Recursive thinking

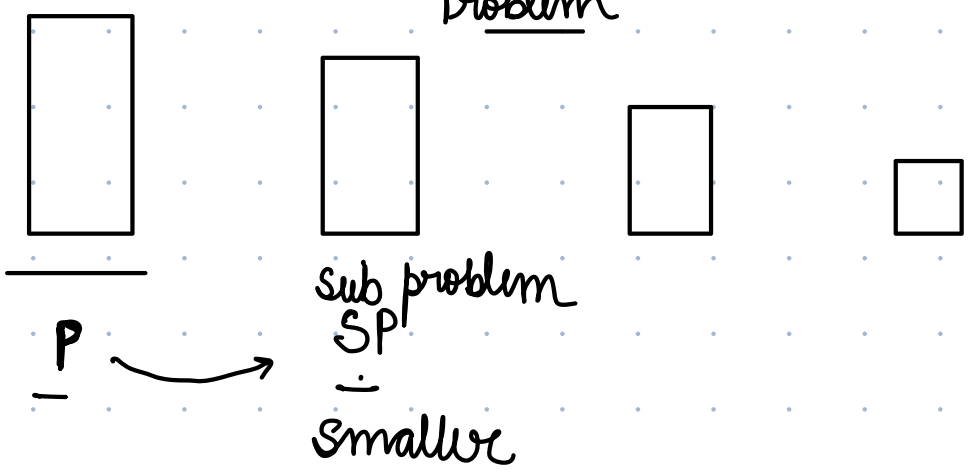
Basic

- Sum of first N natural
- Function call tracking
- Factorial of N
- Printing → Number ✓
  → Array ✓

Time & Space complexity.

→ Fibonacci → Dry Run

Recursion → Function calling itself.



Recursion → built solution to P, from answers of subproblem

P: given N, find sum of first N natural numbers ⇒ int Sum(int N){

Thinking 1 level down

int ans = sum(N-1)

return ans + N

}

P = Sum(N)

SP = sum(N-1) ans

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

ans

P

P = SP + EW

sum(5) = sum(4) + 5

10

15

1) P, SP

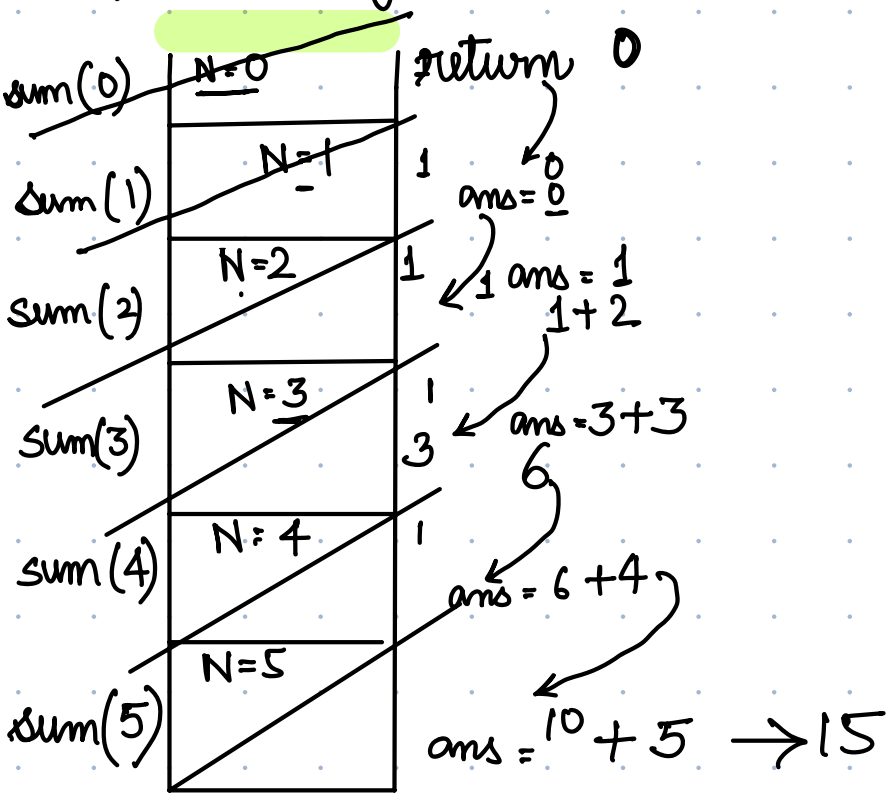
2) from ans of SP ⟶ P

P = SP + EW

ans

3) High level code

4) Dry Run → N=5




```
int Sum(int N){
  if (N== 0)
      return 0;
  int ans = sum(N-1)

  return  ans + N
}
```

```
if (N== 0)
    return 0;
```

P → SP

smallest problem — ans — STOP

base case

1) P , SP

2) from ans of SP ⟶ P

$P = SP_{ans} + EW$

3) High level code

4) Dry Run ⟶ N=5 and find base case & add it.

```
1  public class MainClass {
2
3    static int add(int x, int y) {
4        return x + y;
5    }
6
7    static int mul(int x, int y) {
8        return x * y;
9    }
10
11   static int sub(int  x, int y) {
12       return x - y;
13   }
14
15
16   public static void main(String[] args) {
17     int x = 10, y = 20;
18     System.out.println(sub(mul(add(x, y), 30), 75));
19   }
20 }
```

Edit this code

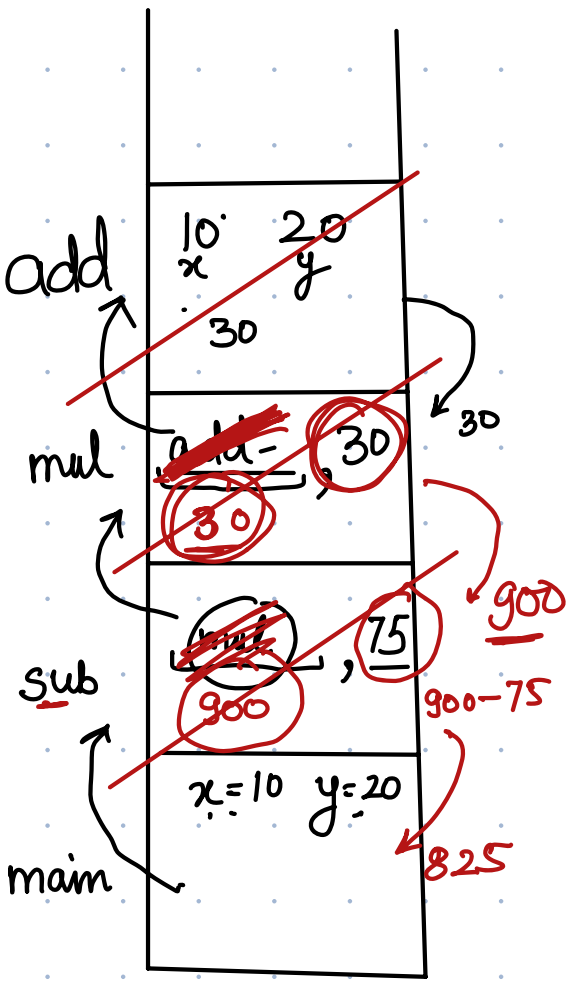
Frames Objects

main:17

```
1  public class MainClass {
2
3    static int add(int x, int y) {
4        return x + y;
5    }
6
7    static int mul(int x, int y) {
8        return x * y;
9    }
10
11   static int sub(int  x, int y) {
12       return x - y;
13   }
14
15
16   public static void main(String[] args) {
17     int x = 10, y = 20;
18     System.out.println(sub(mul(add(x, y), 30), 75));
19   }
20 }
```

1 2 3

825



Recursion — Function call mechanism

Q3 factorial

$N=5 \quad = 5! = 1*2*3*4*5 = 120$

function

P

```
int factorial ( int N) {
```

```
}
```

P = factorial (N)

SP = factorial (N-1)

fact (5) = fact (4) * 5

120

24

P = SP + EW

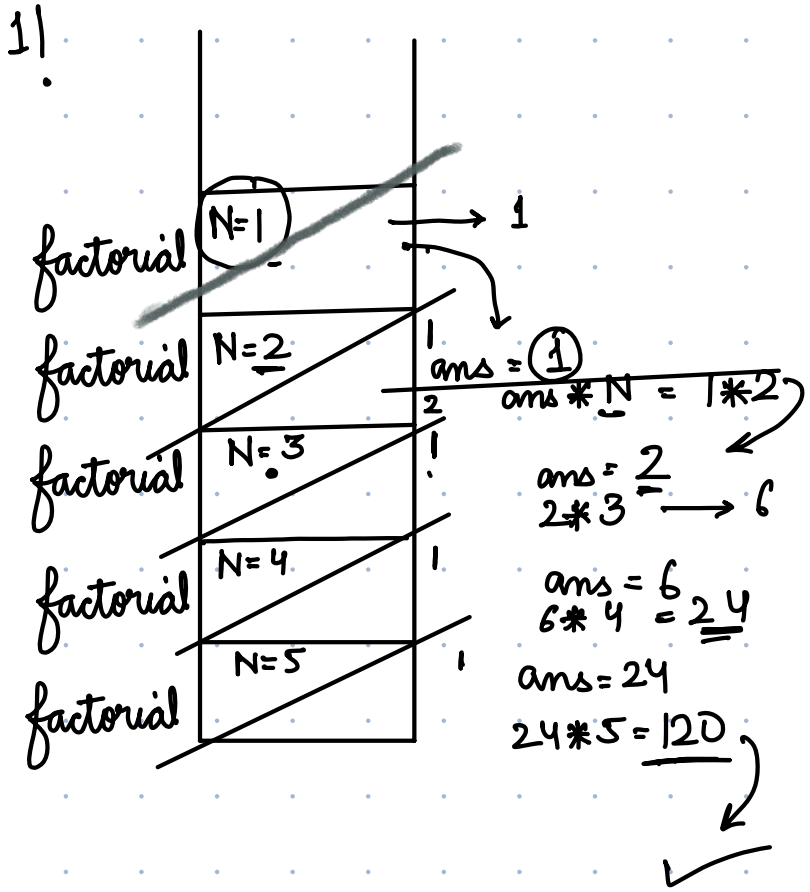
ans

ans * N

High level

```
int factorial (int N) {
    int ans = factorial (N-1);
    return ans * N;
}
```

N = 5




```
int factorial (int N) {
    if (N == 1) return 1;
    int ans = factorial (N-1);   1
    return ans * N;              2
}
```

Q4 print N number in increasing order

N=5 1 2 3 4 5

4 1 2 3 4

```
void Inc ( int N) {

P =

}
```

P = Inc(N) 5 1 2 3 4 5

SP = Inc(N-1) => 1 2 3 4

ans

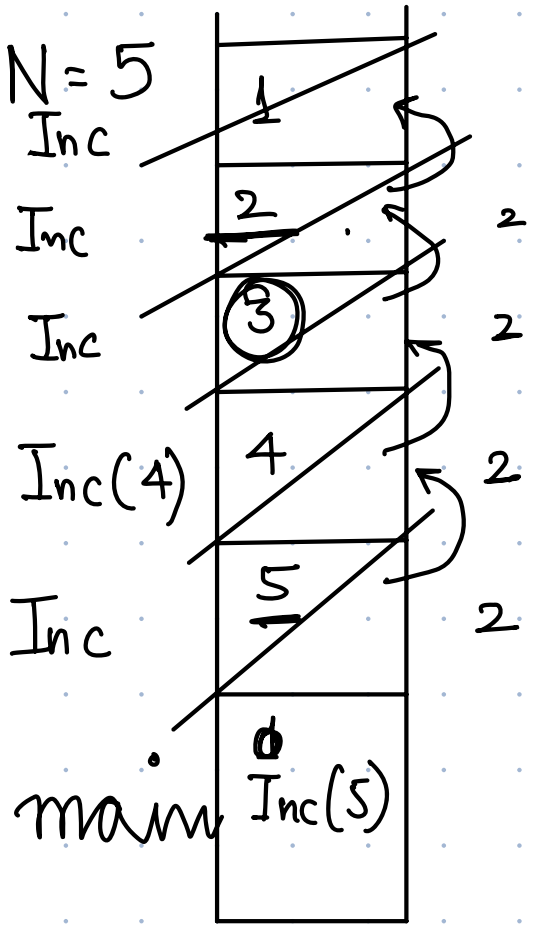
EW = print (N) ✓

High

Dry Run

void

```
Inc ( int N) {
if (N == 1) { print (1)
return; }
Inc ( N-1);   1
print (N)     2
}
```



1 2 3 4 5

int ↗ vary ⟶ function calls

arrays ⟶ function calls

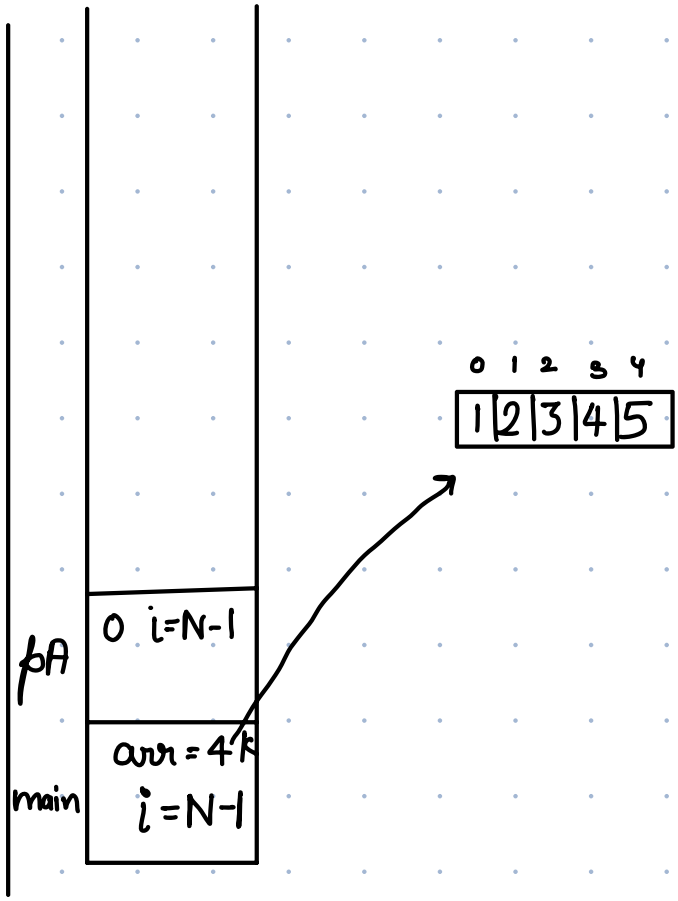
Recursion with arrays

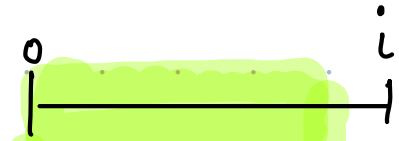
Q5 Print array

variable

A = [1, 2, 3, 4, 5] ⇒ 1 2 3 4 5

```
void printArray(int[] A,
                int i){
}
```



P = 

= printArray (arr , i)

SP = printArray (arr , i-1)

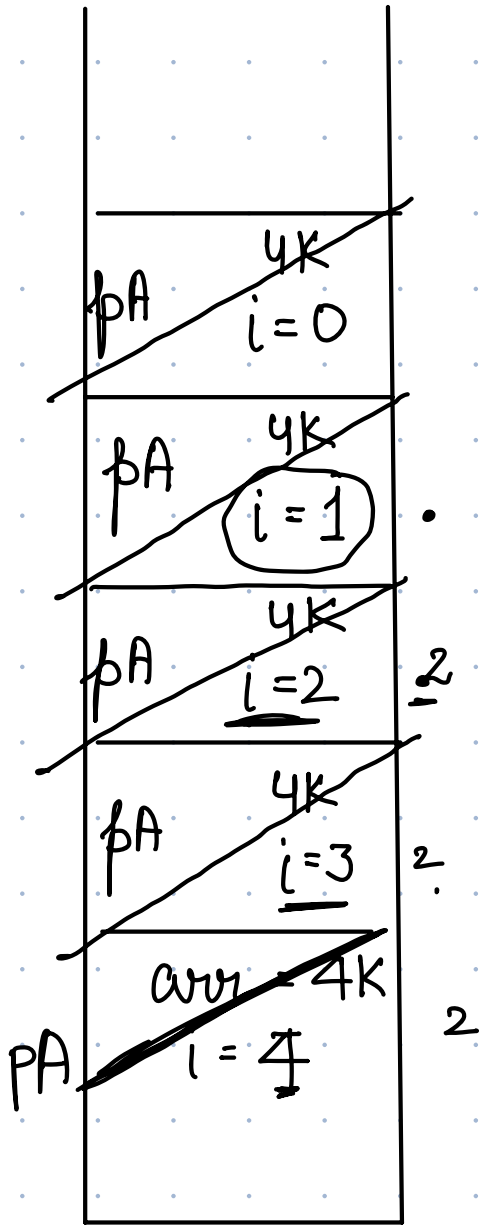
print (arr [i])

void printArray (int[] arr , int i) {

printArray (arr , i-1)

print (arr [i])

}




```
void printArray(int[] arr, int i){
    if (i==0){ print(arr[0])
               return;
    }
    printArray(arr, i-1)
    print(arr[i])
}
```

0	1	2	3	4
1	2	3	4	5

4K

12345

Time and Space Complexity

Time Complexity = O(Number of function calls * Time per function call)

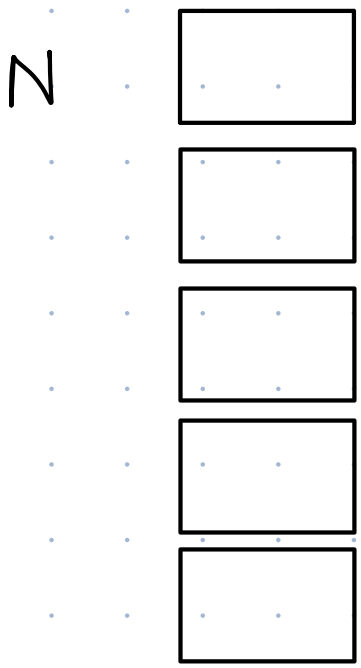
Space Complexity = O(Maximum depth of recursion tree/stack space * Space per function call)

$TC = O(\text{No of calls} * \text{TC of 1 call})$

$N * O(1) = O(N)$

```
int Sum(int N){
    if (N == 0)
        return 0;
    int ans = sum(N-1)

    return ans + N
}
```

O(1)

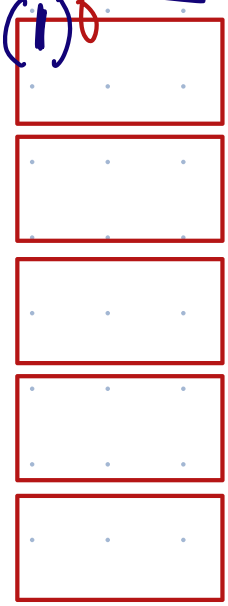


Fact

$TC = O(\text{No of calls} * \text{TC of 1 call})$

$N * O(1) = O(N)$

```
int factorial (int N) {
    if (N == 1) return 1;
    int ans = factorial (N-1);

    return ans * N;
}
```



Time Complexity = O(Number of function calls * Time per function call)

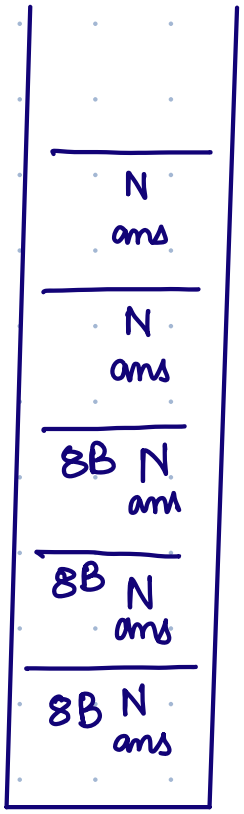
Space Complexity = O(Maximum depth of recursion tree/stack space * Space per function call)

```
int Sum(int N){
    if (N == 0)
        return 0;
    int ans = sum(N-1)

    return ans + N
}
```

N * 8B

8N

O(N)



Space taken by 1 function call

```
int factorial (int N) {
    if (N == 1) return 1;
    int ans = factorial (N-1);

    return ans * N;
}
```

8B

N * O(1)

8N

O(N)

Q6 Fibonacci

fib(0) = 0

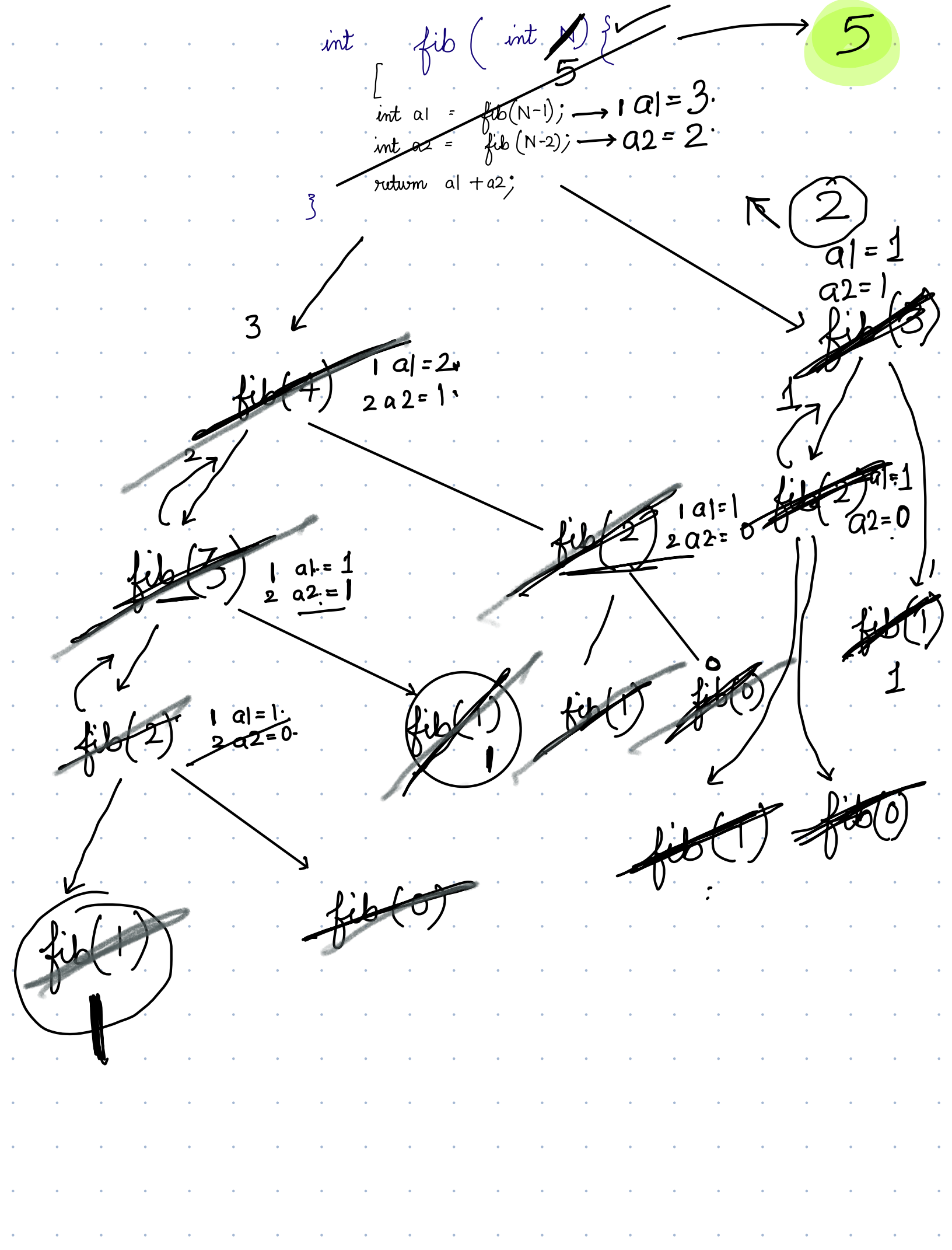
fib(1) = 1

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

N	0	1	2	3	4	5	6	7	8 ...
fib(N)	0	1	1	2	3	5	8	13	21 ...

P =

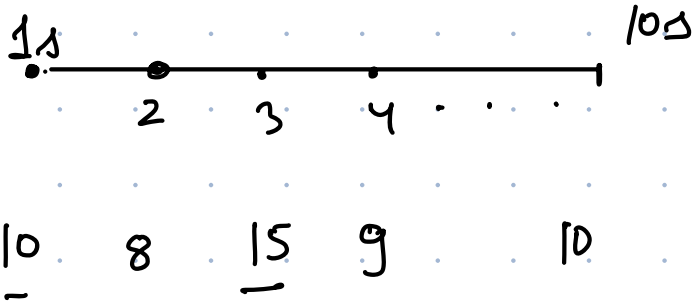
```
int fib ( int N) {
[
int a1 = fib(N-1);
int a2 = fib(N-2);
return a1 + a2;
}
```


int fib (int N) {
[
int a1 = fib(N-1); → a1 = 3
int a2 = fib(N-2); → a2 = 2
return a1 + a2;
}
5
5
3
2
a1 = 1
a2 = 1
fib(4)
1 a1 = 2
2 a2 = 1
fib(3)
1
fib(3)
1 a1 = 1
2 a2 = 1
fib(2)
1 a1 = 1
2 a2 = 0
fib(2)
a1 = 1
a2 = 0
fib(2)
1 a1 = 1
2 a2 = 0
fib(1)
1
fib(1)
fib(0)
fib(1)
1
fib(1)
fib(0)
fib(1)
fib(0)

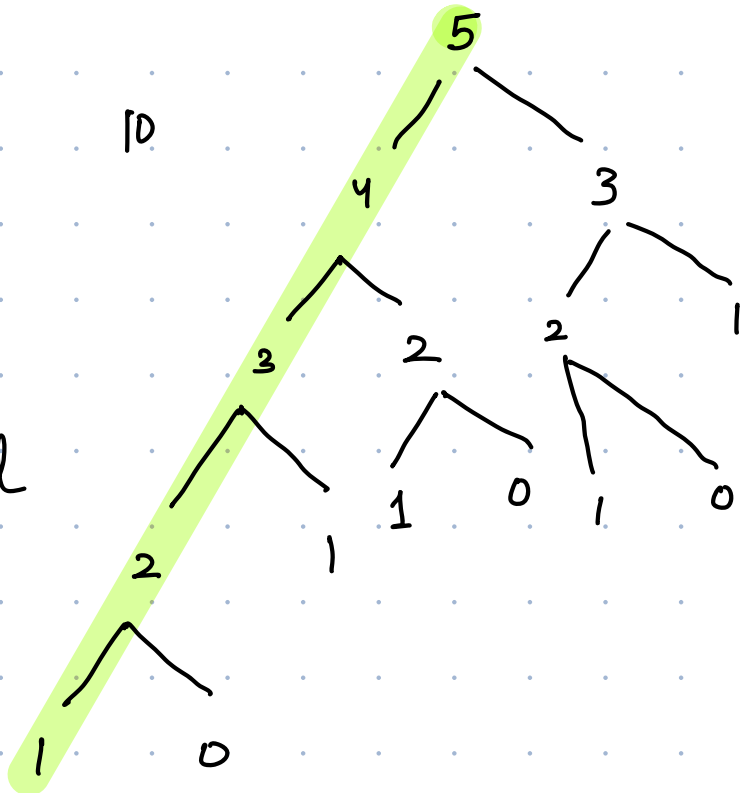
Complete Code:

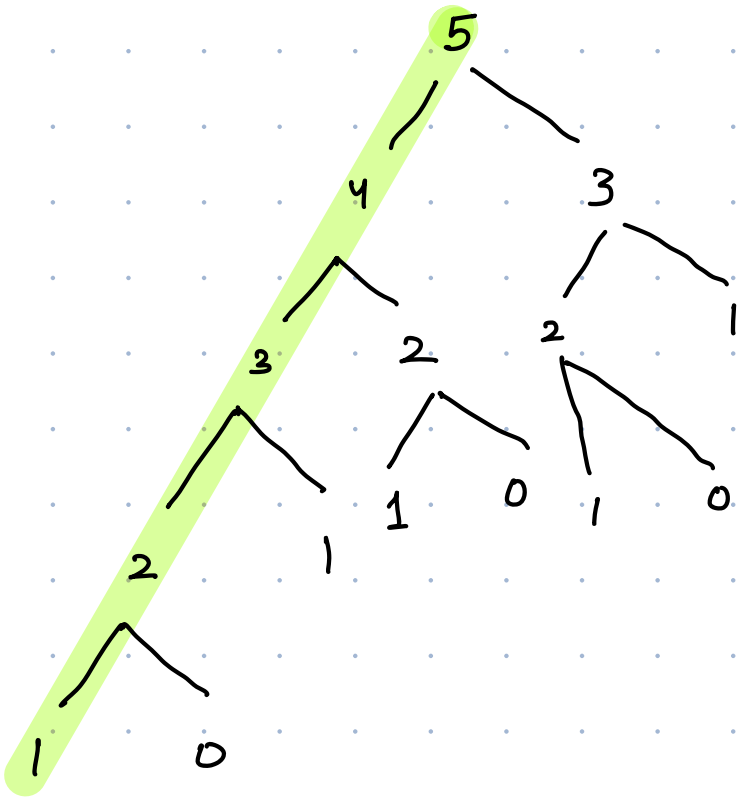
```
int fib(int N) {
    if (N == 0) return 0;
    if (N == 1) return 1;

    int a1 = fib(N-1);
    int a2 = fib(N-2);
    return a1 + a2;
}
```

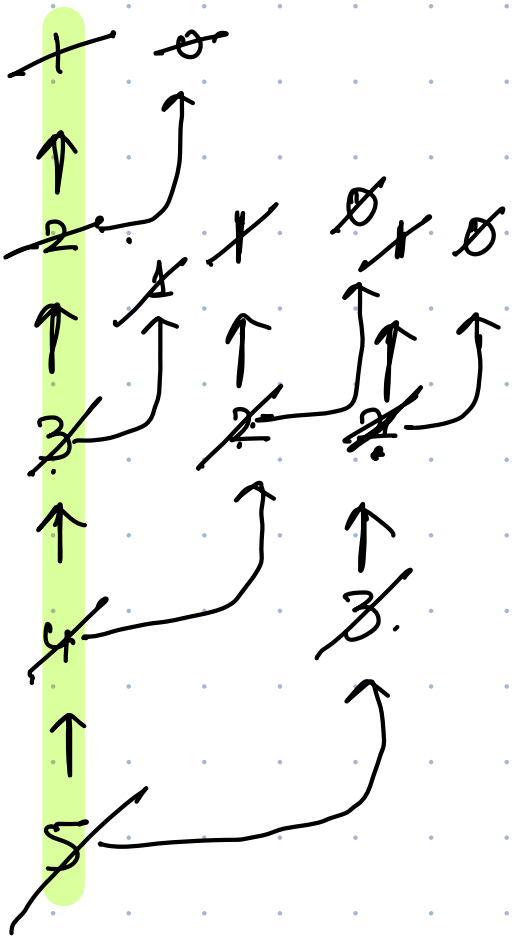
TC =



SC = max depth of stack * S of 1 call







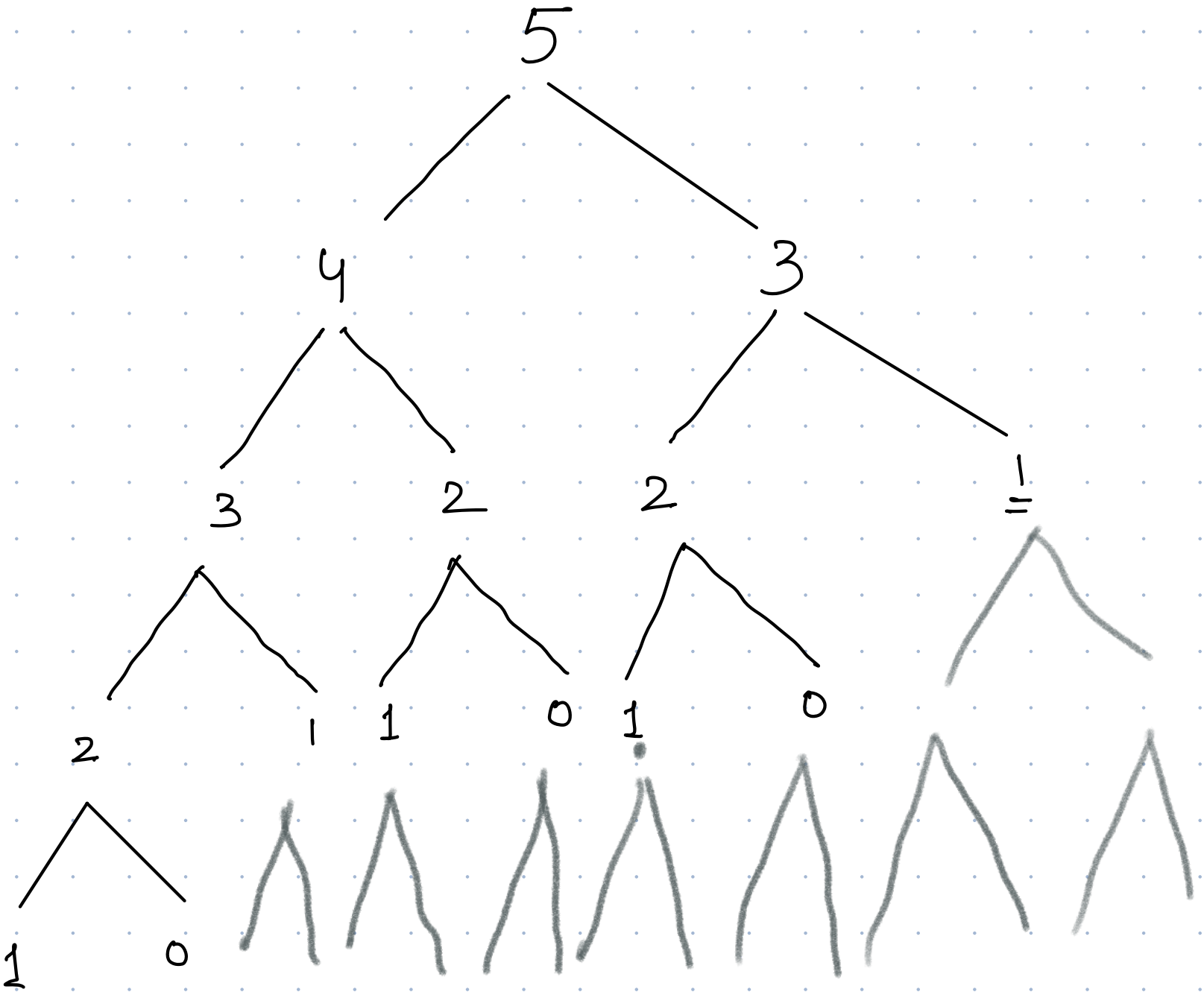
N * Space of 1 call

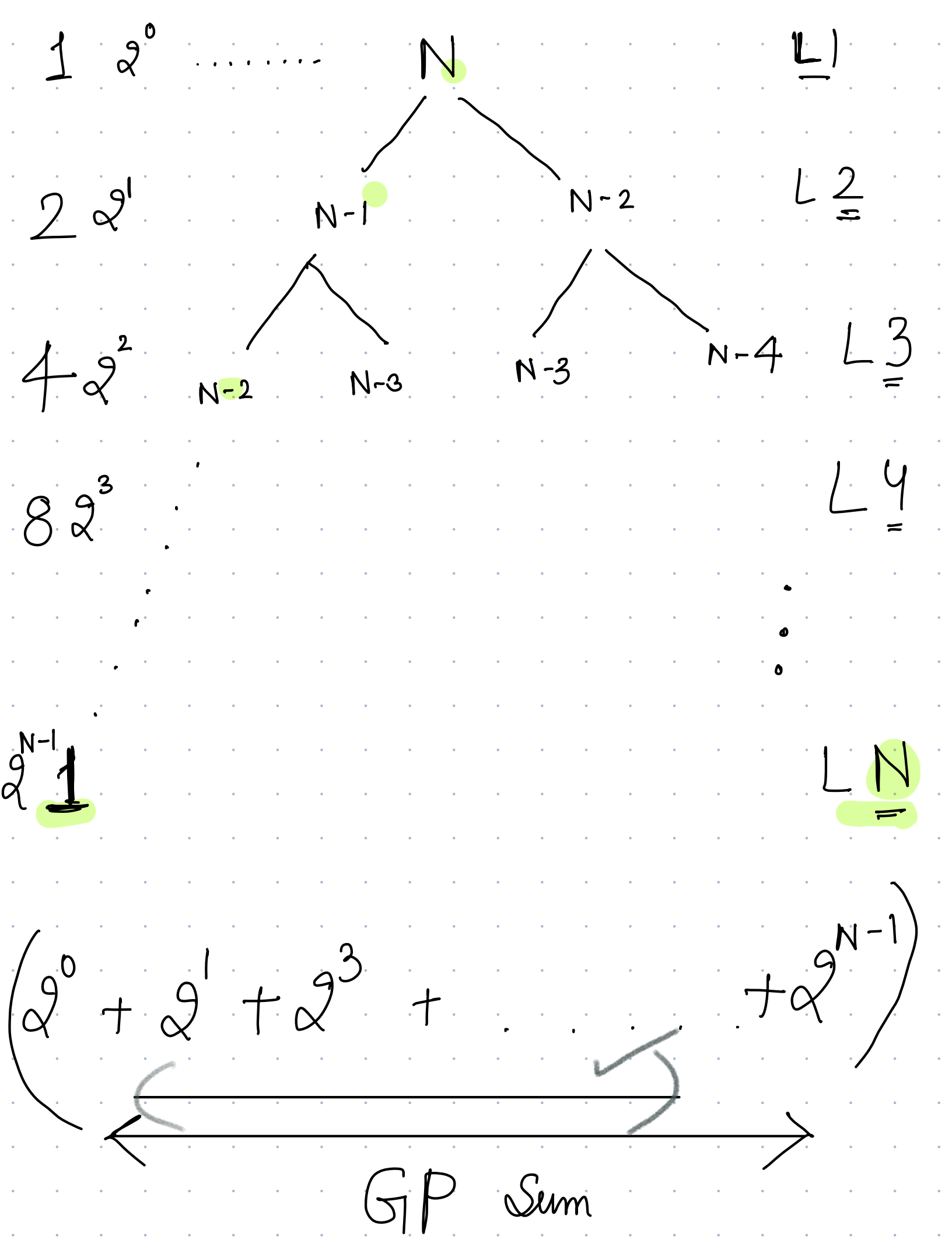
12 B

12 N

O(N)

TC = Total No of calls * Time of 1 call




1 2^0 ..... N L1
2 2^1 N-1 N-2 L2
4 2^2 N-2 N-3 N-3 N-4 L3
8 2^3 L4
2^{N-1} 1 L N
$(2^0 + 2^1 + 2^3 + \ldots + 2^{N-1})$
GP Sum

$=$ Sum $*$ Time of 1 call

$=$ 2^N $O(1)$

$=$ $O(2^N)$