Analysis of Algorithms (Background)

Given a number n, write the sum of first n natural numbers

Sol:

```
*int fun1(n) {
    return n(n+1)/2;
}
```

fun1() → C_1

```
*int fun2(n){
    int sum=0
    for (int i=1; i<=n; i++)
        sum=sum+i;
    return sum;
}
```

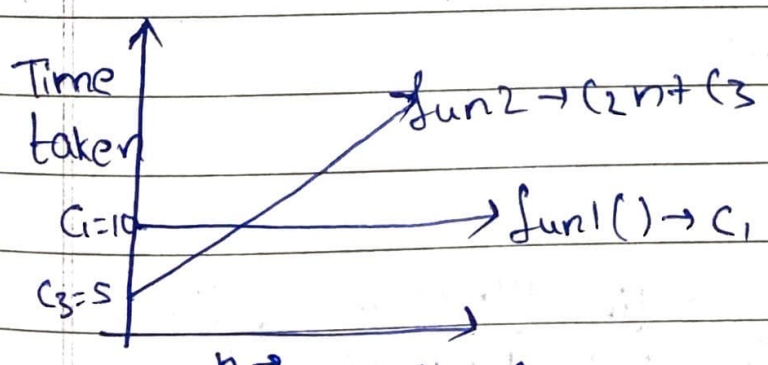
fun2() → $C_2n + C_3$

```
*int fun3(int n) {
    int sum=0;
    for (int i=1; i<=n; i++){
        for (int i=1; i<=n; i++){
            for(int j=1; j<=i; j++)
                sum++;
    return sum,
}
```

$n(\frac{n+1}{2}) + \ldots$

fun3() → $C_4n^2 + C_5n + C_6$

Comparison of fun1() and fun2()



$n \geq 0$

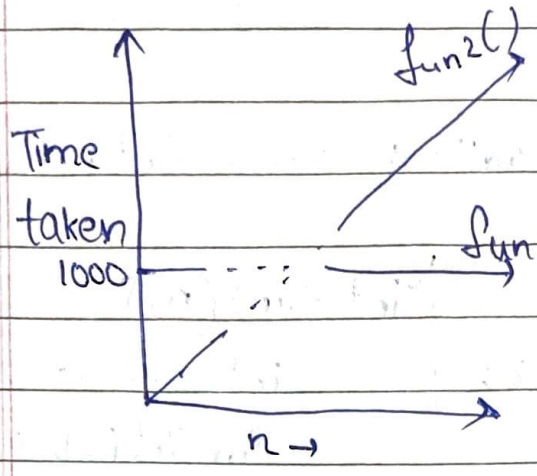
fun1(), fun2() ≥ 0

$2n+5 \geq 10$

$n \geq 2.5$

$n \geq 3$

Every value after $n \geq 3$ then it is going to take more time on fun2 no matter the machine and everything



fun1 = 1000

fun2 = n+1

$n+1 \geq 1000$

$n \geq 999$

After a value less than 1000 even tho the system is slower, but the constant time complexity will always take less time after a certain value.

ORDER OF GROWTH

There might be a situation where n never leaves its highest constant time.

A function $f(n)$ is said to be growing faster than $g(n)$ if

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = \infty$$

OR

$$\lim_{n \to \infty} \frac{g(n)}{f(n)} = 0$$

Assumptions

$n \geq 0$

$f(n).g(n) \geq 0$

Ex.

$$\lim_{n \to \infty} \frac{2n+5}{n^2+n+6} \Rightarrow \lim_{n \to \infty} \frac{n/n+5/n^2}{1+1/n+6/n^2} \rightarrow \lim_{n \to} \frac{0+0}{1+0+0} = 0$$

Direct Way

1. Ignoring lower order terms
2. Ignore leading Constant

How do we know which terms are lower order?

~~$c > \log n \log n > \log n > n^{1/3} > n^{1/2}$~~

$c < \log n \log n < \log n < n^{1/3} < n^{1/2} < n < n^2 < n^3 < n^4 < 2^n < n^n$

Example :-

1. $f(n) = 2n^2 + n + 6 \rightarrow$ $f(n)$'s order of growth is n^2
   $g(n) = 100n + 3 \rightarrow$ $g(n)$'s order of growth is n (Better)

2. $f(n) = c_1 \log n + c_2 \rightarrow \log n$ (Better)
   $g(n) = c_3 n + c_4 \log \log n + c_5 \rightarrow n$

3. $f(n) = c_1 n^2 + c_2 n + c_4 \rightarrow n^2$
   $g(n) = c_3 n \log n + c_6 n + c_7 \rightarrow n \log n$ (Better)

BEST, AVERAGE AND WORST CASES & ASYMPTOTIC NOTATIONS

(1)
```
int getSum(int arr[], int n){
    int sum=0
    for(int i=0; i<n; i++)
        sum=sum+ arr[i];
    return sum;
}
```

~~$cn + c_2$~~ n is order of growth

(2)
```
int getsum(int arr[], int n){
    if (n%2==0)
        return 0;
    for (int i=0; i<n; i++)
        sum=sum+ arr[i];
    return sum;
}
```

if n → odd
O(1) → c_1
if n → even
O(n) → n

For situations, like these where are different time complexities for different input values of n, we consider Best case, Average case, Worst case

In (2) program,

Best case: Constant. (n is odd)

Average Case: Linear (n and odd equally likely)

Worst Case: Linear (n is even)

In general, we never consider
→ Best case time complexity.
And, we will like to know
→ Average time complexity
But most of the time, we find
→ Worst Case time Complexity.

Big O: Represents exact bound (or) upper bound.

Theta: Represents exact bound

Omega: Represents exact (or) lower bound.

Big O Notation (upper bound on order of growth)

We say $f(n) = O(g(n))$ iff there exists constants c and n_0 such that $f(n) \leq cg(n)$ for all $n \geq n_0$

Example :-

$f(n) = 2n+3$ can be written as $O(n)$

$f(n) \leq cg(n)$ for all $n \geq n_0$

$(2n+3) \leq cn$ for all $n \geq n_0$

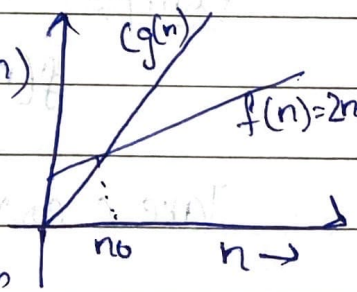
Take constant of highest gravity term

$c = 3$ (next value of $2n$)

$(2n+3) \leq 3n$

$3 \leq n$

$n_0 = 3$



Application

Suppose, this linear search algorithm

```
int linearSearch(in arr[], int n, int x)
{
  for(int i=0; i<n; i++)
    if(arr[i]==x)          → O(n)
      return i;
  return -1;
}
```

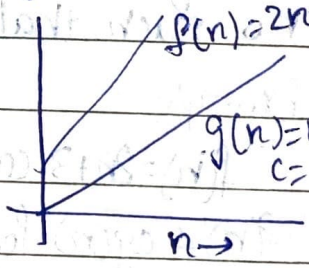
Omega Notation (Lower Bound)

$f(n) = \Omega(g(n))$ iff there exists positive constant c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0$.

Example :-

$f(n) = 2n+3$

$= \Omega(n)$



Take c as value next smaller than $2n \to 1$

$c = 1$

$cg(n) = n$

$n \leq 2n+3$

$n_0 = 0$

1) $\{\frac{n}{4}, \frac{n}{2}, 2n, 3n, 2n+3, n^2, 2n^2, \ldots n^n\} \in \Omega(n)$
in general use tight bounds

2) if $f(n) = \Omega(g(n))$ then $g(n) = O(f(n))$

3) Omega notation is useful when we have lower bound on order of growth

Ex:- if there is an endless running game, the upper bound time complexity is infinite, but the lower bound will be $\Omega(n)$ as there are n players that have started the game.

Theta Notation (Bounds from both upper and lower)
(Exact order of growth)

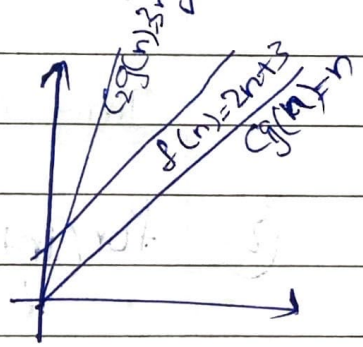
$f(n) = \theta(g(n))$ iff there exists positive constraints c_1, c_2 and n_0 such that $0 <= c_1 g(n) <= f(n) <= c_2 g(n)$ for all $n >= n_0$

Example :-

$f(n) = 2n+3$

$= \theta(n)$

$c_1 = 1$, $c_2 = 3$



$c_1 g(n) \leq f(n) \leq c_2 g(n)$

$n \leq 2n+3 \leq 3n$

$n \geq 0$ ~~~ $2n+3 \leq 3n$

$3 \leq n$

$n \geq 3$

max :-

$n_0 = 3$

① if $f(n) = \Theta(g(n))$
then $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$
and $g(n) = O(f(n))$ and $g(n) = \Omega(f(n))$

② Theta is useful to represent time complexity when we know exact bound. For ex, time complexity to find sum, max and min in an array is $\Theta(n)$.

③ $\{\frac{n^2}{4}, \frac{n^2}{2}, \ldots 2n^2, 2n^2+1000n, 4n^2+2n\log n + 30, \ldots\} \in \Theta$

Analysis of common loops

①
```
for(int i=0; i<n; i=i+c)
{
    Some Θ(1) work
}
```

Time Complexity → $\Theta(n/c)$ $n=10$, $c=2$: 0 2 4 6 8 $= \frac{10}{2} = 5$

②
```
for(int i=n; i>0; i=i-c)
{
    Some Θ(1) work
}
```

$\Theta(n)$

③ for (int i=1; i<n; i=i*c)
{
some θ(1) work
}

n=32, c=2
2, 4, 8, 16

θ(log n)

$1, c, c^2, c^3, \ldots c^{K-1}$

$c^{K-1} < n$

$K-1 < \log_c n$

$K < \log_c n + 1$

base doesn't matter

④ for(int i=n; i>1; i=i/c)
{
Some θ(1) work)
}

n=32, c=2
32, 16, 8, 4, 2

θ(log n)

n=33, c=2
33, 16, 8, 4, 2

⑤ for (int i=2; i<n; i=pow(i,c)){
Some θ(1) work
}

$2, 2^c, (2^c)^c, \ldots$

$2, 2^c, 2^{c^2}, \ldots 2^{c^{K-1}}$

$2^{c^{K-1}} < n$

$c^{K-1} < \log_2 n$

$K-1 < \log_c \log_2 n$

$K < \log_c \log_2 n + 1$

θ(log log n)

6) void fun(int n) {
for (int i=0; i<n; i++)
// some θ(1) work
} θ(n) work

for (int i=1; i<n; i=i*2)
// some θ(1) work
} θ(log n) work

for (int i=1; i<100; i++)
// some θ(1) working
} θ(1) work

$\theta(n) + \theta(\log n) + \theta(1)$
✓ ✗ ✓

7) void fun(int n) {
for (int i=0; i<n; i++) θ(n)
for (int j=1; j<n; j=j*2) θ(log n)
// some θ(1) work
}

$\theta(n)\,\theta(\log n)$

8) void fun (int n) {
for (int i=0; i<n; i++) θ(n)
for (int j=1; j<n; j=j*2) θ(log n)
// some θ(1) work
for (int i=0; i<n; i++) θ(n)
for (int j=1; j<n; j++) θ(n)
// some θ(1) work

$\theta(n \log n) + \theta(n^2) = \theta(n^2)$

Analysis of Recursion (Introduction)

Example 1

```
void fun(int n)
{
    if (n<=0)
        return;
    print("GFG");
    fun(n/2);
    fun(n/2);
}
```

Explanation

when n>0

$T(n) = T(n/2) + T(n/2) + \Theta(1)$

$T(n) = 2T(n/2) + \Theta(1)$

$T(0) = \Theta(1)$

Example 2

```
void fun(int n){
 if (n<=0)
     return;
 for (int i=0; i<n; i++)
     print("GFG");
 fun(n/2);
 fun(n/3);
}
```

Exp.

```
while(n>0)
```

$T(n) = T(n/2) + T(n/3) + \Theta(n)$

$T(0) = \Theta(1)$

Example 3

```
void fun(int n)
{
  if (n<=1)
    return;
  print("GFG");
  fun(n-1);
}
```

Expl. :-

$T(n) = T(n-1) + \Theta(1)$

$T(0) = \Theta(1)$

Recursion Tree Method for Solving Recurrence

→ We consider the recursion tree and compute total work done

→ We write non-recursive part as root of the tree and write the repursive part as children

→ We keep expanding until we see a patter

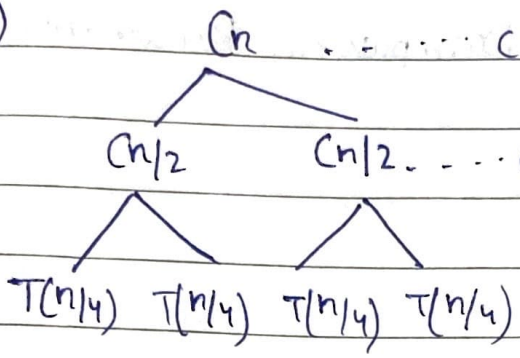
$T(n) = 2T(n/2) + cn$

$T(1) = c^2$

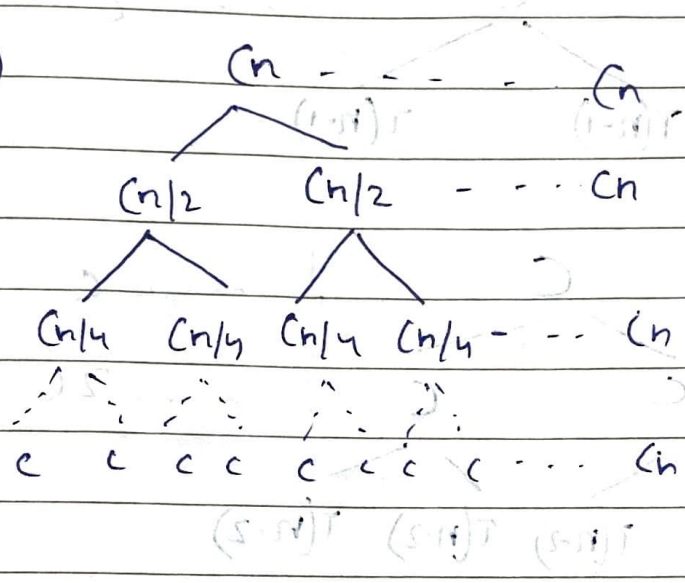
(1) cn

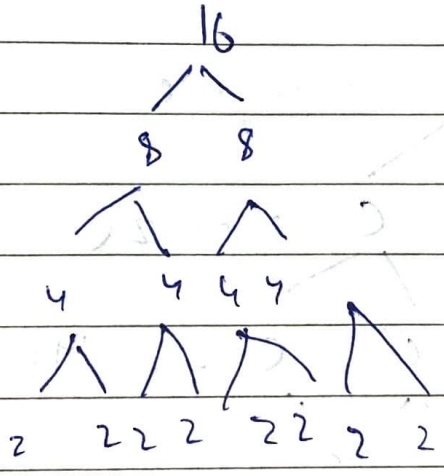
$T(n/2)$ $T(n/2)$

2)



3)





$\lceil \log_2 n \rceil + 1$
$= \log n$ (height of tree)

$= \Theta$ Cn (time for each level)

~~Tototal~~ Total $= \Theta(n \log n)$

↓

Even if $C_1 C_2$ contents change it is

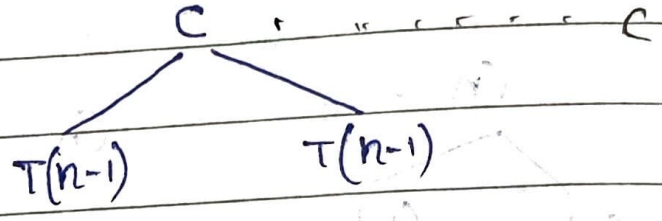
$\Theta(n \log n)$

More Examples on Recurrences

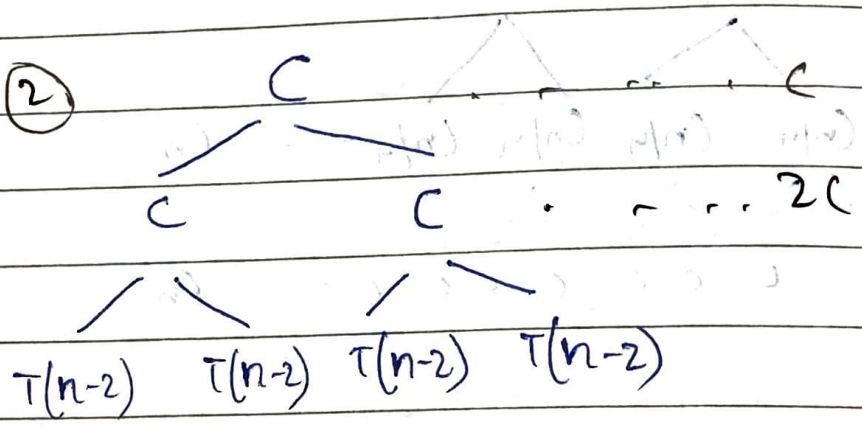
$T(n) = 2T(n-1) + c$

$T(1) = c$

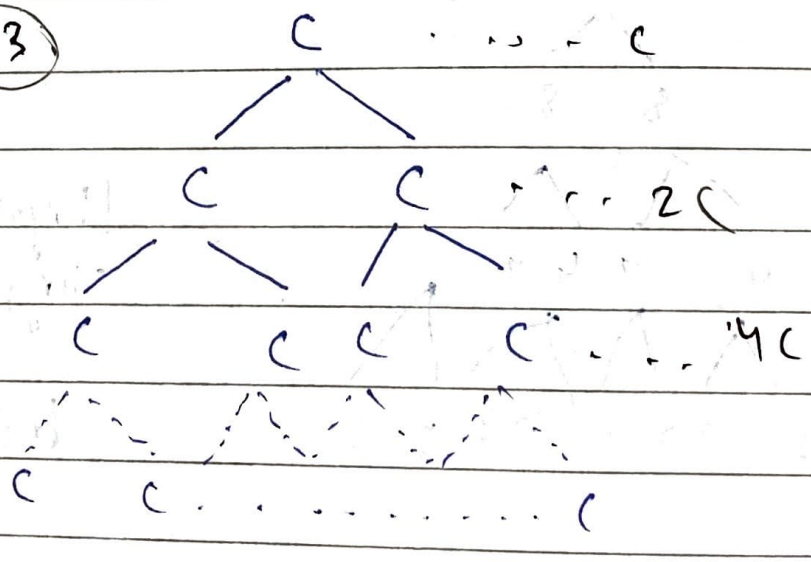
(1)



(2)



(3)



$c + 2c + 4c + \cdots$

$c(1 + 2 + 4 + \cdots)$

$c\left[\frac{2^n - 1}{2 - 1}\right]$

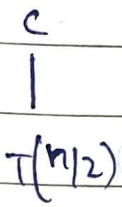
$\Theta(2^n)$

Tower of Hanoi Problem

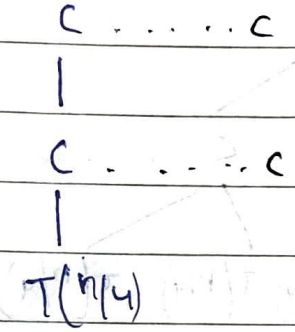
$T(n) = T(n/2) + c$

$T(1) = c$

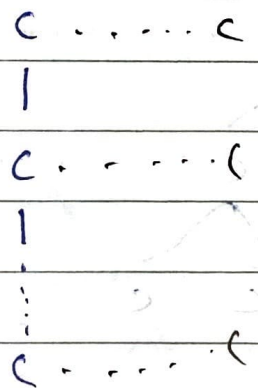
(1)



(2)



(3)



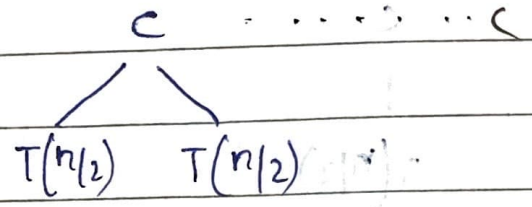
$\underbrace{c + c + c + \cdots + c}_{\lfloor \log_2 n \rfloor + 1}$ Binary Search

$\lfloor \log_2 n \rfloor + 1 \quad \Theta(\log_2 n)$

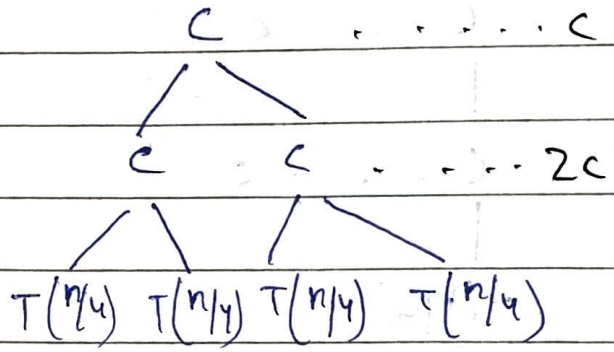
$T(n) = 2T(n/2) + c$

$T(1) = c$

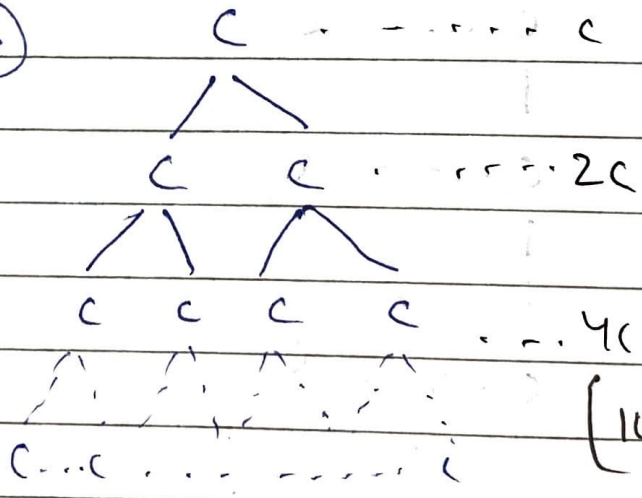
(1)



(2)



(3)



$c + 2c + 4c + \ldots \quad \theta(\log_2 n)$

$\theta\left(\frac{2^{\log_2 n} - 1}{2-1}\right)$

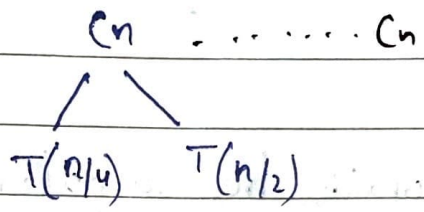
$\theta(n)$

Upper Bounds using Recursion Tree Method

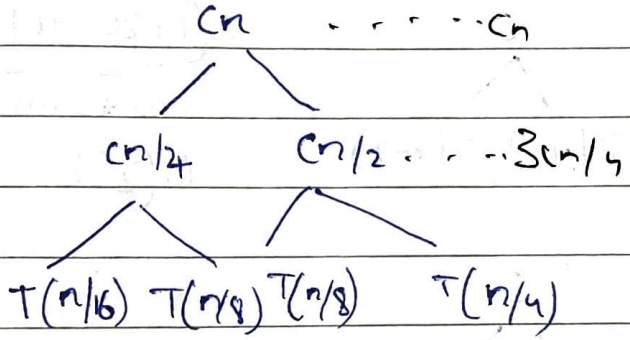
$T(n) = T(n/4) + T(n/2) + Cn$

$T(1) = c$

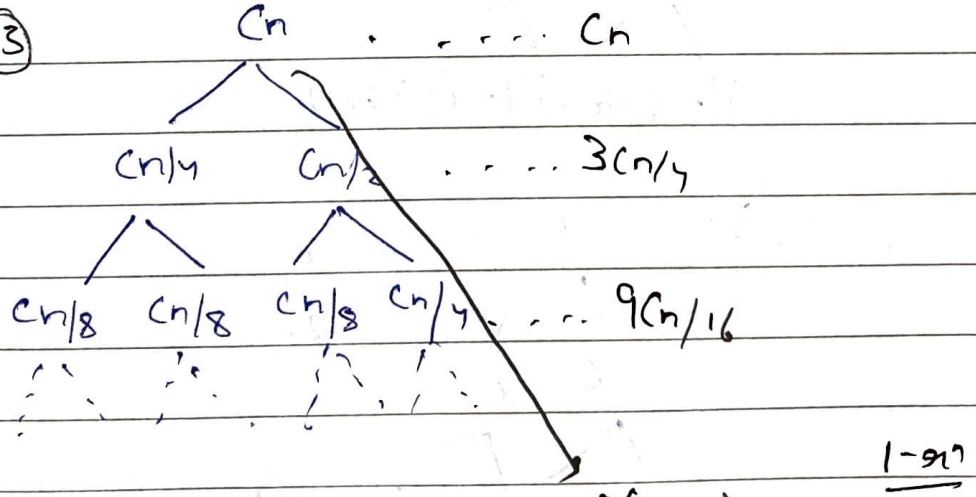
(1)



(2)



(3)



$Cn + 3Cn/4 + 9Cn/16 + \ldots\ldots$ $\theta(\log n)$

$\frac{1-r^n}{1-r}$

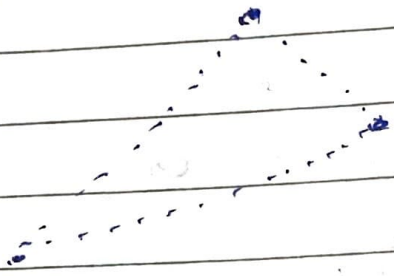
$O\left(\frac{Cn}{1-3/4}\right)$ $\because \frac{a}{1-r}$

$O(n)$

$T(n) = T(n-1) + T(n-2) + C$

$T(1) = C$

$T(0) = C$



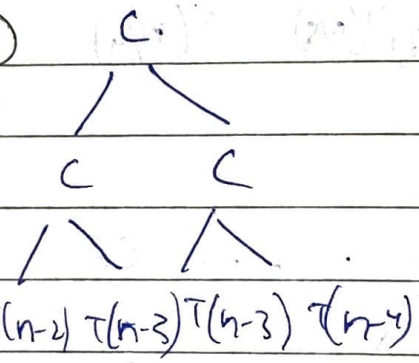
In these, consider tree as perfect and solve the problem

(1)

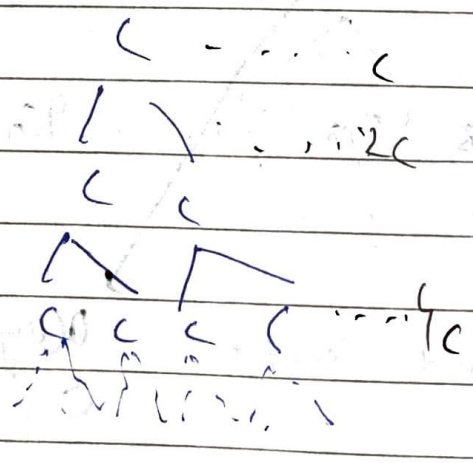
$O(\underbrace{C + 2C + 4C + \ldots}_{n \text{ terms}})$

$O(2^n)$

(2)



(3)



Space Complexity

Order of growth of memory (or RAM) space in terms of input size

```
int getSum1(int n)
{
  return n*(n+1)/2;
}
```

```
int getSum2(int n)
{
  int sum=0
  for(int i=1;i<=n;i++)
      sum=sum+i
  return sum;
}
```

$\theta(1)$ or $O(1)$

```
int arrSum(int arr[], int n)
{
  int sum=0;
  for (int i=0; i<n ; i++){        θ(n)
      sum = sum + arr[i];
  return sum;
}
```

Auxiliary Space:

Order of growth of extra space or temporary space interms of input size.

*
```
int arrSum(int arr[], int n)
{
    int sum=0;
    for(int i=0; i<n; i++){
        sum=sum+arr[i];
    }
    return sum;
}
```

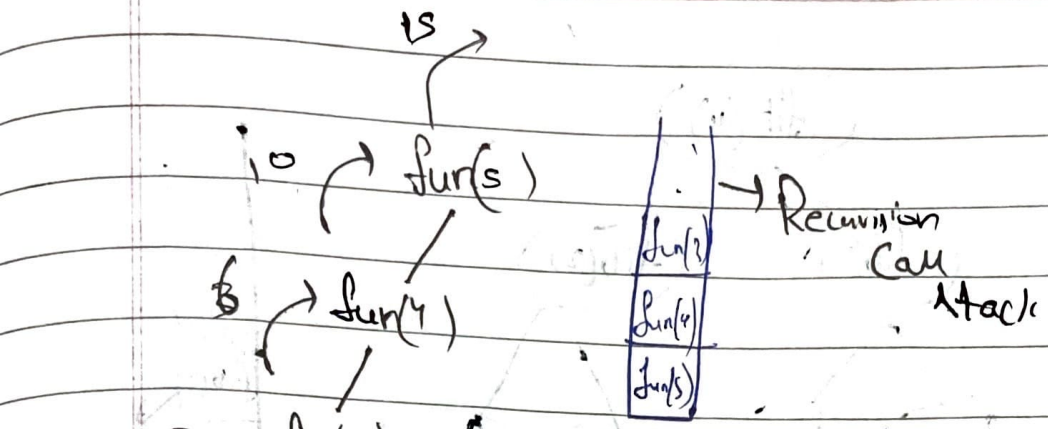
As this program doesn't require another array to solve the auxiliary space remains constant

Auxiliary Space : $\Theta(1)$
Space Complexity : $\Theta(n)$

Note:-

* We use Auxiliary Space more than space Complexity in Algorithm Analysis
* We mostly use $\Theta(n)$ space complexity in all the algorithms that needs an array.
* This caused used to use Auxiliary space.

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



* For this program to execute,

n + (fun(n-1))

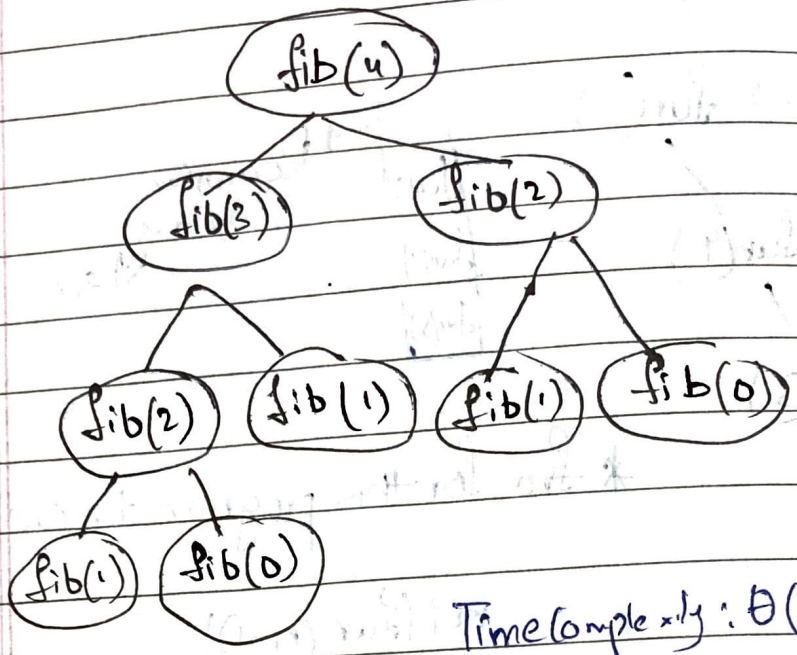
First it needs the value of fun(n-1), which is fun(n-2) ... And the process repeats until the value 0. Meanwhile All the fun's that are waiting are held in a stack. They were again using callback

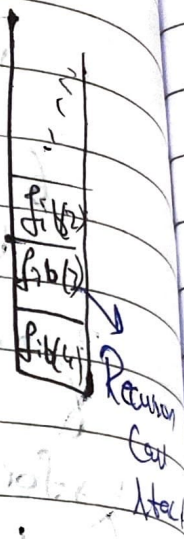
Auxiliary Space
$\Theta(n)$

Fibanocci No.'s :-

0, 1, 1, 2, 3, 5, ...
n=0 n=1 n=2 n=3

```
int fib(int n)
{
if(n==0 || n==1)
    return n;
return fib(n-1) + fib(n-2);
}
```

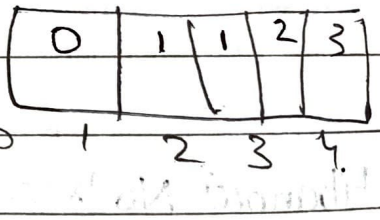




Time Complexity: $\Theta(n)$

Auxiliary Space: $\Theta(n)$

```
int fib(int n)
{
  int f[n+1];
  f[0] = 0;
  f[1] = 1;
for (int i = 2; i <= n; i++)
    f[i] = f[i-1] + f[i-2];
  return f[n];
}
```



Time Complexity: $\Theta(n)$

Auxiliary Space: $\Theta(n)$

```
int fib(int n)
{
    if (n==0 || n==1)
        return n;
    int a=0, b=1;
    for (int i=2; i<=n; i++){
        c=a+b;
        a=b;
        b=c;
    }
    return c;
}
```

Time: $\Theta(n)$

Auxiliary space: $\Theta(1)$

Space complexity: $\Theta(1)$