Recursion

Page No.
Date :

*How do function call works?

Let's say we have a program as shown below:

```
public class Main {
    public static void main(String[] args){
        print_num(1);
    }
    static void Print_num2(2){
    Static void Print_num(int n) {
        sout(n);
        Print_num_2(2);
    }
    Static void Print_num_2(int n){
        sout(n);
        Print_num_3(3);
    }
    static void Print_num_3(int n){
        sout(n);
    }
}
```

Output :-

1
2
3

Let's see how this works

1) while the function is not finished executing, it will remain in the stack memory. (Prints)

stack

main

- main fuction will be the 1st fuctn to be stored in stack.
- In our case it calls another fuctn named print-num(1)

→

After calling the fuctn, it will not be removed from stack, instead it will wait until the print-num fuctn has finished executing.

stack

print-num(1)
main

Print-num(1) will 1st print 1 and will do the same thing as main fuctn. It will tell the print-num-2(2) fuctn to finish it's work until then it will just wait in stack.

Prints 3

stack

Print-num-3(3)
Print-num-2(2)
Print-num(1)
main

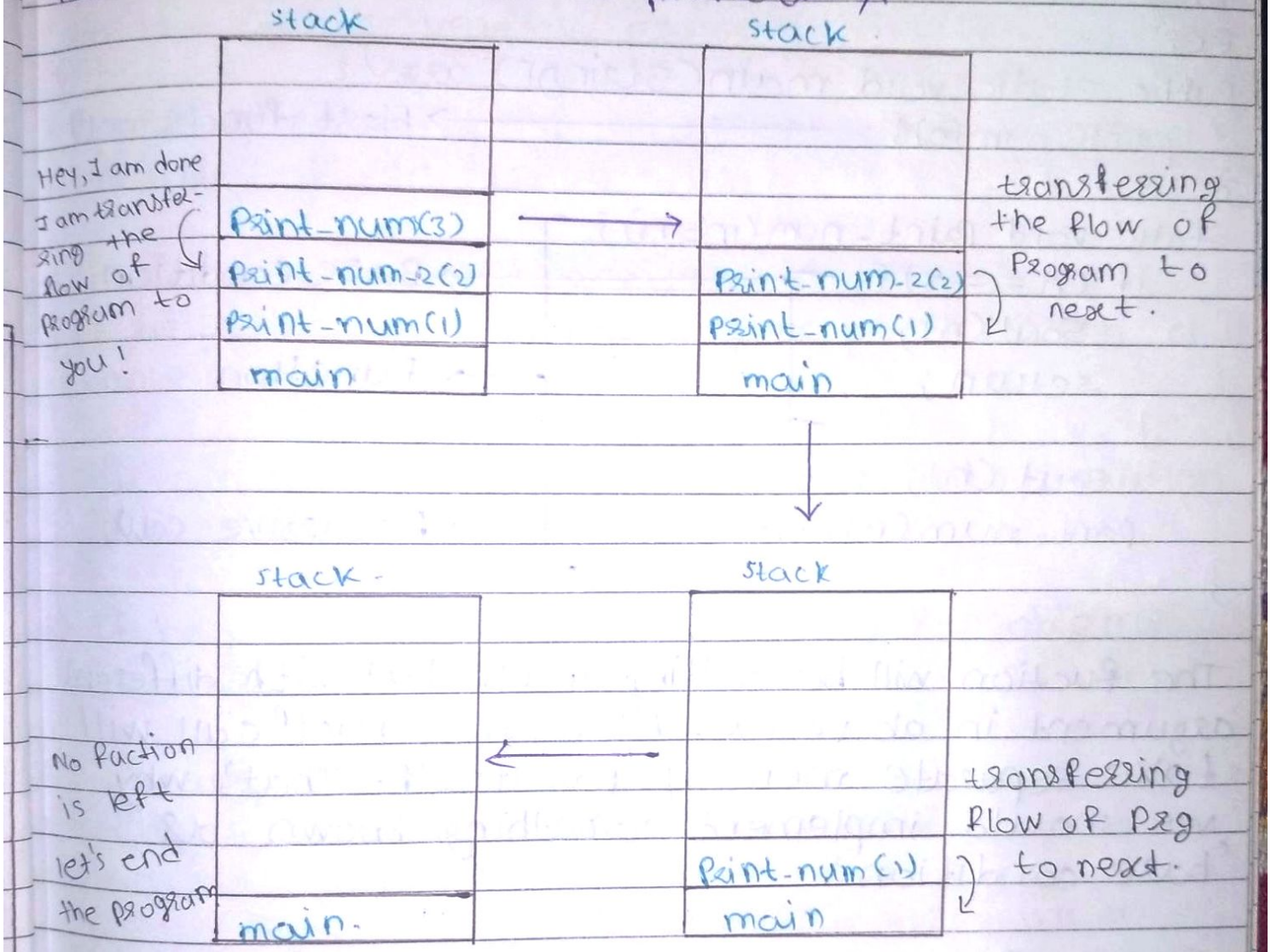
Prints 2

stack

print-num-2(2)
Print-num(1)
main

Now, the last fuctn i.e. print-num-3(3) does not have any other function to call and so, its work is done! Now it will tell the previous fuctn that its work is done!

2) when the fuction finishes executing, it is removed from stack & the program flow is restored to where that function was called previously.



Date :

Recursion — The process in which a function calls itself is known as recursion.

Ex.

```
Public static void main(String[] args){
    print_num(0);                          → First function call
}
Static void print_num(int n){
    if (n == 8){                           → Base condition
        Sout(n);
        return;                            → Function
    }
    Sout(n);
    print_num(n+2);                        → Recursive call
}
```

The fuction will be calling itself but with different argument in above ex. Also, every fuct^n call will take separate memory for itself. That's why we should implement something known as 'base condition'.

Base Condition

- The function will stop making recursive calls if this condition is fullfilled
- In above example base condition is n == 8

Page No.
Date :

- But what if there's no base condition ?
well, the fuctn will keep making recursive calls & the stack memory will get filled. This will lead to the stackoverflow error!

- space complexity of recursion is not constant because of recursive calls.

why dow we need recursion ?

1) It helps us in solving complex problems in a simple way!

2) You can convert recursive solution into iteration & vice versa.

3) It help us in breaking down bigger problems to smaller problems

For example fibonacci

- Break it down into smaller problems
- Fibo(N) = Fibo(N-1) + Fibo(N-2)

when you write down a formula in recursion it is known as 'recurrance relation'

- The base condn is represented by answers we already know.
In this case we know F(0) = 0
& F(1) = 1

Tail Recursion :- The last fuction call is called tail recursion

In the 1st example print_num(); is the last fuct^n call so it is tail recursive.

In case of Fibonacci
return fibo(n-1) + fibo(n-2); // this is not a tail recursive.

Note :- How to understand & approach a problem.

1) Indentify if you can break down problem into smaller problem.

2) Write the recurrence relation if needed

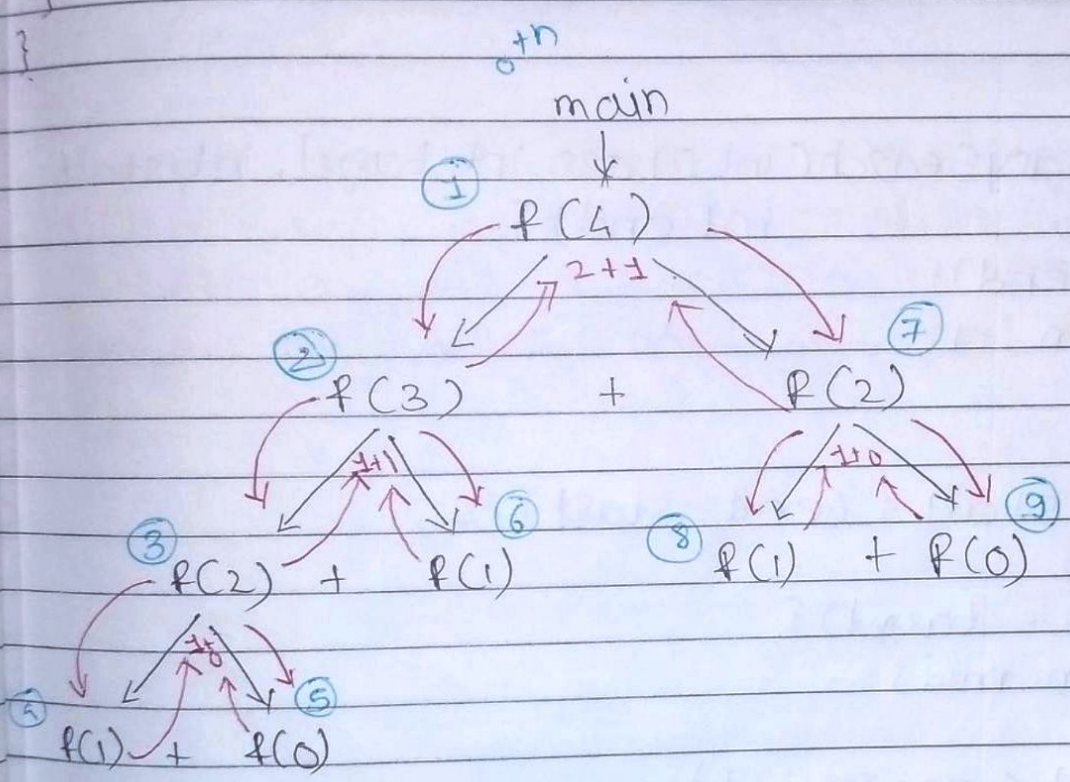
3) Draw the recursive tree.

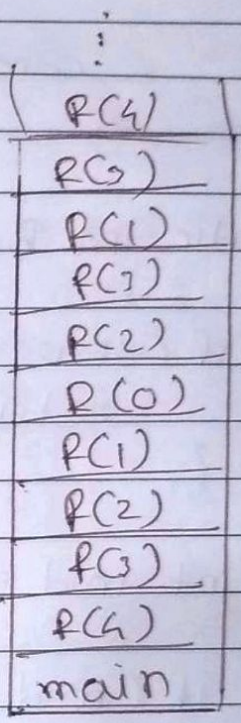
4) About the tree.
- see the flow of fuc^n, How they are getting in stack
- Identify & focus on left tree calls and right tree calls
- Draw the tree & pointers again & again using pen & paper
- Use a debugger to see the flow

5) See how the values are returned at each steps; see where the ~~fuct~~ function call will come out off.

Fibonacci –

```
public static void main (String[] args){
    System.out.println(fibo(4));
}
static int fibo(int n){
    if (n<2){
        return n;
    }
    return fibo(n-1) + fibo(n-2);
}
}
```





Make sure to # return the result of a fuct^n call of the return type.

Types of recurrence
1) Linear recurrence rel^n → fibo
2) Divide & conquer rec. rel^n → Binary S.
(reduced by a factor)

Page No.
Date :

* Variables :- 1) Arguments
2) Return Type
3) Body of Function

2.1) Binary search with recursion

$$f(N) = O(1) + f(N/2)$$

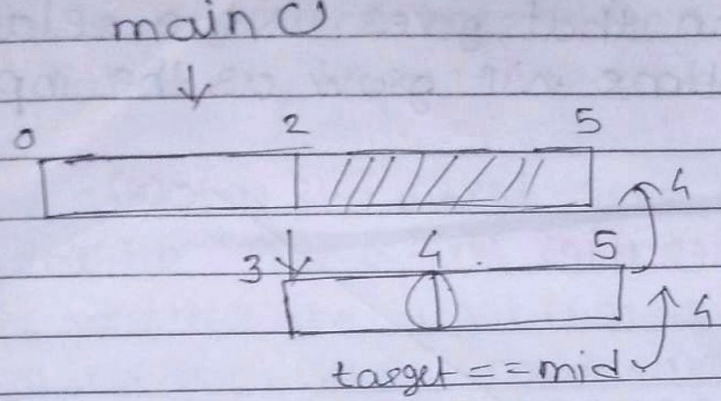
$O(1)$ ↓ comparison
$f(N/2)$ ↓ Divide arr by 2
└ Recurrence Rel^n

*

```
static int BinarySearch(int[] arr, int target, int start
                        int end) {
    if (start > end){
        return -1;
    }

    int mid = start + (end - start) / 2;

    if (mid == target) {
        return mid;
    }
    if (target < arr[mid]){
        return BinarySearch(arr, target, start, mid - 1);
    }

    return BinarySearch(arr, target, mid + 1, end);
}
```

arr = [3, 5, 7, 8.8, 99, 101];

target = 99.



whatever you are passing in the arguments it will be used in the future function calls, & whatever you are using in the body of function that will be used for that function call only.