→ Function calling itself! (To solve some problem)

→ Solving a problem using smaller sub-problems

sum(N) => Sum of N natural nos

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

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

sum(N) → Bigger Problem

sum(N-1) → smaller Problem

sum(4)
→ 4 + sum(3)
→ 3 + sum(2)
→ 2 + sum(1)
→ 1 + sum(0) ← Base Case.
→ 0 + sum(-1)

Base Case: if (N == 0)
return 0;

3 steps of Recursion

1) Assumption: Decide what you want your function to do.
sum(N) returns the sum of first N natural numbers

2) Main Logic: solve the problem using smaller sub-problems
sum(N) = sum(N-1) + N

3) Base Case: When do you want to stop Recursion
if (N == 0)
return 0;

```
int sum(N) {
    if(N == 0)
        return 0;
    return sum(N-1) + N;
}
```

T.C: O(N x 1) => O(N)

S.C: O(N)

sum(5) → sum(4) → sum(3) → sum(2) → sum(1) → sum(0)

Question: Factorial

5! = 5 x 4 x 3 x 2 x 1

5! = 4! x 5

fact(N) = fact(N-1) x N

1) Assumption: fact(N) returns N!

2) Main Logic: fact(N) = fact(N-1) x N

3) Base Case: if(N == 1 || N == 0) return 1;

```
int fact(N) {
    if(N<0) { "ERROR" };
    if(N == 0) return 1;
    return fact(N-1) x N
}
```

fact(4) → fact(3) → fact(2) → fact(1) → fact(0)

O(N x O(1)) => O(N) T.C, O(N) S.C

fact(2) -> 2 x fact(1) -> 2 x 1 x fact(0) => 2 x 1 = 2!

Question: Compute a^N

Approach1:

$a^5 = a^4 \times a$

$pow(a,N) = pow(a,N-1) \times a$

$a^N = a^{N-1} \times a$

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

$= O(N)$

$\Rightarrow O(N)$

Assumption: pow(a,N) returns a^N

Main logic: pow(a,N) = pow(a,N-1) x a

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

Approach2: (Fast Exponentiation)

$a^{12} = a^6 \times a^6$ ($a^N = a^{N/2} \times a^{N/2}$)

$a^{16} = a^8 \times a^8$

$a^{13} = a^6 \times a^6 \times a$ ($a^N = a^{N/2} \times a^{N/2} \times a$)

$a^{17} = a^8 \times a^8 \times a$

Assumption: pow(a,N) returns a^N

Main logic:

```
if(N & 1 == 0){
    pow(a,N) = pow(a,N/2) x pow(a,N/2)
}
else{
    pow(a,N) = pow(a,N/2) x pow(a,N/2) x a
}
```

Base Case : if(N == 0)
return 1;

```
int pow( a, N ) {
    if (N == 0) return 1;
    if (N & 1 == 0) {
        return pow(a,N/2) x pow(a,N/2)
    }
    else {
        return pow(a,N/2) x pow(a,N/2) x a
    }
}
```

O(1) ← pow(a,N/2) : T(N/2), pow(a,N/2) : T(N/2), O(1)

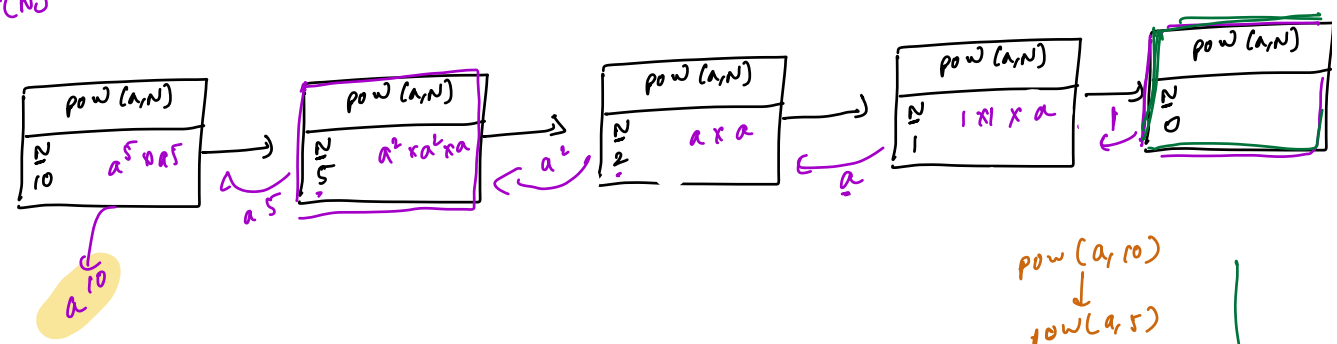
$T(N) = T(\frac{N}{2}) + T(\frac{N}{2}) + O(1)$

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

```
int pow( a, N ) {
    if (N == 0) return 1;                    → O(1)
    half_power = pow(a,N/2);                 T(N/2)
    if (N & 1 == 0) {
        return half_power x half_power       → O(1)
    }
    else {
        return half_power x half_power x a   → O(1)
    }
}
```

O(1)

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



pow(a,10)
↓
pow(a,5)
↓
pow(a,2)
↓
pow(a,1)
↓
pow(a,1)

Time Complexity

1) Recursion Tree Method

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

↓ ↓

logN O(1)

O(logN)

S.C: O(logN)

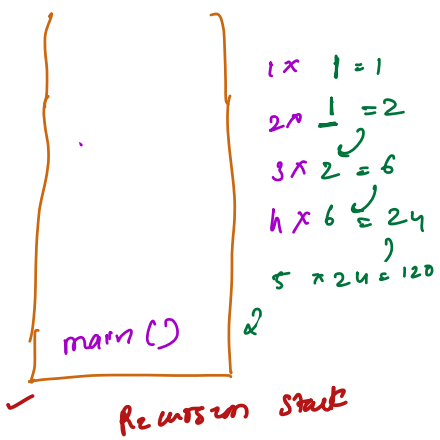
Recursion Stack

1) Whenever a function is called, some space is allocated on top of stack

2) the space is released after the function is completely executed

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

int main() {
    fact(5)
}
```

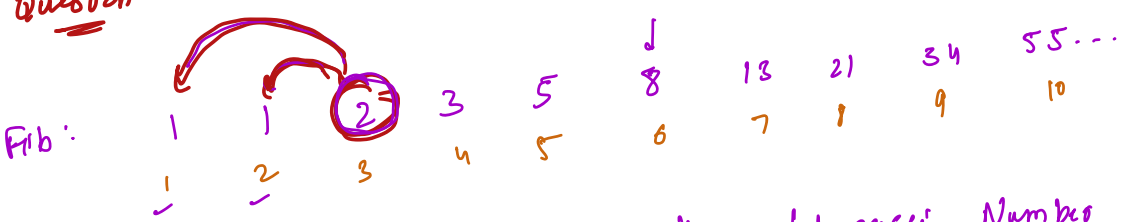


Space Complexity

O(Max # Active Function calls x S.C per function)

N x O(1) = O(N)

Question: N^{th} Fibonacci Number ($N \geq 1$)



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

Main Logic: fib(N) = fib(N-1) + fib(N-2)

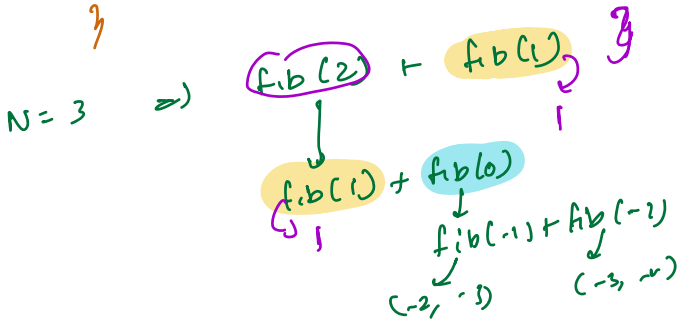
Base Case:

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

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

if(N <= 2) return 1;

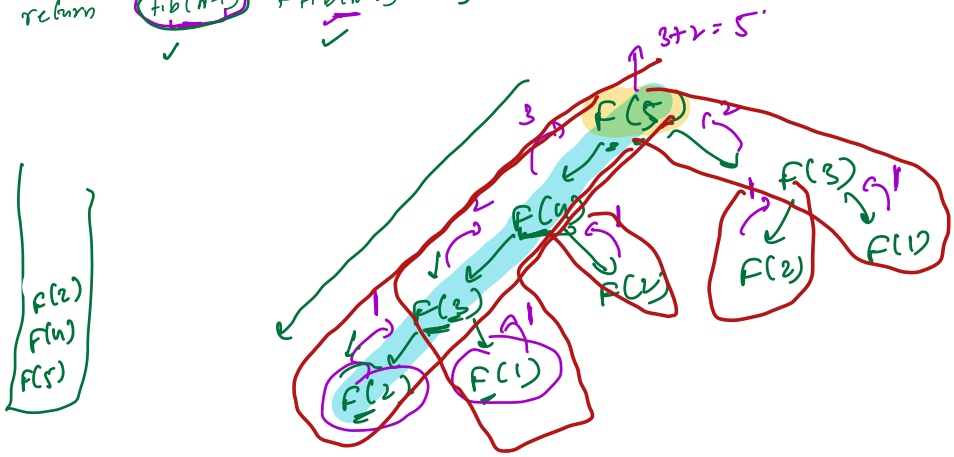
N = 3 =>



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

O(1)

Recursion Tree

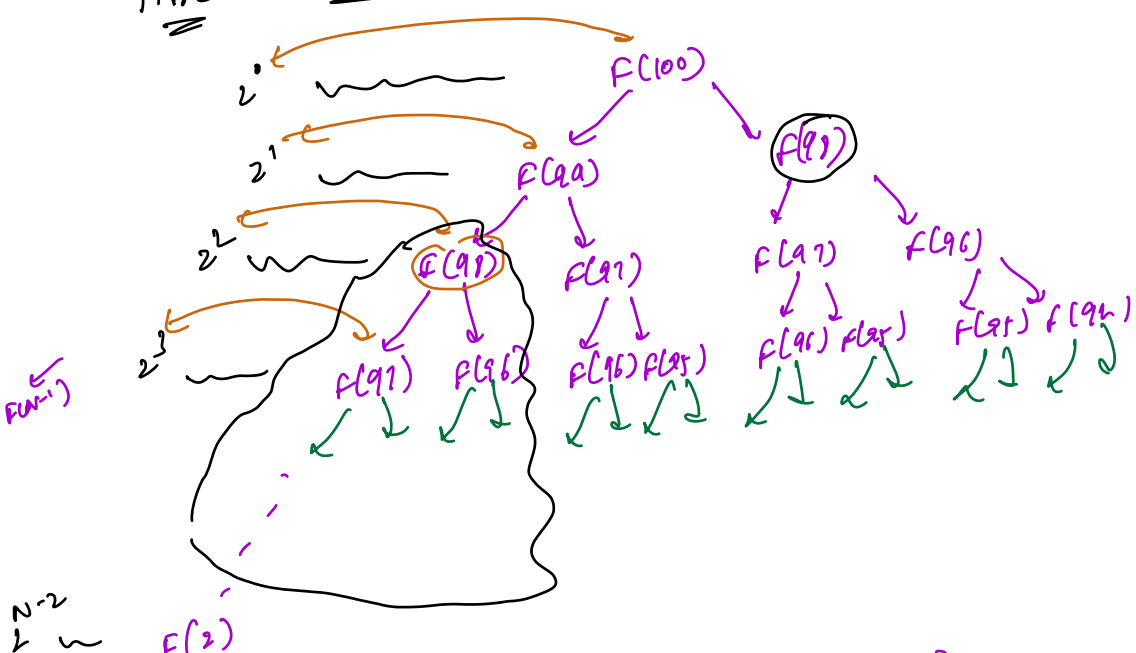


Space Complexity :

max Active Function calls = Height of the tree

O(N x O(1)) = O(N)

Time Complexity



$2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{N-2}$

$[0, 1, \ldots, N-2]$

$n-1$

$a = 1$

$r = 2$

$K = n-1$

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

Function calls $\approx 2^{N-1} - 1 = \frac{1(2^{N-1} - 1)}{2-1}$

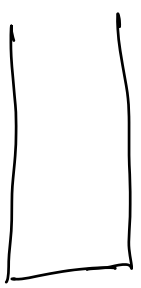
$= \frac{2^N}{2} - 1$

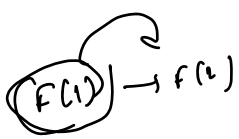
$= 2^N$

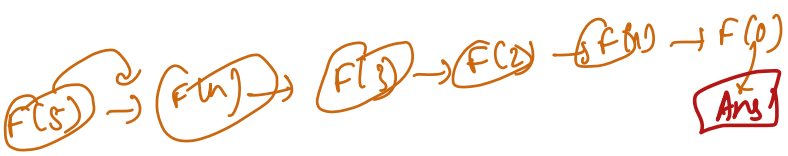
T.C: $O(2^N + 1) = O(2^N) \rightarrow O(N)$

[Memoization]

Break: 8 mins







Tail Recursion

If the recursive call is the last thing to be executed in the function, then its called tail recursion

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

N = 5 => 5 4 3 2 1

YES

```
int sum(N) {
  if(N==0)
    return 0;
  return sum(N-1) + N;
}
```

Last operation

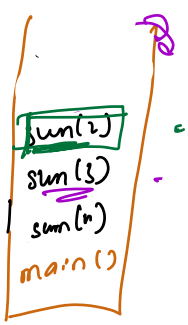
NO

Can be converted to Tail Recursion (HW)

```
int fun(N) {
  return fun(N-1) + fun(N-2);
}
```

More than one distinct recursive function calls: Cannot be Tail Recursed

```
void solve(N) {
  if(N==0) return;
  print(N)
  solve(N-1);
}
main() {
  solve(4)
}
```



sum(0) + 1 + 2
= sum(1) + 2

Output: 4 3 2 1

No need of Recursion stack.

Every Recursive code → Iterative

↓

Can be changed to tail Recursive code → O(1) space

Cannot be changed tail Recursive code → Implicit stack

Time Complexity

1) Recursion Tree Method

O(# Function calls × T.C per function)

2) Substitution Method

```
int sum(N){
  if(N==0) return 0;        -> O(1)   T(N) = T(N-1) + O(1)
  return sum(N-1) + N;
}
```

T(N-1) — O(1)

T(0) = O(1)

→ We want to find the time the function takes for input N

Lets assume T(N) is the time taken to compute sum(N)

sum(N) → T(N)

sum(N-1) → T(N-1)

1) $T(N) = T(N-1) + O(1)$

$T(N-1) = T(N-2) + O(1)$

2) $T(N) = T(N-2) + 2 * O(1)$

$T(N-1) = T(N-3) + O(1)$

3) $T(N) = T(N-3) + 3O(1)$

⋮

K) $T(N) = T(N-K) + K\,O(1)$

$T(0) = O(1)$

$k = N$

$T(N) = O(N-N) + N\,O(1)$

$= O(1) + N \cdot O(1)$

$= O(N)$

$2\,[2T(\frac{N}{2}) + O(1)] + O(1)$

$4T(\frac{N}{2}) + 2O(1) + O(1)$

Eg2 : 1st $T(N) = 2T(\frac{N}{2}) + O(1)$

$T(\frac{N}{2}) = 2T(\frac{N}{4}) + O(1)$

$= 4T(\frac{N}{4}) + 3 \cdot O(1)$

2nd : $T(N) = 2^2 T(\frac{N}{2^2}) + 3 \cdot O(1)$

$T(\frac{N}{4}) = 2T(\frac{N}{2}) + 1$

3rd : $T(N) = 2^3 T(\frac{N}{2^3}) + (2^3 - 1) O(1)$

⋮

K steps : $T(N) = 2^K T(\frac{N}{2^K}) + (2^K - 1) O(1)$

$T(0) = O(1)$

$N \to \frac{N}{2^1} \to \frac{N}{2^2} \to \frac{N}{2^3} \cdots\cdots 1 \to 0$

$K = \log_2^N$

$\frac{N}{2^{\log_2^N}} = \frac{N}{N} = 1$

$= 2^{\log_2^N} T(\frac{N}{2^K}) + \log_2^N$

$= N T(1) + (N-1)$

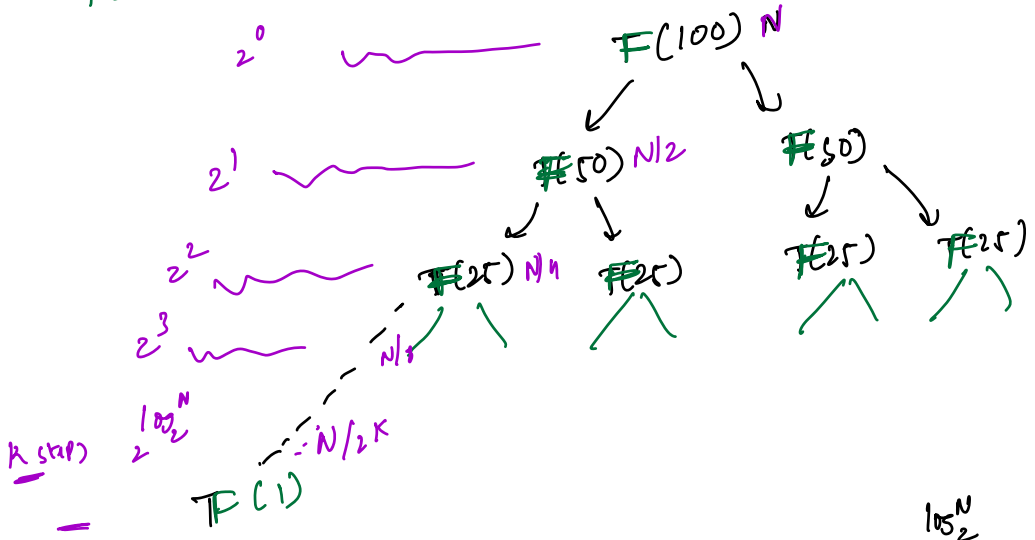
$= N O(1) + (N-1)$

T.C : $O(N)$

HW: $T(N) = T(\frac{N}{2}) + O(1)$

[Recursion Tree Method]

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



$2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{\log_2^N}$

$a^{\log_a^b} = b$

$a = 1$

$r = 2$

$K = \log N$

$S_K = \frac{a(r^K - 1)}{r - 1}$

$= \frac{1(2^{\log_2^N} - 1)}{2 - 1} = \frac{N - 1}{1}$

$= O(N - 1) \Rightarrow O(N)$

Q): $T(N) = 2T(N-1) + 1$, $T(0) = 1$

HW: Recursion Tree

1: $T(N) = 2T(N-1) + O(1)$

$T(N-1) = 2T(N-2) + O(1)$

$2[2T(N-2) + O(1)] + O(1) = 4T(N-2) + 3 \cdot O(1)$

2 $T(N) = 4T(N-2) + 3\,O(1)$

$T(N-2) = 2T(N-3) + O(1)$

$\Rightarrow$ 3 $T(N) = 8T(N-3) + 7\,O(1)$

$= 2^3 T(N-3) + (2^3 - 1)\,O(1)$

K $T(N)$: $2^K \boxed{T(N-K)} + (2^K - 1)\,O(1)$

$\boxed{K = N}$

$= 2^N T(N-N) + (2^N - 1)\,O(1)$

$T(N) = 2^N + 2^N - 1 \Rightarrow O(2^N)$

~~Fib~~ $Fib(N-1) + Fib(N-2)$

$O(1)$

Fib(N): Fib(N-1) + Fib(N-2)

$T(N) = T(N-1) + T(N-2) + O(1)$ ✓

$T(N-1) = T(N-2) + T(N-3) + O(1)$

$T(N-2) = T(N-3) + T(N-4) + O(1)$

$= T(N-2) + 2 \cdot T(N-3) + T(N-4) + 3 \cdot O(1)$

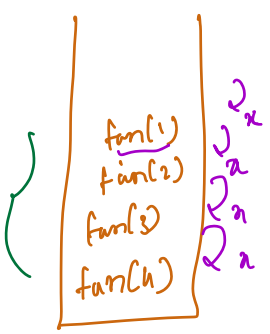
[Recursion Tree method]

If more than 1 distinct function call: Do not use substitution method

Master's Theorem

$$T(N) = aT\left(\frac{N}{b}\right) + O(n^d)$$

```
int fun(N) {
    if(N==0) return 1;
    return fun(N-1)
}
```



N = 4

if (N-1)%2 == 1) {
    task1
}
else {
    task2
}

$a \% m == b \% m$

$a \% m == (b - (b-a)) \% m$

$a \% m == b \% m$