Sum of n natural no's

Approach 1

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

This function takes an int n,
returns Sum.
$n(n+1)/2$ - direct formula
- O(1) time Complexity
- C

Approach 2

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

n+1

- O(n)

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

$O(n^2)$

$i=1 \quad 2 \quad 3$

$1+(1+1)+(1+2+1)$

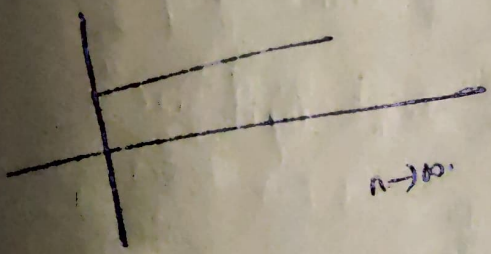
$= 6.$

Upto n times

steps of i will be added.

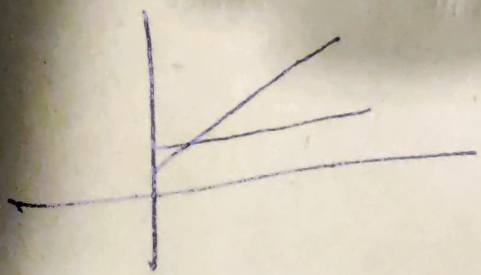
$\frac{n(n+1)}{2}$

$\frac{n^2+n}{2}$. $\frac{1}{2}n^2+n+C$



$O(1) - C -$

So at worst possible outcome, your program will take Constant time.



So your program runs slow initially for low n values.

but for higher n values

it takes Constant time.

(+Hardware
won't be of much issue)

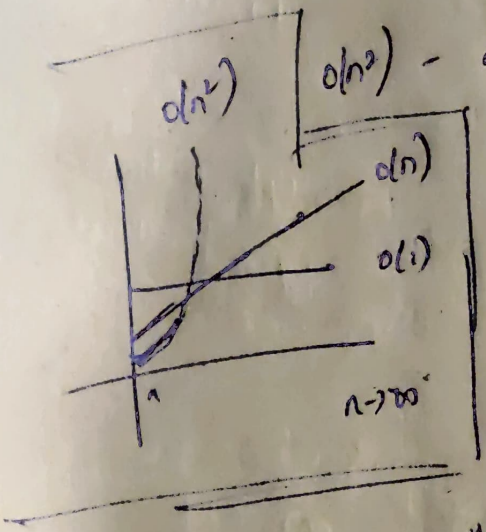
After some time it beats better machine
case.

$O(1) > O(n)$

$O(n)$ - linear

Similarly

$O(1) >> O(n) >> O(n^2)$



Analysis depends on 'n' not time taken by program

"n - input size"

Order of Growth

direct way

- ignore Constants
- ignore leading Constants

$100n+5 \approx O(n)$

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

$n^2 < n^3$

$< n^4 < 2^n$

$< n^n$

Asymptotic Notations :-

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

- $C_1(n) + C_2$
- $c(n)$ //

```
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 ;
        // here add no's - return 0 ;
        // Even no's - Sum - adds up .
```

So

Best Case : Constant

Ex 1 -> 0 (returns)

Average Case : Linear

Under Assumption

Even & odd are equally likely.

So it is depend on n some times,

& Some times its Constant

linear + Const

→ Linear //

n → odd
n → even

Worst Case

n : Always Even

- Linear //

Big O : Represents Exact bound (or) Upper bound

Theta : Represent Exact bound

Omega : Represent Exact (or) lower bound.

Application.

Linear Search

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

return -1;
```

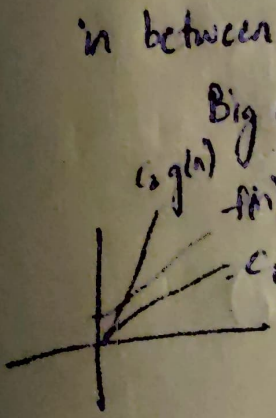
worst Case : O(n). Linear time.

Best Case : O(1). Constant time. Element is at 1st position.

Omega - opposite of Big 'O'.

Theta - Avg Case.

in between of Big O, Omega.



$(c_1 g(n) <= f(n) <= c_2 g(n))$

- We should know Exact bound.
  - best & Worst Case

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

$c^{k-1} < n$ Log on both sides

$k-1 < \log_c n$

$k < \log_c n + 1$

$\therefore \theta(\log n)$

```
for(int i=1; i<n; i=i*c)
{
  ≡≡≡
}
```

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

100, 50, 25, 12, 6 ... $\theta(\log n)$

n n/2 n/4 n/8 n/16

```
for(int i=2; i<n; i=pow(i,c))
{
  ≡≡≡   θ work
}
```

$2, 2^c, (2^c)^c$

$2\ 2^c\ 2^{c^2} \ldots 2^{c^{k-1}}$

$2^{c^{k-1}} < n$ Log on both Sides

$c^{k-1} < \log_2 n$ Log on both Sides

$k-1 < \log_c \log_2 n$

$k < \log_c \log_2 n + 1$.

$\rightarrow \theta(\log_c \log_2 n)$

base can be anything.

in subsequent loop we add

$\theta(n)$
$\theta(\log n)$
$\theta(1)$

Analysis

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

$\theta(n)$ //

in nested loops we multiply.

```
for(int i=1; i<n; i++)          - (n)
    for(int j=1; j<n; j=j*2)    . log n.
```

$n \log n$.

Analysis of Recursion

```
Void fun (int n)
{
    if (n<=0)
        return;          // fun calls itself
    print ("K")
    fun (n/2);
    fun (n/2);
}
```

for $n>0$ (we write recurrence relation)

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

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

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

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

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

$=$

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

```
void fun(int n)
{
    if (n<=1)
        return;
    Print("%d")
    fun(n-1);
}
```

$T(1) = O(1)$

$n > 1$

$T(n) = O(1) + T(n-1)$

$= T(n-1) + O(1)$ //

// Space Complexity.

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

here only 1 variable.

Space doesn't depends on n.

So O(1) (or) c(1).

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

T.C - O(n)

S.C - O(1)

Space doesn't depends on 'n'.

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

here Space depends on n

So if n is 1000, size of n i.e. [1000] will be created

O(n)

Auxillary Space

Space created by memory other than input & output.

A.S - O(1)

S.C - O(n)

```
for (int i = 1; i < n; i++)
{
    Sum += arr[i];
}
return Sum;
```

So in Case of arrays, to optimize space we need to find logic which can use more variable instead of array //

```
for (i=1 ; i<=n ; i++)
for (j=1 ; j<=n ; j=j+i)
    cout << "chanka" ;
```

i=1	i=2	i=3
j = 1 to n	j = 1+2 = 3	j = 1, 4, 7, 10
	i, 3, 5, 7.	1 to n/3
	n/2	

$$n + n/2 + n/3 + \cdots n/n$$

$$n\left[1 + \underbrace{1/2 + 1/3 + \cdots 1/n}_{\text{H.P.}}\right]$$

$$n\left[\log_e n\right]$$

$$= O(n \log_e n)$$

for(int i=1 ; i<=n ; i=i*2)

1 2 4 . . . n

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

$2^0+2^1+2^2+2^3 \cdots 2^k$ $\quad k+1 = p-1$

$n = 2^k$

$\log n = k \log 2$

$\frac{\log n}{\log 2} = k$ $\quad k = \log_2 n$

3 6 9 = $\log_3 n$

4 8 12 $\log_4 n$

11

for(i=n/2 ; i<=n ; i++) $\quad n/2 - n \quad n/2$

for(j=1 ; j<=n/2 ; j++) $\quad 1 - n/2 \quad n/2$

for(k=1 ; k<=n ; k++) $\quad 1-n \quad n$

$k = k*2$

$\log n$

$= n/2 \cdot n/2 \cdot n$

$= n/2 \, n/2 \log n$

$= n^2 \log n$

$= O(n^3)$ //

```
for(i=1; i<=n; i++)
    for(j=1; j<=m; j++)
        Cout << "hello";
```

input n,m

n m.

O(n)(m) O(n×m)

1 2 3 4 ... n

1 to m 1 to m 1 to m 1 to m 1 to m

m m m m m

[m + m + m + m + ... m]

m[1 + 1 + 1 + ... 1] — n times

m[n]

= O(m(n))

if both n

n×n

= O(n²)

```
int i=1; i<=n; i++
2  i=1; i<=n^2; i++
c    i=1; i<=n^3; i++
4
```

$O(n^3)$.

$i <= n^3$

$n^3 \in n^x$

$n^3 \in n^4$

```
for (int i=1; i<=n; i++)
for (int j=1; j<=i^2; j++)

    for (k=1; k<=n/2; k++.
```

i=	1	2	3	...	n
j=	1	4	9		n^2
k=	$n/2$	$4*n/2$	$9*n/2$		$n^2 n/2$.

$n/2$.

$n/2 + 4n/2 + 9n/2 + \cdots n^2 n/2$

$n/2 \left(1 + 4 + 9 + \cdots n^2\right)$

$\left(1^2 + 2^2 + 3^2 + \cdots n^2\right)$

$n/2 \left(n(n+1)(2n+1)/6\right)$

$= O(n^4)$ //

$\frac{n(n+1)}{2}$ → Arithmetic progression

prime no

```
for(int i=2 ; i<=n-1 ; i++)
    if(no%2==0)
        not prime
    return 0;
return 1;
```

worst - $O(n)$

Best Case - $\Omega(1)$

$\theta(n/2) = \theta(n)$ //

```
for(i=1; i<=n ; i++)
for(j=1 ; j<=i^2 ; j++)
```

1	2	3	4	m
1-1	1-4	1-9	1-16	$1-n^2$
1	4	9	16	n^2

$(1+4+9+16+\ldots n^2)$ Sum of 1st n natural square

n^2 - natural no $\left[\frac{n(n+1)(2n+1)}{6}\right]$ nos

$n^2+n(2n+1)$

$= 2n^2+n^2+2n^2+n$

$= 2n^3+3n^2+n = (n^3)$ //

for (i=1; i<=n; i++)

for (j=1; j<=n; j++)

"Hello"

$O(n^2)$

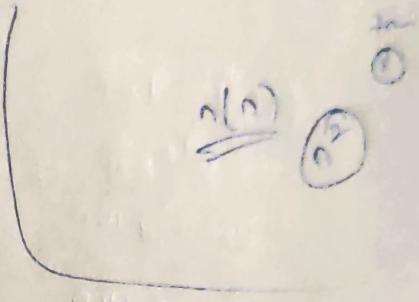
1 2 ... n

1 to n 1 to n 1 to n

n n n

$n + n + n + \dots$ n

$n(1+1+\dots 1)$



for (int i=1; i<=n; i++)

for (int j=1; j<=i; j++)

"

1 2 3 ... n

1 to n

1 2 3 n

1 1-2 1-3

1 2 3

1

$[1+2+3+\dots n]$ Sum of n natural no

$= n(n+1)/2$

$= n^2+n/2$ ignore /2 , n

$= O(n^2)$ //