Intro to Recursion

Recursion ⇒ Dynamic Programming

Recursion: Function calling itself
↳ to solve a problem (Objective)

Solving a problem using smaller instance of same problem is called Recursion

sum(n) ⇒ Returns sum of first 'n' natural nos

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

for(i=1; i<=n; i++)
sum += i;

sum(n-1)

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

Bigger Problem: sum(n)
Smaller Problem :) sum(n-1)

Solving a problem using sub-problems is called Recursion.

Result

1+2+3+4 = ?

if(n==1)

sum(4) = 10

sum(4) = 4 + sum(3)

sum(3) = 3 + sum(2)

sum(2) = 2 + sum(1)

sum(1) = 1 + sum(0)

sum(0) = 0

sum(2) = sum(2) + sum(0)

Infinite Loop { sum(4) → s(3) → s(2) → s(1) → s(0) → s(-1) → ... s(3) ← s(2) ← s(1)

0 + sum(-1)

3 steps to Recursion

1) **Assumption**: What do we want our function to do.

sum(n) → Return sum of first N natural nos

2) **Main Logic / Recursive Relation**

Solve the problem using smaller sub-problems

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

3) **Base condition**

When do you want to stop the recursion

a) if(N == 1) return 1;
b) if(N == 0) return 0;

→ Every Recursive solution WILL have Iterative solution (Extra space)

↓ (Complex Code)

4-5 lines

Problem: N Factorial

Problem:

$5! = 5 \times 4 \times 3 \times 2 \times 1$

fact(n): $n! = 1 \times 2 \times 3 \times \cdots \underbrace{(n-2) \times (n-1)}_{fact(n-1)} \times n$

$\Rightarrow fact(n) = n \cdot fact(n-1)$

1) Assumption

fact(n) $\Rightarrow$ Return N!

2) Main Logic

fact(n) = n x fact(n-1)

3) Base condition:

if(n == 1) return 1; ✗

if(n == 0 || n == 1) return 1;

0! = ? $\Rightarrow$ 1

0! = 1

P1: Sum of n natural nos

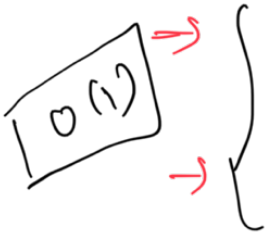
```
int sum(N)
{
    if(N == 0) return 0;

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

O(1)

Fact(n) fact(-5) → (n == 0) ✗

P2:




```
int fact (...
    if(n< 0) { return ...
    if( n == 0
    return n * fact(n-1);
}                                   } return 1;
```

f(-5) → f(-6) → f(-7)

fact(0)

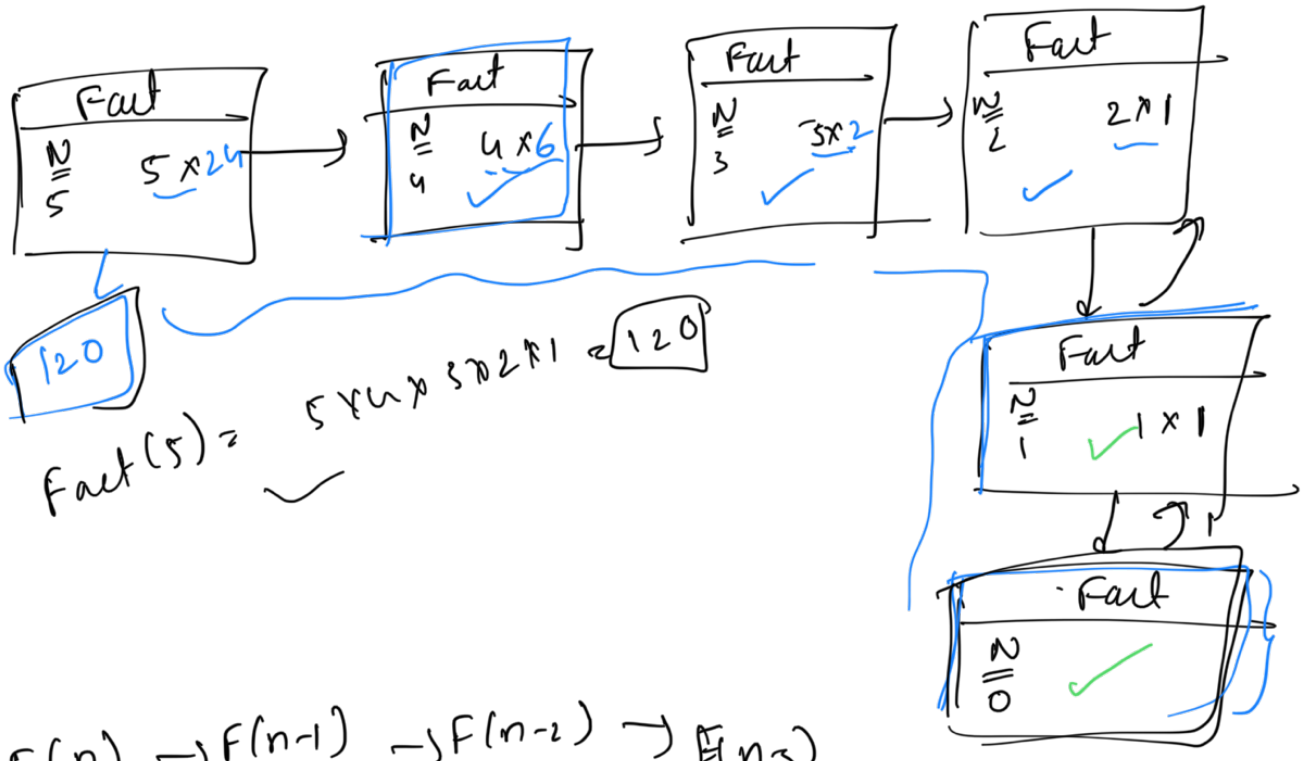
0 * fact(-1)

f(0) → f(-1) → f(-2) . . . .

fact(2) = 2 * fact(1) = 1 * fact(0) = 1

fact(-5)

Fact(5)



Fact(5) = 5 x 4 x 3 x 2 x 1 = 120

F(n) → F(n-1) → F(n-2) → F(n-3)

F(n-4) → F(n-5)

Stack

↓

LIFO => Last In First Out

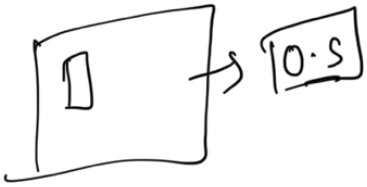
→ Put chair on top

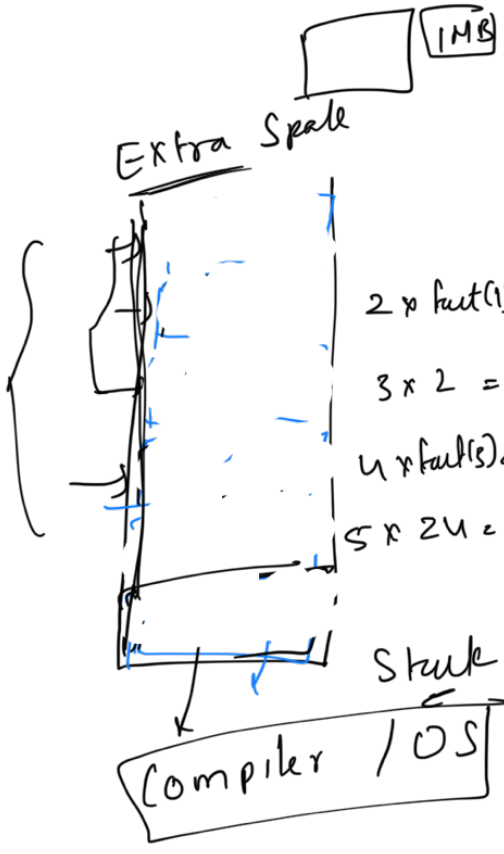
→ Remove from top

stack of stools

Stack: We can access only data at the top

stack






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

10^5

int: 2×10^9

$N = 10^5$

$(N)! \leq 10^{18}$

$fact(N) \leq 10^{18}$

fact(N) % M

Time complexity

$\Rightarrow$ O(# function calls × T.C per function call)

fact(n) → fact(n-1) → fact(n-2) → fact(n-3) → ⋮ → fact(1)

N function calls ☆

T.C of factorial: $O(N \times 1) \Rightarrow O(N)$

T.C of sum of N natural nos?

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

Space complexity

fact(n)

↑ function calls ↓ (n)

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

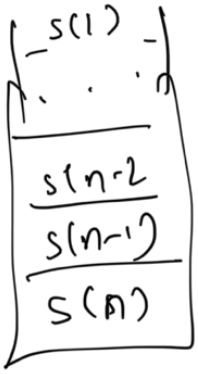
Max no. of active functions in the stack at any point

S.C: (Max No. of active functions)

$= O(N)$

S.C of sum of n natural nos.

n function calls



2) O(N) S.C (Merge Sort)

O(N)

```
function(N) {
    int arr[N];
}
```

(3 hrs)

S.C: (N x N) = O(N^2)

Iterative solution

```
for(i=1; i<=n; i++)
    ans = ans * i;
```

T.C: O(N)

S.C: O(1)

Question: Fibonacci series

1 1 2 3 5 8 13 21 34 55

N=1 N=2 N=3 N=4 5 6 7 8

Find N^{th} Number in Fibonacci series.

Fib(8) = 21

N = 20

Fib(10) = 55

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

2 calls: 2 Base Cases

1) Assumption

Fib(n) => Return the N^{th} Fiborazcci Number

2) Main Logic

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

3) Base Condition

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

Fib(3) = Fib(2) + Fib(1)

(1)

Fib(2) = Fib(1) + Fib(0)

Sequentially

→ code Runs

int fib(N)

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

return fib(n-1) + fib(n-2);

O(1)

(N-1)

}

(3)

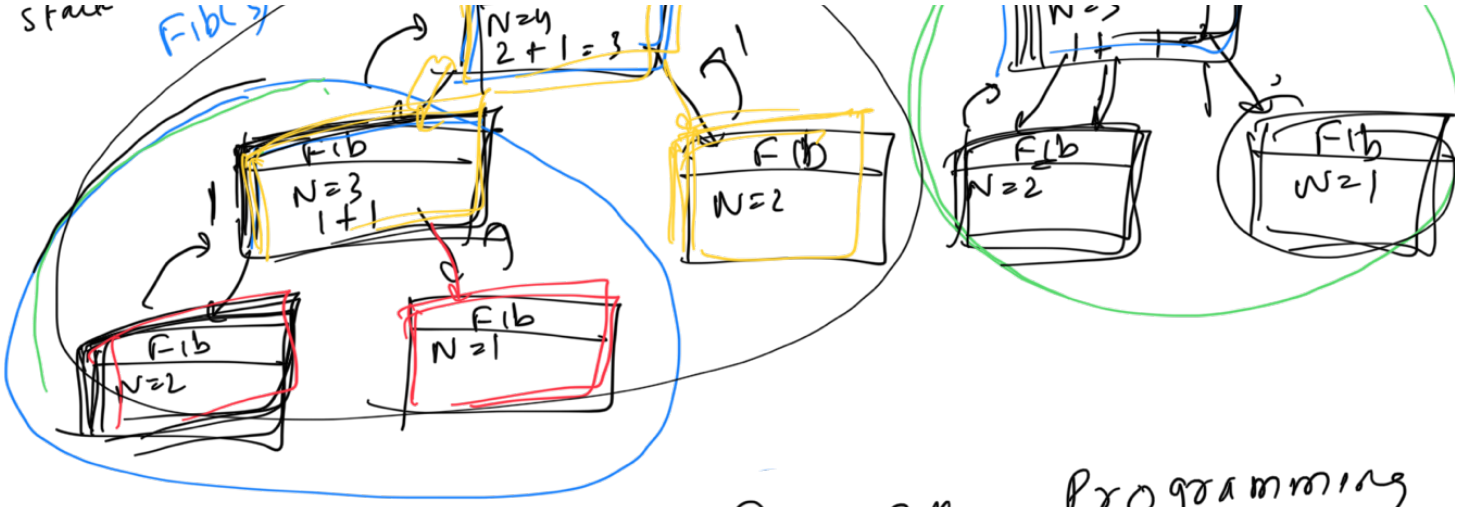
(2)

= 3 + 2 = 5

Fib N=5

Fib

Fib



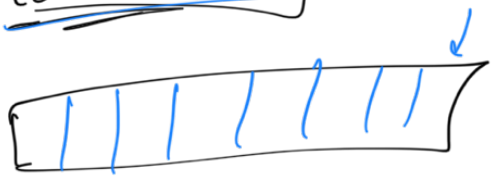
Idea of Dynamic Programming

Memoization

15

Array:

Hashmap:



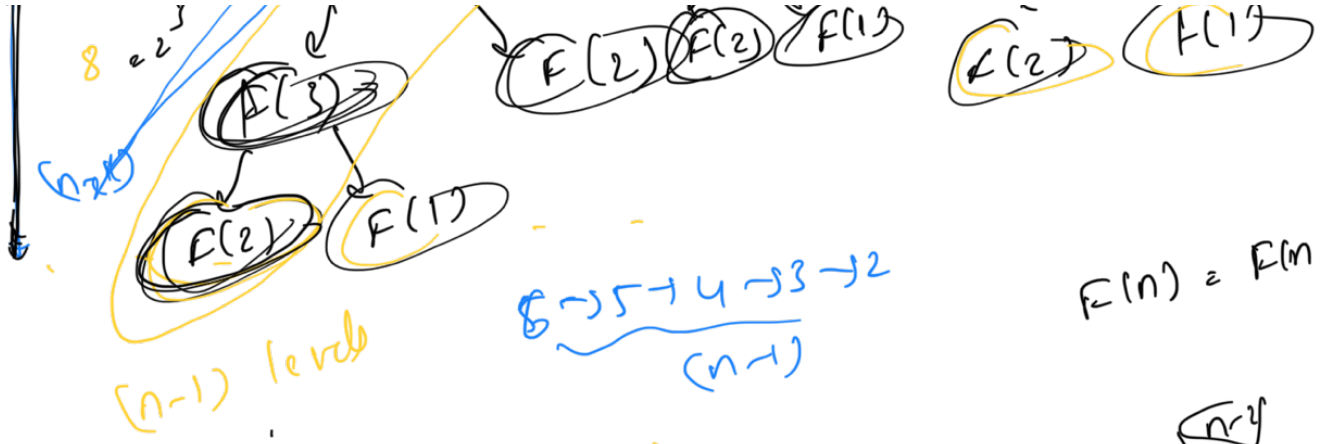
Advanced

S.C.: (n-1) function calls

S.C: O(n)

$(2^{n-1}-1) \times 1 \Rightarrow O(2^n)$

T.C: O(# function calls x T.C per function)



$8 = 2^3$

$n \times k$

$(n-1)$ levels

$5 \to 4 \to 3 \to 2$

$(n-1)$

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

$1 + 2 + 4 + 8 + \dots$

$1 + 2^1 + 2^2 + 2^3 + \dots$ $(n-1)$ terms

$n-1$

2

$a = 1$

$r = 2$

$n = n-1$

$\frac{a(r^n - 1)}{r-1}$

$a \Rightarrow$ 1st term

$r \Rightarrow 2$

$n \Rightarrow$ No of terms

$1 \left(\frac{2^{n-1} - 1}{2-1}\right) = \left(2^{n-1} - 1\right) \times 1$

Function calls

$= O(2^{n-1} - 1)$

$= O(2^n)$

$O(2^{n-1}) = O\left(\frac{2^n}{2}\right)$

$= O(2^n)$

T.C: $O(2^n)$

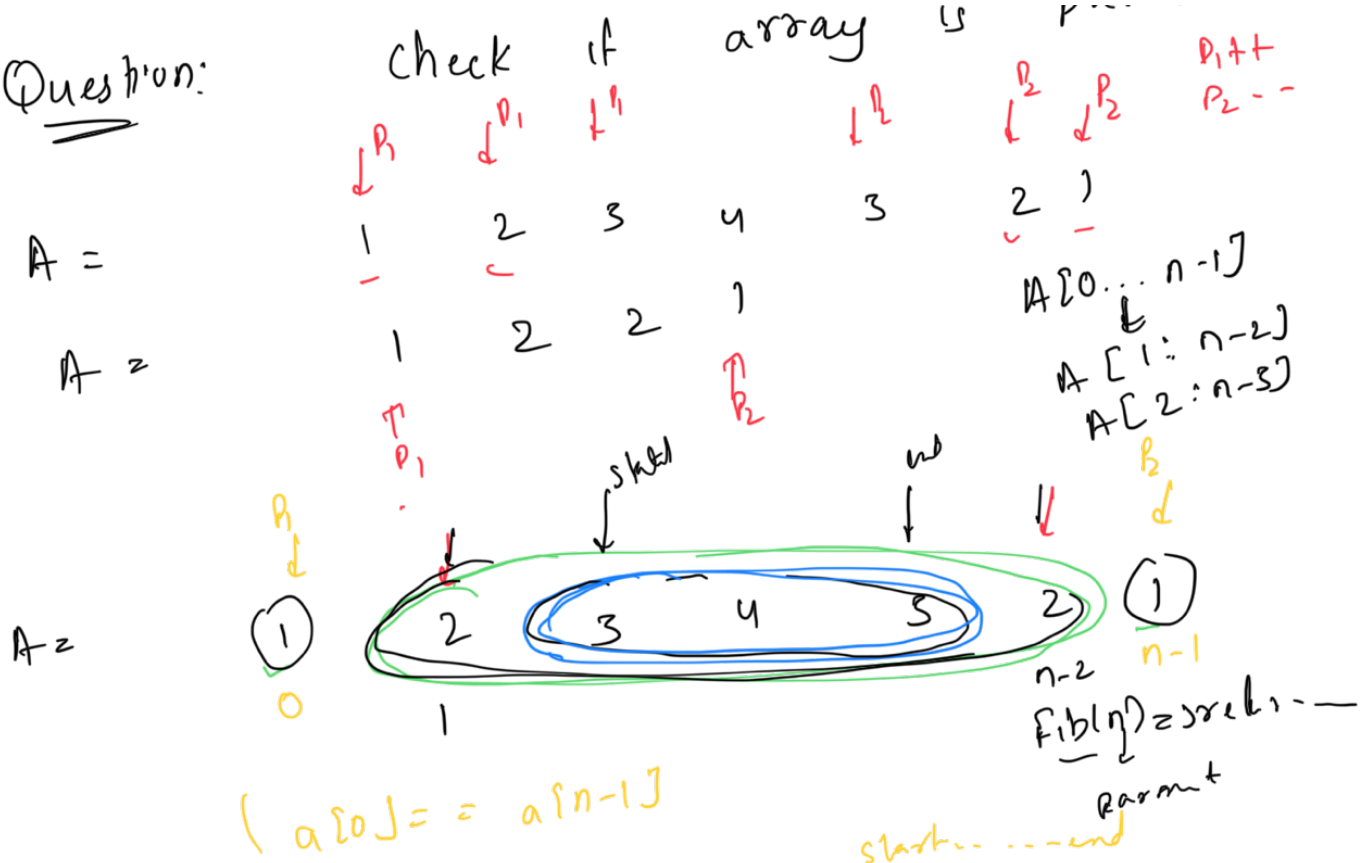
S.C: $O(n)$

S.C: $O(n)$

```
fun(n) {
  int arr[N];
}
```

palindrome

Question: Check if array is



1) Assumption :

isPalindrom(arr, start, end) :
check if arr[start ... end] is a palindro

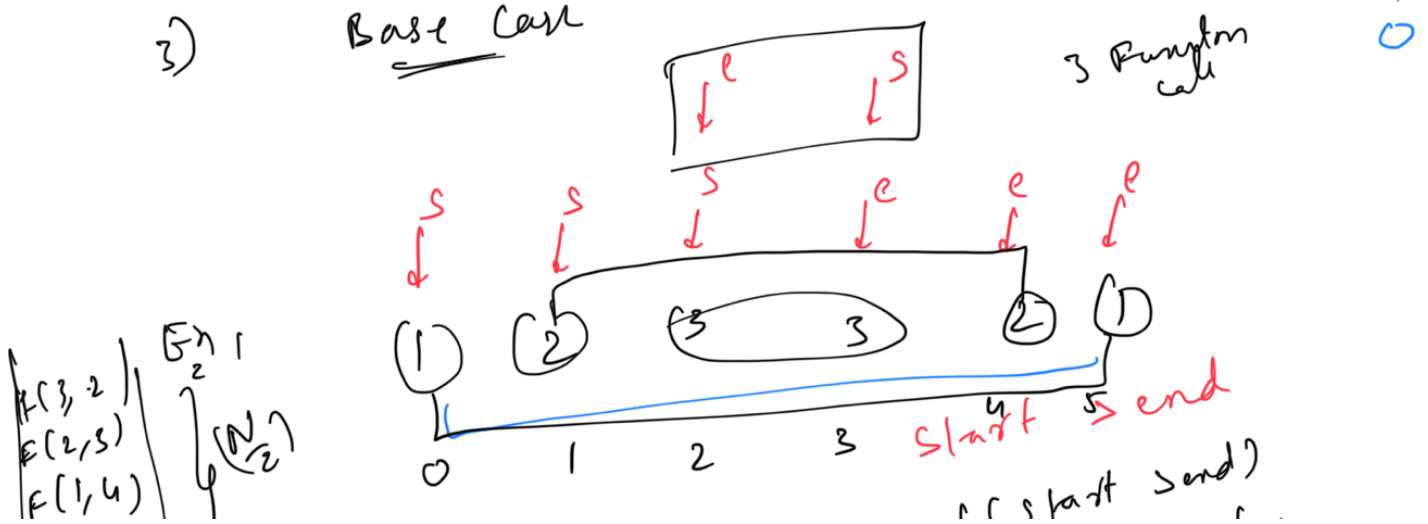
2) Main Logic

O(1)

{ { arr[start] == arr[end] && isPalindrome(arr, start+1, end-1) == true

n → (n-2) → (n-4) ...

3) Base case



F(0,5)

return true

1 2 3 2 1

return true;

if (start ≥ end)
return true

T.C: (# function calls × T.C per function)

F(0, n-1)
↓
F(1, n-2)
↓
F(2, n-3)

n → (n-2) → (n-4) 0

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

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

S.C: O(N)

F(2,2)
F(1,3)
F(0,4)

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

$O(\frac{n}{2} \times 1) = O(N)$

F(0, n-1)

true

F(1,3) = true

True

1 2 3 4 5 6 5 4 3 2 1

0 1 2 3 4 5 6 7 8 9 10

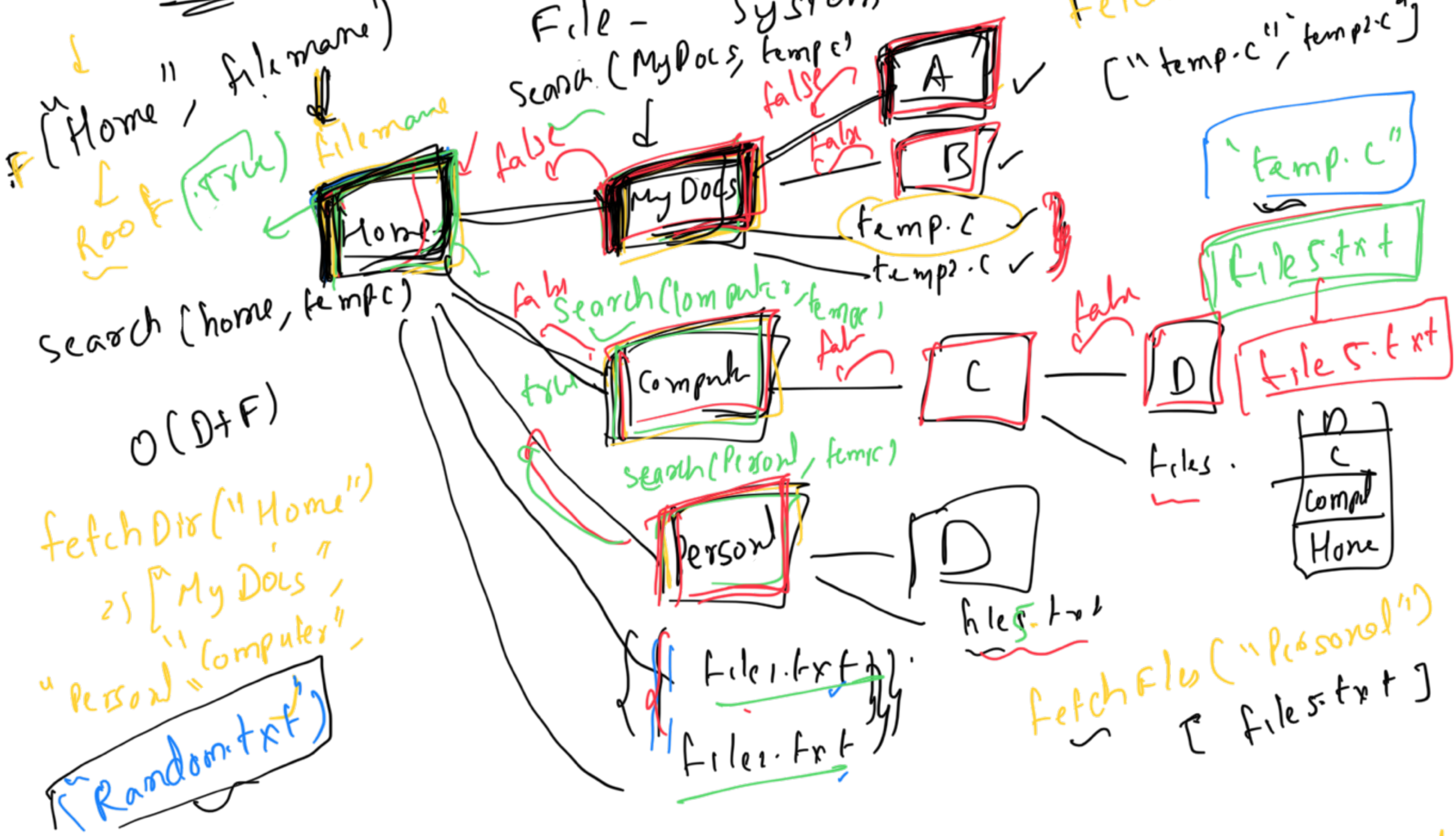
(0, 10)
(1, 9)
(2, 8)
(3, 7)
(4, 6)
(5, 5)

6 times

$\frac{n}{2} + 1 \Rightarrow O(n)$

$\frac{n}{2} + 1 = O(n)$

Question: Search for a file in a File - System



filename

Take directory as an argument

1) fetchDir(x): takes ... returns list of Directories inside it

2) fetchFiles(x): Returns list of files present inside it.

1) Assumption

search(Dir, filename): to check if this file is present in my Directory or not.

2) Main Logic

if filename is in fetchFiles(Dir)

OR

for all sub-dir in fetchDir(Dir):
if(search(sub-dir, filename) = tr
return true

return False

search(NULL, temp)

bool search(Dir, file) {
if(Dir == NULL) return False
if(file in fetchFiles(Dir)
return true;
for sub-dir in fetchDir(Dir)
if(search(sub-dir, file) == true)
return true;
}
return false;

O(F)

O(f+d)

(List) (f) O(f)

(List) (d) O(d)

"A" "B" "C" "D"

return

= ["A", "B", "C"]

Approach 1

T.C : (# Function Calls × T.C Per Function)

D : Total No of Directories

F : Max No. of Files in any Directory

T.C : O(D × F)

Approach 2

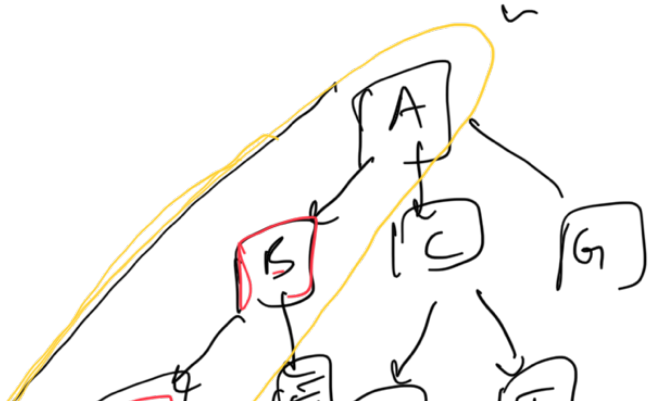
D : Total No. of directories.

F : Total No. of files

T.C : O(D+F)

S.C : (# Max No. of active Function × S.C Per Function)

Depth of Directory structure



Depth of Directory structure

S.C: O(Depth * (f+d))

f: Max no. of files in any directory

d: " " sub dirs in any directory

N = No of files + Directories

D, F, N

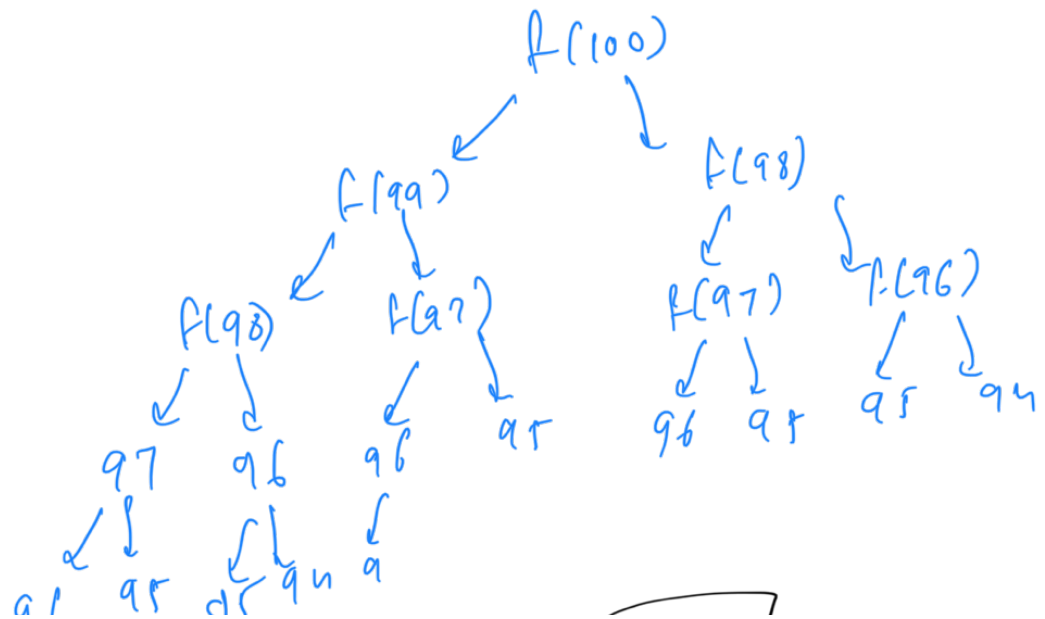
O(N)

N dirs
m F:

O(N+M)

fact(3) → 3 * fact(2) → (2 * fact(1))

F(n-1) will give control to F(n)



2^0 2^1 2^2 2^3 2^4

Binary Search Tree

1) Hash map (More Space than Required)

hashmap< username, feed >

(O(1))

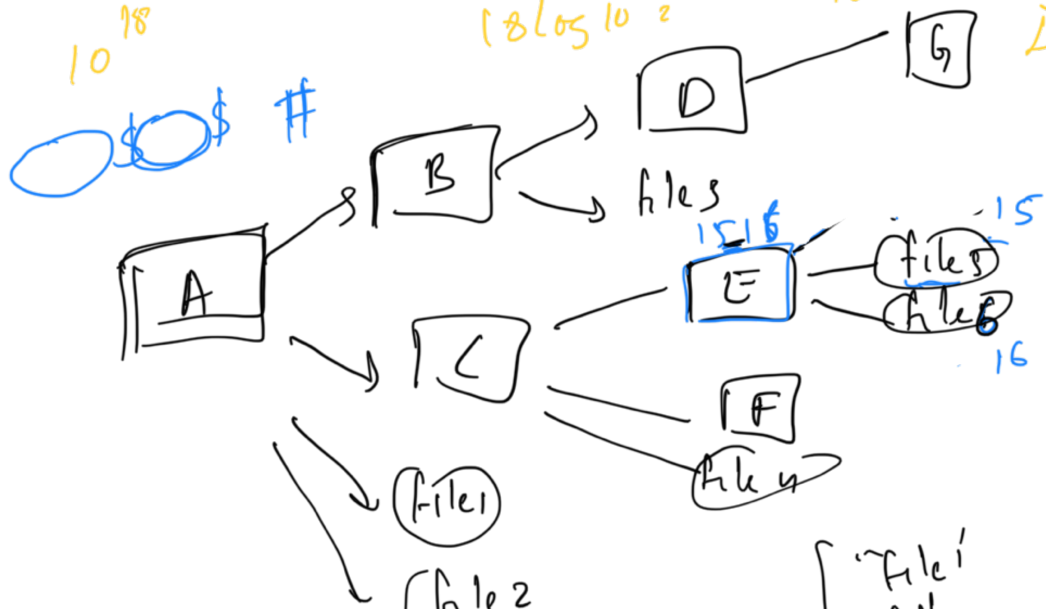
2) Binary Search Trees => (Consume Space that is req.

N Users

O(log n) times

18 log 10 ≈

18 × 3 ≈ (54) oper.



["file" f1

[Precomputation :

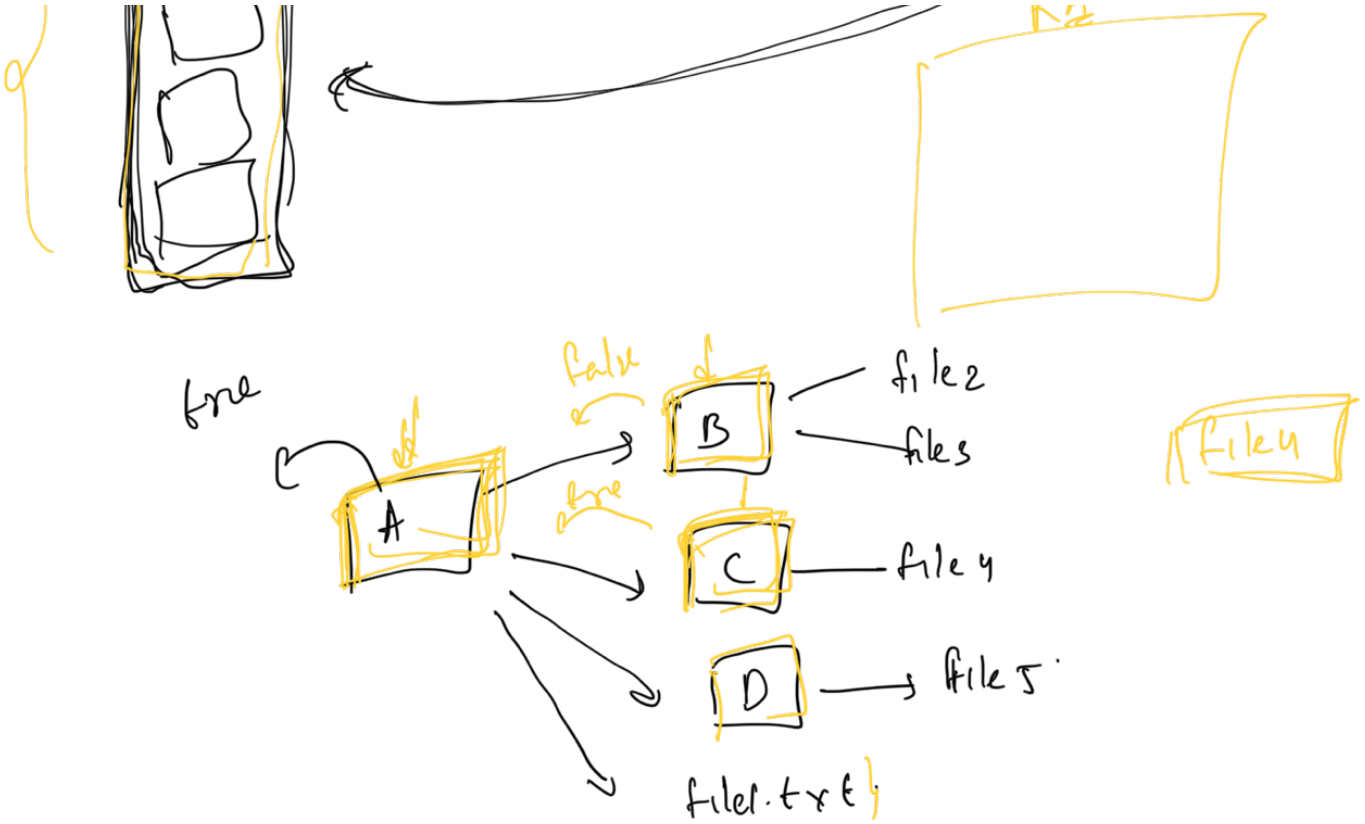
]

[Precomputation

(Experiment)

$(10^9) \times 4$ Bytes)

RAM



yahnit.sirineni_1@scaler.com