DESIGN ANALYSIS AND ALGORITHM

LAB 3

FIBONACCI AND GCD SERIES

NAME: NAOMI GEORGE

SLOT: L25+L26+L33+L34+L13+L14

REGISTRATION NO. : 19BCE7572

COURSE CODE: CSE3004

CODE:

```
(NAIVE GCD)
import java.lang.Math;
import java.util.Scanner;
public class Main{
 static int GCD(int a,int b){
 int maximum=Math.max(a,b);
 int currentNumber=maximum-1;
 while(currentNumber>1){
 if((a%currentNumber==0)&&(b%currentNumber==0)){
 return currentNumber;
 }else{
 currentNumber--;
 }
 }
 return 1;
 }
 public static void main(String[] args){
 System.out.println(GCD(245,65));
 }
}
```

OUTPUT:

CODE:

```
(EUCLIDEAN GCD)
import java.util.*;
import java.lang.*;
public class Main
{
 public static int gcd(int a, int b)
 {
 if (a == 0)
 return b;

 return gcd(b%a, a);
 }
 public static void main(String[] args)
 {
 System.out.println(gcd(936574,736572));
 }
}
```

OUTPUT:

ANALYSIS:

GCD :

Recursively it can be expressed as :

$$gcd(a,b) = gcd(b, a\%b) \qquad [a,b : \text{integers}]$$

Suppose, a & b are 2 integers such that $a > b$ then according to Euclid's Algo :

$$g(a,b) = gcd(b, a\%b)$$

Using the above formula, repetitively until reaching a step, $b = 0$. At this step, the result will be the GCD of 2 integers, will be equal to a. So, it can be said,

time complexity $T(n) \propto$ no. of steps req. to reduce b to 0

Assuming,

$gcd(a,b)$ ----> N steps

Now, if Euclidean Algo for 2 nos. a & b reduces in N steps then, a should be at least

$f(N+2)$ & $b = f(N+1)$

$\Rightarrow$ $gcd(a,b)$ ----> N steps

Then, $a >= f(N+2)$ & $b >= f(N+1)$

where f_N is the N^{th} term in fibonacci series & $N > 0$

∴ Using Principle of Mathematical Induction,

Lets assume $a = 2$ & $b = 1$

then $gcd(2,1) \Rightarrow gcd(1,0)$ in 1 step (gets reduced)

$\Rightarrow N = 1$

This means a should at least f_3 & 1 should be at least f_2 & $f_3 = 2$ & $f_2 = 1$

$\Rightarrow$ a is at least $f_{(N+2)}$ & b is at least $f_{(N+1)}$

∴ It is proved that if the Euclidean algorithm for a number a & b reduces in N steps then,

a is at least $f_{(N+2)}$

& b is at least $f_{(N+1)}$

$\Rightarrow$ $f_{N+1} \approx \min(a,b)$

$N+1 \approx \log_\phi \min(a,b)$

$O(N) = O(N+1)$

$= \log(\min(a,b))$

———.

CODE:

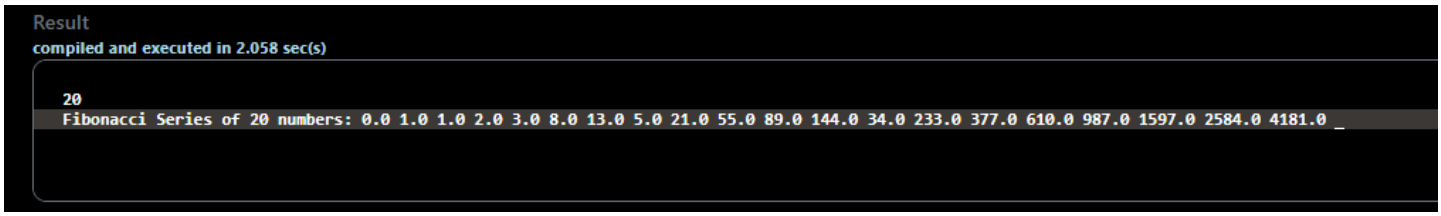
```
(EFFICIENT FIBONACCI SERIES)
import java.util.Scanner;
public class FibonnaciEff{
    static int fib(int n){
        int f[]=new int[n+2];
        int i;
        f[0]= 0;
        f[1]= 1;
        for(i=2; i<=n; i++){
            f[i]=f[i-1]+ f[i-2];
        }
        return f[n];
    }
    public static void main(String args[])
    {
        Scanner sc = new Scanner(System.in);
        int m= sc.nextInt();
        for(int i = 0; i < m;i++)
        System.out.println(fib(i)+ " ");
    }
}
```

OUTPUT:

CODE:

```
(NAIVE FIBONACCI)
import java.util.Scanner;
public class Fibonacci{
        public static double fibRec(double n){
        if(n == 0){
                return 0;
        }
        if(n == 1 || n == 2){
                        return 1;
                }
        return fibRec(n-2) + fibRec(n-1);
        }
    public static void main(String args[]) {
        Scanner sc= new Scanner(System.in);
        int m= sc.nextInt();
        System.out.print("Fibonacci Series of "+m+" numbers: ");
        for(int i = 0; i < m; i++){
                        System.out.print(fibRec(i) +" ");
```

OUTPUT:



ANALYSIS:

(next page)

Fibonacci : (Name)

Let F(n) = returns the value of F_n.

To evaluate F(n) for $n > 1$, we can reduce our problem into 2 smaller problems of the same kind

$F(n-1)$ & $F(n-2)$

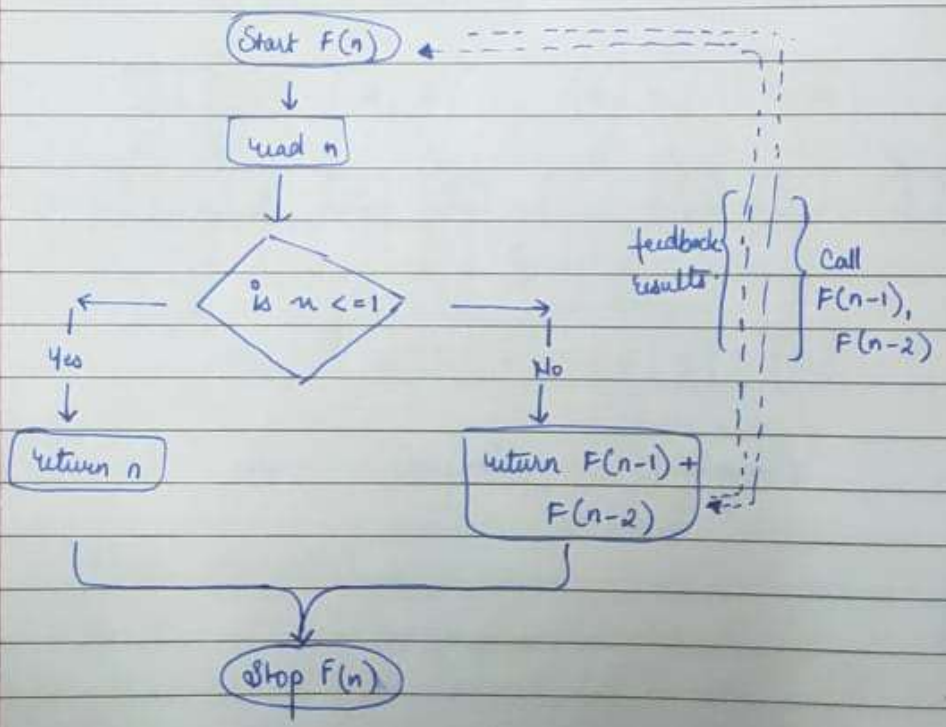
$\Rightarrow$ $F((n-1)-1)$ & $F((n-1)-2)$ &

$F((n-2)-1)$ & $F((n-2)-2)$ resp.

If we repeat this reduction, we'll get our base cases & get a solution to F(n)

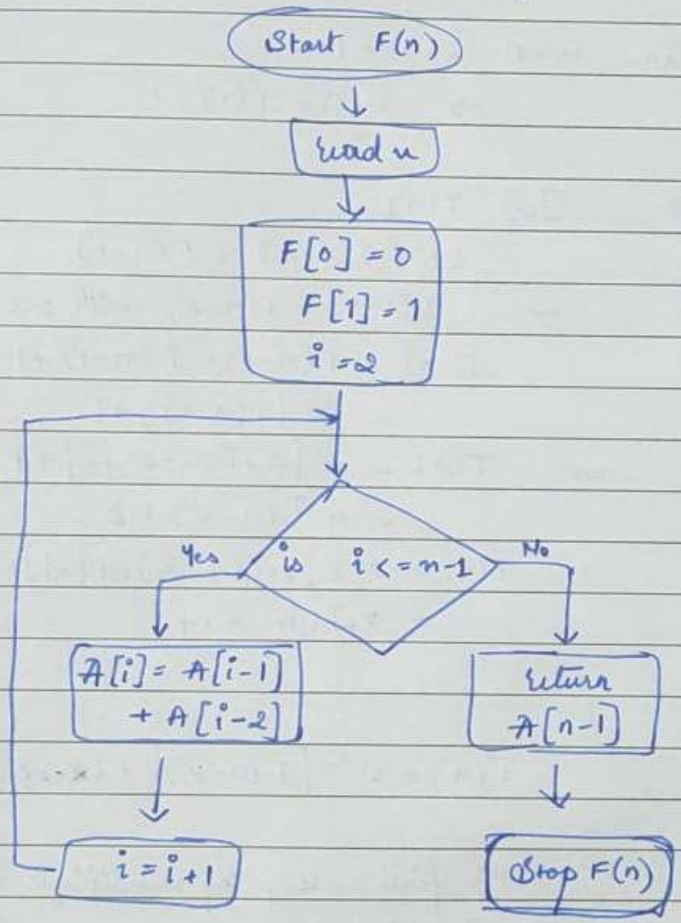
Employing this logic, F(n) will have 2 steps:

1. Check if $n \leq 1$. If so, return n.
2. Check if $n > 1$. If so, call our func^n F with inputs $n-1$ & $n-2$ & return the sum of 2 results.



Efficient fibonacci :

First, we'll store 0 & 1 in F[0] & F[1], resp.
Next, we'll iterate through array positions 2 to n-1.
At each position i, we store the sum of the 2 preceding array values in F[i].
Finally, we return the value of F[n-1], giving us the number at position n in the sequence.



The time complexity in this case, is assignment. Firstly our assignment of F[0] & F[1] cost O(1) each.

Secondly, our loop performs one assignment per iteration & executes $(n-1)-2$ times, costing a total of $O(n-3)$

$\therefore \Rightarrow O(n)$