Function calling itself.

Use → Solve the problem using solution of subproblems.

Q → Find sum of first N natural numbers. $\frac{N*(N+1)}{2}$

N = 4 sum(4) = 1 + 2 + 3 + 4 = 10

N = 5 sum(5) = 1 + 2 + 3 + 4 + 5 = 15

sum(4) + 5

sum(N) = sum(N-1) + N

How to use recursion?

1) Define exactly what the function do.
2) Identify how to use subproblems to get the answer.
3) Define base case i.e. smallest subproblem.

```
int sum(N) {
    if (N == 1) return 1
    return sum(N-1) + N
}
```

Function call tracing

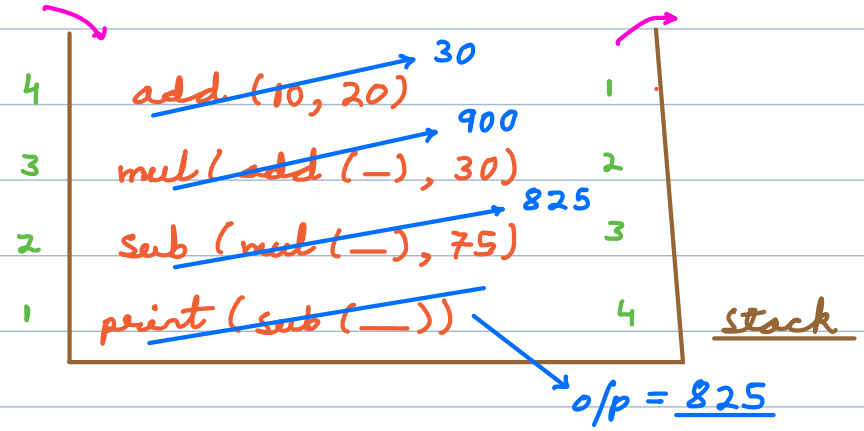
```
int add(x, y) {
    return x+y
}
```

```
int sub(x, y) {
    return x-y
}
```

```
int mul (x, y) {
    return x * y
}
```

x = 10 y = 20

```
print ( sub (mul ( add (x, y), 30), 75))
```



Q → Find factorial of N using recursion.

N = 5 fact (5) = 1 * 2 * 3 * 4 * 5 = 120

fact (4)

1) Define the fuction → int fact (N) {...}
2) Use of subproblem → fact (N) = N * fact (N-1)
3) Base case → fact (0) = 1

```
int fact (N) {
    if (N == 0) return 1
    return N * fact (N-1)
}
```

TC = O (N * 1) = O(N)

SC = O(N)

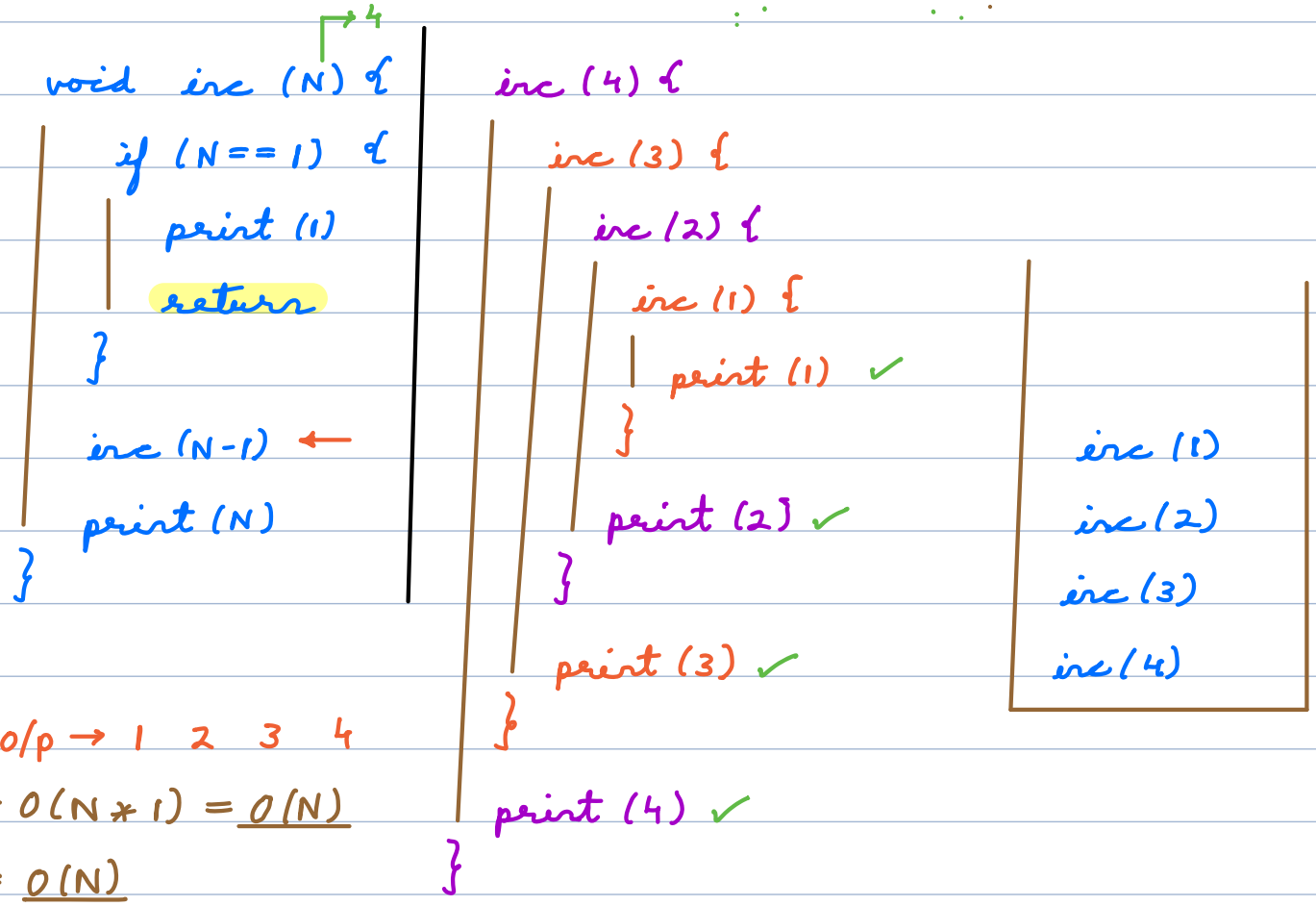
```
fact (3) {                    6
    return 3 * fact (2) {     2
        return 2 * fact (1) { 1
            return 1 * fact (0) { 1
                return 1
            }
        }
    }
}
```

}

Q → Print numbers 1 to N in increasing order.

N = 3 o/p → 1 2 3

1) Define the fuction → void inc (N) {...}
2) Use of subproblem → inc (N) → inc (N-1) print (N)
3) Base case → inc (1) → print (1)



o/p → 1 2 3 4

TC = O(N * 1) = O(N)

SC = O(N)

Q → Print numbers N to 1.

dec (3) → 3 2 1

dec (N) → print (N)

dec (N-1)

```
void dec (N) {
    if (N == 1) {
        print (1)
        return
    }
    print (N)
    dec (N-1)
}
```

```
dec (3) {
    print (3) ✓
    dec (2) {
        print (2) ✓
        dec (1) {
            print (1) ✓
        }
    }
}
```

o/p → 3 2 1

Time Complexity → O ((# function calls) * (time per function call))

Space Complexity → O (Max stack size at any point + space in function call)

Q → Given an integer N, print numbers N to 1 followed by 1 to N using recursion.

N = 3 o/p → 3 2 1 1 2 3

1) Define the fuction → void decInc (N) { ... }

2) Use of subproblem → decInc (N) → print (N)
decInc (N-1)
print (N)

3) Base case →
decInc (0) → return

```
void decInc(N) {
    if (N == 0) return
    print (N)
    decInc (N-1)
    print (N)
}
```

TC = O(N)

SC = O(N)

```
decInc(2) {
    print (2)
    decInc(1) {
        print (1)
        decInc(0) { }
        print (1)
    }
    print (2)
}
```

Fibonacci Numbers

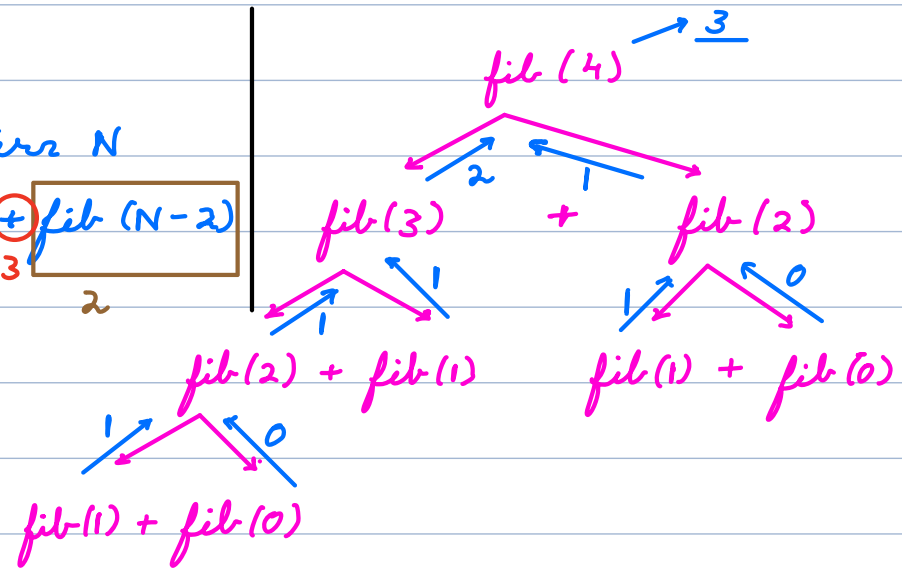
N =	0	1	2	3	4	5	6	7	...
f(N) =	0	1	1	2	3	5	8	13	...

1) Define the fuction → int fib (N) {...}

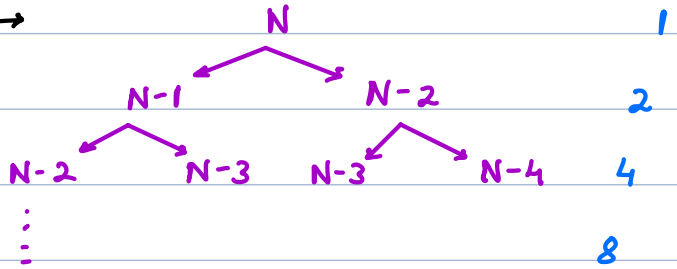
2) Use of subproblem → fib (N) = fib (N-1) + fib (N-2)

3) Base case → if (N <= 1) return N

```
int fib (N) {
    if (N <= 1) return N
    return fib (N-1) + fib (N-2)
}
```



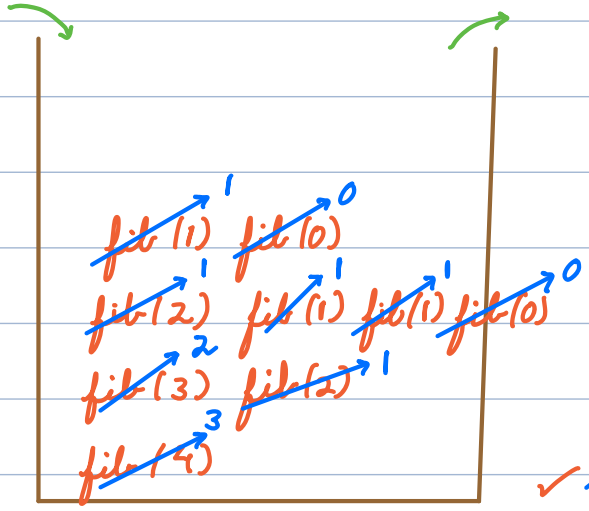
Time Complexity →

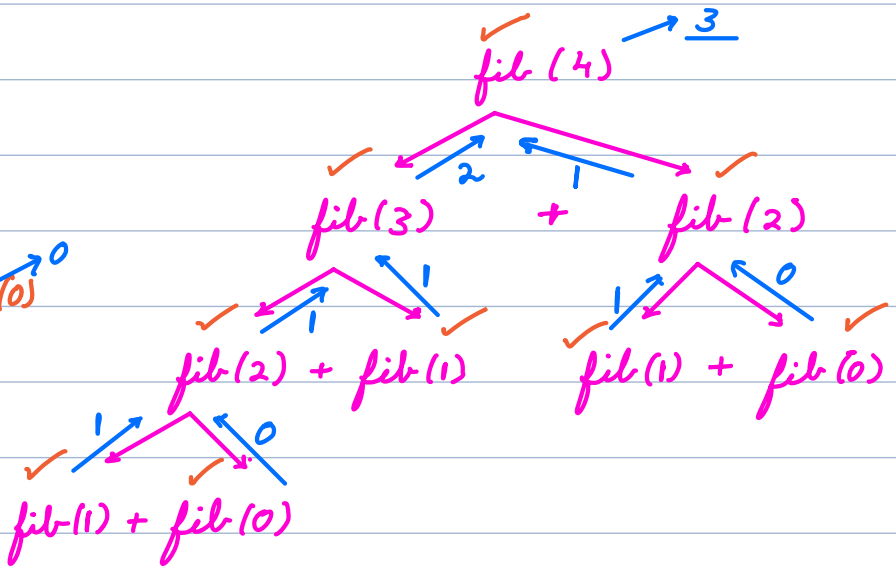


$$\frac{a(r^n - 1)}{r - 1} \longrightarrow \frac{1(2^{N+1} - 1)}{2 - 1} \rightarrow 2^{N+1} - 1$$

$O(2^N)$

Space Complexity →





SC = $O(N)$

> Max size of recursion stack at any point = Height of recursion tree

A = [1 0 1] (0 1 2)

OR → any 1 ⇒ 1

1 → 1
1 0 → 1
1 0 1 → 1
0 → 0
0 1 → 1
1 → 1

5

Ans = (# subarrays) − (# subarrays with all 0's)

7

A = [1 0 1 1 0 0 0 1]

1

6

consecutive 0's = K ⇒ # subarrays = $\frac{K*(K+1)}{2}$

A = [1 3 5 2] 1 — 6

[1 3 5 2 1 2 3 (4) 5 (6)]

Find 2 unique numbers where others are repeating twice.
