Analysis of Algorithm

Given a number n, write a fn to find sum of first n natural nos.

I/p: n=3 → 1+2+3 = 6
O/p: 6

I/p: n=5 → 1+2+3+4+5 = 15
O/p: 15

3 solns written by 3 diff. prgmers.

① fun1() ⇒ 9

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

doing constant work (this work always happen exactly once) for n = highest n = 14000

*, +, / (not depend upon n)

the no of opns that we are doing is constant for any value of n.

⇒ 3*(3+1)/2 = 3*4/2 = 6

② fun2() ⇒ C3n + C4

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

depends upon n

n=3

1+2+3 = 6

initialisation

constant work [always happen exactly once for any value of n]

③ fun3() = C1n² + C5n + C6

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

n=3

⇒ (1) + (1+1) + (1+1+1) = 6

1 + 2 + 3 ..... n = n*(n+1)/2 = (n²+n)/2

sum++ depends upon n

By above 3 prgms we can decide which prgm is best by using asymptotic analysis

⇓

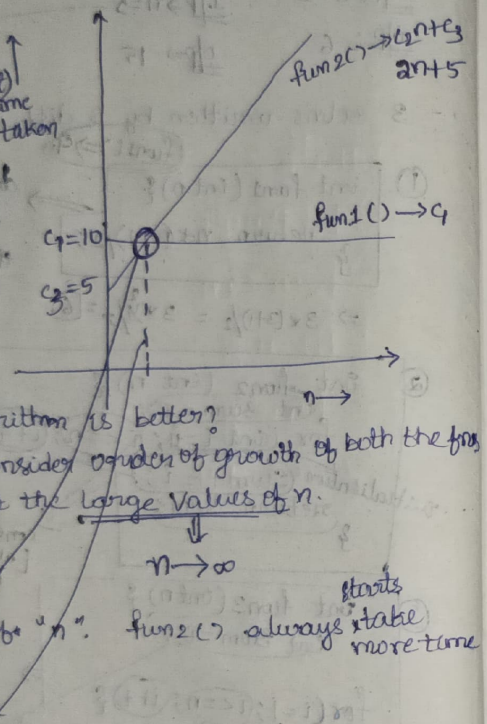
measuring order of growth of a prgm or algo in terms of I/P size.

Comparison of fun1() and fun2()

* When we do analysis of algorithm we always talk about size of I/P is must be ≥ 0.
* I/P size never be -ve.
* Time take by the functions should be ≥ 0.
* Time " "

always take Q_1 (1st quadrant)

* $n \geq 0$
* fun1(), fun2() ≥ 0



To consider which algorithm is better?
→ To decide that we consider order of growth of both the fns and always talk about the large values of n.

$n \to \infty$

after certain value of "n", fun2() always starts take more time

To find value ⇒

$2n+5 \geq 10$

$n \geq 2.5$

$n \geq 3$

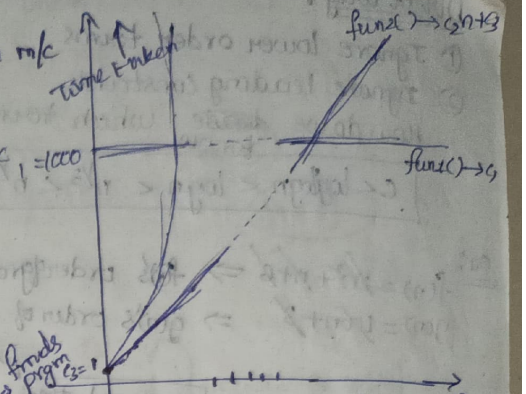
If $n \leq 3$ ⇒ fun2() takes less time for $n=1,2,3$

If $n \geq 3$ ⇒ fun2() starts takes more time. [after any value of 3]

fun2() ⇒ executes on faster m/c

$c_2=1$ & $c_3=1$

fun1() ⇒ mobile ⇒ $c_1=1000$ $c_1=1000$



$n \to \infty$

$n+1 \geq 1000$

$n \geq 999$

py(prgm) finds prgm $c_3=1$

fun1() takes less time

If $n \geq 999$ ⇒ fun2() takes more time.

$n \geq 999$

fun3() grow much faster than fun1() & fun2()

assumptions

$n \geq 0$

Time taken ≥ 0

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

Order of Growth:

we always talk about (lt $n \to \infty$) ⇒ large I/P size.

⇒ 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)

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

∴ $f(n)$ is a bad algorithm
$g(n)$ is a good algorithm (or) better.

⇒ $f(n) = n^2+n+6$

$g(n) = 2n+5$

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

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

Direct way:

① Ignore lower order terms
② Ignore leading constants
How do we know which terms are lower order?

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

eg:
$f(n) = n^2 + n + 6 \Rightarrow$ f(n)'s order of growth is n^2
$g(n) = 100n + 3 \Rightarrow$ g(n)'s order of growth is n

g(n) is better

$\Rightarrow f(n) = c_1 \log n + c_2 \Rightarrow \log n$
$g(n) = c_3 n + c_4 \log\log n + c_5 \Rightarrow n$
} f(n) is better.

$\Rightarrow f(n) = c_1 n^2 + c_2 n + c_4 \Rightarrow n^2$
$g(n) = c_5 n \log n + c_6 n + c_7 \Rightarrow n \log n$
} g(n) is better.

or

$f(n) = \frac{n^2}{n} = n$
$g(n) = \frac{n \log n}{n} = \log n$
} g(n) is better.

Best, Average, and worst cases & Asymptotic notations

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

$c_2 n + c_3$
$+$
$c_4 n$
$\Rightarrow c_5 n + c_6$

Ignoring constants
Ignoring leading constants
n

$\Theta \rightarrow$ exact bound.
$\Leftrightarrow n$ is order of growth

```
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;
}
```

n=3
n=4

$\Rightarrow$ If n is odd $\Rightarrow$ we return 0.
otherwise we find the sum of elements & return sum.

Now we can't decide about order of growth.
If n is odd $\Rightarrow$ constant time
If n is even $\Rightarrow$ linear time.

There are multiple cases $\Rightarrow$ so we categorise into 3 cases.
① Best ② Average & ③ worst.

Best case: constant (n is odd)

Average case: linear (under the assumption that even and odd are equally likely)
even + odd = linear
linear + const = linear

Worst case: linear (n is even)

In general, when we analyse algorithms, we don't consider Best cases $\Rightarrow$ it is considered as bogus.
For creating S/W $\Rightarrow$ when we know about worst case but we don't know about ~~average~~ worst case.
it's not a best S/W.

Avg case $\Rightarrow$ we would like to know [but it's impractical to do]

So we always find worst case.

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]

eg: $f(n) = 3n^2+5n+6$

~~$3n^2+5n+6$~~

$f(n) \Rightarrow O(n^2)$

eg: $f(n) = \sqrt{n} + \log n + 30$

$f(n) = O(n)$

we say $f(n) = O(g(n))$ iff there exist constants c and n_0 such that $f(n) \le cg(n)$ for all $n \ge n_0$

($f(n) \le cg(n)$: equal & higher order of growth)

ex:
$f(n) = 2n+3$ can be written as $O(n)$ ⇓ $O(g(n))$

$f(n) \le cg(n) \quad \forall\ n \ge n_0$

$(2n+3) \le cg(n)$

$(2n+3) \le cn \quad \forall\ n \ge n_0$

Take leading constant of highest growing term. & take next integer ⇒ $c = 2+1 = 3$

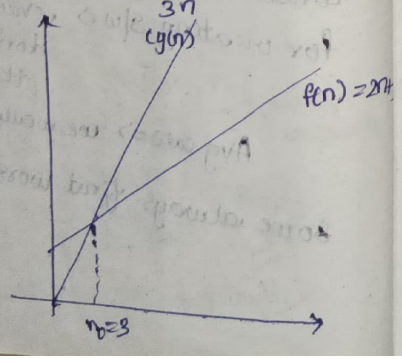
$(2n+3) \le 3n$

$3 \le n \Rightarrow n_0 = 3$

After $n_0 = 3$ $3n$ must be $> 2n+3$ for values of n

$n = 4$ $2(4)+3 \le 3(4)$ ✓

$O(n^2+n) \Rightarrow O(n^2)$



$\{\frac{n}{4}, 2n+3, \frac{n}{100}+\log n, n+1000+n/1000, 100, \log n+100\} \in O(n)$

If the exprns or fns which have order of growth $\le n$. they also come under Big O notation

equal & more

It represents the fns which have order of growth equal or less

$\{n^2+n, 2n^2, n^2+1000n+n^2+2\log n, \frac{n^2}{1000}, \dots\} \in O(n^2)$

$\{1000, 2, 1, 3, 10000, 10000.00, \dots\} \in O(1)$

Application:

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

Best case: $O(1)$ constant
Avg: linear ⇒ ith element is not present at all.
Worst case: linear

$O(n)$ — It covers all 3 cases & everything.

Omega Notation [Lower Bound]

$f(n) = \Omega(g(n))$ iff there exist positive constants c & n_0 such that $0 \le cg(n) \le f(n) \quad \forall\ n \ge n_0$.

ex: $f(n) = 2n+3$

$\Omega(n)$

⇓

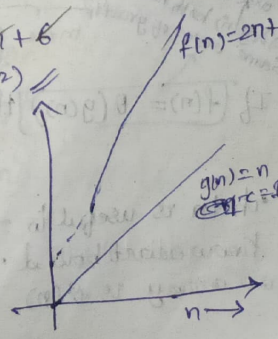
Take leading constant of highest growing term & take before integer $c = 2-1 = 1$

$cg(n) = n$

$n \le 2n+3 \Rightarrow n_0 = 0$

$2n^2+3n+6$

$\Omega(n^2)$



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

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 bound on time complexity

eg: Nplayer game ?

$\Omega(n)$ & we dont know about upper bound. bcz it differs based on players playing game.

Theta Notation (Θ) [exact bound of order of growth of f

$f(n) = \Theta(g(n))$ iff there exist +ve constants c_1, c_2 and n_0 such that $0 <= c_1 g(n) <= f(n) <= c_2 g(n)$ $\forall$ $n \geq n_0$

eg: $f(n) = 2n+3$ order of growth $\Rightarrow \Theta(n)$

$c_1 g(n) \leq f(n) \leq c_2 g(n)$ $\forall$ $n \geq n_0$

$c_1 = 1$ $c_2 = 3$

$1 * n \leq f(n) \leq 3 * n$

$2n+3$

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

can bound $f(n)$ from above & below with other

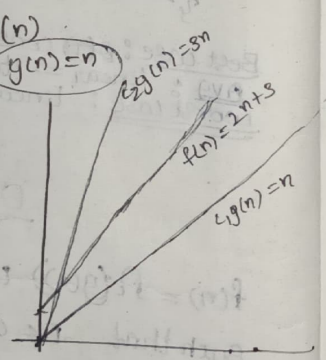
$f(n)$ & $g(n)$ both have same order of growth

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

$3 \leq n$

more from two is $n \geq 3$

$n \geq n_0$ $n_0 \leq 3$



① If $f(n) = \Theta(g(n))$ then $f(n) = \Theta(g(n))$ & $f(n) = \Omega(g(n))$ & $g(n) = O(f(n))$ & $g(n) = \Omega(f(n))$

② theta is useful to represent time complexity when we know exact bound. for eg: T.c to find sum, max & min in an array is $\Theta(n)$.

for eg: Quick sort: exact bound

Best case: $\Theta(n \log n)$
Avg case: $\Theta(n \log n)$
worst case: $\Theta(n^2)$
} T.C = $O(n^2)$

③ $\{\frac{n^2}{4}, \frac{n^2}{2}, \ldots 2n^2, 2n^2+100n, 4n^2+2n\log n+30, \ldots\} \in \Theta(n^2)$

Analyses of common loops

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

n=10
c=2
0, 2, 4, 6, 8

$\Rightarrow \frac{10}{2} = 5$ ⇒ loop runs floor value

$\lfloor n/c \rfloor$ $\Theta(n/c) = \Theta(n)$

n=11
c=2
0, 2, 4, 6, 8, 10 = 6

$\frac{11}{2}$

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

n=10
c=2
10, 8, 6, 4, 2 ⇒ 5 times

n=11
i = 11, 9, 7, 5, 3, 1

$\lceil n/c \rceil$ ⇒ ceiling of n/c ⇒ $\Theta(n)$

③
```
for (int i=1; i<n; i=i*c)
{
  // some Θ(1) work
}
```
⟹ $\Theta(\log n)$

c is a constant not a user i/p so we need to find order of growth in terms of i/p size n

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

* n=33, c=2
2, 4, 8, 16, 32

c^0
* 1, c, c^2, c^3 ... c^{k-1} ← no of iterations

$c^{k-1} < n$

$k-1 < \log_c n$

$k < 1 + \log_c n$

⇒ to no of times loop runs is $k \Rightarrow c^0$ to c^{k-1}

$\Theta(\log n)$

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

* n=32 c=2

i = 32, 16, 8, 4, 2 — $c^{k-1}, c^{k-2}, c^{k-3}$ ---- 1 ⇒ $\theta(\log_c n)$

* n=33 c=2

i = 33, 16, 8, 4, 2

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

* n=32 c=2

$i = 2, a^2, (a^2)^2 \Rightarrow a, a^c, (a^c)^c$

2, 4, 16 — $a, a^c, a^{c^2}, \ldots a^{c^{k-1}}$

$a^{c^{k-1}} < n$

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

$k-1 < \log_c \log_2 n$

$k < \log_c \log_2 n + 1 \Rightarrow \theta(\log\log n)$

⑥ Void fun (int n).
```
{
  for (int i=0; i<n; i++)
    // some θ(1) work                    ⇒ θ(n)
  for (int i=1; i<n; i=i*2)
    // some θ(1) work                    ⇒ θ(log n)
  for (int i=1; i<100; i++)
    // some θ(1) work  θ(1)  ⇒ constant loop it never grows with i/p
}                                        on Asymptotic analysis we always talk about n→∞
```

Subsequent 3 adding $\theta(n) + \theta(\log n) + \theta(1) \Rightarrow \theta(n)$

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
}
```
⇒ $\theta(n) * \theta(\log n) = \theta(n \log n)$

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
}
```
$\theta(n \log n) + \theta(n^2) \Rightarrow \theta(n^2) + \theta(n \log n)$ ⇓ $\theta(n^2)$ (Ignore lower order terms)

Analysis of Recursion (Introduction)

ex:
```
Void fun (int n)
{
  if (n<=0)
    return;
  Printf("GiFGi");
  fun(n/2);
  fun(n/2);
}
```

* n>0

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

$T(n) = 2T(n/2) + \theta(1)$ ⇒ Recurrence relation

* n<=0

$T(n) = \theta(1)$

when n is +ve $T(0) = \theta(1)$ ⇒ Recurrence relation Base case

*
```
Void fun (int n)
{
  if (n<=0) ⇒ θ(1)
    return;
  for (int i=0; i<n; i++) ⇒ θ(n)
    Printf("GiFGi");
  fun(n/2); ⇒ T(n/2)
  fun(n/3); ⇒ T(n/3)
}
```

*
$n > 0$

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

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

```
Ex:  Void fun (int n)
     {  if (n<=1)
          return;
        printf ("GFG");
        fun (n-1);
     }
```

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

$T(1) = \theta(1)$

Recursion Tree Method for solving Recurrences

→ we consider the recursion tree and compute total work done

→ we write non-recursive part as root of the tree and write the recursive part as children.

→ we keep expanding until we see a pattern

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

$T(1) = c$

① ⇒ cn → $T(n/2)$ $T(n/2)$ ⟹ $T(n/2) = 2T(n/4) + c(n/2)$

② ⇒ cn → $c(n/2)$ $c(n/2)$ → $T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$

③ ⇒ cn → cn/2 cn/2 → cn/4 cn/4 cn/4 cn/4 → c ... c

$cn + cn + cn - - - - cn$

⇓

$\theta(n \log n)$

Height of a Binary tree:

1 ⇒ 1

2 ⇒ 3

3 ⇒ 4

4 ⇒ 5

$\Rightarrow \lfloor \log_2 n \rfloor + 1$ (floor)

$\log_2^4 + 1 = 4$

$\log_2^{16} + 1 = 5$

$\lfloor \log_2 n \rfloor + 1 \Rightarrow \theta(\log n)$

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

$T(1) = c_2$

① ⇒ cn → $T(n/2)$ $T(n/2)$

② ⇒ cn → $cn/2$ $cn/2$ → $T(n/4)$ $T(n/4)$ $T(n/4)$ $T(n/4)$; $cn - - - - - c_1 n$; $cn/2 - - - - - c_1 n$

③ ⇒ $c_1 n$; $c_1 n$; $c_2 n$; $c_2 n$; $c_2 - - c_2$

$\lfloor \log n \rfloor$ (floor) terms in that fn except $c_2 n$.

$c_1 n * \theta(\log n) + c_2 n$

$\Rightarrow \theta(n \log n)$

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

$T(1) = c$

① ⇒ c → $T(n-1)$ $T(n-1)$

② ⇒ c → c c → $T(n-2)$ $T(n-2)$ $T(n-2)$ $T(n-2)$

③ ⇒ c - - - - c ; c - - - - 2c ; c c c c - - - 4c ; c - - c

$\Rightarrow c + 2c + 4c + - - -$

$c[1 + 2 + 4 + - - -]$

$c \frac{1 * (2^n - 1)}{2 - 1}$

$\Rightarrow \theta(2^n)$

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

$T(1) = c$

① ⇒ c ; ② ⇒ c → $T(n/2)$; ③ ⇒ c → c → $T(n/4)$; c → c → $T(n/8)$; c - - c

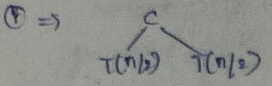
$c + c + c + - - - c$ ($\lfloor \log_2 n \rfloor + 1$ terms)

$\Rightarrow \theta(\log n)$

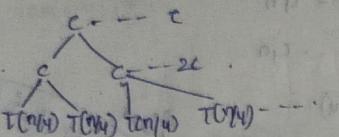
$\Rightarrow T(n) = 2T(n/2) + c$

$T(1) = c$

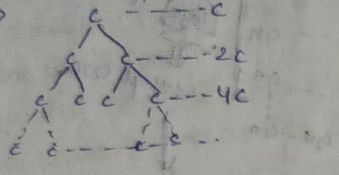
① ⇒



② ⇒



③ ⇒



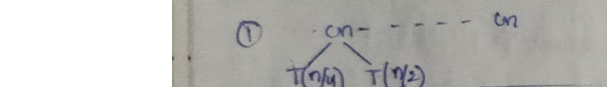
$c + 2c + 4c + \dots$

Upper Bounds using Recursion Tree method:

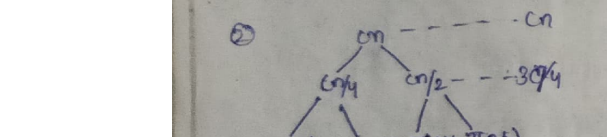
$T(n) = T(n/4) + T(n/2) + cn$

$T(1) = c$

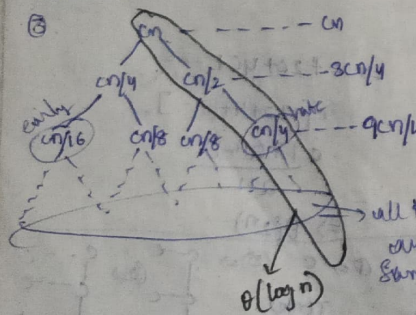
①



②



③



we assume tree to be full

$cn + 3cn/4 + 9cn/16 \dots$

sum of infinite terms

$\Downarrow \quad r = 3/4$

$\frac{a}{1-r}$

$O\left(\frac{cn}{1-3/4}\right)$

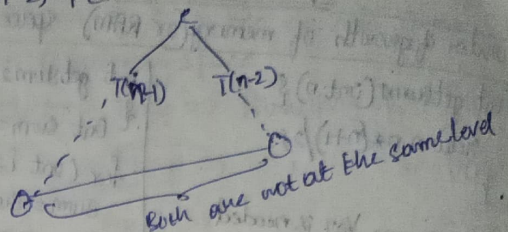
$O(n)$

as it is not having exact bounds

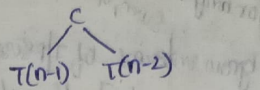
$T(n) = T(n-1) + T(n-2) + c$

$T(1) = c$

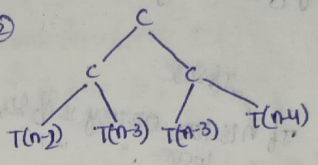
$T(0) = c$



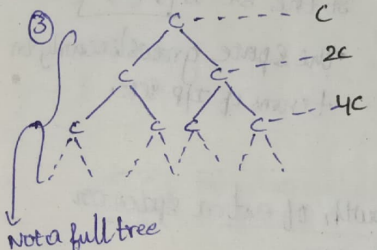
⇒ ①



②



③



$O(c + 2c + 4c - - -)$

n terms

$\Downarrow$

$O(2^n)$

① Draw a recursive tree for given recurrence relation

② calculate cost at each level & count total no of levels in recursion tree

③ count the total no of nodes in last level & calculate the cost of last level.

④ sum up the cost of all the levels in the recursive tree

Space Complexity:

Order of growth of memory (or RAM) space in terms of I/p size.

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

only one var is needed
So S.C ⇒ $\theta(1)$ / (on) / $O(1)$

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

for any value of n we need only 3 Variables sum, i, n ⇒ so S.C is $\theta(1)$ / (on) / $O(1)$

* the amount of memory space doesn't grow interms of I/p size. If no of Variables are constant [not array of variables] then its considered as constant S.C.

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

I/p size
If n is 1000 array is of size 1000
" 10000 " " 10000
So the S.C is $\theta(n)$
The space grows linearly in terms of I/p size.

Auxiliary Space: Order of growth of extra space or temporary space in terms of I/p size.

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

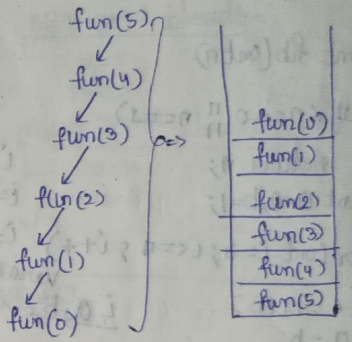
extra space is not required So $\theta(1)$
for arr space, S.C is $\theta(n)$

Auxiliary space ⇒ $\theta(1)$
Space Complexity ⇒ $\theta(n)$

* all sorting algorithms have S.C $\theta(n)$ but we cannot compare them in terms of S.C bcz all are same we can compare in Auxiliary space.
Insertion, Bubble, Selection, Heap ⇒ Auxiliary Space $\theta(1)$
Merge ⇒ " " $\theta(n)$
Scud Array
Quick ⇒ " " $\theta(\log n)$ ⇒ Best

So S.C is less used, Auxiliary Space is more used

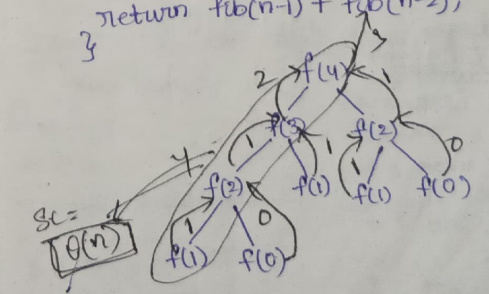
```
⇒ int fun(int n)
{  if (n<=0)
      return 0;
   return n + fun(n-1);
}
```

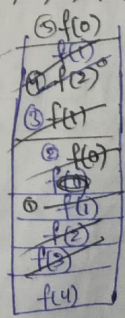


S.C & Auxiliary Space is ⇒ $\theta(n)$ ← for value n, in stack n+1 space is required.

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

Auxiliary Space = Height of the tree
0, 1, 1, 2, 3, 5, 8, 13 - - - ⇒ $\theta(n)$





```
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];
}
```

depends upon n

n=4

0	1	1	2	3
0	1	2	3	4

Both S.C & Auxiliary Space is $\theta(n)$

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

n=4
a=0 b=1

i=a; c=1; a=1, b=1
i=3, c=2; a=1, b=1
i=4; c=3; a=2; b=3

Variables have constant ~~i, a, b, c~~ then S.C or Auxiliary Space is $\theta(1)$