Recursion 1

Content

- Introduction
- Function call tracing.
- Factorial
- Fibonacci
- Power function
- TC & SC of recursive code.

Imp** Announcement

DSA Contest 1 : Topics: Arrays + Bit manipulation

Date : 19^{th} Jan 2024 9:00 PM IST.

PSP	Today
64%	65.3% {avg}

personal Target ⟶ 100%.

Introduction

Definition of Recursion — A function calling itself.

⟶ A problem solved using subproblems.

Q> Find sum of first N natural no. using recursion

$\frac{N*(N+1)}{2}$

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

How to write recursive code ?

Magic Steps

1. Assumption — Decide what the function will do.

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

2. Main logic —

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

3. Base case — Smallest problem we already know the answer for.

   sum (1) = 1

Pseudocode

```
int sum ( int N) {
    // Base case
    if (N == 1) { return 1 }

    // Main logic
    return sum (N-1) + N
}
```

Function Call Tracing

```
int add (int x, int y) {
    return x + y
}

int mul (int x, int y) {
    return x * y
}

int sub (int x, int y) {
    return x - y
}

void main () {
    int x = 10
    int y = 20

    print ( sub ( mul ( add (x, y), 30), 75))
}
```

# of function calls = 4 {print, sub, mul, add}

825

print (sub (mul (add (x, y), 30), 75))

↑ return 825 900

sub (mul (add (x, y), 30), 75)

↑ return 900

30

mul (add (x, y), 30)

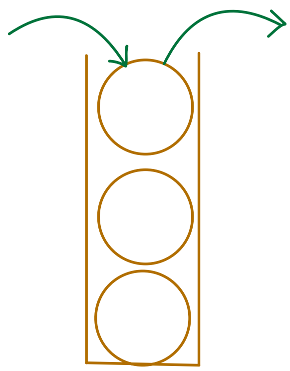
↑ return 30

output

825

add ($\overset{10}{x}$, $\overset{20}{y}$)

NOTE : All the function calls are stored in call stack.



Last in First Out.

print (sub (mul (add (x , y), 30), 75))

add { ~~add ($\overset{10}{x}$, $\overset{20}{y}$)~~ → return 30

mul { ~~mul ($\overset{30}{\text{add (x , y)}}$, 30)~~ → return 900

sub { ~~sub ($\overset{900}{\text{mul (add (x , y), 30)}}$, 75)~~ → return 825

print { ~~print (sub (mul (add (x , y), 30), 75))~~ → return null.

Output : 825

Q> Given a +ve integer N. find factorial of N.

factorial (N) = 1 * 2 * 3 * ..... N-1 * N
factorial (3) = 1 * 2 * 3 = 6
factorial (5) = 1 * 2 * 3 * 4 * 5 = 120

step 1> Assumption
factorial (N) ⟶ 1 * 2 * 3 * ..... * N-1 * N

step 2> Main Logic
factorial (N) = factorial (N-1) * N

step 3> Base case
factorial (1) = 1

Pseudocode

```
int factorial (int N) {
    // Base condition
    if (N == 1) return 1

    // Main logic
    return factorial (N-1) * N
}
```

```
int factorial ( 3 ) {  ———→  return 6
    if (N == 1) return 1
    return factorial (N-1) * 3
}
        return 2
int factorial ( 2 ) {
    if (N == 1) return 1
    return factorial (N-1) * 2
}
        return 1
int factorial ( 1 ) {
    if (N == 1) return 1
    return factorial (N-1) * N
}
```

Q) Given a +ve no. N. Find N^{th} fibonacci number

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

step 1 ⟶ fib(N) = N^{th} number in fibonacci series.

step 2 ⟶ fib(N) = fib(N-1) + fib(N-2)

step 3 ⟶ Base case
fib(0) = 0
fib(1) = 1

Pseudocode

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

    return fib(N-1) + fib (N-2)
}
```

$$\text{return } \underbrace{fib(N-1)}_{1} \underset{\downarrow \atop 3}{+} \underbrace{fib(N-2)}_{2}$$

Function call tracing

int fib (int ~~N~~ 4) { → return 3

if (N ==0) return 0

if (N = = 1) return 1

return fib (~~N-1~~ 3) + fib (~~N-2~~ 2)

}

return 2

int fib (int ~~N~~ 3) {

if (N ==0) return 0

if (N = = 1) return 1

return fib (~~N-1~~ 2) + fib (~~N-2~~ 1)

}

return 1

int fib (int ~~N~~ 2) {

if (N ==0) return 0

if (N = = 1) return 1

return fib (~~N-1~~ 1) + fib (~~N-2~~ 0)

}

return 1

int fib (int ~~N~~ 1) {

if (N ==0) return 0

if (N = = 1) return 1

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

}

return 0

int fib (int ~~N~~ 0) {

if (N ==0) return 0

if (N = = 1) return 1

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

}

return 1

int fib (int ~~N~~ 1) {

if (N ==0) return 0

if (N = = 1) return 1

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

}

return 1

int fib (int ~~N~~ 2) {

if (N ==0) return 0

if (N = = 1) return 1

return fib (~~N-1~~ 1) + fib (~~N-2~~ 0)

}

return 1

int fib (int ~~N~~ 1) {

if (N ==0) return 0

if (N = = 1) return 1

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

}

return 0

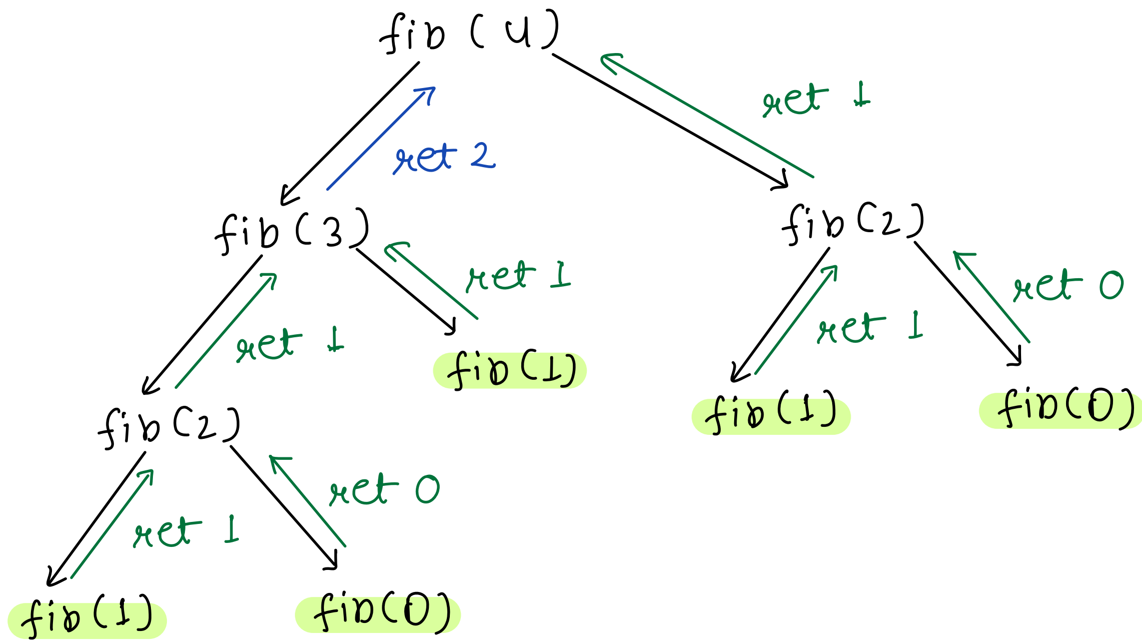
int fib (int ~~N~~ 0) {

if (N ==0) return 0

if (N = = 1) return 1

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

}


fib(4)
ret 2
ret 1
fib(3)
fib(2)
ret 1
ret 1
ret 1
ret 0
fib(2)
fib(1)
fib(1)
fib(0)
ret 1
ret 0
fib(1)
fib(0)

Q> Given 2 numbers a & n , find a^n using recursion.

$2^3 = 2 * 2 * 2$

step 1 pow (a, n) ⟶ return a^n

step 2 $pow(a,n) = \underbrace{a * a * a \dots * a}_{N}$

pow (a, n) = pow(a, n-1) * a

step 3 Base condition.

pow (a, 0) = 1

if (n==0) return 1.

Pseudocode

```
int pow ( int a, int n ) {
    if (n==0) return 1

    return pow(a, n-1) * a
}
```

TC: O(N)

pow (2, 3) ⟶ pow (2, 2) ⟶ pow (2, 1) ⟶ pow (2, 0)

$\xleftarrow{4*2}$ $\xleftarrow{2*2}$ $\xleftarrow{ret\ 2}$ $\xleftarrow{ret\ 1}$

= 8

Alternative approach

main logic

$pow(a, n) \rightarrow$ n is even: $pow(a, n/2) * pow(a, n/2)$

$pow(a, n) \rightarrow$ n is odd: $pow(a, n/2) * pow(a, n/2) * a$

$2^4 \longrightarrow 2^2 * 2^2$

$2^5 \longrightarrow 2^2 * 2^2 * 2$

Pseudocode

```
int pow ( int a , int n ) {
    if ( n = = 0 ) { return 1 }

    if ( n & 1 = = 1 ) { // n is odd
        return pow(a, n/2) * pow (a, n/2) * a
    }
    else {
        return pow(a, n/2) * pow (a, n/2)
    }
}
```

$\because$ Since we are calling $pow(a, n/2)$ again & again
Lets re-use it.

```
int Fastpow(int a, int n){
    if(n==0){ return 1 }

    int half = Fastpow(a, n/2)

    if(n & 1 == 1){ // n is odd
        return half * half * a
    }
    else {
        return half * half
    }
}
```

fpow(2, 3) ⟶ fpow(2, 1) ⟶ fpow(2, 0)

fpow(2, 0) ⟵ ret 1

fpow(2, 1) ⟵ 1 * 1 * 2

fpow(2, 3) ⟵ 2 * 2 * 2

8

$N \longrightarrow N/2 \longrightarrow N/4 \; \ldots \; 0$

TC : $O(\log N)$

Break : 22 : 50

Time Complexity of Recursion — Recurrence Relation

Main Logic

factorial (N) = factorial (N-1) * N

Pseudocode

```
int factorial (int N) {
    // Base condition
    if (N == 1) return 1

    // Main logic
    return factorial (N-1) * N
}
```

TC of any recursive code = # of function calls * Time taken per function call.

$T(n) = \text{factorial}(N)$

Time taken per call.

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

$T(1) = 1$

Defines TC for factorial (N)

$\implies T(N-1) = T(N-2) + 1$

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

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

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

$$T(N) = T(N-3) + 1 + 2$$
$$= T(N-3) + 3$$
$$\vdots$$

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

substitute $k = N-1$

$$N - k = 1$$
$$k = N-1$$

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

Logical way

TC of any recursive code = # of function calls * Time taken per function call.

$$\underbrace{\overset{O(1)}{fact(N)} \longrightarrow \overset{O(1)}{f(N-1)} \longrightarrow f(N-2) \ldots\ldots\ldots\ldots\ldots \overset{O(1)}{f(1)}}$$

per function call the TC $= O(1)$

$\Rightarrow \quad O(N)$

```
int Fastpow(int a, int n) {
    if (n==0) { return 1 }

    int half = Fastpow(a, n/2)

    if (n & 1 == 1) { // n is odd
        return half * half * a
    }
    else {
        return half * half
    }
}
```

$T(N) \longrightarrow$ TC for fastPow(a, n)

$$T(N) = T(N/2) + 1 \quad \ldots\ldots \quad 1$$

$$T(N/2) = T(N/4) + 1$$

$$\Rightarrow \quad T(N) = T(N/4) + 1 + 1$$

$$T(N/4) = T(N/8) + 1$$

$$T(N) = T(N/8) + 1 + 1 + 1$$

$$T(N) = T(N/2^3) + 3$$

$\vdots$

k^{th} step

$$T(N) = T(N/2^k) + k$$

$\frac{N}{2^k}$ should become 0

$\because$ Integer division

$$2^k > N$$

$$k > \log(N)$$

$$T(N) = T(0) + \log(N) + 1$$

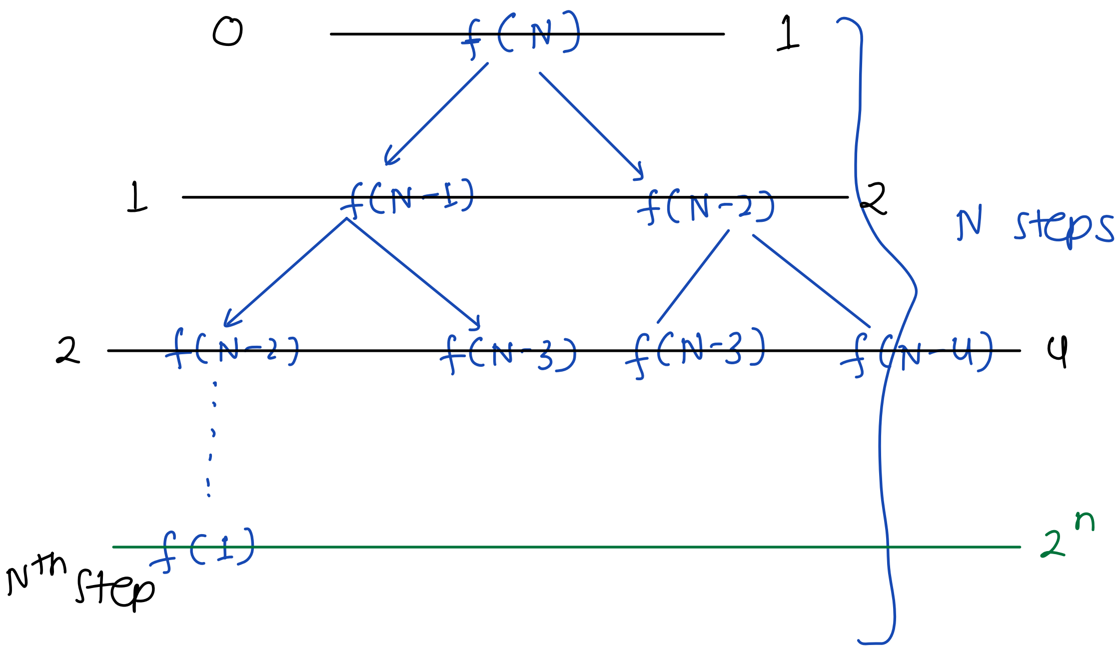
$$T(N) = \log(N) + 1$$

TC : $O(\log(N))$

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

    return fib(N-1) + fib(N-2)
}
```

$$\underbrace{fib(N-1)}_{1} \; \underset{\downarrow\, 3}{+} \; \underbrace{fib(N-2)}_{2}$$



TC : $1 + 2 + 4 + \ldots\ldots \; 2^n$

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

\# recursive calls in factorial (6)

$$\underbrace{f(6) \longrightarrow f(5) \longrightarrow f(4) \ldots\ldots \longrightarrow f(1)}_{\text{6 calls.}}$$

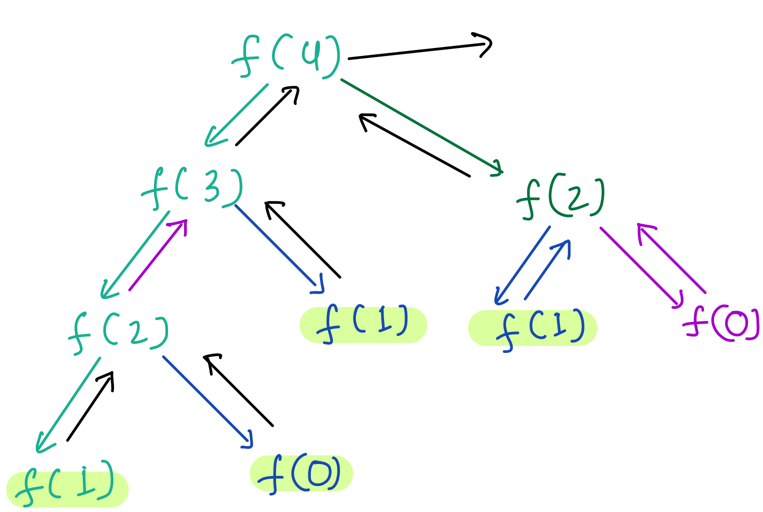
\# recursive calls in FastPow(2, 5)

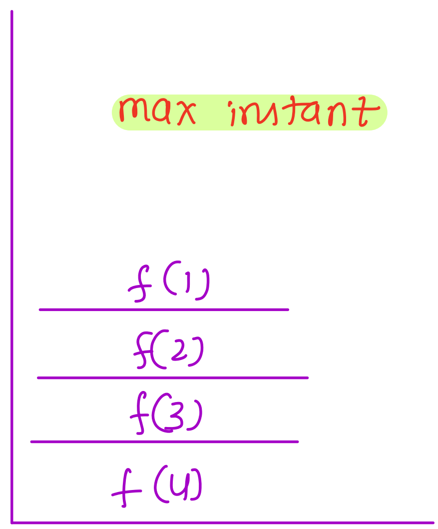
$$\underbrace{fp(2,5) \longrightarrow f(2,2) \longrightarrow f(2,1) \longrightarrow f(2,0)}_{\text{4 calls.}}$$

space complexity of recursive code

Max amount of space used at any instant of time.

HW: Draw the call stack for fib(4).





stack

SC : O(N)

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

    return fib(N-1) + fib(N-2)
}
```

$\underbrace{fib(N-1)}_{①} + \underbrace{fib(N-2)}_{②} \rightarrow ③$

Doubt Session

worst case scenario → 0 q/s solved.