* Problem statement :-

→ Calculate the n^{th} fibonacci number.

→ $F(n) = F(n-1) + F(n-2)$ for $n > 1$ where $F(0) = 0$ & $F(1) = 1$

* Test cases :-

$n = 0$ $\quad\quad$ $n = 1$

o/p = 0 $\quad\quad$ o/p = 1

$n = 2$

o/p = $f(2) = f(1) + f(0) = 1$

$n = 3$

o/p = $f(3) = f(2) + f(1) = 1 + 1 = 2$

* Solution :-

→ There a various approaches to solve this problem.

1) Approach 1 :- Binet's formula

→ This formula is used to calculate the n^{th} fibonacci number directly.

→ The n^{th} fibonacci number can be written as :-

$$F(n) = \frac{\phi^n - \psi^n}{\phi - \psi}$$

where, $\phi = \dfrac{1 + \sqrt{5}}{2}$ (The golden ratio)

$\psi = \dfrac{1 - \sqrt{5}}{2}$ (The conjugate of the golden ratio)

→ You can explore the derivation later.

```java
public static binetFormula (int n) {
    double sqrt5 = Math.sqrt(5);
    double phi = (1+ sqrt5) / 2;
    double psi = (1- sqrt5) / 2;
    double result = (Math.pow(phi, n) - Math.pow(psi, n)) / sqrt5;
    return (int) Math.round(result);
}
```

TC = SC = O(1)

2) Approach 2 :- Simple iterative approach

- **Algorithm**

→ If n is 0 or 1, return n.

→ Initialize prev ← 0 , curr ← 1

→ For i from 2 to n do:

    next ← prev + curr

    prev ← curr

    curr ← next

→ Return curr

TC :- O(n)

SC :- O(1)

3) Approach 3 :- Simple recursive approach

$$F(n) = \begin{cases} 0, & \text{if } n=0 \\ 1, & \text{if } n=1 \\ F(n-1) + F(n-2), & \text{if } n>1 \end{cases}$$

→ If n is 0 or n is 1, return n.

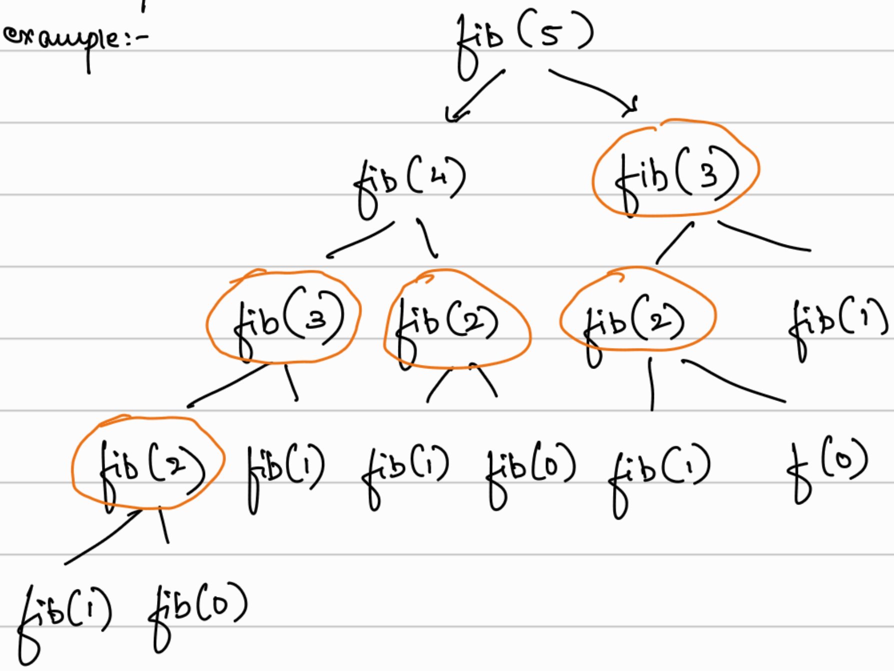
→ Return fibo (n-1) + fibo (n-2)

TC :- $O(2^n)$

SC :- $O(n)$ → Because of call stack

→ Each call spawns two or more calls.

→ For example:-



→ We see that there are many repeated calls.

→ The no. of function calls doubles almost at every level forming a binary recursion tree with $\sim 2^n$ nodes.

→ ∴ Total calls ≈ nodes in binary tree ≈ $O(2^n)$

→ In recursion, space comes from the call stack where we see how many nested calls exist at a time.

→ The max call stack depth from the root to the leaf :-

fib (n) → fib (n-1) → fib (n-2) → fib(n-3) → ... → fib (1) = n calls deep

∴ Max depth of recursion tree = n

∴ SC :- $O(n)$

4) Approach 4 :- Matrix exponentiation

→ Read about this later & explore more problems where you can use this approach.

→ TC :- $O(\log n)$

SC :- $O(1)$