Balanced String

Non-Recursive

Pseudo-code:

```
ALGORITHM BalancedString(S)
  n ← length of s
  maxLen ← 0
  For i := 0 to n - 1 do
    create array count[0..127] and initialize all to 0
    distinct ← 0
    For j := i to n - 1  do
      if count[s[j]] = 0 then  distinct ← distinct + 1
      count[s[j]] ← count[s[j]] + 1
      if distinct > 2 then   break
      if distinct = 2 then
      {
        create array values[0..1]
        index ← 0
        For k := 0 to 127  do
          if count[k] > 0 then
            values[index] ← count[k]
            idx ← idx + 1
          if index = 2 then    break
}
        if vals[0] = vals[1] then    maxLen ← max(maxLen, j - i + 1)
  return maxLen
```

Analysis:

Outer Loop:

- The outer loop runs from i = 0 to i = n - 1, where n is the length of the string s.

Inner Loop:

- The inner loop starts from j = i to j = n - 1. So, for each starting index i, it explores all possible substrings starting at index i.
- Operations in each iteration of the inner loop:
- The count array (size 128) is built to store the frequency of each character.
- The distinct count is updated to track how many distinct characters are in the substring.
- If there are exactly 2 distinct characters, the code checks if their frequencies are equal. If they are, it updates maxLen with the length of the current valid substring.

Time Complexity Analysis:

1. Outer Loop:

- The outer loop runs n times, where n is the length of the string s.

2. Inner Loop:

- The inner loop runs from j = i to n - 1. So, in the worst case, the inner loop also runs n times for each value of i.

3. Operations inside the inner loop:

- The count array, which has a fixed size of 128, is created in each iteration.
- The distinct count is updated as we iterate through the characters of the substring.
- When there are exactly 2 distinct characters, the code checks the frequency of the characters by iterating over the count array (of size 128). This check takes constant time O(128), which is essentially O(1) since the size of the array is fixed.

Time Complexity:

- The two loops (i and j) are nested, and each loop runs up to n times. So, in the worst case, the time complexity of the two loops is O(n^2).
- The inner checks for frequency comparisons are O(1) since the size of the count array is constant (128).
- $O(n^2)$

Recursive

Pseudo-code:

Divide and Conquer

```
Algorithm BalancedString(S,l,)
    create array count[0..127] and initialize all to 0
    distinct ← 0
  for i := l to r do
    if count[s[i]] = 0 then     distinct  ← distinct + 1
    count[s[i]] ← count[s[i]] + 1
    if distinct != 2 then      return FALSE
  values[2] ← {0}
  index ← 0
  for i := 0 to 127:
    if count[i] > 0 then    values[index] ← count[i]
      index ← index + 1
    if index = 2 then     break
  if vals[0] != vals[1] then      return FALSE
return TRUE

function longestBalanced (s, l, r):
  if r - l + 1 < 2  then       return 0
  if BalancedString (s, l, r)  then       return r - l + 1
  max1 ← longestBalanced (s, l, r - 1)
  max2 ← longestBalanced (s, l + 1, r)
 return max(max1, max2)
Function max(a, b):
  if a > b:
```

```
    return a
Else
  return b

Function main
  n ← length of s
  return longestBalanced (s, 0, n - 1)
```

Analysis:

Function BalancedString(s, l, r):

Purpose: Checks if the substring from index l to r contains exactly two distinct characters with equal frequencies.

Time Complexity:

First loop: Iterates over the substring and updates the frequency of each character, taking O(r - l + 1) time.

Second loop: Iterates over the fixed-size count array of size 128, which takes O(1) time.

Overall time complexity: O(r - l + 1).

Function longestBalanced(s, l, r):

Purpose: Finds the longest balanced substring (two characters with equal frequency).

Time Complexity:

- The BalancedString function is called for every substring.
- The function makes two recursive calls for each range (l, r-1 and l+1, r), leading to a recurrence relation T(n) = 2T(n - 1) + O(n).

Overall time complexity: $O(2^n)$ due to the nested recursive calls.

Function longestBalanced(s):

Purpose: Calls longestBalanced(s, 0, n - 1) to find the longest balanced substring.

Time Complexity: Same as longestBalanced(s, l, r), i.e., $O(2^n)$.

Overall Time Complexity:

- BalancedString(s, l, r): O(r - l + 1)
- longestBalanced (s, l, r): $O(2^n)$ due to the recursive calls.

Comparison :

Aspect	Recursive Approach	Non-Recursive Approach
Time Complexity	$O(2^n)$ (Exponential)	$O(n^2)$ (Quadratic)
Ease of Understanding	Easier to conceptualize for divide-and-conquer	More straightforward with loops
Efficiency	Less efficient for larger inputs	More efficient for larger inputs
Use Case	Suitable for small input sizes	Suitable for larger input sizes