Hands-on-4 1002233393

DAA - Rashmitha R

problem - 0

Aim: To implement the Fibonacci sequence and debug the function for fib(5).

```
x = Fib(n)
 if n == 0
  return 0
 if n == 1
  return 1
return fib(n-1) + fib(n-2)
```

Step-1: To get fib(5).

The first recursive call is fib(4)
The Second recursive call fib(3)

Step-2: For Fib(4), which is recieved from fib(5).
The first recursive call fib(3).
The second recursive call is fib(2)

Step-3: fib(3), it
The first recursive call is fib(2)
The second recursive call is fib(1)

Step-4: fib(2)

The first recursive call is fib(1)
The second recursive call is fib(0)

5. fib(1), which is received from fib(2)

fib(1) returns 1

6. fib(0), which is received from fib(2)
fib(0) will return 0

Hence, fib(1) returns 1
fib(0) returns 0

fib(2) = fib(0) + fib(1)

fib(2) = 0+1 = 1

7. fib(1), which got from fib(3), returns 1

So we can say that

fib(3) = fib(2) + fib(1)

fib(3) = 1+1 = 2

8. fib(2), got from fib(4).

It returns fib(1) and fib(0) [∵ fib(1) = 1 fib(2) - 1
fib(0) = 0

we can say that fib(4) = 2+1 = 3

9. fib(3), which is received from fib(4)

It returns fib(2) and fib(1)

fib(2) : returns fib(1) and fib(0).

fib(1) : returns 1 and fib(0) returns 0

From this we can say that

fib(5) = 3+2 = 5

fib(4) = 5

fib(5) calls fib(4) → fib(3)

fib(4) calls fib(3) → fib(2)

fib(3) calls fib(2) and fib(1)

fib(2) calls fib(1) → fib(0)

Time Complexity of Fibonacci

The time complexity is $O(2^n)$. It is an

Time Complexity

The time complexity of fibonacci is $O(2^n)$
It is an exponential.

It is because each call to fib(n) generates two or more recursive calls to
fib(n-1) and fib(n-2)

The recursion tree grows exponentially, making it inefficient for large values of n.

The ways to improving fibonacci Implementation

Memoization

A method to optimize the fibonacci by using memoization, which saves previously calculated.

fibonacci values in a table. this avoids the redundant computations and reduces the time complexity is O(n).

Iterative Approach

In this method, fibonacci numbers are computed in a loop rather than recursion. This achieves O(n) time complexity with O(1) space complexity.

problem-1

Time Complexity

To take elements and store it in one array, we can assume a time complexity of O(N*K), then have sorted the array based on merge sort. So overall time complexity is O(NK) + O(NKlogm) = O(NK log(NK))

Improved Approach

Since each array is sorted could have used a min heap that would return smallest element in constant time.

Since the array is sorted we can take one element from each array and compare the first element of the array. After finding the smallest element we must take the next element from the same array and then compare again.

Analysing the approach worst case complexity is O(N*k(log k)).

problem-2:

Time Complexity:

Array is traversal is done only once so time Complexity would be O(n).

Improved Approach:

Could you think of any other approach as this is both space and time efficient, as only one array is used, have modified the input array to form the output. So, I assume the approach is efficient.