How function calls work in languages

NOTE:- When the function is not finished executing, it remains in stack Memory.

Example

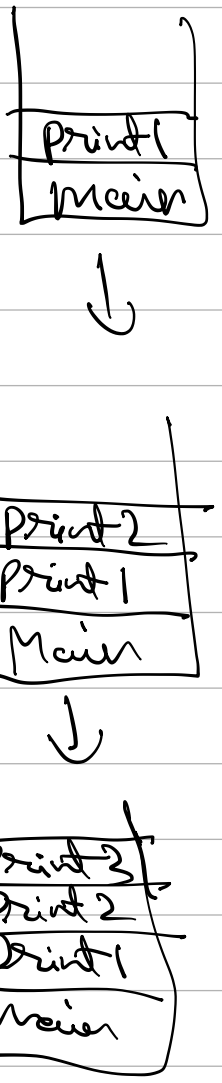
```
main() {
  print1();
}

print1() {
  ---------
  print2();
}

print2() {
  ---------
  print3();
}

print3() {
  ---------
}
```

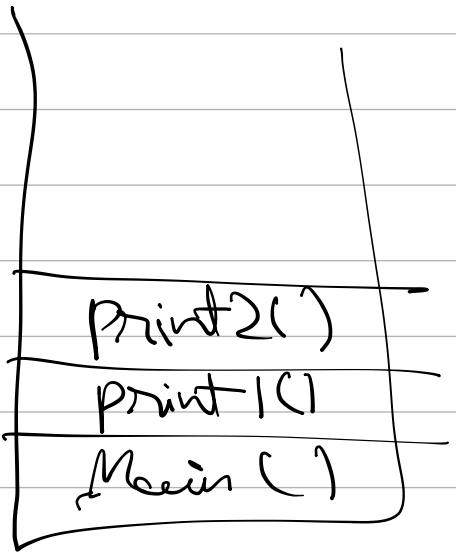
How stack works like this —



★ When a function finishes executing, it is removed from the stack and the flow of program is restored to where it was called.

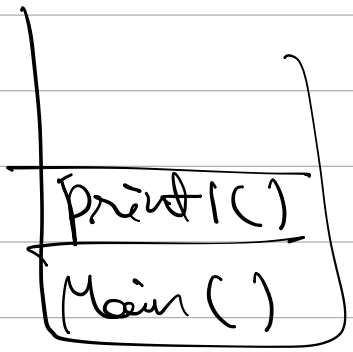
For example:- print3() doesn't call any other function and it finishes executing. So,

- remove print3() →

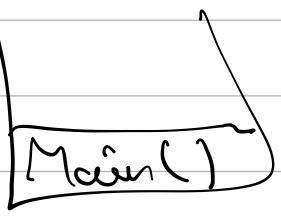


- Then remove print2() as nothing else need to be done.

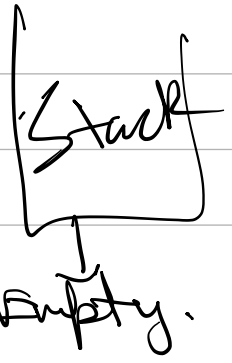
↳



- Then,



- Then,



Program to print first 5 numbers using Recursion.

```
psvm () {
    print (1);
}

Static void print(int n)
{
    if (n==6) {
        System.out.println();
        return;
    }
    System.out.println(n);
    print(n+1);
}
```

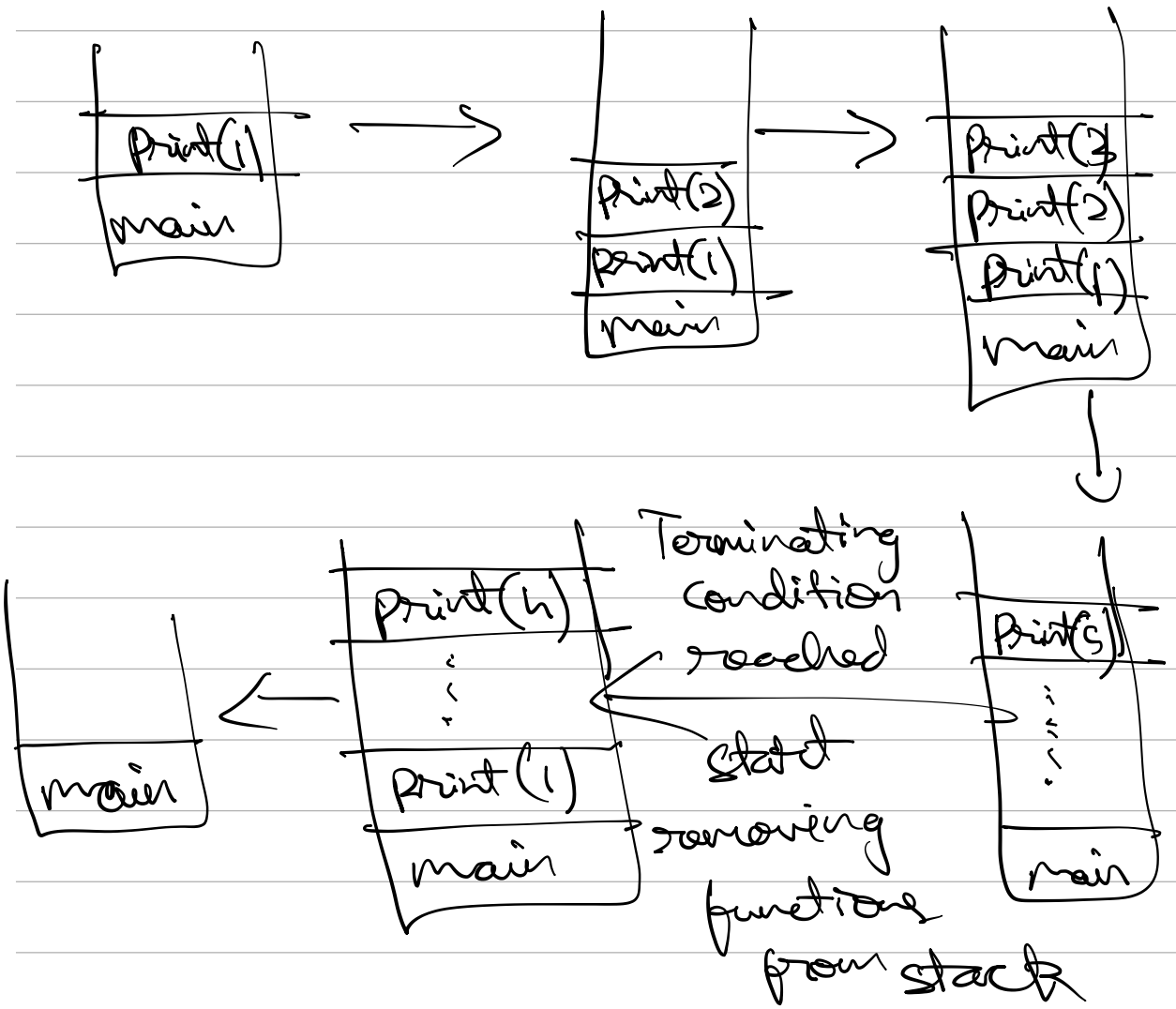
Terminating condition

function calling itself

Output:-

1
2
3
4
5

Stack diagram for above program



NOTE:- Even if you call same function

Each function call will take separate memory in stack. (i.e print() will be called again and again, with 1, 2, 3, 4, 5 as arguments).

☆ 7. If there is no base condition (i.e terminating condition), the function keeps on calling, and stack gets filling and it overflows → Stack overflow error

---

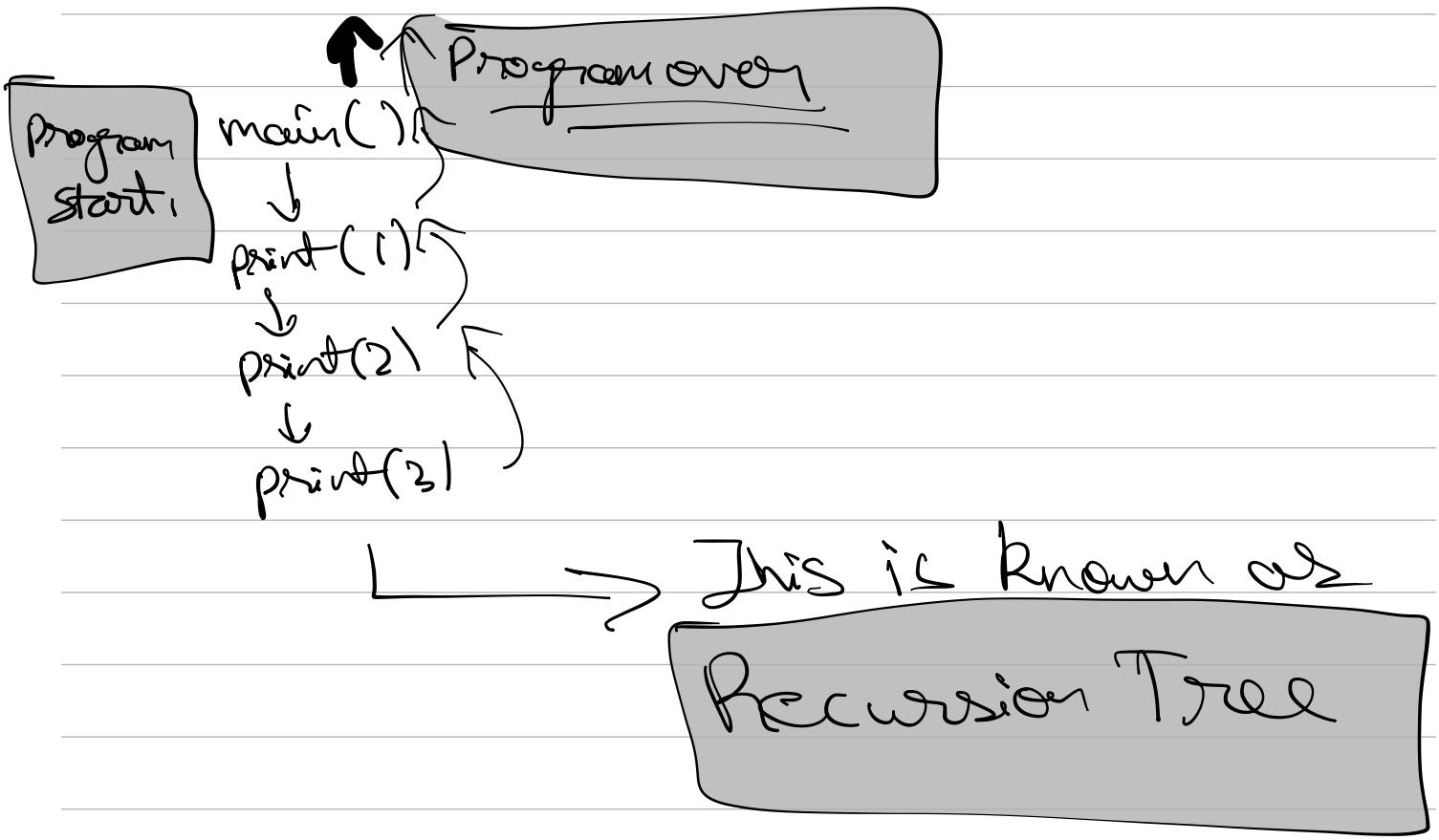
Recursion

→ function calling itself.

→ We can convert recursion to iterative solution & vice versa.

→ i.e we convert recursive soln to loops, to reduce space complexity. as. function calls occupy space.

Visualising Recursion



Q] Find n^{th} Fibonacci number.

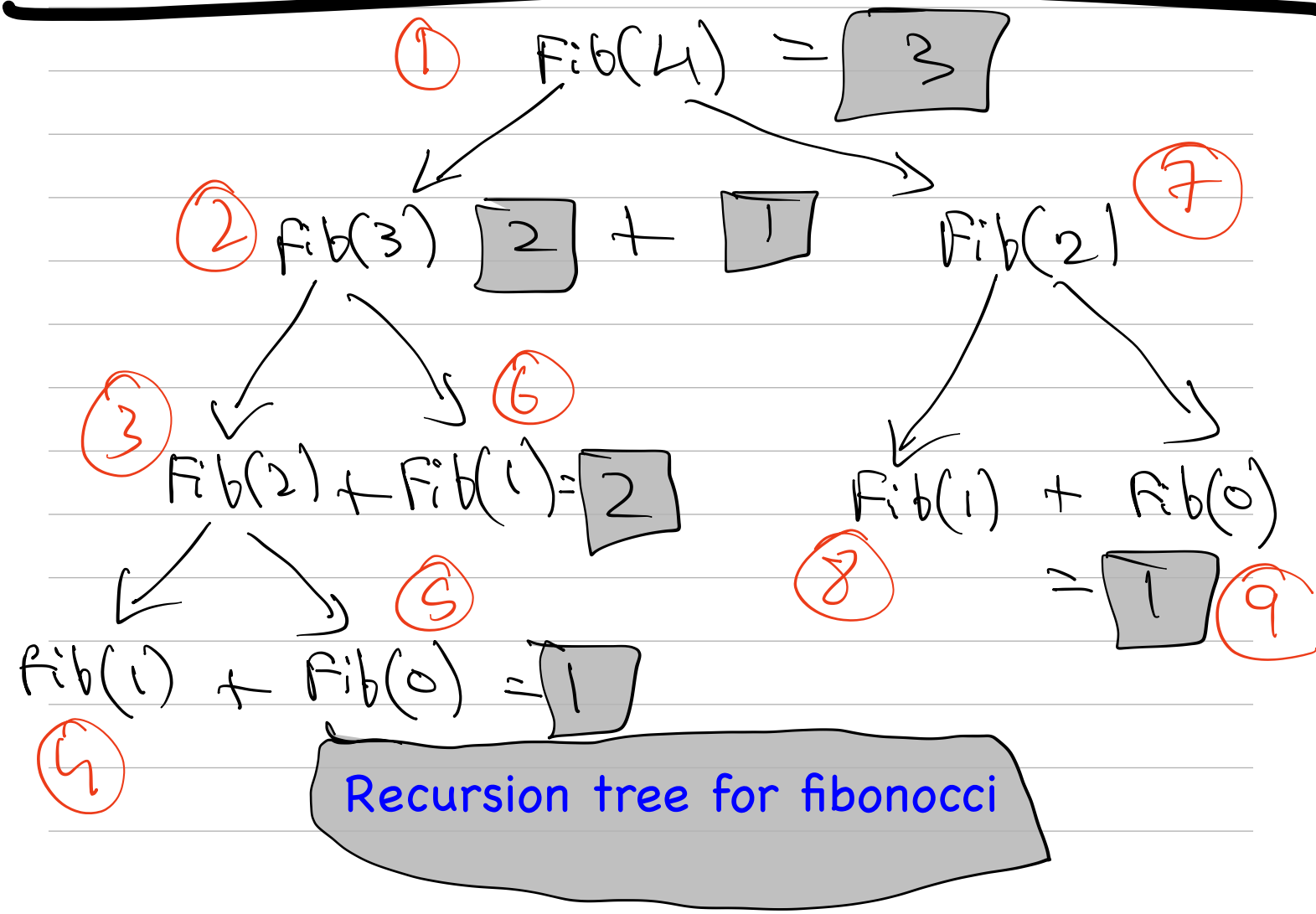
0^{th}	1^{st}	2^{nd}	3^{rd}	4^{th}	5^{th}
0	1	1	2	3	5

$$Fib(n) = Fib(n-1) + Fib(n-2)$$

Recurrance Relation

Ans:-

```
static void fib(int n)
{
    if(n==0)
        return 0;
    if(n==1)
        return 1;
    else {
    return fib(n-1)+fib(n-2);
}
```



NOTE:- How to understand and approach a problem.

1]. Identify if you can break a problem to smaller problems

2]. Write the recurrence relation if needed.

3]. Draw the recursive tree.

4]. About the tree.

*]. See the flow of functions. i.e how they are getting in the stack.

*]. Identify & focus on left tree calls and right tree calls.

- In the above tree, the order of called functions are indicated in red.

5] See how the values are returned at each step. See where the function call will come out in the end, you will come out of the main function.

Variables :- ① Arguments
② Return type
③ Body of function.

For Example :-

```
static int func( int a )
{
    // Body
}
```

Return type → int

Argument variable → int a

Variables in body ← // Body

Q. Binary search with recursion.

Step 1: Recurrence relation

$$F(N) = O(1) + F\left(\frac{N}{2}\right)$$

- $F(N)$ → call input size.
- $O(1)$ → Comparison
- $F\left(\frac{N}{2}\right)$ → Dividing array to half.

Types of recursive relation

1. Linear recursive relation → fibonacci
2. Divide & Conquer recursive relation → Binary search

```
static int search(int[] arr, int target, int s, int e)
{
    if(s>e) {
        return -1;
    }
    int m = s + (e-s)/2;

    if (arr[m] == target)
        return m;
    if(target < arr[m]) {
        return search(arr, target, s, m-1);
    }
    else
        return search(arr, target, m+1, e);
}
```

TIP:- for return variables, include those variables which are useful for further function calls.

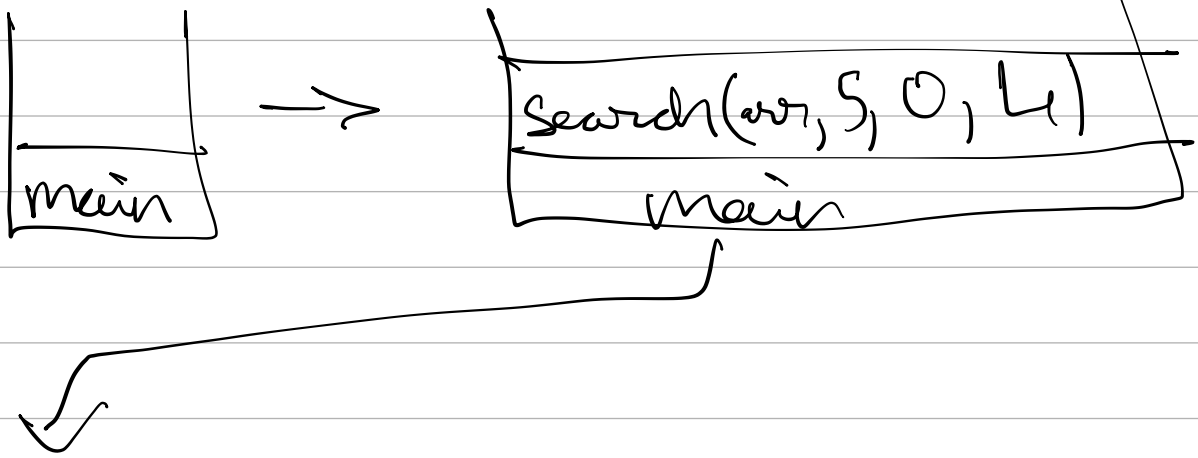
For example :- Here s and c are returned and also included in function arguments.

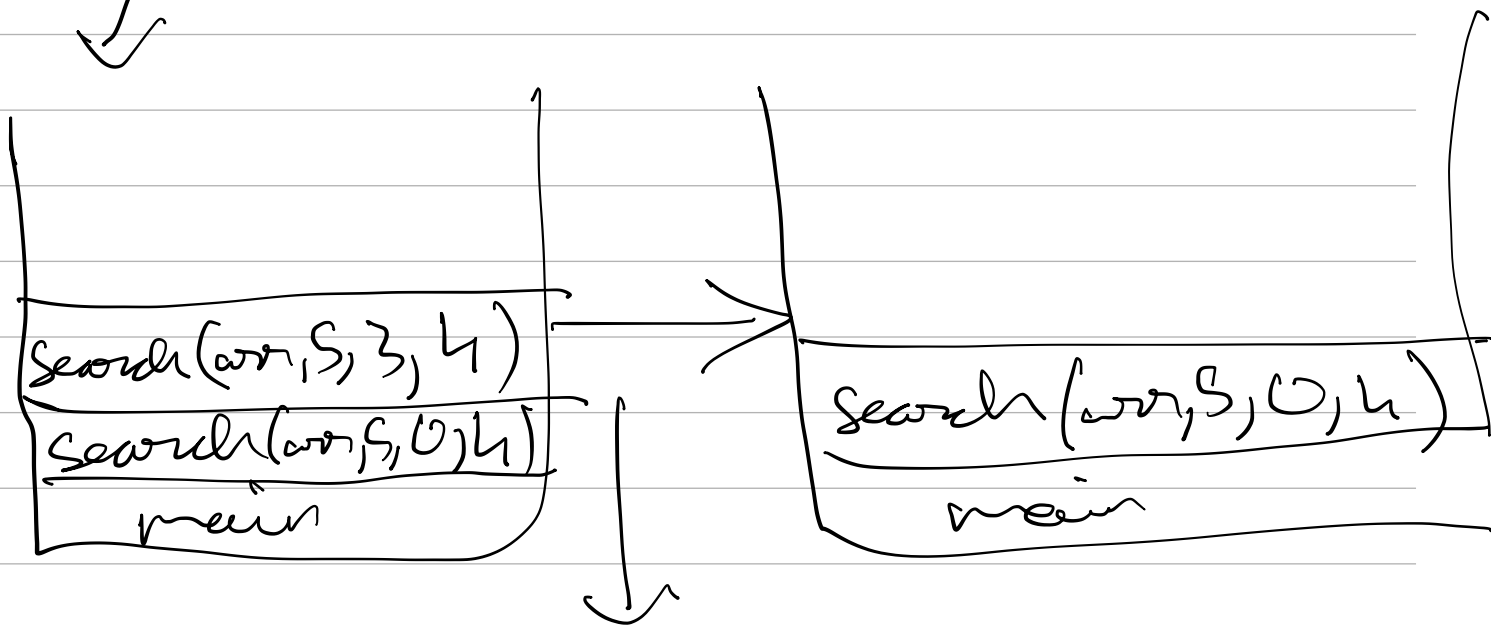
*) Here 'm' is body variable, no need to include in arguments or return variables.

For arr = [2, 3, 4, 5, 6]

Target = 5.

Stack Implementation





element found at

m = $\frac{3+4}{2}$ = 3rd position

So, search(arr, 5, 3, 4) returns 3 and function call exits stack.