Name: Pitta Pavitya
ID: 2030126 Sec: 03

Ans. To The Ques. No: 2

Implementation-1

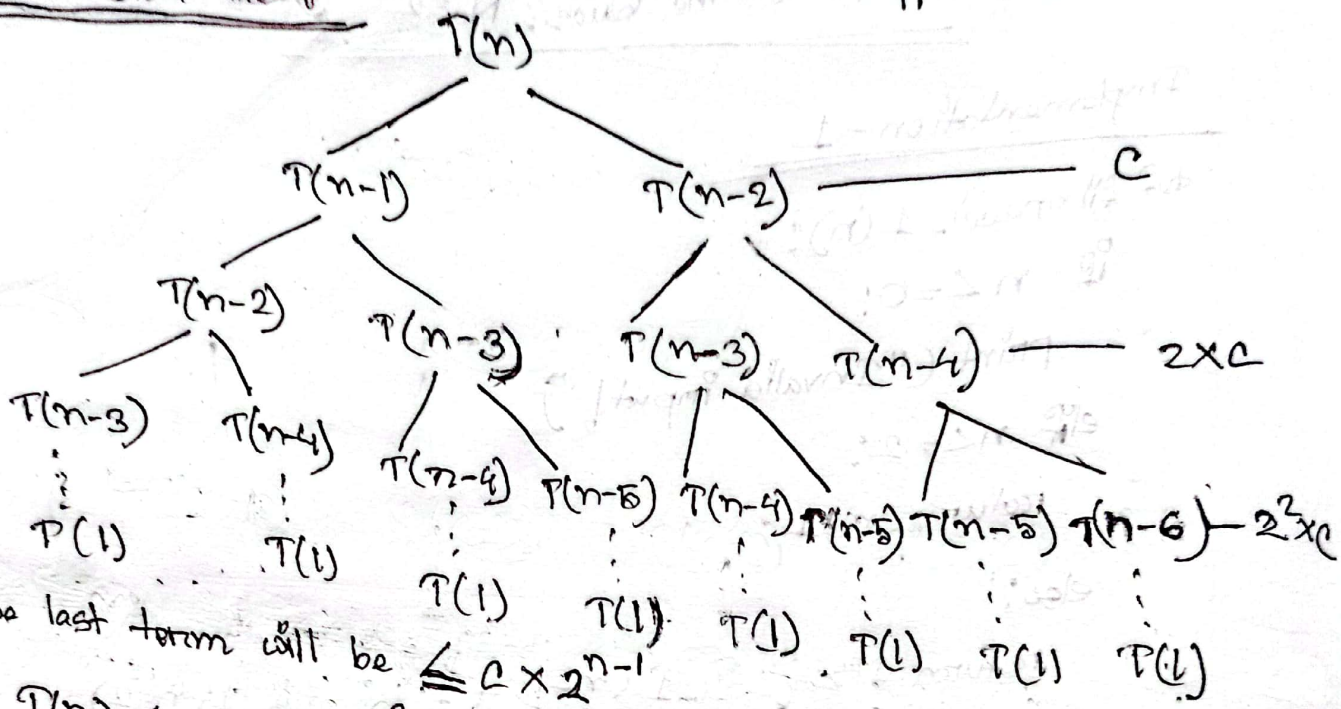
```
def fibonacci_1(n):
    if n<=0:
        print("Invalid input!")
    elif n<=2:
        return n-1
    else:
        return fibonacci_1(n-1) + fibonacci_1(n-2)
n= int(input("Enter a number: "))
n_th fib = fibonacci_1(n)
print("The %d-th fibonacci number is %d" % (n, n_th_fib))
```

The above implementation is a native recursive approach. This code implementation operates on two sub-problems. The size of one sub problem is (size of the original problem -1) and the size of other sub problem is (size of the original problem-2) Then the above implementation performs O(1) additional work. So, the running time equation we can have,

$$T(n) = T(n-1) + T(n-2) + 1 ; T(0)=1, T(1)=1$$

Recursion Tree:

Suppose, $c = 1$



The last term will be $\leq c \times 2^{n-1}$

$$T(n) \leq c + 2c + 2^2c + 2^3c + \ldots + 2^{n-1}c$$

$$\Rightarrow T(n) \leq c \times \{1 + 2 + 2^2 + \ldots + 2^{n-1}\}$$

$$\Rightarrow T(n) \leq c \times \{2^n - 1\}$$

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

Implementation-2

```
def fibonacci_2(n):
    fibonacci_array = [0, 1]
    if n<0:                                  # executes constant time — O(1)
        print("Invalid input!")
    elif n<=2:                               # executes constant time — O(1)
        return fibonacci_array[n-1]
    else:
        for i in range(2,n):                 # executes n times — O(n)
            fibonacci_array.append(fibonacci_array[i-1]+fibonacci_array[i-2])
        return fibonacci_array[-1]
n=int(input("Enter a number:"))
nth_fib = fibonacci_2(n)
print("The %d-th fibonacci number is %d" (n, nth_fib))
```

So, time complexity $= O(1) + O(1) + O(n)$

$= O(n)$

The above implementation is faster than implementation-1 ($O(n) < O(2^n)$). Basically, in case of implementation-2, we iterate 'n' number of times to find the nth fibonacci number. That is why, time complexity is $O(n)$.

On the other hand, in the implementation 1, if $n>2$, then

$T(n) = T(n-1) + T(n-2) + 1$. Because each recursion would call two further recursions. This aspect increases the time complexity exponentially. Hence, we get $O(2^n)$ as the time complexity of implementation-1.

Ans. To The Ques. No: 4

```
Procedure Multiply_matrix(A,B)
    Input A,B nxn matrix
    Output C nxn matrix.
begin
    Initialize C as a nxn matrix
    for i=0 to n-1 ---------- n
      for j=0 to n-1 ---------- nxn
        for k=0 to n-1 ---------- nxnxn
            C[i,j] += A[i,k] * B[k,j]
        end for
      end for
    end for
end Multiply_matrix
```

In the above algorithm, the inner, middle and outer loop

excute n times. Basically, there are three nested loops.

∴ The time complexity is $O(n^3)$

Ans. To the Ques. No-5

Recursion Tree Time Complexity

1] $T(n) = T(n/2) + n - 1$; $T(1) = 0$

Using Master Theorem for $T(n/2) + n$ part,

$T(n)' = T(n/2) + n$

After comparing $T(n)'$ with $T(n) = aT(n/b) + f(n)$; $f(n) = \Theta(n^k \log^p n)$, we get, $p = 0$, $a = 1$, $b = 2$, $k = 1$

Therefore, $T(n)' = n$; [Since $\log_b a < k$ and $p \geq 0$]

∴ $T(n) = n - 1$

So, time complexity is $O(n)$.

2. $T(n) = T(n-1) + n-1 \; ; T(1) = 0$

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

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

$\Rightarrow T(n) = T(n-k) + (n-k) + (n-(k-1)) + \ldots + (n-2) + (n-1)$ [In kth term]

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

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

Assume,

$n - k = 1$

$\Rightarrow n = k+1$

$\therefore T(n) = T(1) + 1 + 2 + \ldots + (n-2) + (n-1)$

$= 0 + \frac{n(n-1)}{2}$

$= \frac{n^2}{2} - \frac{n}{2}$

Therefore, time complexity is $O(n^2)$.

3. $T(n) = T(n/3) + 2T(n/3) + n$

Using Master Theorem in $2T(n/3)+n$ part,

$T(n)' = 2T(n/3) + n$

After comparing $T(n)'$ with $T(n) = aT(n/b) + f(n)$,

we get, $f(n) = \Theta(n^k \log^p n)$

$a = 2, b = 3, p = 0, k = 1$

$\therefore T(n)' = O(n)$; [Since $\log_b a < k$ and $p \geq 0$]

So, time complexity $= O(n)$.

Again, using Master Theorem,

$T(n) = T(n/3) + n$

We get, $a = 1, b = 3, p = 0, k = 1$

$\therefore T(n) = O(n)$; [Since $\log_b a < k$ and $p \geq 0$]

Therefore, the time complexity is $O(n)$.

(Ans)

[4] Given, $T(n) = 2T(n/2) + n^2$

Using Master Theorem,

$a = 2,\ b = 2,\ k = 2,\ p = 0$ [Comparing with $aT(n/b) + f(n)$, Here, $f(n) = n^k \log^p n$]

$\therefore T(n) = O(n^2)$; [Since $\log_b a < k$ and $p \geq 0$]

Therefore, the worst case complexity will be $O(n^2)$.

(Proved)