

2914. Minimum Number of Changes to Make Binary String Beautiful

You are given a **0-indexed** binary string S having an even length.

A string is **beautiful** if it's possible to partition it into one or more substrings such that:

- Each substring has an **even length**.
- Each substring contains **only** 1's or **only** 0's.

You can change any character in S to 0 or 1.

Return *the **minimum** number of changes required to make the string S beautiful.*

Example 1:

Input: $s = "1001"$

Output: 2

Explanation: We change $s[1]$ to 1 and $s[3]$ to 0 to get string "1100".

It can be seen that the string "1100" is beautiful because we can partition it into "11|00".

It can be proven that 2 is the minimum number of changes needed to make the string beautiful.

Example 2:

Input: $s = "10"$

Output: 1

Explanation: We change $s[1]$ to 1 to get string "11".

It can be seen that the string "11" is beautiful because we can partition it into "11".

It can be proven that 1 is the minimum number of changes needed to make the string beautiful.

Example 3:

Input: $s = "0000"$

Output: 0

Explanation: We don't need to make any changes as the string "0000" is beautiful already.

Constraints:

- $2 \leq s.length \leq 10^5$
- s has an even length.
- $s[i]$ is either '0' or '1'.

2914. Minimum Number of Changes to Make Binary String Beautiful

We can make the implementation much more concise by making a key observation: any even-length sequence can be split into pairs of two characters. This is the smallest valid even sequence we can have. If we can organize the entire string into pairs where both characters are the same — either both '0's or both '1's — we'll end up with a beautiful string. This is illustrated in the diagram below:

1	0	0	0	1	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---

String divided into pairs

1	0	0	0	1	0	0	0	0	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---

3 dissimilar pairs

1	1	0	0	1	1	0	0	1	1	1	1
---	---	---	---	---	---	---	---	---	---	---	---

3 changes (in this case, 0 -> 1)

To put this idea into practice, we'll look at the string two characters at a time. If the two characters in each pair are the same, we can move on without any changes. If they don't match, we know that one of the bits will need to be flipped to make them identical.

We'll keep a counter to track how many bits we've flipped throughout the process. At the end, we can return this count, giving us the total number of changes needed to create a beautiful string.

/*

One pointer

Time complexity: $O(n)$

Space complexity: $O(n)$

*/

```
class Solution {
public:
    int minChanges(std::string s){
        int n=s.size();
        int ans=0;
        for(int i=0;i<n-1;i+=2){
            if(s[i]!=s[i+1]) ans++;
        }
        return ans;
    }
};
```

Complexity Analysis

Let n be the length of the input string S .

- Time complexity: $O(n)$

The algorithm iterates through the input string using a step size of 2 . We examine each pair exactly once, performing $n/2$ comparisons in total. Each comparison takes constant time.

Thus, the time complexity is $O(n)$.

- Space complexity: $O(1)$

No additional space is used which scales with the input size, so the space complexity remains constant.