Intro to Recursion

What is Recursion?

Function calling itself.

↳ To solve a problem

→ Solving a bigger problem using smaller instances of the same problem is called Recursion

sum(n) ⇒ Returns sum of 'N' natural numbers

sum(n) = sum(n-1) + n

(sum(n)) = 1+2+3 .... n-2 + n-1 + n

sum(n-1)

sum(n) = sum(n-1) + n ✓

Bigger Problem: sum(n)

Smaller Instance: sum(n-1)

(Subproblem)

→ Solving a problem using sub-problems is called Recursion.

⇒ sum(4) = 1+2+3+4 = 10

sum(4) 10

↳ return 4+sum(3) 6

↳ 3+sum(2) 3

↳ 2+sum(1) (1) n=0

↳ (1+sum(0))

4+3+2+1+0 -1 -2 ... ∞

Stack Overflow Error.

sum(4) → sum(3) → sum(2) → sum(1) → sum(0) → sum(-1)

sum(-INF) ...

3 steps for Recursion

1) Assumption:
what do you want your function to do?
sum(n) => Return sum of n natural nums

2) Main Logic / Recursive Relation
sum(n) = n + sum(n-1)

3) Base condition
When do you want the Recursion to stop
if(n == 0) return 0;

```
int sum(n) {
    if(n == 0) return 0;

    return n + sum(n-1);
}
```

if(n == 1) return 1;

Factorial of number

N = N x N-1 x N-2 ... 1

120

5! =

Assumptions: fact(n) return factorial of N

Main Logic: return n * fact(n-1)

Base condition: if(N <= 1) return 1; 0! = 1

fact(N) = 1 x 2 x 3 x 4 x ..... (N-2) x (N-1) x N

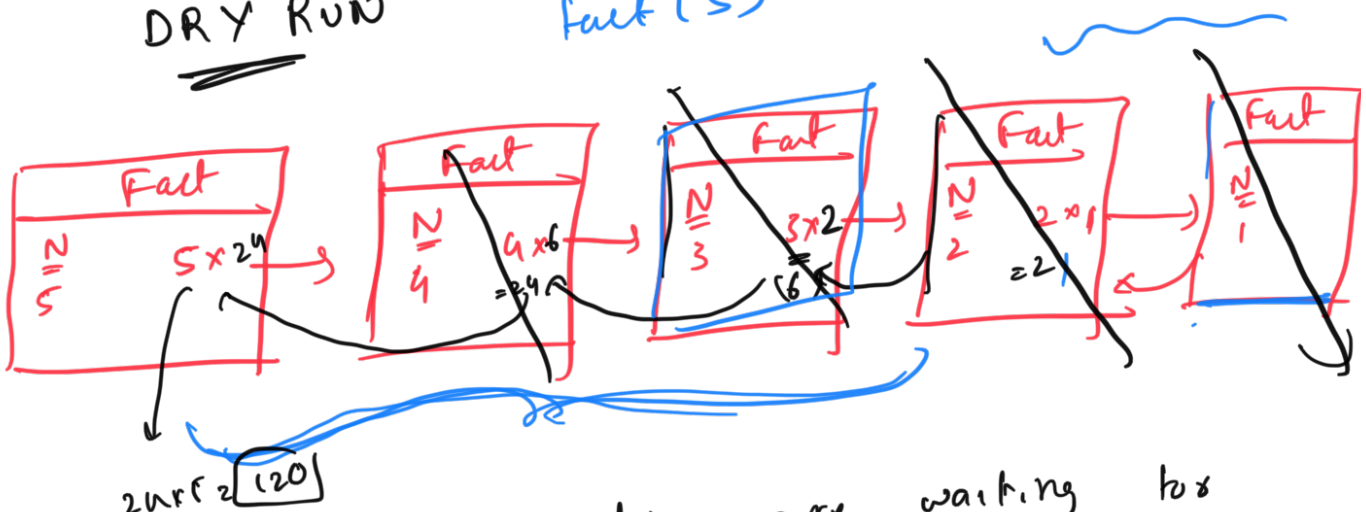
fact(n-1) x N

fact(N) = N * fact(N-1)

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

fact(5)

DRY RUN fact(5)



→ All the functions are waiting for others to complete

How a system keeps track of these functions?

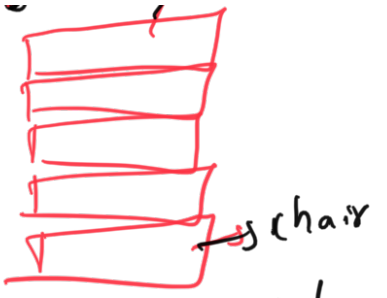
(0! = 1)

Stack

Last In First Out

(LIFO)

Recursion stack



→ We can access only top element

fact(0)=1

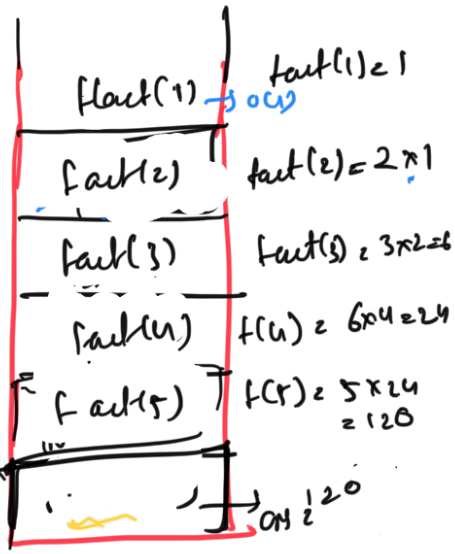
fact(1)

fact(0)
N=0
(0 ≤ 1) ✓

```
int fact(N) {
    if(N<=1)
        return 1;
    return N * fact(N-1);
}
main() {
    ans = fact(5);
}
```

O(1)

O(N)



fact(2) = 2*fact(1)

→ Whenever you call any function, some space is allocated on the stack.

Inbuilt stack

Function is called: Push on top of stack

Function is completed: Pop from top of stack.

T.C

fact(5) → fact(4) → fact(3) → fact(2) → fact(1)

5 function calls

11.

'n' Function calls

T.C: $O(N \times 1) = O(N)$

S.C: $O(n)$ [Recursion Stack] (Extra space for recursion)

$(N \times 1) = O(N)$

Any problem solved using Recursion, can definitely be solving using iteration ↑

Code looks very Simple

```
for(i=0; i<n; i++)
    sum+ = i
```

Recursion: Bigger Problem → Smaller

Iterative

Fibonacci: $F(n) = F(n-1) + F(n-2)$

$F(8) = F(6) + F(7)$

n	1	2	3	4	5	6	7	8	...
	1	1	2	3	5	8	13	21	34 ...
		N=1	N=2			fib(n-2)	fib(n-1)	F(n)	

fib(N)

Assumption: fib(n) returns the N^{th} Fibonacci Number

Main logic: return fib(n-1) + fib(n-2)

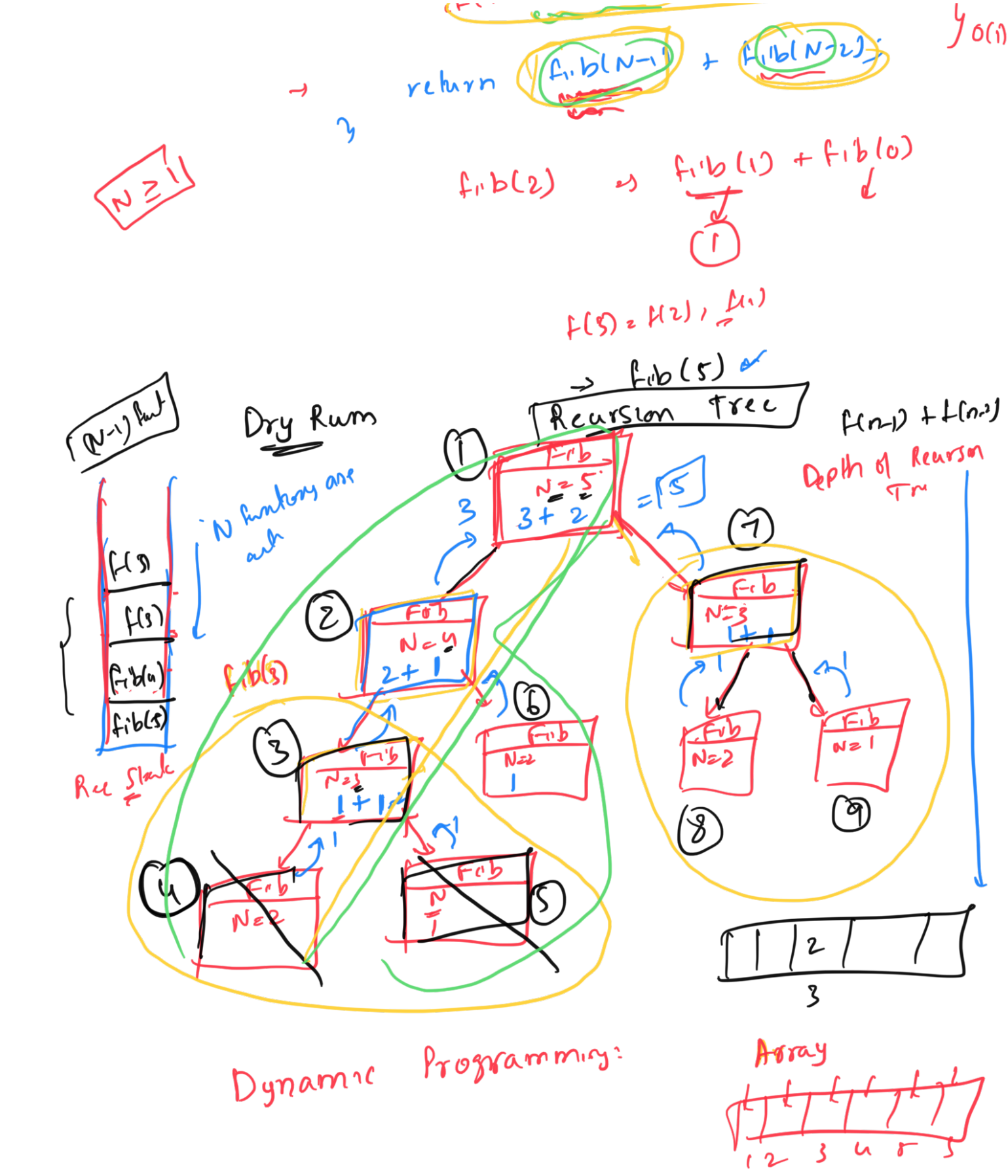
2 Base Cond!

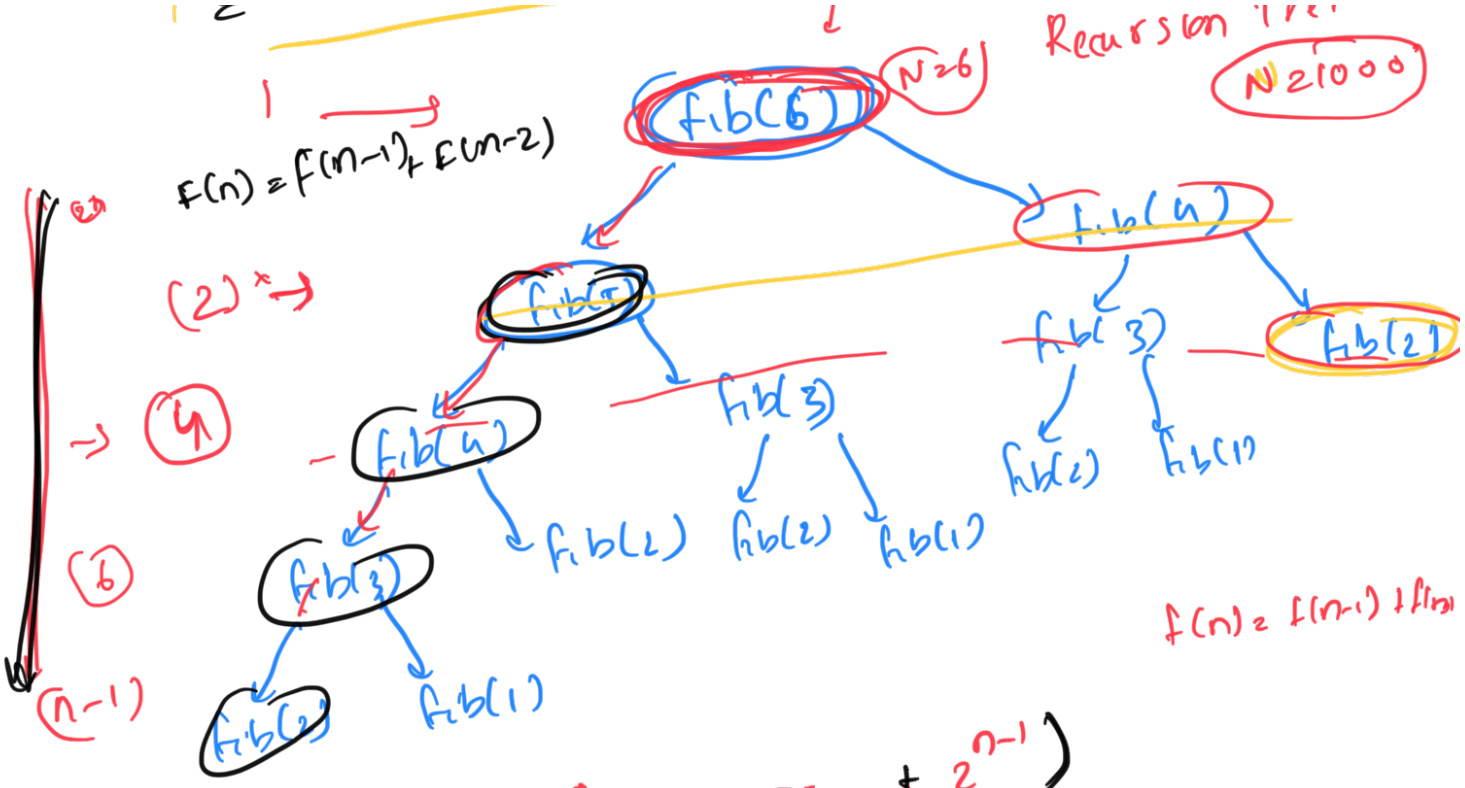
Base Condition: if (N == 1) return 1;

→ int fib(N) {

if (N == 1 || N == 2) return 1;

}


return fib(N-1) + fib(N-2);
}
N ≥ 1
fib(2) → fib(1) + fib(0)
1
f(5) = f(4) + f(3)
fib(5)
Recursion Tree
Dry Run
(N-1) func
N functions are each
Rec Stack
fib(5)
fib(3)
Depth of Recursion Tree
f(n-1) + f(n-2)
Fib
N=5
3+2
N=4
2+1
N=3
1+1
N=2
N=1
1
2
3
Dynamic Programming:
Array
1 2 3 4 5
S.C : O(n)
T.C : O(# function calls × T.C Per function)



f(n) = f(n-1) + f(n-2)

fun = $(1 + 2 + 4 + 8 + \ldots + 2^{n-1})$

6 → 5 → 4 → 3 → 2 (n-1)

(n-1)

$1 + 2 + 2^2 + 2^3 + \ldots 2^{n-1} = 2^n$

functions = 2^n

T.C: $(2^n \times 1) = O(2^n)$

$(1 + 2 + 4 + 8 + \ldots 2^{n-1})$ → G.P

$a \quad ar \quad ar^2 \quad ar^3$

$= \frac{a(r^n - 1)}{r - 1}$ (No. of fun)

Question: check if a string is palindrome

T.C: $O(n)$

S.C: $O(1)$

$0, \quad n-1$

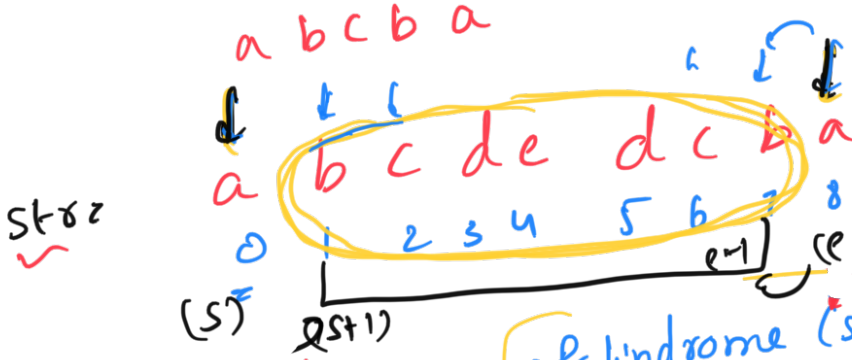
Recursion

1, n-2 ✓
2, n-3 ✓

if(str[start] == str[end] && isPalindrome(start+1, end-1).

NAMAN

a b c b a



Assumption: Is the string str[start...end] a Palindrome or not

isPalindrome(start, end) =>

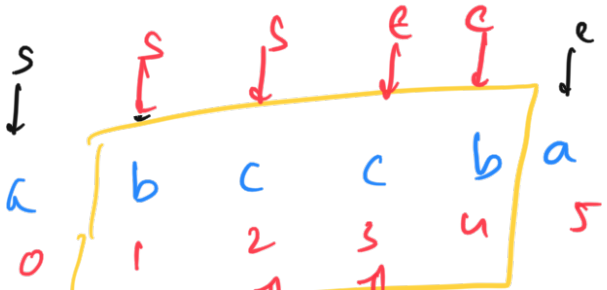
Main Logic:

isPalindrome(start, end) = str[start] == str[end] && isPalindrome(start+1, end-1).

Base Condition: (start ≥ end) return True.

(start > end)
Empty String: True

Ex



start > end

isPalindrome(0, 5)
isPalindrome(1, 4)
isPalindrome(2, 3)
isPalindrome(3, 2)

if (start > end)
return True

F(0, 4) → F(1, 3) → F(2, 2)

Single character

if(start ≥ end)
True;

Str

bool isPalindrome (int s, int e) {
if(s ≥ e)
return true;

O(1)

→ return str[s] == str[e] && isPalindrome (s+1, e-1)

(True) (False) (False)

0 1 2 3 4

Str: a b d c a

F(0,4) → F(1,3)

Str: a b b c
0 1 2 3

F(0,3)

string str;

str[1] ≠ str[3]

str[0] == str[2]

T.C: $(\frac{n}{2}) \times 1 = O(n)$

S.C: $(\frac{n}{2}) = O(n)$

0 1 2 ---- 9 8 6 4

F(0,9) → F(1,8) → F(2,7) → F(3,6) → F(4,5)

Start: 0 → $\frac{n}{2}$ Recursion

end n-1 → $\frac{n}{2}$

F(5,4)

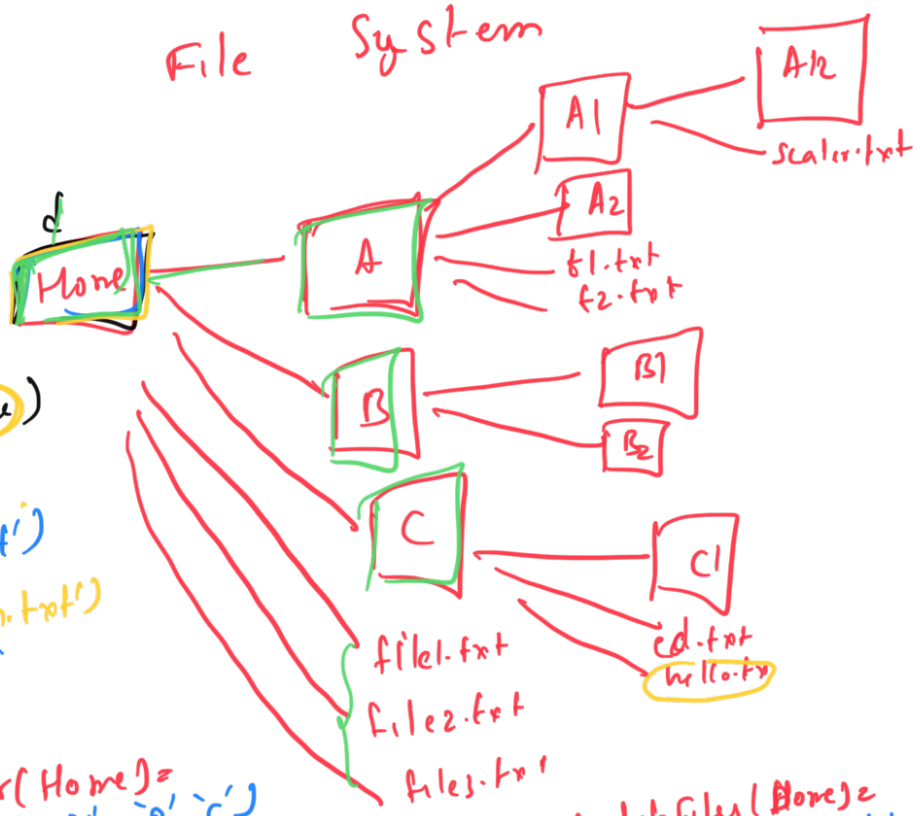
(n) → 0

$\frac{n}{2}$

$(\frac{n}{2})$

Question.

File System



(Home, filename)

F(Home, 'hello.txt')

F(Home, 'random.txt')
Later

fetchDir(Home) = ['A', 'B', 'C']

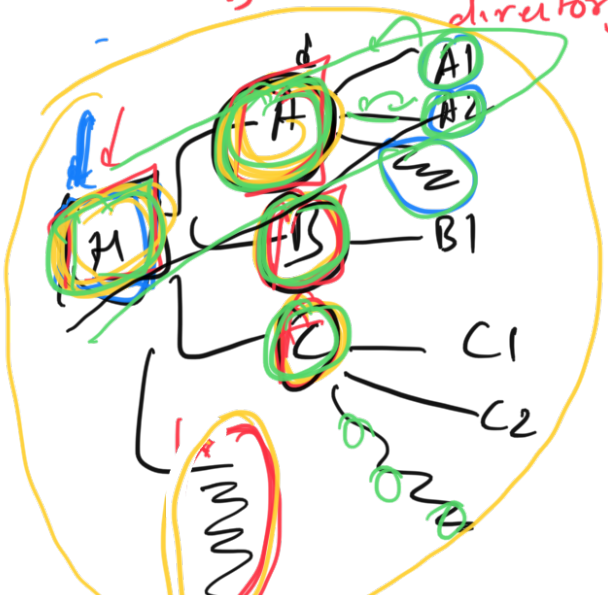
fetchFiles(Home) = [file1.txt, file2.txt, file3.txt]

Available Functions

{

1) fetchDir(x) : takes directory as argument and returns all the directories inside x.

2) fetchFiles(x) : takes directory as argument & returns all files inside that directory.



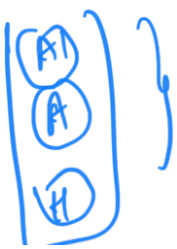
'H' : ['A', 'B', 'C']

'A' : 'A1', 'A2'

(filename)

D directories

F files



Assumption: fileInDir(dir, file) : check if file exists in the directory

Main Logic : check if file is there in fetchFiles(dir) : True;

```
for d in fetchDir(dir):
    if (filePresent(d) == True)
        return True;
return False;
```

```
bool filePresent(Dir, target) {
    if (target is in fetchFiles(Dir)) return true;
    dir_list = fetchDir(Dir);
    for d in dir_list:
        if filePresent(d, target) return True;
    return False;
}
```

Pseudocode

O(F) D(F) Expl fetchFiles(Dir) [] fetchDir(Dir) Temp dir_list (Brahm)

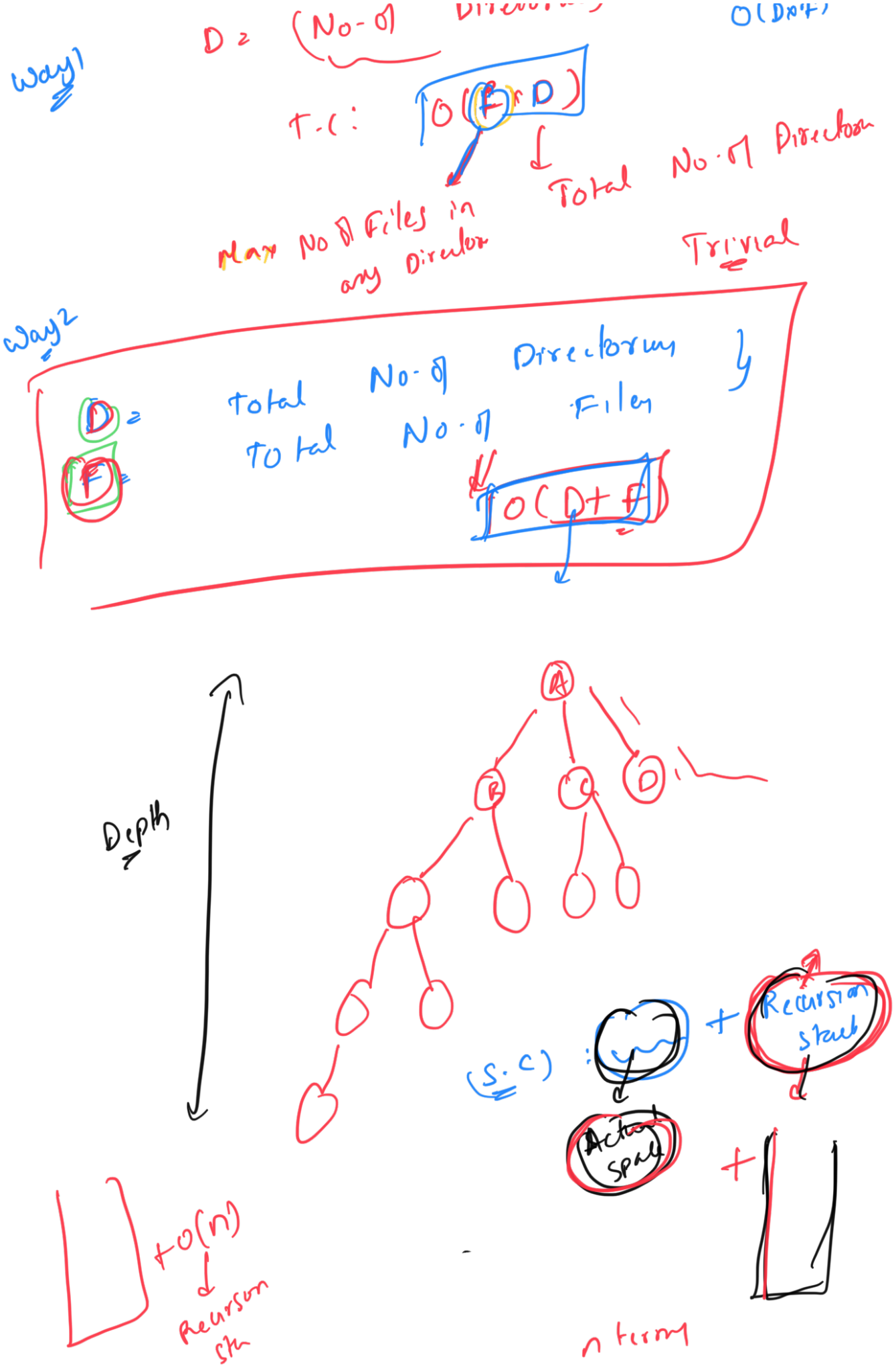
$F(n) = F(n-1) + n$

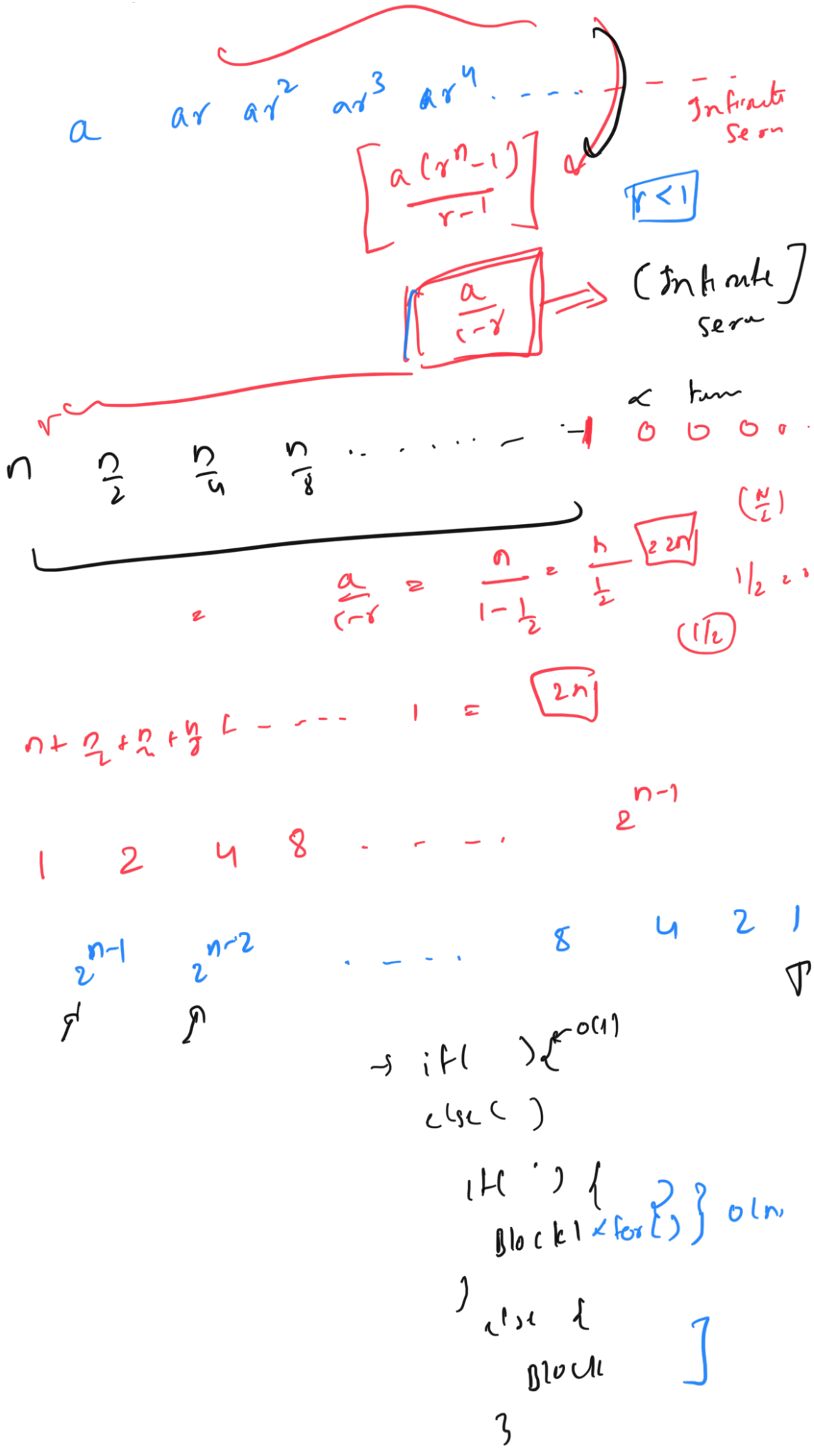
$F(n) - F(3) - F(2) - F(1)$

$F(-2) \quad F(-1) \quad F(0)$

T.C : # Functions x T.C Per Function

(Dir.length) x t


Way1
D = (No-of
O(D*F)
T.C:
O(F*D)
Total No. of Directories
Max No of Files in any Director
Trivial
Way2
D =
F =
Total No. of Directories
Total No. of Files
O(D+F)
Depth
A
B
C
D
(S.C) :
+
Recursion stack
Actual space
+
+O(n)
Recursion stk
n terms


a ar ar^2 ar^3 ar^4 . . . .
Infinite Series
[a(r^n - 1) / r - 1]
r < 1
a / 1 - r
(Infinite Series)
n n/2 n/4 n/8 . . . . .
∝ turn
0 0 0 0 .
(n/2)
a / 1 - r = n / 1 - 1/2 = n / 1/2 = 2n
1/2
(1/2)
n + n/2 + n/4 + n/8 + ... 1 = 2n
1 2 4 8 . . . . 2^(n-1)
2^(n-1) 2^(n-2) 8 4 2 1
O(1)
if( ){
else( )
if( ) { Block1 <for{ } } else
} else {
Block
}