Algorithm

1- Print **this** cover sheet
2- Please write all your names **in Arabic**
3- Please make sure that your ID is correct
4- Attendance Handwritten Signature should be filled in **before** discussion
5- Please, come on your time. Teams who will be **late** will subject to **-2** of the total grade

- Project Idea: **balanced string**
- Team Discussion Time: **1:30 pm**
- ID : **21**

	ID [Ordered by ID]	Full Name [IN ARABIC]	Attendance Handwritten Signature
1	20220229	شروق مصطفى طه علي رشوان	
2	20220160	رشا صلاح محمود حفني موسى	
3	20220140	حور العين احمد حسن مرسي	
4	20220535	نرفانا السيد فرج الله السيد اسماعيل	
5	20220200	سارة ياسر جابر محمد	
6	202001134	ندى نبيل محمد محمد محمود	
7			

Project Requirements

First algorithm (non-recursive)	Pseudo code	2	
	Implementation	2	
	Analysis & complexity	2	
Second algorithm (recursive)	Pseudo code	2	
	Implementation	2	
	Analysis & complexity	2	
Comparison	1		
Understanding	2		

15

Analysis Of Non-Recursive& Recursive Algorithms (Balanc)

Non-recursive :

Pseudo code :

```
1.  // Function to find the maximum of two numbers
2.  algorithm max(a, b)  {
3.     Result  ←  (a > b) ? a : b;
4.     return Result;          }

5.  algorithm strlen_custom(str)  {
6.     length  ←  0;
7.     while str[length] != '\0' do  {
8.        length  ←  length + 1;     }
9.     return length;      }

10. algorithm longestBalanced(str)  {
11.    maxLength  ←  0;
12.    length  ←  strlen_custom(str);

13.    for i  ←  0 to length - 1 do    {
14.       // Reset count1 and count2 for each new starting character
15.       count1  ←  0;
16.       count2  ←  0;
17.       char1  ←  str[i];
18.       char2  ←  '\0'; // Initialize char1 with current character

19.       for j  ←  i to length - 1 do      {
20.          if str[j]  ←  char1 then         {
21.             count1  ←  count1 + 1;         }
```

```
22.         else if char2 = '\0' or str[j] = char2 then     {
23.           char2 ← str[j]; // Set char2 if encountering a new
    character
24.           count2 ← count2 + 1;        }
25.         else {
26.           break;    }    // Exit the loop if there are more than 2
    different characters

27.  // Check if there are exactly two different characters and they
    occur the same number of times
28.         if (count1 = count2) and ((count1 + count2) = (j - i + 1)) then {
29.           maxLength ← max(maxLength, j - i + 1);  }    }  }

30.   return maxLength;
31. }

32. // Main function
33. algorithm main() {
34.   str[100];
35.   write("Enter a string: ");
36.   read(str);

37.   result ← longestBalanced(str);
38.   if result > 0 then  {
39.     write("Length of the longest balanced substring is: ");
40.     write(result);  }
41.   else  {
42.     write("Length of the longest balanced substring is: 0");  }
43.   return 0; }
```

Code :

```
int max(int a, int b) {
    return (a > b) ? a : b;
}

int strlen_custom(char* str) {
    int length = 0;
    while (str[length] != '\0') {
        length++;
    }
    return length;
}

int longestBalanced(char* str) {
    int maxLength = 0;
    int length = strlen_custom(str);

    for (int i = 0; i < length; i++) {
        // Reset count1 and count2 for each new starting character
        int count1 = 0, count2 = 0;
        char char1 = str[i], char2 = '\0'; // Initialize char1 with current
character

        for (int j = i; j < length; j++) {
            if (str[j] == char1) {
                count1++;
            } else if (char2 == '\0' || str[j] == char2) {
                char2 = str[j]; // Set char2 if encountering a new character
                count2++;
            } else {
                break; // Exit the loop if there are more than 2 different
characters
            }

            // Check if there are exactly two different characters and they
occur the same number of times
            if ((count1 == count2) && ((count1 + count2) == (j - i + 1))) {
                maxLength = max(maxLength, j - i + 1);
            }
        }
    }

    return maxLength;
```

```c
}


int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);

    int result = longestBalanced(str);
    if (result > 0) {
        printf("Length of the longest balanced substring is: %d\n", result);
    } else {
        printf("Length of the longest balanced substring is:0");
    }

    return 0;
}
```

Analysis :

```c
#include <stdio.h>
#include <string.h>

int max(int a, int b) {
    return (a > b) ? a : b;
}

int strlen_custom(char* str) {
    int length = 0;
    while (str[length] != '\0') {
        length++;
    }
    return length;}
```

O(1)

O(n)

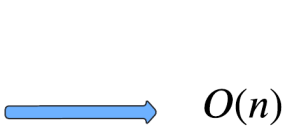
```c
int longestBalanced(char* str) {
    int count[2] = {0};
    char char1 = '\0', char2 = '\0';
    char array[100];
    int length = strlen(str);
    int arrayIndex = 0;
    int maxLength = 0;

    for (int i = 0; i < length; i++) {
        if (char1 == '\0') {
            char1 = str[i];
            array[arrayIndex++] = str[i];
        } else if (char2 == '\0' && str[i] != char1) {
            char2 = str[i];
            array[arrayIndex++] = str[i];
        } else if (str[i] == char1 || str[i] == char2) {
            array[arrayIndex++] = str[i];
        }
    }

    if (count[0] != count[1]) {
        printf("Not balanced\n");
        return 0;
    }

    for (int i = 0; i < arrayIndex; i++) {
        for (int j = i + 1; j <= arrayIndex; j++) {
            int count1 = 0, count2 = 0;
            for (int k = i; k < j; k++) {
                if (array[k] == char1) {
                    count1++;
                } else {
                    count2++;
                }
            }
            if (count1 == count2) {
                maxLength = max(maxLength, j - i);
            }
        }
    }

    return maxLength;
}
```



$O(n)$

$$\Sigma_{i=0}^{n}\ \Sigma_{j=i}^{n} = 1$$
$$= \Sigma_{i=0}^{n}(n - i + 1)$$
$$= -(n+1)\Sigma_{i=0}^{n} i$$
$$= -(n+1)\Sigma_{i=0}^{n} 1 = -(n+1)\frac{(n(n+1))}{2} = n^2$$

$$O(n^2)$$

TIME COMPLXETY: T(n)= n^2

```
int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);
    int result = longestBalanced(str);
    if (result > 0 ) {
        printf("Length of the longest balanced substring: %d\n", result);
    } else {
        printf("Length of the longest balanced substring: 0");
    }
    return 0;
}
```

=o(n)^2

Recursive:

Pseudo code:

1.Start

2.Enter string s

3.Call getLongestBalanced (S) to print the longest balanced substring in the string

- declare maxLength = 0
- declare int startIndex = 0
- declare endIndex = 0
- declare size = 0
- Increment size by 1 as long as s[size]=! '\0'
- Declare length = size
- Call LongestBalance (S, 0, length - 1, &maxLength) to find longest balanced substring in string
- Print maxlength

LongestBalance (char* S, int startindex, int endindex, int* maxLength)

- Check if startindex > endindex then return
- End if
- Check if (isBalanced (S, startindex, endindex)) is true do
- currentLength = endindex - startindex + 1
- Check if (currentLength > *maxLength) do
- Set *maxLength = currentLength
- End if

- End if
- Call function LongestBalance (S, startindex, endindex - 1, maxLength)
- Call function LongestBalance (S, startindex + 1, endindex, maxLength)

I sBalanced (char* S, int startindex , int endindex)

- Declare count[26] = {0}
- Declare distinctChars = 0
- for i <-- startindex to i <-- endindex do
- increament count[S[i] - 'a'] by 1
- End for
- for i <-- 0 to i < 26 do
- check if count[i] > 0
- Then increament distinctChars by 1
- End for
- if (distinctChars == 2) is true then
- Declare CharCount1 = 0;
- Declare CharCount2 = 0;

- for i <-- 0 i < 26 do
- check if (count[i] > 0) then
- check if (CharCount1 == 0) then
- Set CharCount1 = count[i]
- End if
- Else set CharCount2 = count[i];
- End else

- End if
- End for
- return CharCount1 == CharCount2
- End if
- Else return false

Code :

```
bool isBalanced (char* S, int startindex , int endindex ) {
    int count[26] = {0};

    for (int i = startindex; i <= endindex; i++) {
        count[S[i] - 'a']++;
    }

    int distinctChars = 0;
    for (int i = 0; i < 26; i++) {
        if (count[i] > 0) {
            distinctChars++;
        }
    }

    if (distinctChars == 2) {
        int CharCount1 = 0;
        int CharCount2 = 0;

        for (int i = 0; i < 26; i++) {
            if (count[i] > 0) {
                if (CharCount1 == 0) {
                    CharCount1 = count[i];
                } else {
                    CharCount2 = count[i];
                }
            }
        }

        return CharCount1 == CharCount2;
    }

    return false;
}

void LongestBalance (char* S, int startindex, int endindex, int* maxLength) {
    if (startindex > endindex) {
        return;
    }

    if (isBalanced (S, startindex, endindex)) {
        int currentLength = endindex - startindex + 1;
        if (currentLength > *maxLength) {
```

```
            *maxLength = currentLength;
        }
    }

    LongestBalance (S, startindex, endindex - 1, maxLength);
    LongestBalance (S, startindex + 1, endindex, maxLength);
}

void getLongestBalanced (char* S) {
    int maxLength = 0;
    int startIndex = 0;
    int endIndex = 0;
    int size = 0;
    while ( S[size]!='\0'){
        size++;
    }
    int length = size;

    LongestBalance (S, 0, length - 1, &maxLength);

    printf("Longest balanced string length: %d\n", maxLength);
}

int main() {

    char S[100];
    printf("Enter a string: ");
    scanf("%s", S);

    getLongestBalanced (S);

    return 0;
}
```

Analysis:

```
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

bool isBalanced (char* S, int startindex , int endindex ) {
   int count[26] = {0};

   for (int i = startindex; i <= endindex; i++) {
      count[S[i] - 'a']++;
   }

   int distinctChars = 0;
   for (int i = 0; i < 26; i++) {
      if (count[i] > 0) {
         distinctChars++;
      }
   }

   if (distinctChars == 2) {
      int CharCount1 = 0;
      int CharCount2 = 0;

      for (int i = 0; i < 26; i++) {
         if (count[i] > 0) {
            if (CharCount1 == 0) {
               CharCount1 = count[i];
            } else {
               CharCount2 = count[i];
            }
         }
      }

      return CharCount1 == CharCount2;
   }

   return false;
}
```

$\Sigma_{i=startin}^{endin} 1 = n$

$\Sigma_{i=0}^{25} 1 = 26$

$\Sigma_{i=0}^{25} 1 = 26$

TIME COMPLXETY OF isBalanc :

$T(n) = n + 26 + 26 = n$

```
void LongestBalance (char* S, int startindex, int endindex, int*
maxLength) {
   if (startindex > endindex) {
      return;
   }

   if (isBalanced (S, startindex, endindex)) {
      int currentLength = endindex - startindex + 1;
      if (currentLength > *maxLength) {
         *maxLength = currentLength;
      }
   }

   LongestBalance (S, startindex, endindex - 1, maxLength);
   LongestBalance (S, startindex + 1, endindex, maxLength);
```

$T(n) = 2T(n-1) + 1$

$T(n) = 2T(n-1) + 1$

$T(n) = 4T(n-2) + 2$

$T(n) = 8T(n-3) + 3$

$T(n) = 2^k T(n-k) + (2^k - 1)$

$T(0) = n - k \longrightarrow k = n$

$T(n-1) = 2T(n-2) + 1$

$T(n-1) = 2(2T(n-2) + 1) + 1$

$T(n-2) = 2T(n-3) + 1$

$T(n-2) = 4(2T(n-3) + 1) + 2$

$T(n) = 2^n \cdot T(0) + (2^n - 1)$

$T(n) = O(2^n)$

TIME COMPLXETY OF LongestBalanc :

$T(n) = 2^n + n = O(2^n)$

isBalance

```
void getLongestBalanced (char* S) {
   int maxLength = 0;
   int startIndex = 0;
   int endIndex = 0;
   int size = 0;
   while ( S[size]!='\0'){
      size++;
   }
   int length = size;

   LongestBalance (S, 0, length - 1, &maxLength);

   printf("Longest balanced string length: %d\n",
maxLength);
}

int main() {

   char S[100];
   printf("Enter a string: ");
   scanf("%s", S);

   getLongestBalanced (S);

   return 0;
}
```

while loop

TIME COMPLXETY OF getLongestBalanc :

$\Sigma_{i=0}^{n} 1 = n$

2^n

$T(n) = O(2^n)$

TIME COMPLXETY OF MAIN :

$T(n) = O(2^n)$

Comparison between Time Complexity :

	Non-Recursive&	*Recursive*
Best case	Ω (n)	Ω (n)
Worst case	O(n^2)	O(2^n)
Average case	between Θ (n^2) and Θ (n^3)	Θ (2^n)

Non-Recursive :

1. The best-case scenario occurs when the input string contains only two distinct characters, and these characters appear an equal number of times in the string.
   In this case, the function longestBalanced will iterate through the string once, and the condition (count1 == count2) && ((count1 + count2) == (j - i + 1)) will be satisfied for the entire string.
   Therefore, the best-case time complexity is O(n), where n is the length of the input string.

2. The worst-case scenario occurs when the input string contains multiple distinct characters, and each character occurs only once or does not form a balanced substring with any other character.
   In this case, the function longestBalanced will iterate through the string twice, once for the outer loop and once for the inner loop.
   Within the inner loop, it may need to compare each character with every other character, resulting in a time complexity of O(n^2). Therefore, the worst-case time complexity is O(n^3), where n is the length of the input string.

3. The average-case time complexity depends on the distribution of characters in the input string.
   If the input string contains a relatively small number of distinct characters and they are evenly distributed, the average-case time complexity may be closer to O(n^2).
   However, if the input string contains many distinct characters or if the characters are not evenly distributed, the average-case time complexity may still approach O(n^3).
   Overall, the average-case time complexity is typically between O(n^2) and O(n^3), depending on the specific characteristics of the input data.

Recursive :

1. In the best-case scenario, the input string contains only two distinct characters, and these characters appear an equal number of times in the string.
   In this case, the function isBalanced will return true for the entire string, and the function LongestBalance will only need to check the entire string once.
   Therefore, the best-case time complexity is O(n), where n is the length of the input string.

2. In the worst-case scenario, the input string contains multiple distinct characters, and each character occurs only once or does not form a balanced substring with any other character.
   In this case, the function isBalanced will need to iterate through the entire substring for each possible combination of start and end indices.

3. The average-case time complexity depends on the distribution of characters in the input string.
   If the input string contains a relatively small number of distinct characters and they are evenly distributed.

   or, if the input string contains many distinct characters or if the characters are not evenly distributed, depending on the specific characteristics of the input data.