the string.
- 4. Return the calculated number of swaps. In essence, the algorithm is keeping track of how "deep" into unbalanced territory the string has gone with respect to opening brackets, and then using that depth to calculate the minimum number of swaps necessary to reintroduce balance.
- ans = 0for c in s:

def minSwaps(self, s: str) -> int:

ans += 1

if c == '[':

elif ans:

Here is the implementation detail from the given solution:

ans -= 1 return (ans + 1) >> 1

```
It's noteworthy to mention that this implementation has a time complexity of O(n), where n is the length of the string. This is because
it goes through the string once, performing constant time operations for each character.
Example Walkthrough
```

Let's walk through a small example to illustrate the solution approach.

Consider the string s = "]]][[[". The string length <math>n = 6, with n / 2 = 3 opening brackets and n / 2 = 3 closing brackets.

1. Initialize ans as 0.

Following the solution approach:

1 class Solution:

2. Start iterating over each character in the string:

For the second character], ans is still 0, do nothing.

For the third character], ans remains 0, do nothing.

 For the fifth character [, increase ans to 2 (two unpaired opening brackets). For the sixth character [, increase ans to 3 (three unpaired opening brackets).

For the fourth character [, increase ans to 1 (one unpaired opening bracket).

For the first character], since ans is 0 (no unpaired opening brackets), do nothing.

3. After iterating, ans is 3, representing three unpaired opening brackets. 4. To calculate the minimum number of swaps, we apply (ans + 1) >> 1, which is (3 + 1) >> 1 equals 4 >> 1 which is 2.

Swap indices 2 and 3 to form] [[(now the first pair is balanced).

Swap indices 4 and 5 to form [][] (all brackets are now balanced).

Iterate through each character in the string

for c in s:

10

if c == '[':

imbalance += 1

The implementation uses a simple counter to handle this tracking without additional data structures, which makes it efficient and easy to understand.

Thus, it will take a minimum of 2 swaps to transform s into a balanced string. One possible sequence of swaps:

Python Solution class Solution: def minSwaps(self, s: str) -> int:

imbalance = 0 # This variable tracks the number of imbalanced pairs

max_imbalance = 0 # This will hold the maximum imbalance encountered

If the character is an opening bracket, we increment imbalance

* Calculates the minimum number of swaps to make the brackets sequence balanced.

int imbalance = 0; // This variable will keep the count of current imbalance

// A closing bracket opposite to an opening one balances out,

// If the number of imbalances is odd, it's divided by 2 and then rounded up.

} else if (imbalance > 0) { // It's a closing bracket and we have an imbalance

int swaps = 0; // This variable will keep the total number of swaps needed

* @param s Input string containing the brackets sequence.

// Iterate through each character in the input string

// so decrease the current imbalance

// An opening bracket decreases the imbalance

* @return The minimum number of swaps required.

for (char bracket : s.toCharArray()) {

// Return the calculated number of swaps

if (bracket == '[') {

imbalance++;

imbalance--;

public int minSwaps(String s) {

```
# If the character is a closing bracket
               elif imbalance:
                   # If there's an imbalance, we decrement it as the closing bracket balances an opening one
                   imbalance -= 1
14
               # Track maximum imbalance
               max_imbalance = max(max_imbalance, imbalance)
16
17
           # The minimum number of swaps is the maximum imbalance divided by 2 (rounded up)
18
           # because each swap can fix two imbalances.
19
           return (max_imbalance + 1) // 2
20
21
Java Solution
   class Solution {
```

22 23 24 25 // The number of extra opening brackets is divided by 2 to get the number of swaps, 26 // because each swap will fix two misplaced brackets.

/**

6

8

9

10

11

12

13

14

15

16

17

18

19

20

21

```
28
           // The rightward shift operation (imbalance + 1) >> 1 is effectively dividing by 2
29
           // and rounding up in case of an odd number.
30
           swaps = (imbalance + 1) >> 1;
31
32
           return swaps;
33
34 }
35
C++ Solution
 1 class Solution {
2 public:
       int minSwaps(string s) {
           int openBrackets = 0; // Variable to keep track of the number of unmatched '['
           int swaps = 0;
                                 // Variable to keep track of the minimum number of swaps
           // Loop through each character in the string
           for (char& c : s) {
               // If the current character is an opening bracket
               if (c == '[') {
11
                   // Increase the count of unmatched opening brackets
12
                   openBrackets++;
13
               // If it is a closing bracket and there are unmatched opening brackets
               else if (openBrackets > 0) {
                   // Match the bracket and decrease the count of unmatched opening brackets
16
17
                   openBrackets--;
18
19
20
           // The number of swaps needed is half the number of unmatched opening brackets (rounded up)
22
           // because each swap can fix two unmatched opening brackets
23
           swaps = (openBrackets + 1) / 2;
24
```

// Function to calculate the minimum number of swaps required to balance the brackets

let openBrackets: number = 0; // Variable to keep track of the number of unmatched '['

// Variable to keep track of the minimum number of swaps

// Loop through each character in the string for (let c of s) {

return swaps;

Typescript Solution

function minSwaps(s: string): number {

let swaps: number = 0;

25

26

27

29

28 };

```
// If the current character is an opening bracket
           if (c === '[') {
               // Increase the count of unmatched opening brackets
10
11
               openBrackets++;
12
           // If it is a closing bracket and there are unmatched opening brackets
13
           else if (openBrackets > 0) {
14
               // Match the bracket and decrease the count of unmatched opening brackets
16
               openBrackets--;
           // When an unmatched closing bracket is found and there are no open brackets to match
           // it directly contributes to the number of swaps needed
19
20
21
       // The number of swaps needed is half the number of unmatched opening brackets (rounded up)
22
       // because each swap can fix two unmatched opening brackets
23
       swaps = Math.ceil(openBrackets / 2);
24
25
26
       // Return the calculated number of swaps
27
       return swaps;
28 }
29
Time and Space Complexity
```

loop that iterates over each character in the input string exactly once to count the balance of brackets.

Time Complexity The time complexity of the function is O(n) where n is the length of the input string s. This is because the function contains a single

Space Complexity

constant.

The space complexity of the function is 0(1). The only extra space used is for the variable ans which keeps track of the net number of open brackets ('[') encountered during the loop. This does not depend on the input size, and thus, the space complexity is