Task 4"Birthdayday Candles"

Documentation for task 4:

We will cover main points in this documentation

1-purpose

2-expected outputs

3-define the user_ defined function and its purpose and parameters and return

4-define every step

5- Pseudocode and its complexity time

6-The complexity for the whole code (Best//Worst)

7-some test cases to check the validity of code

1-Purpose: This problem aims to obtain the age of someone in his birthday and put number of candles ...this number is the same to his age but with different length

For example:

Age=5

So, we will put 5 candles but they can be with different length...and then store these lengths of candles in an array to make some processes on this array which is given by user.

2--expected outputs:

The tester or the user will expect some outputs which is:

Firstly, Output1: is the integer which represent the tallest length of candles.

Secondly, Output2: the number of occurrence of this tallest length in an array. ==how many times this tallest length repeated in an array.

Thirdly, Output3: is Boolean ((True or false))

It returns true if it is symmetric

And false if it is nonsymmetrical

If array size of numbers of candles lengths which is represented by n is even So, It is exactly not symmetric.

But if the n is odd .Now, we can check firstly the position of largest number if it is exactly in the middle, we will continue our check then

Secondly, we will compare the left side of it with the right side of it if…As we will consider this number as a mirror.

But if it is not in the middle we will return and do not continue.

3: Define the main functions

Here, There is the main function as it's the backbone of code...it cannot run without calling main function to start program execution.

It is simply.

But, there is some user defined functions

Firstly// (((find _largest)))

```
def find_largest (array) :
    largest = array[0]
    for i in range (0,n):
        if array[i]>largest:
            largest=array[i]
    return largest
```

It obtains the array of candles lengths as parameter...and returns an integer which represent the tallest length which is the same as maximum number of array.

Secondly: (count_occurences ())

```
def count_occurences(array,largest):
    counter=0
    for j in range (0,n):
        if array[j]==largest:
            counter+=1

    return counter
```

This user defined function gets 2 parameters. They are the array and the largest number which is (((return of function [find_ largest]))).

And

Return an integer number which represents the number of occurrence of this largest value.

Thirdly: is the is_symmetric::

```
def is_symmetric (array, largest,n):

    if (n%2==0):
        return False
    else:
       for i in range (0,n):
           if array[i]==largest:
            index=i
            break

       if index!=(n//2):
            return False
```

It gets 2 parameters

The array and the largest number and the size (n)

Then we check if it is even or not

If it is even it will print “not symmetric”

If it is odd … Now we have different scenario as we should determine “the largest number position”

If position as index = the median of array (n//2)

We use this double slash (//) to check that output will be integer. This this the first step to check that is symmetric …..But it is not enough to prove this as we should check that this largest as a mirror as the left side of it matches with the right side.

```
if index!=(n//2):
    return False
else:
   for j in range(0,n):
           if array[j]!=array[n-1-j]:
               return False
   return True
```

We defined all user-defined functions I have used and the rest of functions is

The (((main)))

```
print("Happy birthday to you")
n= int (input("enter your new age"))
array=[]
for i in range(0,n):
    element =int(input(f"enter element of array which is candles length{i+1}:"))
    array.append(element)
```

It is used only in this code to scan the age and store it in variable n and generate an array with size n

Then ask the user to fill this array with candles lengths then start my processes that

I send this array as a parameter to each of functions that I mentioned before to make process on then I just call these functions and print their outputs from main ().

This figure about calling functions from the main to make it return their outputs.

```
largest=find_largest(array)
print(f"The largest length of candles is: {largest}")
x = count_occurences(array, largest)
print(f"Largest is repeated for {x} times ")
if (is_symmetric(array,largest,n)):
    print ("symmetric")
else:
    print ("not symmetric")
```

4—The complexity of the whole code is:

1. find_largest Function: **Time Complexity:** O(n), where n is the length of the array.

2. count_occurences Function: **Time Complexity:** O(n).

3. is_symmetric Function:

a-Check if n is even: O(1).

b-Find the index of the first occurrence of largest: Time Complexity: O(n).

c-Compare elements for symmetry:Time Complexity: O(n).

Overall Time Complexity: O(n).

4. Main Program: **Time Complexity:** O(n).

Overall Time Complexity

- **The three key functions (find_largest, count_occurences, and is_symmetric) each takes O(n) time.**
- **The input loop also takes O(n).**

- **Thus, the overall time complexity of the program is: O(n)+O(n)+O(n)+O(n)=O(n)**

The program's time complexity is O(n)

5- The pseudocode of this code

1. **Print congratulation:**
    - **Output: "Happy birthday to you"**
2. **Input Data:**
    - **Read integer n (number of candles).**
    - **Initialize empty list array.**
    - **For i from 0 to n-1:**
        - **Prompt: "t Enter he length of candle {i+1}: "**
        - **Read integer value and add it to array.**
3. **Find the Largest Candle Height:**
    - **Set largest = array[0].**
    - **For each element height in array:**
        - **If height > largest, update largest.**
    - **Save largest.**
4. **Count the Occurrences of the Largest Height:**
    - **Set counter = 0.**
    - **For each element height in array:**

- **If height == largest, increment counter.**
- **Save counter.**

5. **Check Symmetry:**
   - **If n is even:**

     Return "not symmetric".
   - **Otherwise:**
   - **Find the first index of largest:**
   - **For i from 0 to n-1:**

     If array[i] == largest, save index and stop loop.

     If index != n // 2:

     Return "not symmetric".
   - **Otherwise:**
   - **Compare elements from both ends of array:**
   - **For j from 0 to n-1:**

     If array[j] != array[n - j - 1]:

     Return "not symmetric".

     Return "symmetric".
6. **Output Results:**
   - **Print the largest candle height: "The largest length of candles is: {largest}"**

- **Print the number of occurrences: "Largest is repeated for {counter} times"**
- **Print symmetry result: "symmetric" or "not symmetric".**

6-Trace for code

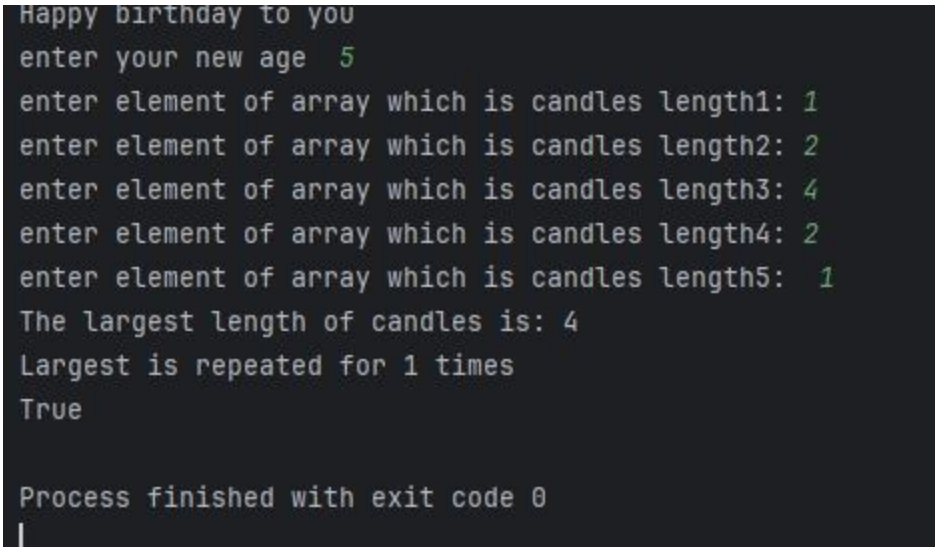
```
Happy birthday to you
enter your new age 4
enter element of array which is candles length1: 1
enter element of array which is candles length2: 2
enter element of array which is candles length3: 4
enter element of array which is candles length4: 3
The largest length of candles is: 4
Largest is repeated for 1 times
False

Process finished with exit code 0
```

Input:

- **The program first displays a greeting: Happy birthday to you.**
- **The program prompts the user to input 4 elements , Final array: [1, 2, 4, 3]**
- **From [1, 2, 4, 3], the largest value is 4.**
- The program counts how many times the largest value 4 appears in the array: 4 appears **once**.
- Symmetry is checked as follows:

  First, the program checks if `n` is even:

Since `n = 4` (even), symmetry is **immediately false** because symmetry is only possible for an odd-length array.



- **The program first displays a greeting: Happy birthday to you.**
- **The program prompts the user to input 5 elements , Final array: [1, 2, 4, 2, 1]**
- **From [1, 2, 4, 2, 1], the largest value is 4.**
- The program counts how many times the largest value 4 appears in the array: 4 appears **once**.
- Symmetry is checked as follows:

  First, it verifies that `n` is **odd**:

  `n = 5` → **odd** → Symmetry check continues.

- **Largest value 4 is at index 2, which is the middle index of the array (n // 2 = 2)**

- Then, the program checks the array is symmetric by comparing elements from the left and right sides:
  - Compare:
    4 `array[0] == array[4]` → `1 == 1` → True
    5 `array[1] == array[3]` → `2 == 2` → True
  - All corresponding pairs are equal, so the array is **symmetric**.

```
Happy birthday to you
enter your new age  5
enter element of array which is candles length1: 1
enter element of array which is candles length2: 2
enter element of array which is candles length3: 4
enter element of array which is candles length4: 1
enter element of array which is candles length5:  2
The largest length of candles is: 4
Largest is repeated for 1 times
False

Process finished with exit code 0
```

- **The program first displays a greeting: Happy birthday to you.**
- **The program prompts the user to input 5 elements , Final array: [1, 2, 4, 1, 2]**
- **From [1, 2, 4, 1, 2], the largest value is 4.**
- The program counts how many times the largest value 4 appears in the array: 4 appears **once**.
- Symmetry is checked as follows:

  First, it verifies that `n` is **odd**:

`n = 5` → **odd** → Symmetry check continues.

- **Largest value 4 is at index 2, which is the middle index of the array (n // 2 = 2)**
- Then, the program checks the array is symmetric by comparing elements from the left and right sides:
- Compare:

  `array[0] == array[4]` → `1 != 2` → **False**
- **Since the elements are not symmetric**

Task 7- K-th Element of Two Sorted Arrays

Documentation for task 7:

We will cover main points in this documentation

1-purpose

2-expected outputs

3-define the user_ defined function and its purpose and parameters and return

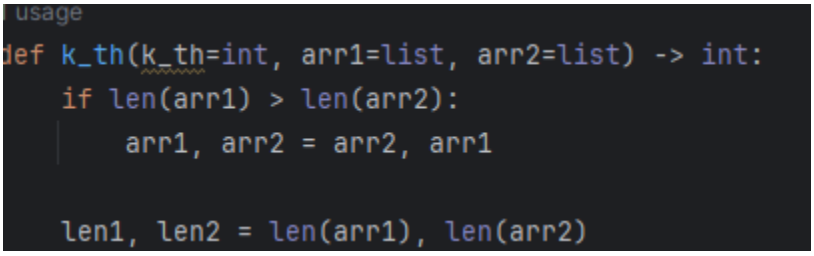
4-define every step

5- Pseudocode and its complexity time

6-The complexity for the whole code (Best//Worst)

7-some test cases to check the validity of code

Step 1: Handle Array Lengths



- **The first step is to ensure that we always perform the search on the smaller of the two arrays. This is because searching on the smaller array can reduce the number of comparisons we need to**

make. If arr1 is longer than arr2, the arrays are swapped.if len(arr1) > len(arr2):

arr1, arr2 = arr2, arr1

Step 2: Bound Check

```
if k_th >= (len1 + len2) or k_th < 0:
    return -1
```

- **Before proceeding, the function checks if the k_th is valid:**
- **If k_th is negative or larger than the total number of elements in both arrays combined, the function returns -1.**
- **if k_th >= (len1 + len2) or k_th < 0:**

  return -1

Step 3: Initialize Binary Search

```
i = (len1 if len1 < k_th else k_th) // 2
```

- **The binary search-like method begins by setting an index i for arr1 (the smaller array). The index i is initially calculated based on the length of arr1 and k_th, with the idea of dividing the search space roughly in half.**
- **However, the calculation of i ensures that we are never searching out of bounds in arr1.**

- i = (len1 if len1 < k_th else k_th) // 2

Step 4: Binary Search Loop

```
while i <= len1:
    j = k_th - i + 1
```

- The loop begins by searching for the correct position in both arrays. The key idea is to partition both arrays into two parts: one part representing the smaller elements and the other part representing the larger elements. The goal is to find a split where:
  - The largest element on the left side of the partition is smaller than the smallest element on the right side of the partition.
  - The binary search logic uses two key variables i and j:
  - i represents the number of elements chosen from arr1 for the left partition.
  - j represents the number of elements chosen from arr2 for the left partition.
- Here's the loop logic:

  while i <= len1:

    j = k_th - i + 1

Step 5: Boundary Values

```
    if i == 0:
        maxlft1 = float('-inf')
    else:
        maxlft1 = arr1[i - 1]

    if i == len1:
        minrit1 = float('inf')
    else:
        minrit1 = arr1[i]

    if j == 0:
        maxlft2 = float('-inf')
    elif j <= len2:
        maxlft2 = arr2[j - 1]
    else:
        maxlft2 = float('inf')
if j >= len2:
    minrit2 = float('inf')
else:
    minrit2 = arr2[j]
```

- **We define the boundary values for both arrays at the left and right partitions.**
- **These boundary values are crucial to check if the partition is valid:**
- **For arr1:**
- **maxlft1: The largest element on the left side of arr1 (if i is 0, it's negative infinity because there are no elements on the left side).**
- **minrit1: The smallest element on the right side of arr1 (if i is equal to len1, it's positive infinity because there are no elements on the right side).**
- **For arr2:**

- maxlft2: The largest element on the left side of arr2 (if j is 0, it's negative infinity).
- minrit2: The smallest element on the right side of arr2 (if j is equal to len2, it's positive infinity).
- maxlft1 = float('-inf') if i == 0 else arr1[i - 1]
- minrit1 = float('inf') if i == len1 else arr1[i]
- maxlft2 = float('-inf') if j == 0 else (arr2[j - 1] if j <= len2 else float('inf'))
- minrit2 = float('inf') if j >= len2 else arr2[j]

Step 6: Check Partition Validity

```
if maxlft1 <= minrit2 and maxlft2 <= minrit1:

    k = maxlft1 if maxlft1 > maxlft2 else maxlft2
    return k
```

- After calculating the boundary values, the function checks if the partition is valid:
- Valid partition condition: The largest element on the left side of arr1 must be less than or equal to the smallest element on the right side of arr2 and vice versa.
- If the partition is valid, the k-th element is the maximum of the two largest values on the left sides of both arrays. This is because, in a sorted order, the larger of the two left-side values will be the k-th element.
- if maxlft1 <= minrit2 and maxlft2 <= minrit1:
- return max(maxlft1, maxlft2)

Step 7: Adjust Binary Search

```
elif maxlft1 > minrit2:
    i -= 1
else:
    i += 1
```

- **If the partition is not valid, the binary search adjusts the search range:**
- **If maxlft1 > minrit2, it means we have taken too many elements from arr1, so we decrease i to move towards the left side.**
- **Otherwise, we increase i to move towards the right side of arr1.**
- **if maxlft1 > minrit2:**

  i -= 1

- **else:**

  i += 1

Step 8: Return -1 if Not Found

```
print(k_th( k_th: 15,  arr1: [2, 3, 6, 10, 12],  arr2: [1, 14, 15, 16, 17, 18, 19, 20, 22, 24]))
```

- **If the loop finishes without finding a valid partition, the function returns -1.**
- **return -1**

Analyzing the Complexity of the Pseudo-code:

1. Initialization and Input Validation:

Swapping the arrays (if needed) and setting up variables like len1 and len2 are O(1) operations.

Edge case checking (e.g., verifying k_th is valid) is also O(1).

2. Binary Search on the Shorter Array (Key Step):

The binary search is performed on arr1, the shorter of the two arrays.

In each iteration of the binary search:

Computing j (the corresponding index in arr2) is O(1).

Comparing partition boundaries (maxlft1, minrit1, maxlft2, minrit2) is O(1).

Adjusting i (moving left or right) involves integer arithmetic, which is also O(1).

Since binary search repeatedly halves the search space in arr1, the number of iterations is at most log(len1).

3. Overall Complexity:

Let and .

Because we always perform binary search on the smaller array, and , the complexity of this step is .

Final Time Complexity:

The most time-intensive step is the binary search, so the overall time complexity is:

O(\log(\min(n_1, n_2)))

Function k_th(k_th, arr1, arr2):

If the length of arr1 is greater than the length of arr2:

Swap arr1 and arr2

Set len1 to the length of arr1

Set len2 to the length of arr2

If k_th is greater than or equal to the sum of len1 and len2, or if k_th is less than 0:

Return -1 (since k_th is out of range)

Set i to the minimum of len1 and k_th // 2 (initial search index)

While i <= len1:

Set j to k_th - i + 1

If i equals 0:

Set maxlft1 to -∞ (no element in the left side of arr1)

Else:

Set maxlft1 to the element in arr1 at index i-1

If i equals len1:

Set minrit1 to +∞ (no element in the right side of arr1)

Else:

Set minrit1 to the element in arr1 at index i

If j equals 0:

Set maxlft2 to -∞ (no element in the left side of arr2)

Else if j is less than or equal to len2:

Set maxlft2 to the element in arr2 at index j-1

Else:

Set maxlft2 to +∞ (no element in the left side of arr2)

If j is greater than or equal to len2:

Set minrit2 to +∞ (no element in the right side of arr2)

Else:

Set minrit2 to the element in arr2 at index j

If maxlft1 <= minrit2 and maxlft2 <= minrit1:

Return the maximum of maxlft1 and maxlft2 (this is the k_th element)

If maxlft1 > minrit2:

Decrease i (move left in arr1)

Else:

Increase i (move right in arr1)

Return -1 (if no valid element is found)