Date 24/12/21
Page

Recursion

→ A function that call itself is a recursive function

* ~~fun~~

```
void fun(int n)
{
    if(n>0)
    {
Ascending  1. _____  → This statement will be executed at calling time
           2. fun(n-1) * 2
Descending 3. _____  → This statement will be executed at returning time

    }
```

* Time Complexity

```
void fun(int n) ——> T(n)
{
  if(n>0) ——— 1
  {
    printf("%d", n); — 1
    func1(n-1); — T(n-1)
  }
}
```

∴ $T(n) = T(n-1) + 2$ → as both are constant
$= T(n-1) + 1$

$T(n) = T(n-1) + 1$
$T(n-1) = T(n-2) + 2$
$T(n-2) = T(n-3) + K$

Static variable is stored in a section for static variable & Global variable in code section

Assume $n-k=0 \therefore n=k$

$T(n) = T(n-n) + n$

$T(n) = T(0) + n$

$T(n) = 1 + n \therefore = O(n)$

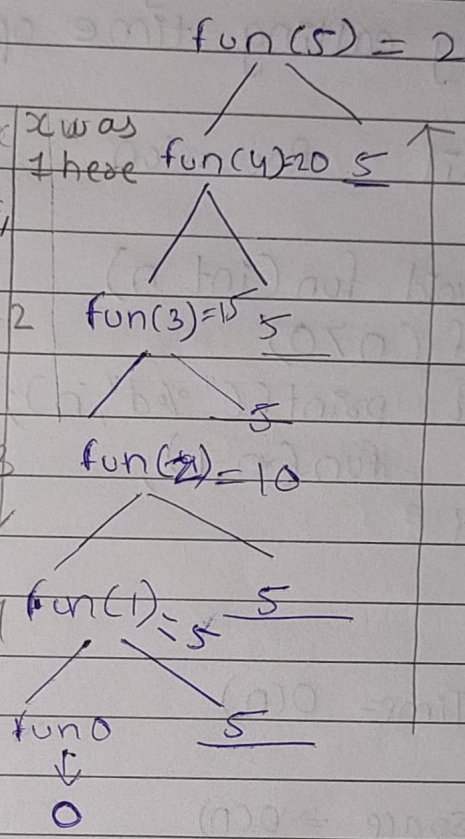
$\therefore O(n)$

↑ Time complexity of recursion function

* Static variable in Recursion.

```
int x=0; Global variable
int fun (int n)
{static int x=0
  if (n>0)
  { x++
    return fun(n-1)+x;
  }
  return 0;
}
```

```
int main()
{ fun(5)
```



Types of Recursion

1. Tail Recursion

→ last statement in function
→ no returning time operation are performed only calling time operation are performed

Tail Recursion	~~Head Recursion~~ loop

void fun(int n)
{ if (n>0)
{ printf("%d", n);
fun(n-1);
}
}
	void fun(int n)
	while(n>0)
	{ printf("%d", n);
	n--;
	}
	}

Time= O(n)	Time = O(n)
Space = O(n)	Space = O(1)

∴ Tail Recursion is not as efficient as loop
∴ It is advisable to use loop instead

2) Head Recursion

→ Opposite of Tail Recursion.
→ Every processing is done in time of returning.

Head Recursion

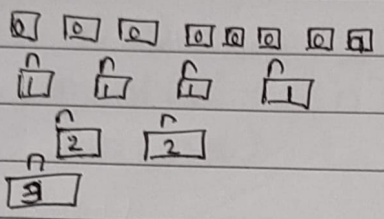
```
void fun (int n)
{ if (n >0)
  {
     fun (n-1);
     printf("%d ", n);
  }
}
```

Tree Recursion

→ linear Recursion → calls itself only one time

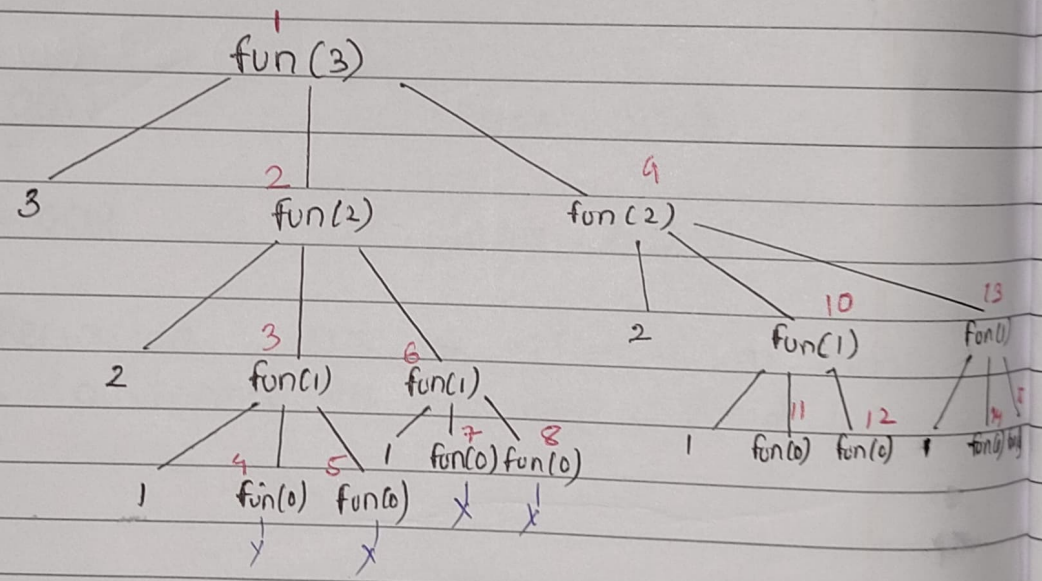
→ Tree Recursion → more than 1 call.

Stack



fun(3)

```
void fun(int n)
{
  if(n>0)
  {
    printf("%d", n);
    fun(n-1);
    fun(n-1);
  }
}
```



→ Time complexity → $O(2^n)$

Calculating Time complexity → $1 + 2 + 4 + 8 = 15$

Here n is 3

$2^0 + 2^1 + 2^2 + 2^3 = 2^{n+1} - 1$

Time required or Time function → $2^{n+1} - 1$

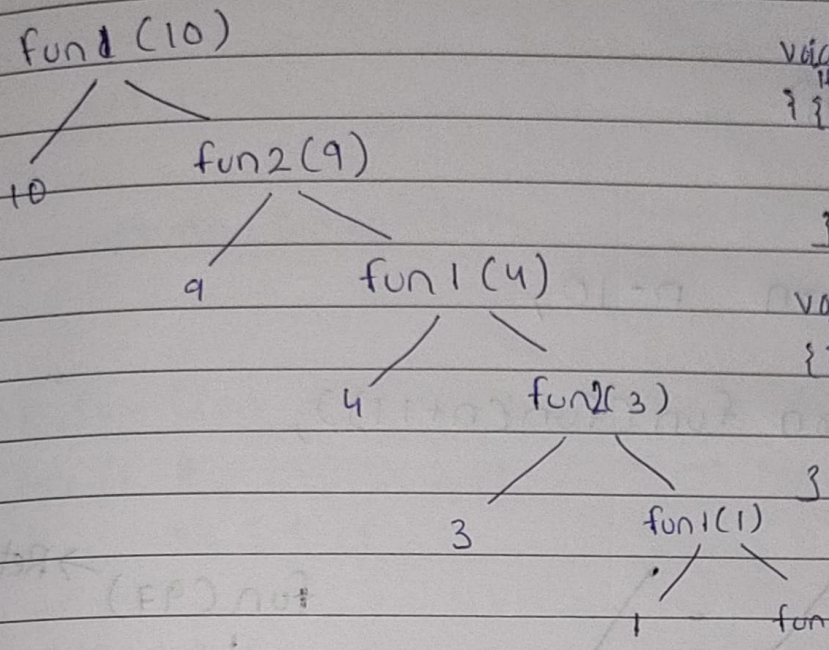
∴ Time complexity $= 2^n$

Now space required is n+1

~~Space → O(~~ ∴ Space complexity $= O(n)$

Indirect Recursion.

Indirect Recursion → more than 1 call but in circular manner involving more than 1 function

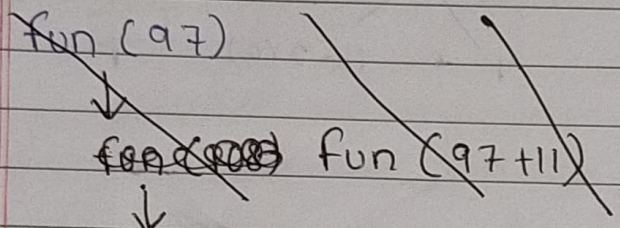



```
void fun1(int n)
{
  if (n>0)
  { printf ("%d", n);
    fun2 (n-1);
  }
}
void fun2(int n)
{
  if (n>1)
  { printf("%d", n);
    fun1 (n/2);
  }
}
```

* Recursion

Nested Recursion - Here parameter will be a recursive call

```
int fun (int n)
{
   if (n>100)
        return  n-10;
   else
        return fun (fun (n+11));
}
```



fun (97) → Returns 91

↓

fun (fun (97+11))

↕

fun (98)

↓

fun (fun (98+11))

↕

fun (99)

↓

fun (fun (99+11))

↕

fun (100)

↓

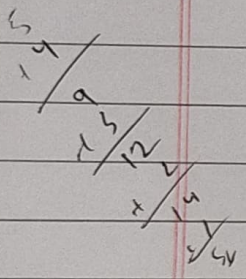
fun (fun (100+11))

↕

fun (101)

↓

91



→ Sum of first n Natural number

→ int Sum (int n)
if (n == 0)
return 0
else
return Sun (n-1) + n;

→ Here Time → O(n)
Space = O(n)

→ int sum (int n) { int i, s=0;
{ for (i=0; i<=n ; i++)
s = s + i;

return s;
}

→ Here Time = O(n)
space = O(1)

→ int sum (int n)
{
return n*(n+1)/2
}

→ Here Time = O(1)
space = O(1)

Factorial using recursion.

→
```
int fac(int n)
{ if (n==0)
    return 1;
  else
    return n x fac(n-1);
}
```

→
```
int fac(int n)
 {
    int f=1,i;
    for (i=1; i<=n; i++)
    {
      f= f *i;
    }
 }
```

* Exponent using recursion

$m^n = m \times m \times m \cdots \cdots n$ time

$m^n = (m \times m \times \cdots \cdots \times n-1 \text{ time}) \times m$

$pow(m, n) = (m \times m \times m \cdots n-1 \text{ time}) \times m$

$pow(m, n) = pow(m, n-1) \times m$

$$pow(m, n) = \begin{cases} 1 & n = 0 \\ pow(m, n-1) & n > 0 \end{cases}$$

Here Time → O(n)
Space → O(n)

* faster version:

```
int pow (int m, int n)
{
    if (n == 0)
        return 1;
    if (n % 2 == 0)
        return pow (m*m, n/2);
    else
        return m * pow (m*m, (n-1)/2);
```

pow (2, 9)
↓
2 * pow (2, 8)
↓
pow (2^4, 2)
↓
pow (2^8, 1)
↓
2^8 * pow (2^16, 0)
↓
1

2^8 × 2 = 2^9

$1 + \frac{4}{1} + \frac{4^2}{2} + \frac{4^3}{3\times2\times1} + \frac{4^4}{4\times3\times2}$

Taylor series → Here Time complexity = $O(n^2)$

$$e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + n \text{ terms}$$

```
e(x, 4)
  ↓
e(x, 3) → 1 + x + x^2/2! + x^3/3! + x^4/4!
  ↓
e(x, 2) → 1 + x + x^2/2! + x^3/3!
  ↓
e(x, 1) → 1 + x/1 + x^2/2!
  ↓
e(x, 0) + 1 + x/1
  ↓
1
```

→ firstly it enters e(x, 4)

```
e(x, 4)
  ↓
  :
  :
e(x, 0)
  ↓
  1
```

Time O(n) → linear.

Date
Page

Taylor Series using Horner's Rule.

→ $e^x = 1 + \frac{x}{1} + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + n \text{ terms}$

$$1 + x + \frac{x^2}{2 \times 1} + \frac{x3}{1 \times 2 \times 3} + \frac{x^4}{1 \times 2 \times 3 \times 4}$$

$$= 1 + x\left[1 + \frac{x}{2} + \frac{x^2}{1 \times 2 \times 3} + \frac{x^3}{1 \times 2 \times 3 \times 4}\right]$$

$$= 1 + x\left[1 + \frac{x}{2}\left[1 + \frac{x}{3} + \frac{x^2}{3 \times 4}\right]\right]$$

$$= 1 + x\left[1 + \frac{x}{2}\left[1 + \frac{x}{3}\left[1 + \frac{x}{4}\right]\right]\right]$$

→ using loops (Taylor series using loops

```
int e(int x, int n)
{
    int s = 1
    for (; n > 0; n--)
    {
        s = 1 + x/n * s;
    }
    return s;
```

2 fun(n-1) → $O(2^n)$ ⇒ Time

calling multiple function with same parameter is excessive recursion

Fibonacci series.

0 1 1 2 3 5 8 13

$$fib(n) = \begin{cases} f(0) = 0 \\ f(1) = 1 \\ f(n) = f(n-2) + f(n-1) \end{cases}$$

$$fib(n) = \begin{cases} 0 & n=0 \\ 1 & n=1 \\ f(n-2)+f(n-1) & n>1 \end{cases}$$

* Storing the result in an array to avoid excessive call is known as memoization

* Using loop - Time complexity → O(n)

initialize $t_0 = 0$

initialize $t_1 = 1$

Now
```
for (i=2; i<=n; i++)
{ S = t0 + t1;
  t0 = t1
  t1 = S
}
```

∴ by this updates & t_0, t_1 keeps results until it reaches n

* Using Recursion.

n = 7

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

f(6) + f(5)

f(5)+f(4) f(4)+f(3)

f(4)+f(3) f(3)+f(2) f(3)+f(2) f(2)+f(1)

f(3)+f(2) f(2)+f(1) f(2)+f(1) f(1)+f(0) f(1)+f(0)

f(2)+f(1) f(1)+f(0) f(1)+f(0)

f(1)+f(0)

Time complexity → $O(2^n)$

as return f(n-2)+f(n-1)

= 2f(n-1) (assume)

= 2^n → Time complexity

→ Using Recursion but with memoization.

f4(7) → 13

f4(6) + f4(5)

f4(5) + f4(4) → 3

f4(4) + f(3) → 2

f(3) + f(2) → 1

f(2) + f(1) → 1

f(0) f(1)

f | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

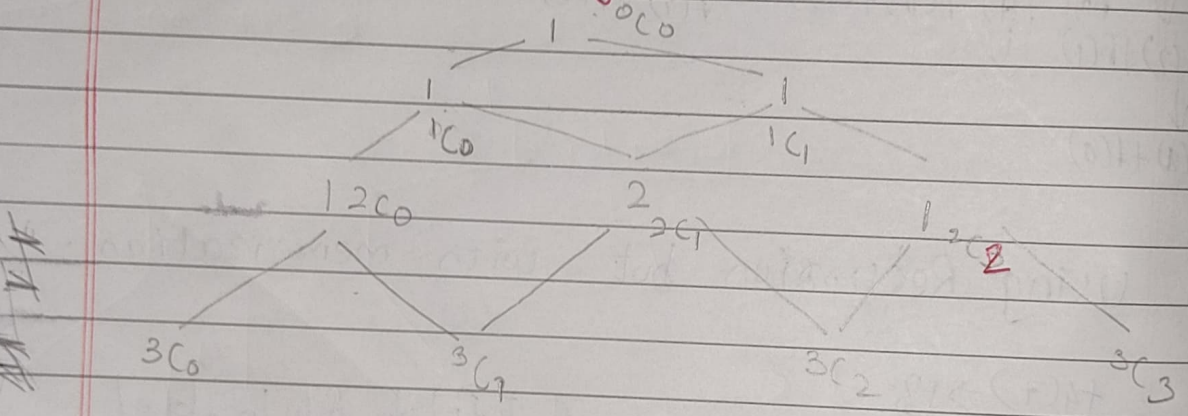
Combination formula

$$^{n}C_{r} = \frac{n!}{r!(n-r)!}$$

```
int c (int n, int r)
{   int t1, t2, t3;
n — t1 = fact(n)
n — t2 = fact(r)
n — t3 = fact(n-r)
    return t1/(t2*t3);
}
```

∴ Time function = 3n
O(n)

nCr using
→ Pascals triangle



★ Logic
→ If its starting / extreme point return 1
→ $^{n}C_{r} = {}^{n-1}C_{r-1} + {}^{n-1}C_{r}$

Towers of Hanoi (using Recursion)

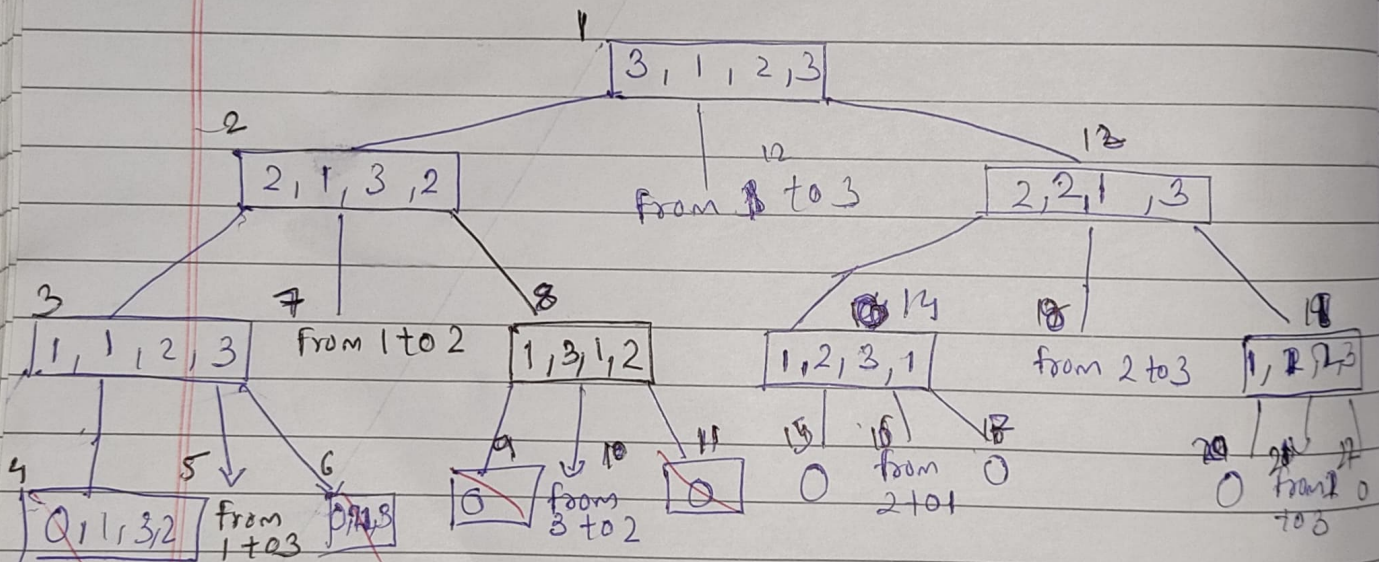
Logic → To move n plates to A to C
1) move n-1 plates to A to B
2) move n plate to C
3) move n-1 plate from A to C

Code :- void TOH (int n, int A, int B, int C)
{
TOH (int n-1, A, C, B)
cout << "Moving from A to C";
TOH (int n-1, B, A, C);
}

Simply repeats moving plate or just to show that n-1 are moved from A to B, C can be now moved easily and then n-1 can be moved from B to C

→ Simply says moving d plate in that function call

For n = 3



→ 1 to 3
→ 1 to 2
→ 3 to 2
→ 1 to 3
→ 2 to 1
→ 2 to 3
→ 1 to 3