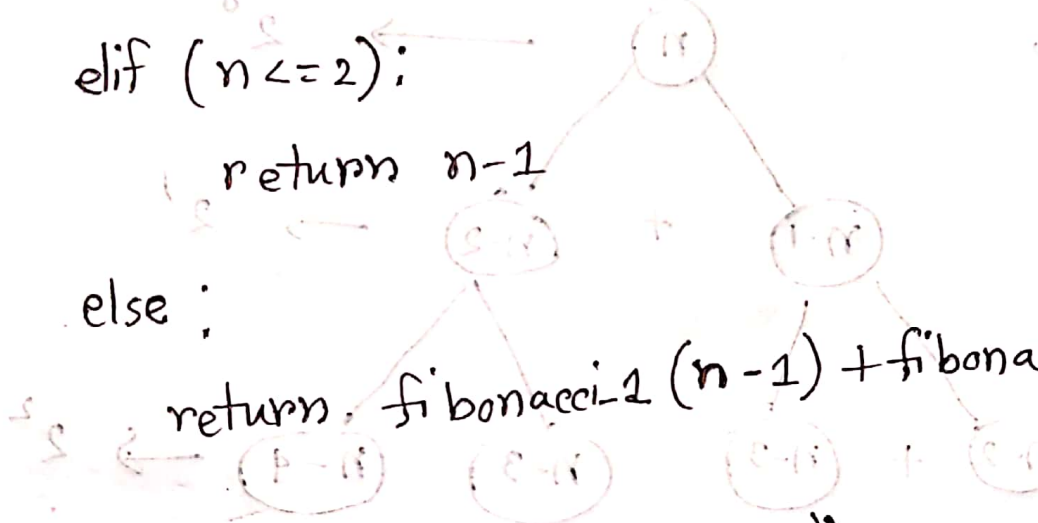
Answer to the question no. 2

Implementation 1

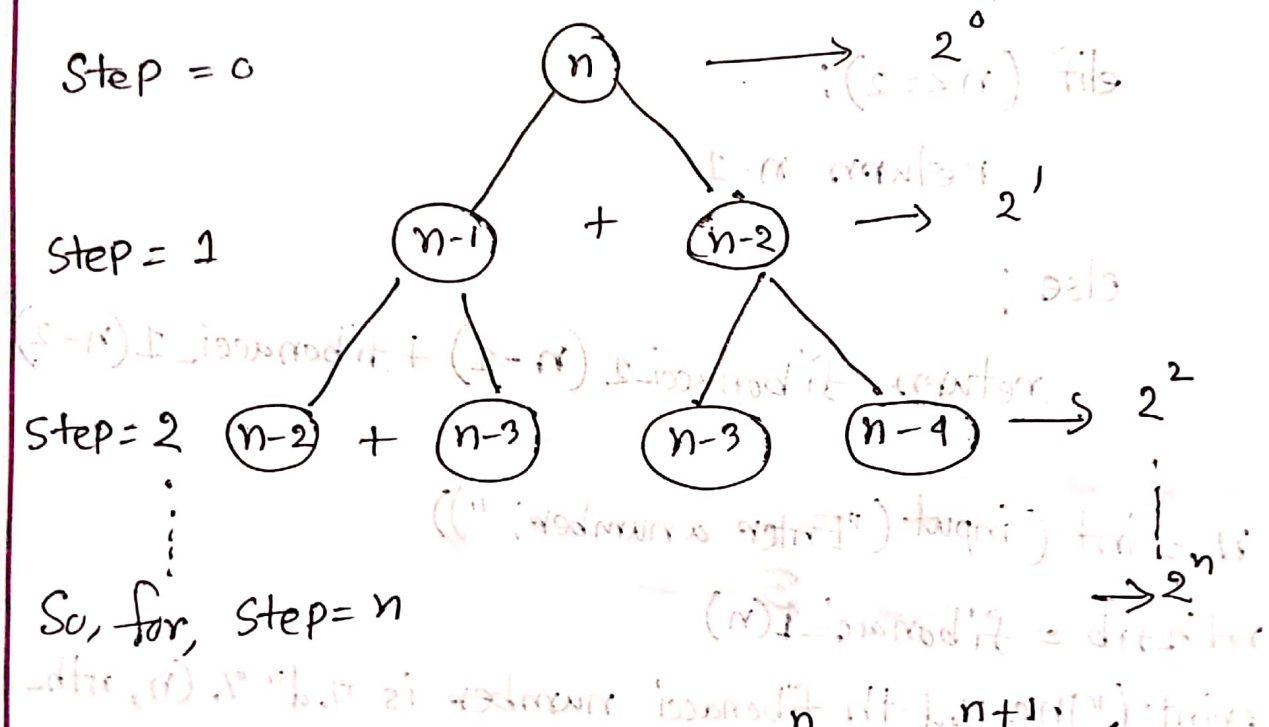
```
def fibonacci_1(n):
    if (n<=0):
        print("Invalid input!")
    elif (n<=2):
        return n-1
    else:
        return fibonacci_1(n-1)+fibonacci_1(n-2)
n = int(input("Enter a number: "))
nth_fib = fibonacci_1(n)
print("The %d-th fibonacci number is %d" %(n, nth_fib))
```



Here,

$$T(n) = \begin{cases} 1 & n <= 0 \\ 1 & n <= 2 \\ T(n-1) + T(n-2) & n > 2 \end{cases}$$

Using recursive tree:



So, for, Step = n

So, $2^0 + 2^1 + 2^2 + \ldots + 2^n = 2^{n+1} - 1$

$\therefore O(2^{n+1} - 1) = O(2^n)$

Implementation - 2

```
def fibonacci_2(n):
    fibonacci_array = [0,1]
    if n<0:
        print("Invalid input!")

    elif n<=2:
        return fibonacci_array[n-1]

    else:
        for i in range(2,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))
```

P.T.O

$\therefore O(1) + O(1) + O(1) + O(n) \times O(1)$

$= O(1) + O(1) + O(1) + O(n)$

$= O(n)$

So,
Time complexity for 1st implementation is $O(2^n)$.

And, for the 2nd implementation is $O(n)$.

So, by comparing these two, 2nd implementation $O(n)$ is better than the 1st one $O(2^n)$.

Ans no : 9

i	j	k
0	0	n
0	1	n
0	2	n
⋮	⋮	⋮
0	n	n
⋮	⋮	⋮
n	n	n

Here in the Multiply_matrix method, when i=0, ~~then~~ j=0, then the k loop will run for n times. similarg, when i=0, j= loop will run for n times. Also, Outer loop will run for n times So, there are 3 loops in the method which will run for n times which makes the complexity of program $O(n^3)$. (Ans')