<https://leetcode.com/problems/minimum-number-of-swaps-to-make-the-string-balanced/description/>

You are given a **0-indexed** string s of **even** length n. The string consists of **exactly** n / 2 opening brackets '[' and n / 2 closing brackets ']'.

A string is called **balanced** if and only if:

It is the empty string, or

It can be written as AB, where both A and B are **balanced** strings, or

It can be written as [C], where C is a **balanced** string.

You may swap the brackets at **any** two indices **any** number of times.

Return *the****minimum****number of swaps to make*s ***balanced***.

**Example 1:**

**Input:** s = "][]["

**Output:** 1

**Explanation:** You can make the string balanced by swapping index 0 with index 3.

The resulting string is "[[]]".

**Example 2:**

**Input:** s = "]]][[["

**Output:** 2

**Explanation:** You can do the following to make the string balanced:

- Swap index 0 with index 4. s = "[]][][".

- Swap index 1 with index 5. s = "[[][]]".

The resulting string is "[[][]]".

**Example 3:**

**Input:** s = "[]"

**Output:** 0

**Explanation:** The string is already balanced.

**Constraints:**

n == s.length

2 <= n <= 106

n is even.

s[i] is either '[' or ']'.

The number of opening brackets '[' equals n / 2, and the number of closing brackets ']' equals n / 2.

**Attempt 1: 2025-07-21**

**Solution 1: Stack (30 min)**

class Solution {

    public int minSwaps(String s) {

        int unmatch = 0;

        Stack<Character> stack = new Stack<>();

        for(int i = 0; i < s.length(); i++) {

            char c = s.charAt(i);

            if(c == '[') {

                stack.push(c);

            } else {

                if(!stack.isEmpty()) {

                    stack.pop();

                } else {

                    unmatch++;

                }

            }

        }

        return (unmatch + 1) / 2;

    }

}

Time Complexity: O(n)

Space Complexity: O(n)

**Solution 2: Greedy + Math (30 min)**

class Solution {

    public int minSwaps(String s) {

        int unmatch = 0;

        int swap = 0;

        for(int i = 0; i < s.length(); i++) {

            char c = s.charAt(i);

            if(c == '[') {

                unmatch++;

            } else {

                if(unmatch > 0) {

                    // Matches with a previous '['

                    unmatch--;

                } else {

                    // Needs a swap

                    swap++;

                }

            }

        }

        // Equivalent to ceil(unmatched / 2)

        return (swap + 1) / 2;

    }

}

Time Complexity: O(n)

Space Complexity: O(1)

**Refer to Deepseek**

**Problem Summary**

Given a string s consisting only of '[' and ']', find the minimum number of swaps needed to make the string balanced. A balanced string has every '[' matched with a ']' in the correct order.

**Approach**

**Greedy Counting**:

Traverse the string while keeping track of unmatched '[' brackets.

When encountering an unmatched ']', it indicates an imbalance. We can fix this by swapping it with a later '['.

The number of such imbalances divided by 2 gives the minimum swaps needed.

**Key Insight**:

Each swap fixes two imbalances (one ']' and one '['), so the number of swaps is ceil(unmatched / 2).

**Solution Code**

class Solution {

public int minSwaps(String s) {

int unmatched = 0;

int swaps = 0;

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

if (c == '[') {

unmatched++;

} else {

if (unmatched > 0) {

unmatched--; // Matches with a previous '['

} else {

swaps++; // Needs a swap

}

}

}

return (swaps + 1) / 2; // Equivalent to ceil(unmatched / 2)

}

}

**Explanation**

**Initialization**:

unmatched counts the number of unmatched '[' brackets.

swaps counts the number of required swaps.

**Traversal**:

For each '[', increment unmatched.

For each ']':

If there's an unmatched '[', decrement unmatched (they form a pair).

Otherwise, increment swaps (this ']' needs to be swapped).

**Result Calculation**:

Each swap fixes two imbalances, so the minimum swaps is (swaps + 1) / 2 (equivalent to rounding up swaps / 2).

**Example**

**Input**: s = "]]][[["

Traversal:

']' → swaps = 1

']' → swaps = 2

']' → swaps = 3

'[' → unmatched = 1

'[' → unmatched = 2

'[' → unmatched = 3

swaps = 3, so result is (3 + 1) / 2 = 2.

**Time and Space Complexity**

**Time Complexity**: **O(n)**, where n is the length of the string (single pass).

**Space Complexity**: **O(1)**, using only a few extra variables.

**Alternative Approach (Stack-Based)**

A stack can also be used to count unmatched brackets, but the greedy approach is more efficient for this problem.

**Final Notes**

The greedy method is optimal for this problem.

The key is recognizing that each swap fixes two imbalances, leading to the formula (swaps + 1) / 2.

This solution efficiently computes the minimum swaps by leveraging simple counting and arithmetic.

**Show me stack based approach**

**Intuition**

The problem requires balancing a string of brackets ('[' and ']') with the minimum number of swaps. A stack can help track unmatched brackets efficiently. Here’s the idea:

**Use a stack** to match '[' and ']' as we traverse the string.

**Count unmatched ']'** that cannot be matched with a '[' in the stack.

**Each swap fixes two unmatched brackets**, so the minimum swaps is ceil(unmatched / 2).

**Algorithm**

Initialize a stack and a counter for unmatched ']'.

Traverse the string:

Push '[' onto the stack.

For ']', pop from the stack if possible (matching '['). Otherwise, increment the unmatched counter.

The number of swaps needed is (unmatched + 1) / 2.

**Solution Code (Stack-Based)**

import java.util.Stack;

class Solution {

public int minSwaps(String s) {

Stack<Character> stack = new Stack<>();

int unmatchedClosing = 0;

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

if (c == '[') {

stack.push(c);

} else {

if (!stack.isEmpty()) {

stack.pop(); // Match with a '['

} else {

unmatchedClosing++; // No matching '[', needs swap

}

}

}

return (unmatchedClosing + 1) / 2;

}

}

**Explanation**

**Stack Initialization**:

stack keeps track of unmatched '['.

unmatchedClosing counts ']' that couldn’t be matched.

**Traversal**:

**'['**: Push onto the stack (waiting for a ']').

**']'**:

If the stack is not empty, pop a '[' (they form a pair).

If the stack is empty, increment unmatchedClosing (this ']' needs a swap).

**Result Calculation**:

Each swap fixes **two** unmatched ']', so the minimum swaps is (unmatchedClosing + 1) / 2 (equivalent to rounding up unmatchedClosing / 2).

**Example**

**Input**: s = "]]][[["

Traversal:

']' → unmatchedClosing = 1 (stack is empty).

']' → unmatchedClosing = 2.

']' → unmatchedClosing = 3.

'[' → pushed to stack (stack = ['[']).

'[' → pushed to stack (stack = ['[', '[']).

'[' → pushed to stack (stack = ['[', '[', '[']).

unmatchedClosing = 3 → **Swaps needed = (3 + 1) / 2 = 2**.

**Time & Space Complexity**

**Time**: **O(n)** (single pass over the string).

**Space**: **O(n)** (stack can grow up to n/2 in the worst case).

**Comparison with Greedy Approach**

|  |  |  |  |
| --- | --- | --- | --- |
| **Approach** | **Time** | **Space** | **Key Idea** |
| **Greedy** | O(n) | O(1) | Count unmatched  ']'  directly. |
| **Stack** | O(n) | O(n) | Explicitly track unmatched  '['  with a stack. |

**When to Use Which?**

**Greedy** is **more efficient** (no extra space).

**Stack** is **more intuitive** for those familiar with bracket-matching problems.

**Final Notes**

The stack approach is **clear and easy to understand**, but the greedy method is **optimal** for this problem.

Both approaches yield the same result, but the greedy one is **preferred** in interviews for efficiency.

Would you like further optimization or a dry run on another example? 😊

**Refer to**

[L301.Remove Invalid Parentheses (Ref.L22)](note://EAF368E1345B4DBFB47321E244D8E03F)