Recursion

Recursion - Introduction

- A function calling itself.
- Solving a problem using a smaller version of the same problems (subproblem).

3 Steps to implement Recursive code

- [ ] **Assumption:** Decide what your function does and assume that it does.
- [ ] **Main Logic:** Break down and solve problems using sub problems.
- [ ] **Base Condition:** Decide when your function should stop.

Function Call Tracing

<table>
<tr>
<td>

```
Int add(N, M){
    return (N+M)
}

Int Square(N){
    Return N*N
}

Int twice(N){
    Return 2*N
}
```

</td>
<td>

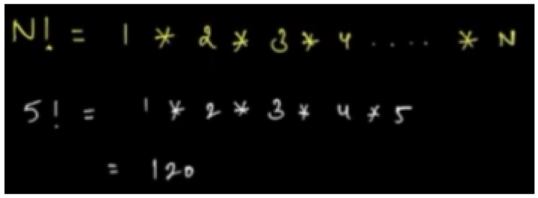
```
main(){
    Int x = add(10, 20)
    Int y = Square(x);
    Int z = twice(y);
    print(z);
}


main(){
    print(twice square(add(10, 20)))
}
```

</td>
</tr>
</table>

Add -> square -> twice
30 -> 900 -> 1800

Problems

Problem 1: Given N, find the factorial of N!



Assumption:

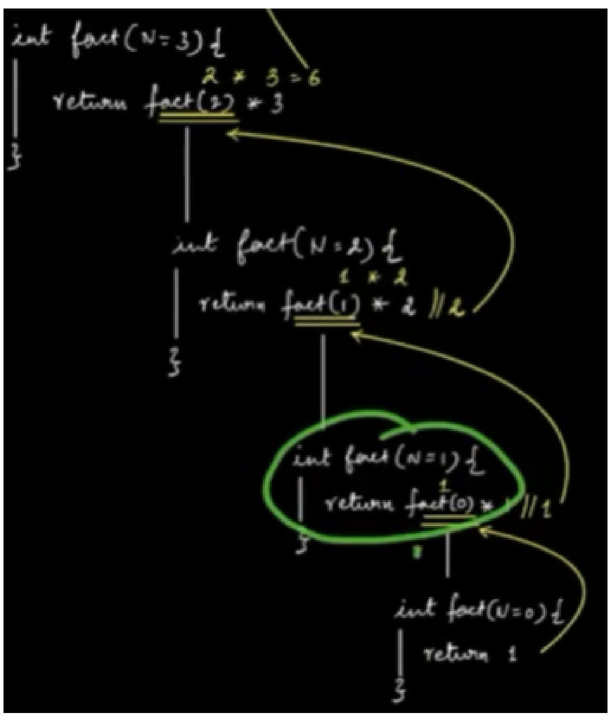
Int factorial(int N){
…..
}

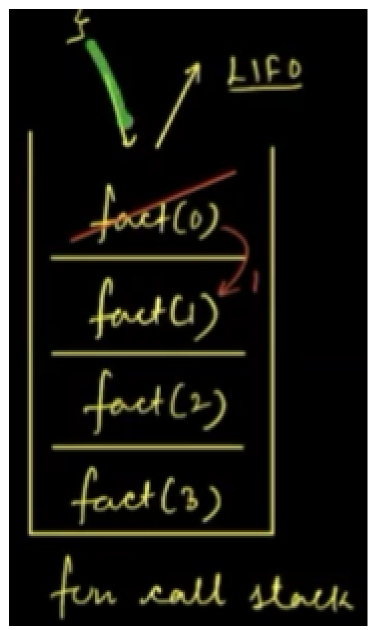
Main Logic: Try to break the problem

N! = 1*2*3*4*5*6*7*.............*(N-1)*N

Base Condition:

When N==1, return 1;





Problem 2: Fibonacci Series / Sequence

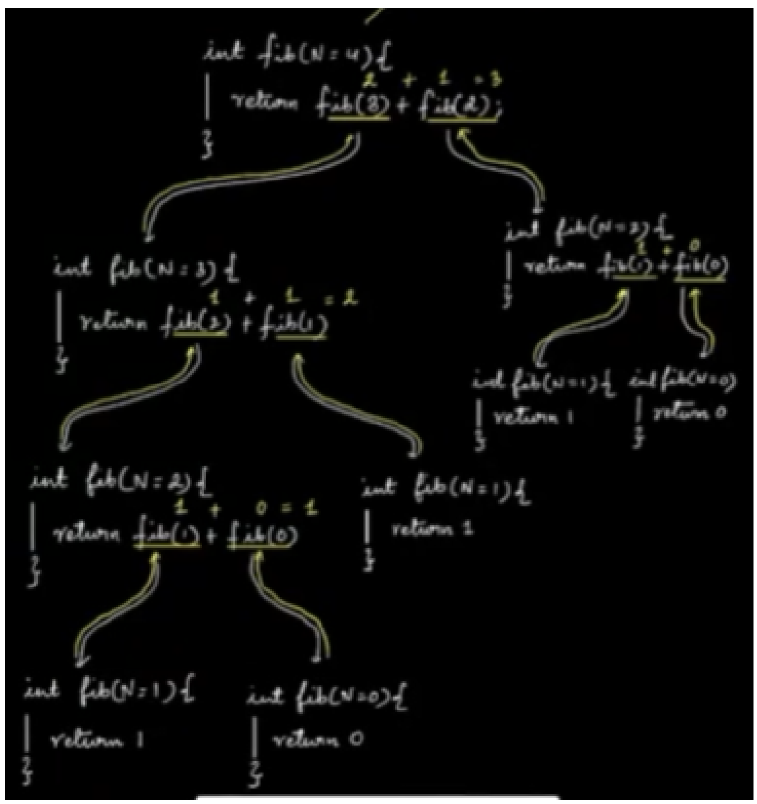
Write a function to compute Nth Fibonacci

0	1	2	3	4	5	6	7	....
0	1	1	2	3	5	8	13	

1) **Given N,** the function will calculate Nth Fib and return
```
int fib(int N){
        ………..
}
```

2) **Main Logic,**
```
return fib(N-1)+fib(N-2);
```
3) **Base Condition**
```
If (N == 0 || N == 1){
        return N;
}
```

Function Call Trace:



Problem 3:
Given 2 integers a and N. Find a^N using Recursion.

1) **Assumption:** given a and N, the function will calculate and return a^N.

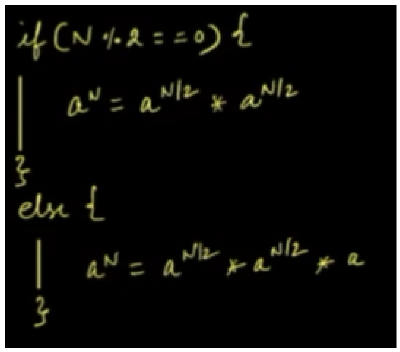
```
Int pow(int a, int N){
        …….
}
```

2) **Main Logic:**

a^N = a*a*a*......................*a (N times)

Approach 1) a*pow(a, N-1);

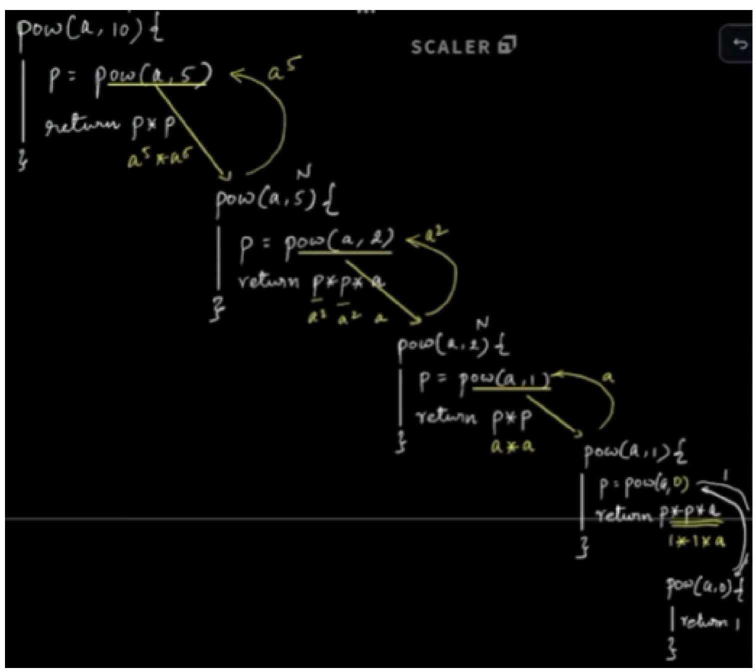
Approach 2)



3) **Base Condition:**

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

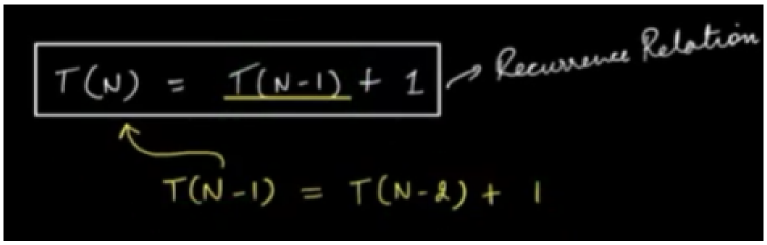
Function Call Stack:



Time Complexity For Recursive Function

- Use the substitution method to find the recursive Functions.

1. **For Factorial:**

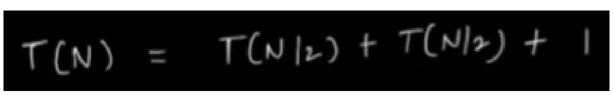


T(N) = T(N-1) + 1
T(N-1) = T(N-2) + 2
…………….
…………….
T(N) = T(N-K) + K

T(0) = 1
N-K = 0
K=N

T.c = O(N)

2. **For Pow Function**



T(N) = **2[T(N/2)] + 1;**
T(N/2) = 2[2T(N/4) + 1] + 1 = **4T(N/4) + 3**
…………………… = **8T(N/8) + 7**
………………….
Generalized Eqn : **T(N) = 2^K[T(N/2^K) + (2^K-1)]**

N/2^K = 1
N = 2^K
K = logN

T.c = O(logN)

- ## Generalized Definition Of Time Complexity

Time Complexity = No of Function Calls * Time Taken in a Single Function Call.