2> Implementation - 1

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

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

Best case : $T(1) = T(0) = 1$

Assumption : $T(n-2) \lesssim T(n-1)$

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

$T(n-1) = 4T(n-2) + 2c$

$= 2^2 T(n-2) + (2^{2-1})c$

$T(n-2) = 8T(n-3) + 4c$

$= 2^3 T(n-3) + (2^{3-1})c$

$\therefore T(n) = 2^K T(n-K) + (2^{K-1})c$ $\quad T(0) = 1$

$n - K = 0$

$n = K$

$T(n) = 2^n T(0) + (2^{n-1})c$

$T(n) = 2^n + (2^{n-1})c$

$T(n) \propto 2^n$

$\therefore$ time complexity $= O(2^n)$

Implementation-2

```
def fibonacci_2(n):                          O(1)
    fibonacci_array = [0, 1]                 O(1)
    if n < 0:
        print ("Invalid Input!")
    elif n <= 2:                             O(1)
        return fibonacci_array [n-1]
    else:
        for i in range (2, n):
            fibonacci_array [-1]             O(1)
```

Time complexity $= O(1) + O(n-2)$
$= O(n)$

We can observe that in Implementation-1, the time complexity is $O(2^n)$ which is in exponential time whereas in Implementation-2 the time complexity is $O(n)$ which is in linear time. Hence, implementation-2 is more efficient.

4> Pseudocode

```
Procedure Multiply_matrix (A, B)
      Input  A, B    n x n  matrix
      Output C       n x n  matrix

begin
      Initialise C as a n x n zero matrix

      for i = 0 to (n-1)            O(n)
          for j = 0 to n-1          O(n)
              for k = 0 to n-1           O(n)
                  c[i, j] += A[i, k] x B[k, j]
              end for
          end for
      end for
end Multiply_matrix
```

time complexity $= O(n^k)$ $k = 3$

$= O(n^3)$