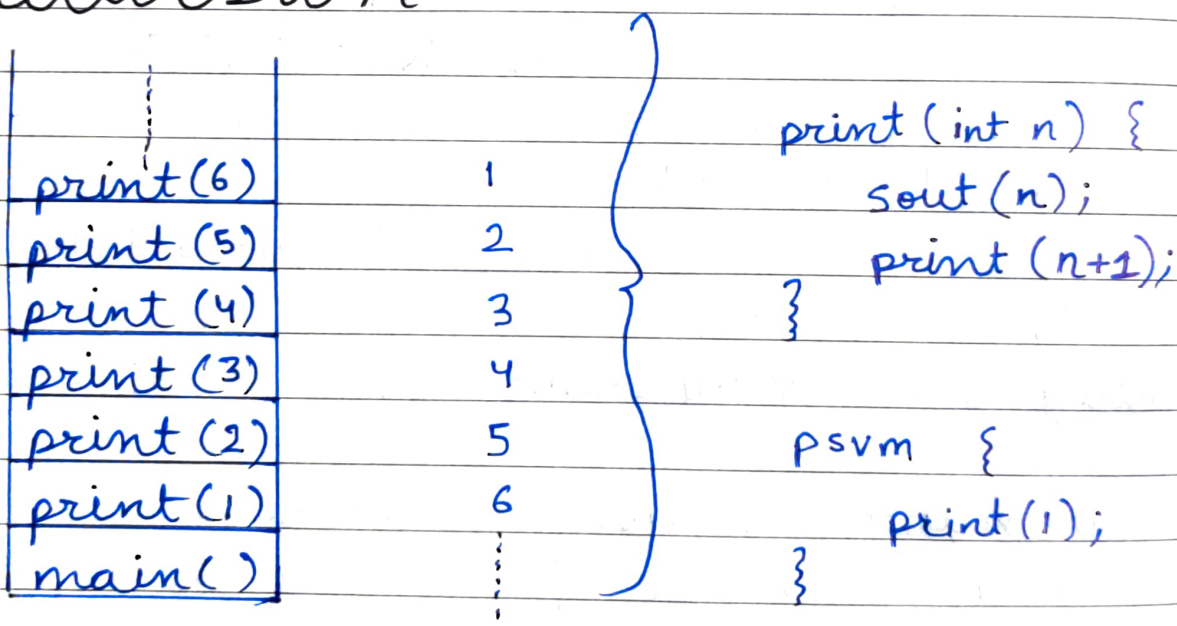
How function calls work in languages:

→ While the function is not finished executing, it will remain in stack.

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

Recursion



print() is calling itself but you're not stopping anywhere. So, in order to handle this problem we need a base condition.

Base condition in recursion is a condition where our recursion will

stop making new calls.

```
print(int n) {
    if (n == 5) {
        sout(5);
        return;
    }
    sout(n);
    print(n+1);
}
```

→ Base condition

If you're calling a function again and again you can treat it as a separate call in the stack. As many times as you call the function it will take memory separately.

No base condition → Function calls will keep happening, stack will be filled again and again → memory of computer will exceed the limit → Stack Overflow error

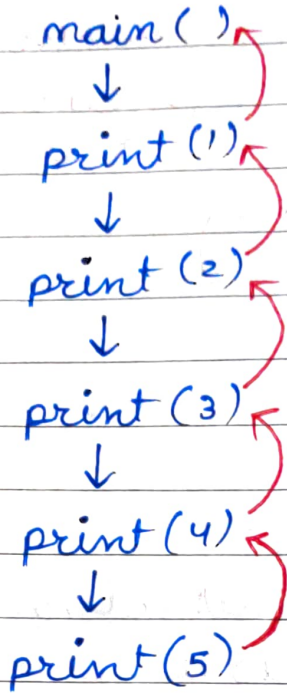
Why Recursion?

→ It helps us in solving bigger/complex problems in a simple way.

→ you can convert recursive solution into iterative solution and vice versa.

→ Space complexity is not constant because of recursive calls.

Visualising Recursion



→ Tailed Recursion

Q Find n^{th} fibonacci number :—

0^{th} 1^{st} 2^{nd} 3^{rd} 4^{th} 5^{th} 6^{th} 7^{th}

0, 1, 1, 2, 3, 5, 8, 13, ...

$$Fibo(N) = Fibo(N-1) + Fibo(N-2)$$

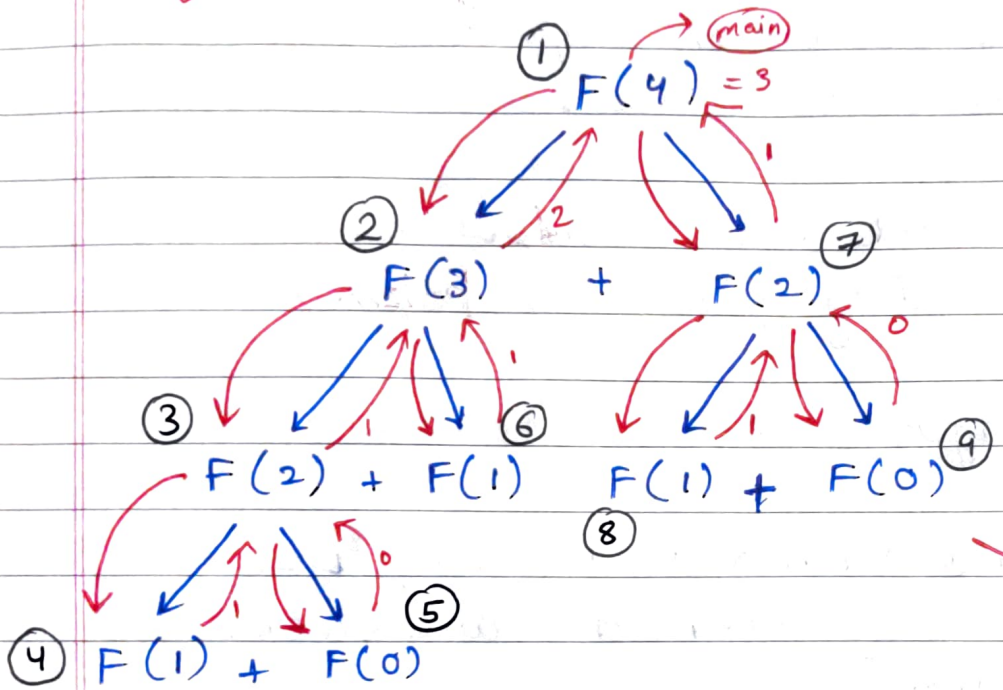
→ by common sense

doesn't this mean that this entire problem is divided into smaller sub problem?

→ hence, you can apply recursion.

Fibo(N) = Fibo(N-1) + Fibo(N-2)

This is known as a recurrence relation, when you write the recursion in a formula it is called recurrence relation.



The base condition is represented by answers we already have. For example, we know that, in this case F(0) = 0, F(1) = 1. This is base condition.

☆ How to understand and approach a problem?

1. Identify if you can break down the problem into smaller problems.
2. Write the recurrence relation if needed.
3. Draw the recursive tree.

④ About the recursive tree :-

→ See the flow of functions, how they're getting in stack.

→ Identify and focus on left tree calls and right tree calls.

→ Draw the tree and pointers again and again using pen and paper.

→ Use a debugger to see the flow.

⑤ See how the values and what type of values (int, String, etc) are returned at each step.

⑥ See where the function call will come out of.

⑦ In the end, you'll come out of the main function.

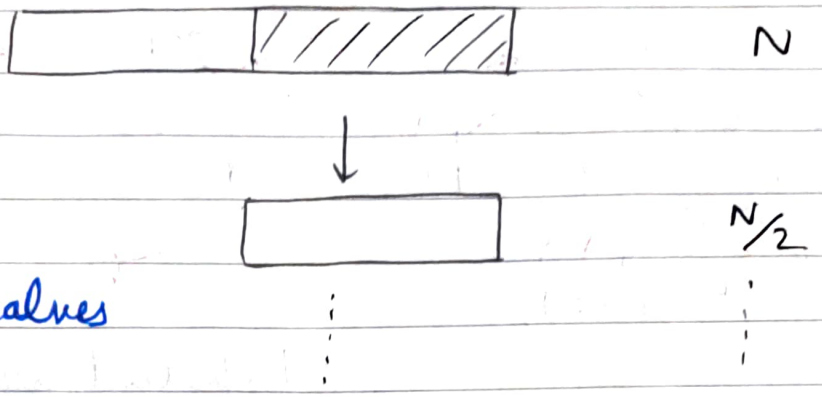
Working with variables in recursion :

① Arguments (will go in the next function call)

② Return Type

③ Body of function (specific to that call) (mid in BS)

Q Binary search with recursion



At every step:-

① Comparing → O(1)
② Dividing into 2 halves

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

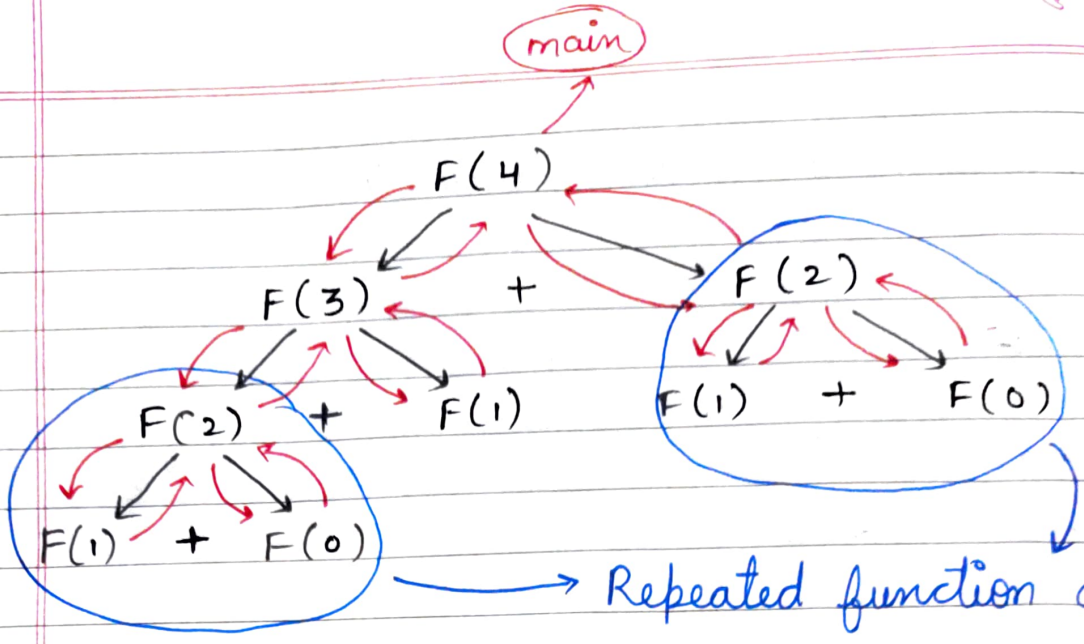
Types of Recurrence Relation :-

① Linear Recurrence Relation → Fibonacci (∵ its reducing it linearly N-1 & N-2)

② Divide & Conquer Recurrence Relation → BS
(∵ search space is reduced by a factor)
(N/2, N/3, N/4, etc)

Our search space is getting reduced much faster in divide and conquer recurrences. (dividing a number by 2 is definitely much faster than substracting 1, 2 from it) Therefore divide and conquer is much more efficient than linear recurrence relation.

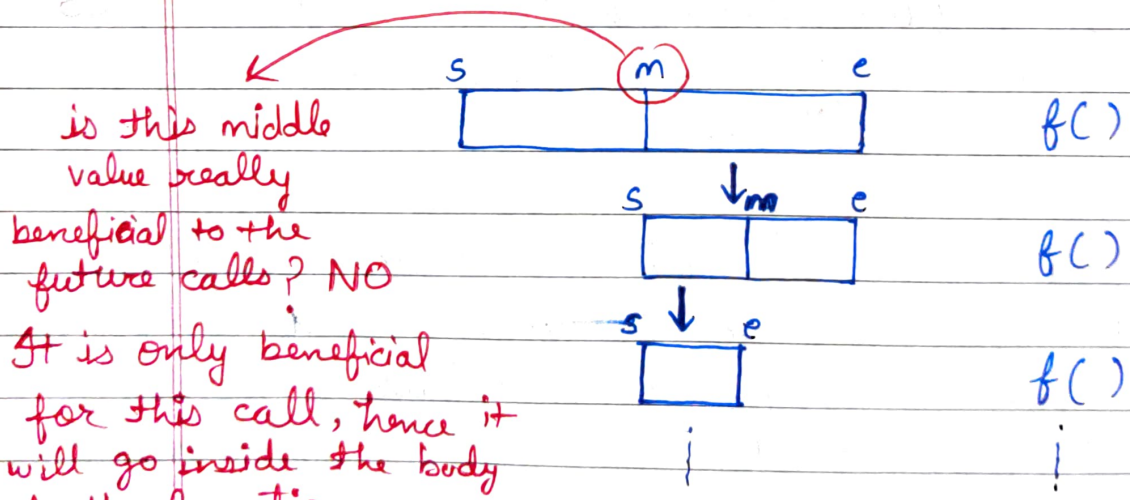
Linear recurrence relations are very very inefficient because the bigger number is actually getting smaller and smaller in very small steps.



→ Repeated function calls

Imagine for F(50) how many repeated function calls will be there. How can we solve this problem ? Dynamic Programming

Continuation of where to take which variables :-



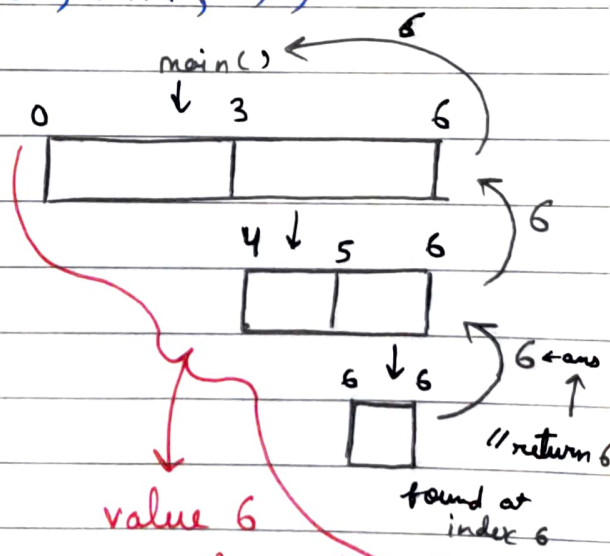
is this middle value really beneficial to the future calls ? NO

It is only beneficial for this call, hence it will go inside the body of the function.

s, e variables are actually being passed in the function calls, these are very important that is why they will go in the arguments. So, in short, if there are some variables that you need to pass

in the future function calls, put it inside the argument without thinking twice and all the variables that you only consider valuable in that function call that you dont need to pass inside the future recursion calls, put them inside the body of that f^n.

```
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);
    }
    return search(arr, target, m+1, e);
}
```



How to figure out when to put return statement? → If there's a return condition over here, make sure you're returning whatever answer from the subcalls you're getting.

value 6 was figured out in the last step travelled through function calls.