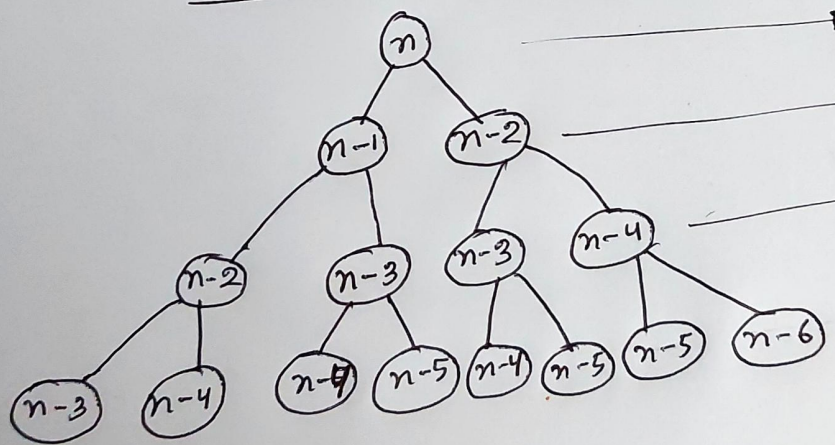
Answer to the question no. 2(a)

implementation 1:

```
def fibonacci_1(n):
    if n<0:
        print("Invalid Input!")
    elif n<=1:
        return n
    else:
        return fibonacci_1(n-1)+fibonacci_1(n-2)
```

Function Call Stack



Function Calls

$1 = 2^0$

$2 = 2^1$

$4 = 2^2$

$8 = 2^3$

2^n

Total Function Calls $= 2^0 + 2^1 + 2^2 + \cdots + 2^n$

$= 2^{n+1} - 1$

$\therefore$ Time taken by the algorithm $= k * (2^{n+1} - 1)$

$= k(2^n * 2 - 1)$

$= 2^n * 2 \times k - 1 * k$

$= 2^n * c$

$= 2^n$

$\therefore$ The Time complexity for implementation 1 is $O(2^n)$

(Ans)

implementation 2:

```
def fibonacci_2(n):
    if n<0:
        return "Invalid Input"

    if n<=1:
        return n

    fib = [0] * (n+1)
    fib[0] = 0
    fib[1] = 1
    for i in range(2, n+1):
        fib[i] = fib[i-1] + fib[i-2]
    return fib(n)
```

In this implementation, the inner loop runs for ~~only~~ $(n-1)$ iterations $\approx (n)$ iterations.

∴ The time complexity for the algorithm is $O(n)$.

(Ans)

Comparison

The recursive approach for finding n^{th} ~~fibonacci~~ fibonacci number takes 2^n time. This can be assumed as a bad approach to solve this problem. Even finding just the 50^{th} fibonacci number would take a massive time.

On the other hand, implementation 2 uses memoization ~~technique technique~~ technique to solve the problem. Even though it requires extra space, but still the time complexity for this algorithm is much better → $O(n)$

So, all in all, implementation 2 is a much better algorithm to choose in order to find n^{th} fibonacci number.

(Ans)