**Algorithm Analysis: Finding the Longest Balanced Substring**

**Problem Statement**

The problem is to find the longest "balanced substring" in a given string. A balanced substring is defined as a substring containing exactly two different characters, with each character appearing the same number of times.

**Algorithm 1: Brute Force Approach**

**Description**

This algorithm uses a brute force approach by examining all possible substrings of the input string and checking if each one is balanced.

**Implementation**

def is\_balanced(string):

letters = {}

for letter in string:

if letter in letters:

letters[letter] += 1

else:

letters[letter] = 1

if len(letters) > 2:

return False

if len(letters) != 2:

return False

values = list(letters.values())

return values[0] == values[1]

def find\_longest\_balanced\_substring(string):

longest = 0

for i in range(len(string)):

for j in range(i + 1, len(string) + 1):

if is\_balanced(string[i:j]):

longest = max(longest, j - i)

return longest

**Pseudocode**

Procedure is\_balanced(string)

1: Create an empty dictionary letters

2: For each letter in string do

3: If letter exists in letters then

4: Increment letters[letter]

5: Else

6: Set letters[letter] = 1

7: EndIf

8: If length of letters > 2 then

9: Return False

10: EndIf

11: EndFor

12: If length of letters ≠ 2 then

13: Return False

14: EndIf

15: Set values = list of values in letters

16: If values[0] = values[1] then

17: Return True

18: Else

19: Return False

20: EndIf

EndProcedure

Procedure find\_longest\_balanced\_substring(string)

1: Set longest = 0

2: For i = 0 to length of string - 1 do

3: For j = i + 1 to length of string do

4: Set Substring = string[i..j]

5: If is\_balanced(Substring) then

6: Set longest = max(longest, j - i)

7: EndIf

8: EndFor

9: EndFor

10: Return longest

EndProcedure

**Time Complexity Analysis**

* The outer loop runs O(n) times.
* The inner loop runs O(n) times for each iteration of the outer loop.
* The is\_balanced function takes O(k) time where k is the length of the substring being checked, which can be up to n.
* Therefore, the overall time complexity is O(n³).

**Algorithm 2: Optimized Iterative Approach**

**Description**

This algorithm optimizes the approach by avoiding unnecessary recomputing of character counts. For each starting position i, it keeps track of two characters and their counts while extending the substring.

**Implementation**

def find\_longest\_balanced\_substring(s):

n = len(s)

longest = 0

for i in range(n):

count1 = count2 = 0

char1 = s[i]

char2 = None

for j in range(i, n):

if s[j] == char1:

count1 += 1

elif char2 is None:

char2 = s[j]

count2 += 1

elif s[j] == char2:

count2 += 1

else:

break

if count1 == count2 and count1 > 0:

longest = max(longest, j - i + 1)

return longest

**Pseudocode**

Procedure find\_longest\_balanced\_substring(s)

1: Set n = length of s

2: Set longest = 0

3: For i = 0 to n - 1 do

4: Set count1 = 0, count2 = 0

5: Set char1 = s[i]

6: Set char2 = None

7: For j = i to n - 1 do

8: If s[j] == char1 then

9: Increment count1

10: Else If char2 is None then

11: Set char2 = s[j]

12: Increment count2

13: Else If s[j] == char2 then

14: Increment count2

15: Else

16: Break loop

17: EndIf

18: If count1 == count2 and count1 > 0 then

19: Set longest = max(longest, j - i + 1)

20: EndIf

21: EndFor

22: EndFor

23: Return longest

EndProcedure

**Time Complexity Analysis**

* The outer loop runs O(n) times.
* The inner loop runs at most O(n) times for each iteration of the outer loop.
* All operations inside the loops are O(1).
* Therefore, the overall time complexity is O(n²).

**Algorithm 3: Recursive Approach**

**Description**

This algorithm uses recursion to find the longest balanced substring. It examines substrings by incrementally extending them and recursively exploring different starting positions.

**Implementation**

def find\_longest\_balanced\_substring(s, i=0, j=0):

if i >= len(s):

return 0

if j >= len(s):

return find\_longest\_balanced\_substring(s, i + 1, i + 1)

count1 = count2 = 0

char1 = s[i]

char2 = None

for k in range(i, j + 1):

if s[k] == char1:

count1 += 1

elif char2 is None:

char2 = s[k]

count2 += 1

elif s[k] == char2:

count2 += 1

else:

return max(find\_longest\_balanced\_substring(s, i, j + 1),

find\_longest\_balanced\_substring(s, i + 1, i + 1))

if count1 == count2 and count1 > 0:

return max(j - i + 1, find\_longest\_balanced\_substring(s, i, j + 1))

else:

return find\_longest\_balanced\_substring(s, i, j + 1)

**Pseudocode**

Procedure find\_longest\_balanced\_substring(s, i = 0, j = 0)

1: If i ≥ length of s then

2: Return 0

3: EndIf

4: If j ≥ length of s then

5: Return find\_longest\_balanced\_substring(s, i + 1, i + 1)

6: EndIf

7: Set count1 = 0, count2 = 0

8: Set char1 = s[i]

9: Set char2 = None

10: For k = i to j do

11: If s[k] == char1 then

12: Increment count1

13: Else If char2 is None then

14: Set char2 = s[k]

15: Increment count2

16: Else If s[k] == char2 then

17: Increment count2

18: Else

19: Return max(

find\_longest\_balanced\_substring(s, i, j + 1),

find\_longest\_balanced\_substring(s, i + 1, i + 1)

)

20: EndIf

21: EndFor

22: If count1 == count2 and count1 > 0 then

23: Return max(

j - i + 1,

find\_longest\_balanced\_substring(s, i, j + 1)

)

24: Else

25: Return find\_longest\_balanced\_substring(s, i, j + 1)

26: EndIf

EndProcedure

**Time Complexity Analysis**

* The recursive approach leads to many overlapping subproblems.
* Without memoization, this algorithm has exponential time complexity O(2^n) in the worst case.
* For each i and j pair, we might need to make two recursive calls.
* The algorithm processes each position multiple times through different recursive calls.

**Comparative Analysis**

| **Algorithm** | **Time Complexity** | **Advantages** | **Disadvantages** |
| --- | --- | --- | --- |
| Brute Force | O(n³) | Simple to understand and implement | Very inefficient for large inputs |
| Optimized Iterative | O(n²) | Much more efficient than brute force; constant space | Still quadratic time complexity |
| Recursive | O(2ⁿ) | Can handle complex cases with backtracking | Very inefficient; can cause stack overflow for large inputs |

**Conclusion**

For practical purposes, the optimized iterative approach (Algorithm 2) provides the best balance of efficiency and simplicity. It avoids the cubic time complexity of the brute force approach and the potential exponential complexity of the non-memoized recursive solution.