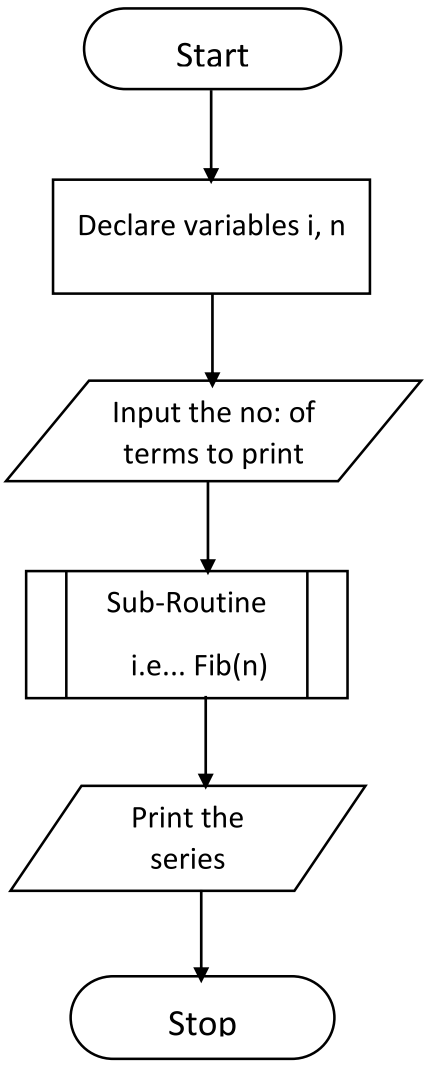
1.Fibonacci series using recursion:

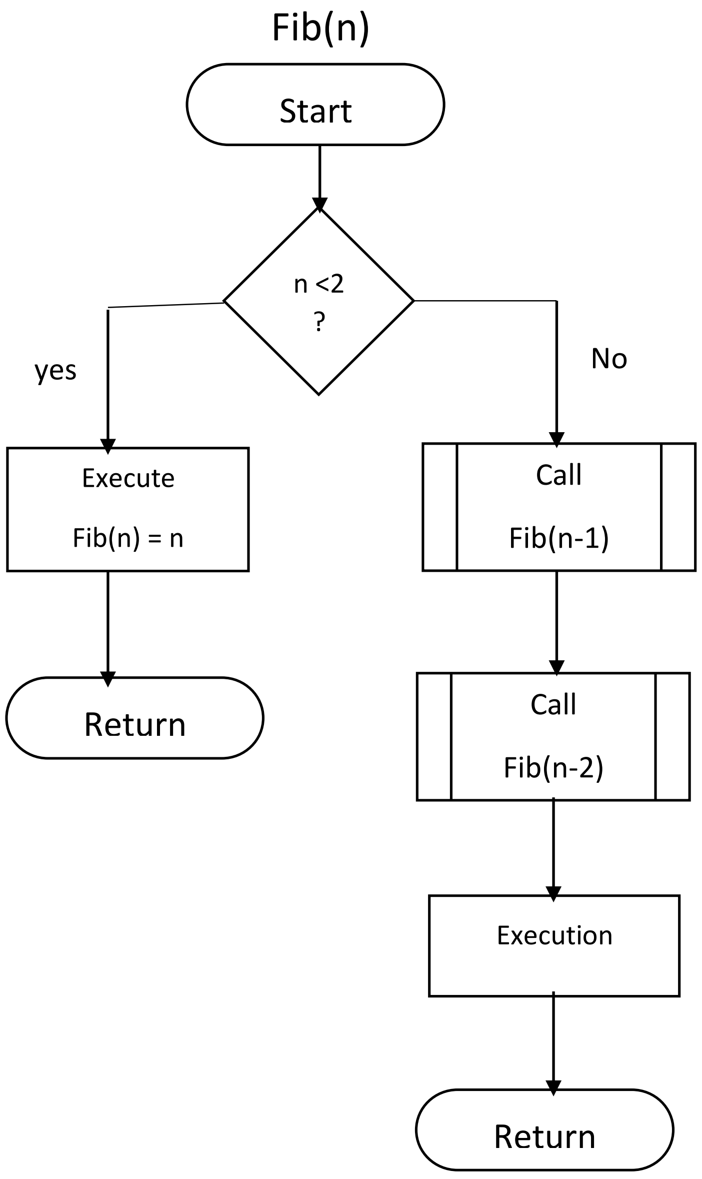
Fibonacci series:

fib(n) = fib(n-1) + fib(n-2) ; for n>1

fib(n) = 1 ; for n=0,1

Flowchart:





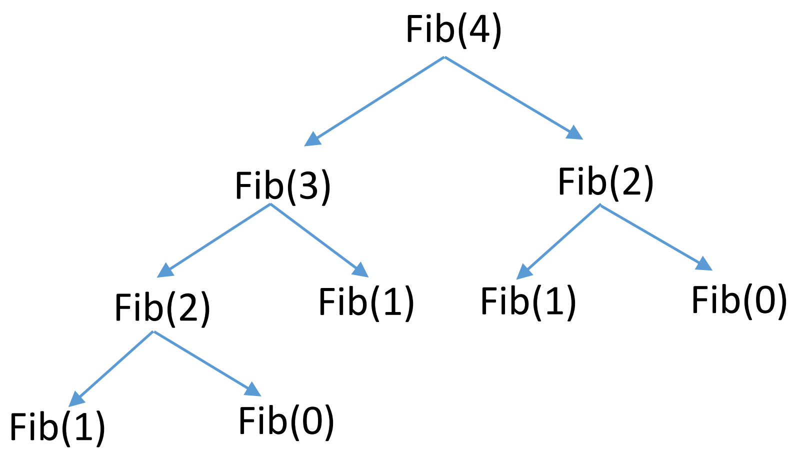
Algorithm (Pseudo code):

```
• Start
• Declare variables i, n
• Initialize i=1
• Read value of 'n' from user
• Subroutine Fib(n)
   Begin
          If n<2 then
                 Return n;
          Else
                 Return Call Fib(n-1) + Call Fib(n-2)
   End

• Repeat until i<n
     Print Fib(i)
      i=i+1

• Stop
```

Total Memory/Space required:

The recursive approach looks easy and simple, but it's not as it calculates the Fibonacci of a number multiple times. Here the space required is proportional to the maximum depth of the recursion tree.

For example, recursion tree for Fibonacci(4) would be:



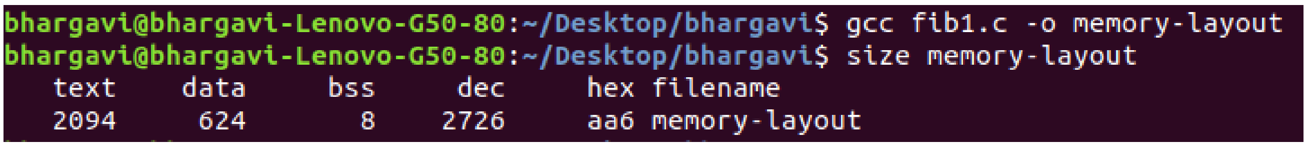
From the above recursion tree for fib(4), it is clearly visible that to calculate fib(4), we need to calculate the Fibonacci of all the branches. So there will be ‘n’ recursive calls and thereby ‘n’ stacks will be used. Hence the space required for finding Fibonacci using recursion is O(n)

- Therefore, space complexity is O(n)

Program code: (Using Recursion)

```
#include <stdio.h>
long long int fib(int n)
{
        if(n<2)
                return n;
        else
                return fib(n-1)+fib(n-2);
}

int main()
{
        int n,i;
        printf("Enter number of terms: ");
        scanf("%d",&n);
        printf("fibonacci series: ");
        for(i=1;i<=n;i++)
        {
                printf("%lld, ",fib(i));
        }
        printf("\n\n");
        return 0;
```

Memory-layout of the above program: (In bytes)



```
bhargavi@bhargavi-Lenovo-G50-80:~/Desktop/bhargavi$ gcc fib1.c -o memory-layout
bhargavi@bhargavi-Lenovo-G50-80:~/Desktop/bhargavi$ size memory-layout
   text    data     bss     dec     hex filename
   2094     624       8    2726     aa6 memory-layout
```

Memory required for the program:

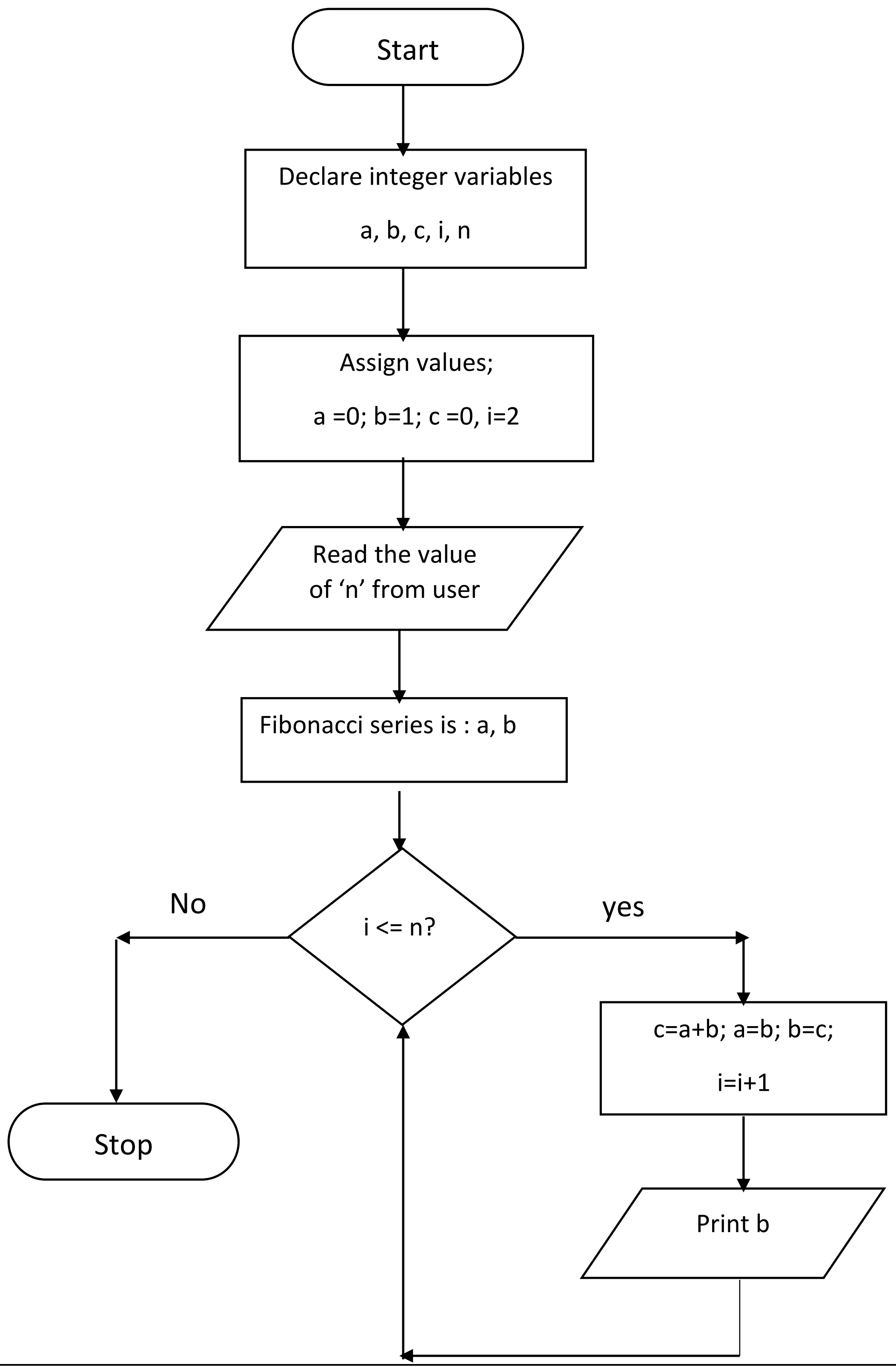
There are two integer declarations ‘i’ and ‘n’. And they are uninitialized, hence they will be stored in **bss segment** of the memory. And size of an integer is complier and program language dependent. Here it takes 4 bytes for int, hence two integers take 8 bytes. And during the execution of ‘for loop’ in the program it calls fib(n) function which creates a stack frame each time it is called. And inside the for loop there is only printing operation which takes constant space Therefore, total space occupied would be O(1*n). This can be taken as O(n) for the worst case scenario.

Observations:

While executing the program with n=5 there was an instant output, while executing again with n=50 it took nearly 3 minutes to give output. So we can understand that the reason behind this is because of the recursive calls which take time to get executed. So proceeding further to calculate fibonacci series for n=500, it would take some hours to print the output.

2. Fibonacci series using iterative method:

Flowchart:



Algorithm (Pseudo code):

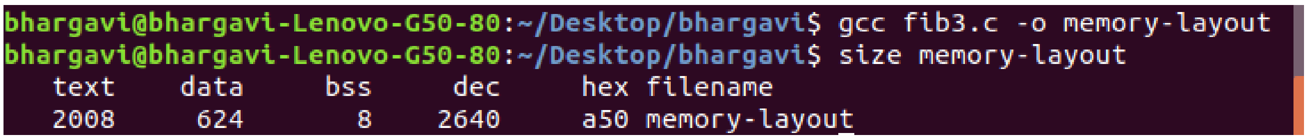
```
• Start
• Declare variables a, b, c, i, n
• Initialize variables with a=0, b=1, i=2
• Read 'n' value from user
• Print a and b
• Repeat until i<=n
     c = a+b
     a = b
     b = c
     i=i+1
• Stop
```

Program code: (Iterative approach)

```
#include <stdio.h>
int main()
{
      long long int a,b,c;
      int n;
      a=0;
      b=1;
      printf("Enter n: ");
      scanf("%d",&n);
      printf("Fibonacci series: ");
      for(int i=1;i<n;i++)
     {
         c = a+b;
         a = b;
         b = c;
```

```
        printf("%lld, ",b);
    }
    printf("\n\n");
    return 0;
}
```

Memory-layout of above program: (In bytes)



```
bhargavi@bhargavi-Lenovo-G50-80:~/Desktop/bhargavi$ gcc fib3.c -o memory-layout
bhargavi@bhargavi-Lenovo-G50-80:~/Desktop/bhargavi$ size memory-layout
   text    data     bss     dec     hex filename
   2008     624       8    2640     a50 memory-layout
```

Memory required for iterative approach:

The memory requirement on whole will be O(1) i.e. constant. Since the operations inside the for loop are only getting overwritten and just printing the value without storing in any datatypes, it only requires a constant space. Therefore, the best case and the worst case space complexity is O(1).

Space Complexity comparison table:

Recursive approach	Iterative approach
Best case: O(1)	Best case: O(1)
Worst case: O(n)	Worst case: O(1)

-----------xxxxxx-----------