Algorithm Complexity

Space Complexity of an algorithm

- The analysis of algorithm based on **how much memory is required by an algorithm** to solve a particular problem is called space complexity of an algorithm.

- The space required by memory includes:
  - Instruction space
  - Data space

- An efficient algorithm is the one that makes the space requirement as low as possible.

- Although space complexity is important, the inexpensive memory has reduced the significance of space complexity.

Time Complexity of an algorithm

- The analysis of algorithm based on **time of computation ie,** ‘time taken to execute an algorithm and get the desired results’ is called *time complexity* of an algorithm.

- Main objective of time complexity is to compute the performance of different algorithms in solving same problem.

- One possible approach to measure the time complexity is to implement the algorithms using programming language and execute them…...

… but this methods have many shortcomings…like:

i. Time is wasted in implementing all the algorithms as finally only one algorithm would be used .
ii. Time complexity depends on number of factors as listed below those may influence the running time.
  - Type of Processor, RAM used in a machine
  - Type of Operating System used
  - Quality of code and the technique used for writing an algorithm

Change in any of the above factors result in absolute time of the algorithm. So, it is not appropriate to measure the algorithm’s performance.

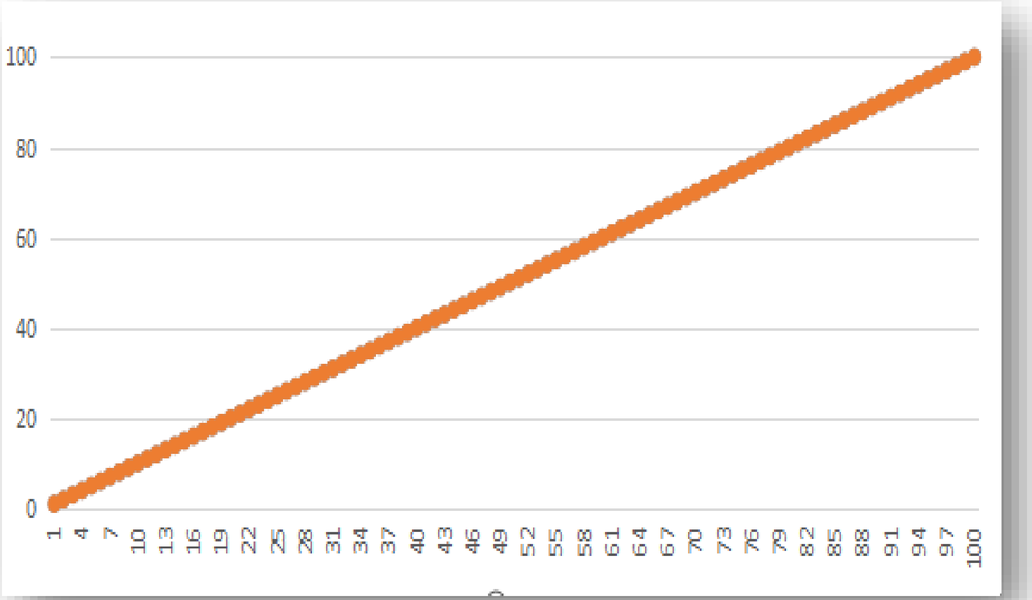
Some examples of finding Time Complexity of an algorithm

Example 1: A segment of algorithm involving simple Loop (with step size 1):

This is an example of a Linear Loop*

Algorithm	Equivalent code

Repeat For i = 1 to 100 by **1**
   Set of statements
<end loop>
	for(i=1; i<=100; **i++**)
	 {
	   ----statements---
	 }



Here, the body of the loop is repeated 100 times.

We know, the time complexity is directly proportional to the number of iterations. So, for a loop with n iterations, it is generally expressed as:

$$f(n) \propto n$$

**Linear Loop* *is the one in which we increment or decrement a counter variable.*

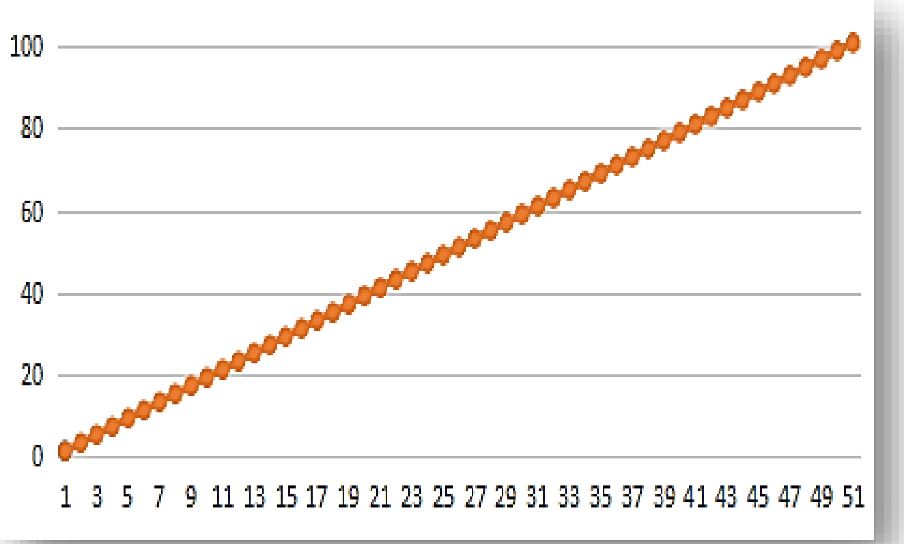
Finding Time Complexity of an algorithm

Example 2: A segment of algorithm involving simple Loop (with step size 2):

Algorithm	Equivalent code

Repeat For i = 1 to 100 by 2
Set of statements
<end loop>
	for(i=1; i<=100; **i=i+2**)
	{
	----statements---
	}

Another example of Linear Loop



Here, the body of the loop is repeated 50 times ie **half of the value of n** because the step size here is 2 (in above example n = 100).

So in this case,

$$\boxed{f(n) \propto n/2}$$

Finding Time Complexity of an algorithm

Example 3: Loop counter is multiplied in each iteration (eg: i = i x 2)

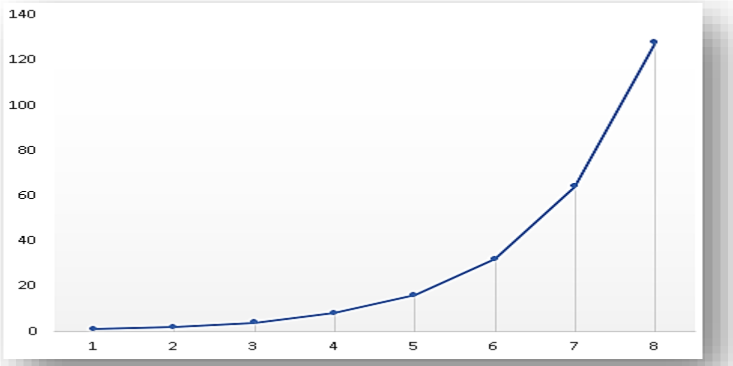
Algorithm	Equivalent code

i=1
Repeat while i <= 100
Set of statements
i = i *2
<end loop>
	for(i=1; i<=100; **i=i*2**)
	{
	---statements---
	}

Table: Value of i for each iteration

Iteration No.	Value of i during each iteration
1	1
2	2
3	4
4	8
5	16
6	32
7	64
8	128 (Exit Loop)

Here, the value of counter variable **i** is **not changing linearly** as it was in previous two examples.



In this example, **no. of iterations are 7 only**. The loop continues as long as following condition is met:

$$2^{\text{iterations}} <= 100$$

Expressing this in terms of logarithms, we can say that there are

$\log_2 100$

ie, **7 iterations**.

So,

$$\boxed{f(n) \propto \log n}$$

Finding Time Complexity of an algorithm

Example 4: Loop counter is divided in each iteration (eg: i = i / 2)

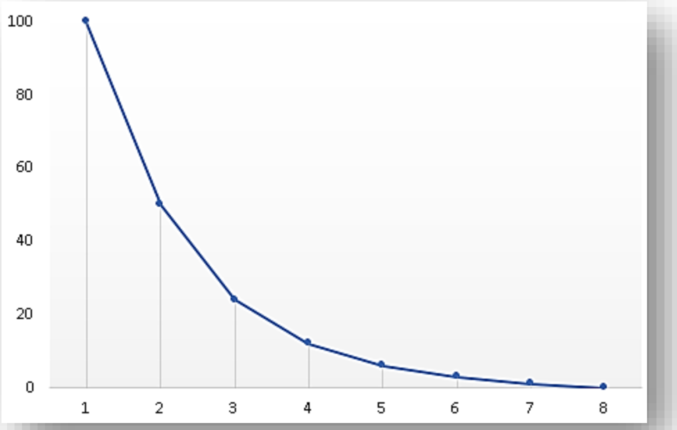
Algorithm	Equivalent code

i=100
Repeat while i >= 1
Set of statements
i = i /2
<end loop>
	for(i=100; i>=1; **i=i/2**)
	{
	---statements---
	}

Table: Value of i for each iteration

Iteration No.	Value of i during each iteration
1	100
2	50
3	24
4	12
5	6
6	3
7	1
8	0 (Exit Loop)

Here also, the value of counter variable **i** is not changing linearly.



In this example again, no. of iterations are 7 only. The loop continues as long as following condition is met:

$$100/2^{\text{iterations}} >= 1$$

Expressing this in terms of logarithms, we can say that there are **$\log_2$ 100** ie, **7 iterations**.

So, $\boxed{f(n) \propto \log n}$

Example 5: In case of Nested Loops

- **Here,** we need to find out how many times each loop is iterating.
- Total number of iterations is the product of number of iterations in the inner loop and number of iterations in the outer loop.

```
Repeat For i = 1 to 10 by 1
   Repeat For j = 1 to 10 by 1
          ----Statements----
   <end of inner loop>
<end of outer loop>
```

Inner loop is iterated 10 times

Outer loop is also iterated 10 times

So, total no. of iterations are **10 x 10**

In a nutshell, if both loops iterates **n** times, then the time complexity is proportional to **n x n** ie, $\mathbf{n^2}$

So,

$$\boxed{f(n) \propto n^2}$$

Example 6: Another case of Nested Loops

```
Repeat For i = 1 to 10 by i=i+1
   Repeat For j = 1 to 10 by j=jx2
         ----Statements----
   <end of inner loop>
<end of outer loop>
```

Inner loop is iterated **log10** times

Outer loop is iterated **10 times**

So, **total no.** of iterations are **10 x log10**

So in general,

$$\boxed{f(n) \propto n\log n}$$

Finding Time Complexity of an algorithm

Example 7: Another variation of Nested Loops

```
Repeat For i = 1 to 10 by i=i+1

  Repeat For j = 1 to i by j=j+1

      ----Statements----

  <end of inner loop>

<end of outer loop>
```

So, Inner loop is iterated (10 +1)/2 times

Outer loop is also iterated 10 times

So, **total no. of iterations** are **10 x (10+1)/2**

In general,

$$f(n) \propto n(n+1)/2$$

When value of ' i ' in the outer loop is	No. of times the body of inner loop iterated is (j = 1 to i)
1	1 time
2	2 times
3	3 times
4	4 times
5	5. times
6	6 times
7	7 times
8	8 times
9	9 times
10	10 times

So, total no. of times the **statements** inside the **inner loop** runs is: 1+2+3+4+5+6+7+8+9+10 = **55**

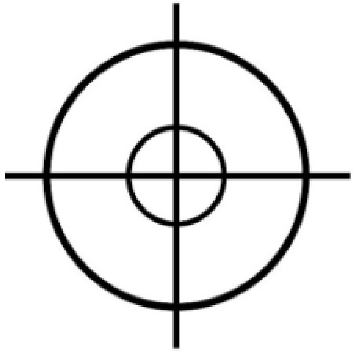
No. of times **outer loop** iterated is : **10**

∴ No. of times **inner loop** iterated is : **55/10 = 11/2 = (10+1)/2**

Time Complexity can further be in terms of …

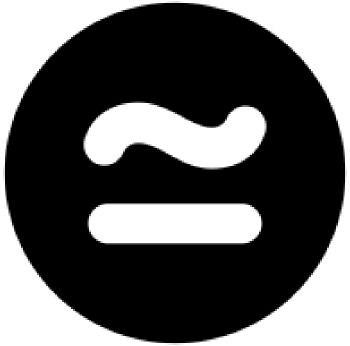
Exact Measurement

of Algorithm's Time Complexity



Approximate Measurement

of Algorithm's Time Complexity



EXACT Measurement of Time Complexity

- When we consider **time taken by each and every statement of an algorithm** to calculate the total time of execution, it is the case of **exact measurement of time complexity** for that algorithm.

Let us understand the exact measurement with help of some code snippets:

Example-1:

```
void  Multiplication_Table( int  num,  int  n)
  {
      int  m;

      for( int  i = 1;  i<=n;  i++)
        {
           m = num * i;

           cout<< num << " * " << i << "=" << m;
        }

    return;
  }
```

Each of these is executed **single** time

Each of these is executed **n** times

executed **single** time

In this example code,

- No. of statements executed **single time** (red colour highlights): **3**
- No. of statements executed **n times** (blue colour highlights): **4**

  ie, **n + n + n + n = 4n**

∴ **Time complexity of this code snippet is :**

$$f(n) = 4n + 3$$

Approx. Complexity:

$f(n) = n^2 + 3n + 10$

ETC.

n	f(n)	n^2 value	%	3n value	%	10 value	%
0	10	0	0	0	0	10	100%
10	140	100	71.4	30	21.4	10	7.9%
100	10310	10000	96.99	300	2.9	10	0.11
1000	1003010	10^6	99.69	3000	0.30	10	0.01

1003010 → 1000000

$\simeq n^2$

Asymptotic Complexity.

$n^2 + 3n + 10$

$0 + 0 + 10$

$f(n) = O(n^2)$

Big-O (O)

Big-Omega (Ω)

Big-Theta (Θ)

1 worst case

2 Best case

3 Avg. case

1 Worst case

11, 55, 12, 33, 14, 26, 88, 54, 39, 62

62 89

62

2 Best Case

11

3.

Big-O Big-oh.

1894.

Paul Bachmann

$f(n) = n^2 + 3n + 10$

$O(n^2)$

$f(n) = 5n^3 + 3n^3 + 2n^2 + 10n + 200$

$f(n) = 8n^3 + 2n^2 + 10n + 200$

$8n^3$

$= O(n^3)$

$n^3 + n^2 + n + 200.$

$\boxed{O(n^3)}$

$n! > 2^n > n^k > \ldots > n^3 > n^2 > n\log n > n > \log n > 1$

fac exp. poly. Quad. linear log. linear log. constant.

P

A_1 A_2 A_3 A_4 A_5

$O(n^3)$ $O(n\log n)$ $O(n!)$ $O(\log n)$ $O(2^n)$

Ranking

$O(n!)$

$O(2^n)$

$O(n^k)$

$O(n^3)$

$O(n^2)$

$O(n\log n)$

$O(n)$

$O(\log n)$

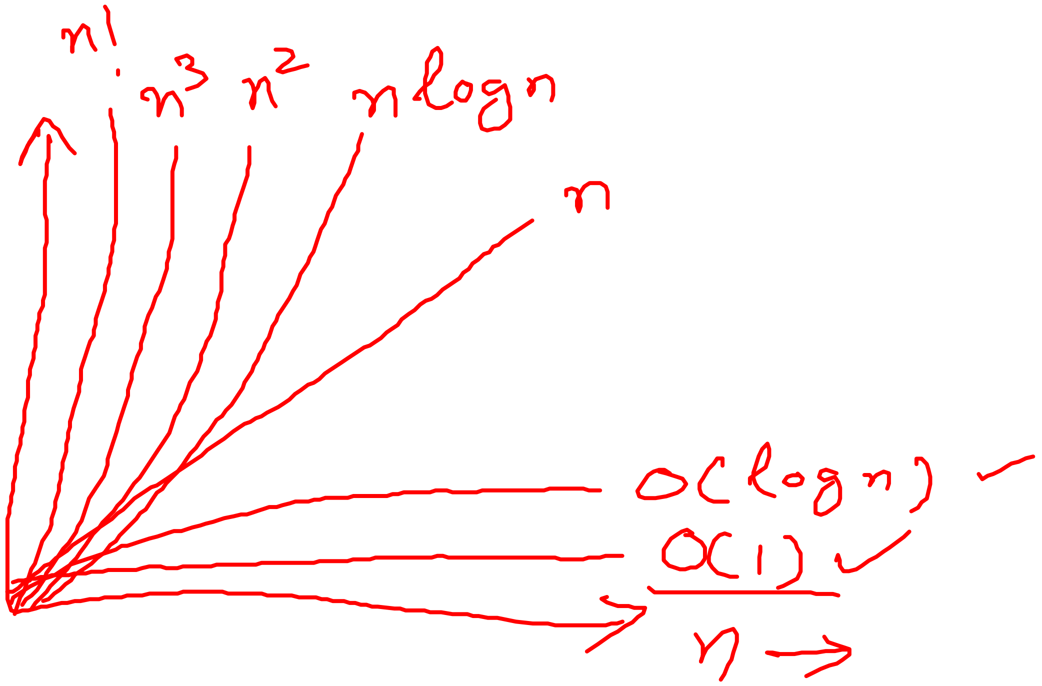
$O(1)$

$f(n) = 2n^3 + 5n + \log n + 20$

$f(n) \simeq n^3$

$O(n^3)$

$O(n)$



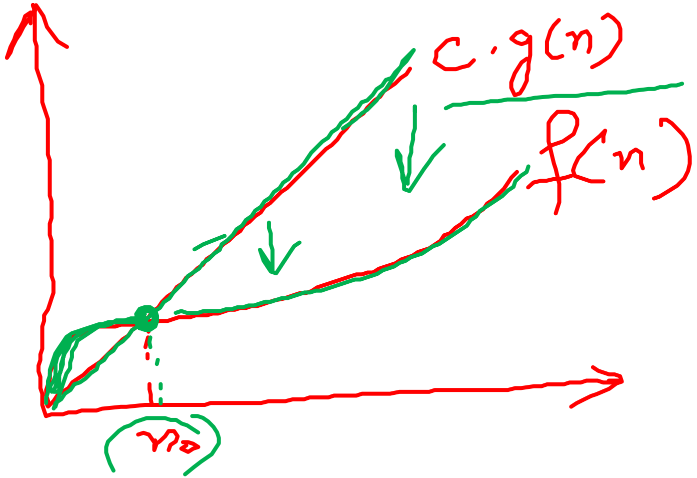
Big-O

Let $f(n)$ and $g(n)$ be two functions where n is a positive integer, then we can say, that $f(n)$ is of order of $g(n)$ as

$$\boxed{f(n) = O(g(n))}$$

iff $\exists$ a $c > 0$ (const. value)

s.t. $\boxed{|f(n)| \le c|g(n)|}$



$n = 0$ ✓
$n = 10$ ✓
$n = 100$
$n = 1000$

Ex: Show that $f(n) = 3n^2 + 3n + 2$ is of order of n^2

Soln:

$f(n) = n(n+1)/2$

$\boxed{O(n^2)}$

$f(n) \leq c\, g(n)$

Big-O.

$f(n)$

$O(g(n))$ $\Omega(g(n))$ $\Theta(g(n))$

$$\boxed{f(n) = O(g(n))}$$

$$f(n) = 8n^2 + 3n + 50$$

$$f(n) \le c\,(g(n))$$

$\rightarrow n \geq n_0$

$\rightarrow c > 0$

$f(n) = 3n^2 + 2n + 5$

$g(n) = n^2$

$f(n) = O(g(n))$

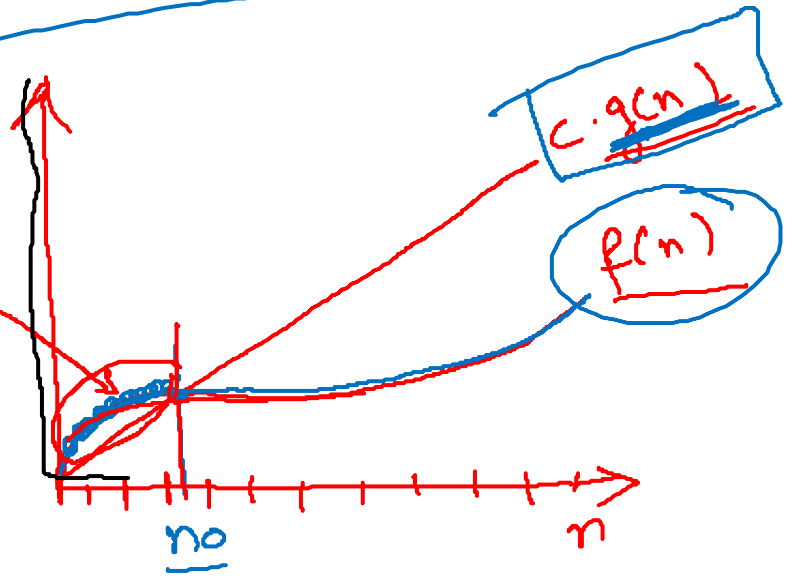
$$\boxed{f(n) \le c \cdot g(n)}$$

$\rightarrow c > 0$

$n \geq n_0$

$g(n) = n^3$

$g(n) = n^4$



$g(n) = n^2$

$g(n) = n^k$ where $k \geq 2$

n^2

n^3

n^4

n^5

$\vdots$

② Big omega notation (Ω)

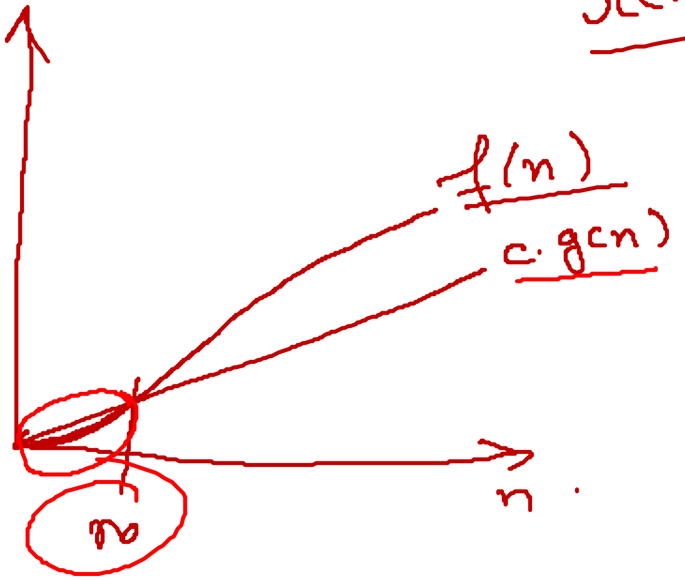
Best case

$$f(n) = \Omega(g(n))$$

$$f(n) \geq c \cdot g(n)$$

$n \geq n_0$

$c > 0$



$\Omega(n^2)$

$\Omega(n\log n)$

Big-O

$f(n) = O(g(n))$

$f(n) \leq c \cdot g(n)$

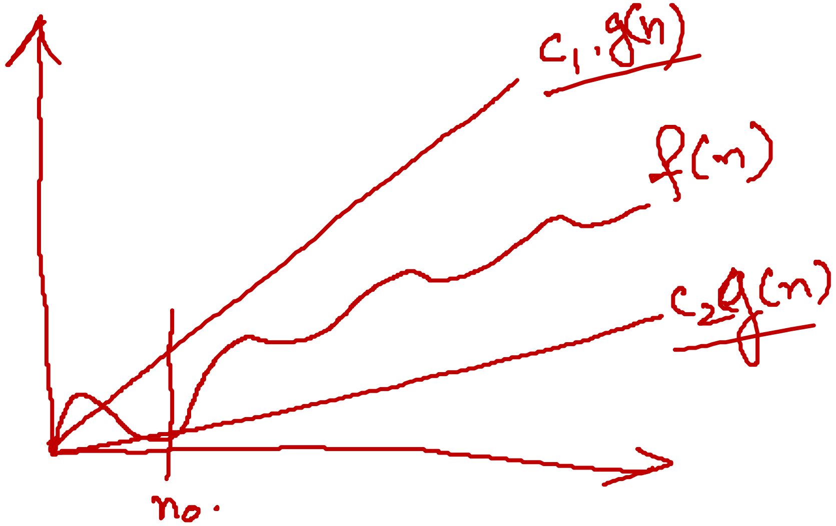
$O(n^2)$

$O(n\log n)$

③ Big-Theta Notation (Θ)

$$\boxed{f(n) = \Theta(g(n))}$$

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



Examples:

P

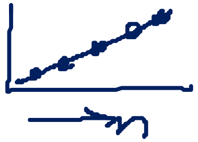
		Big-O
$A1 =$	20	$O(1)$
$A2 =$	$n^{1/2} + 3$	$O(n^{1/2})$
$A3 =$	$n \log n + 2$	$O(n \log n)$
$A4 =$	$n! + n$	$O(n!)$
$A5 =$	$n + 2n^2 + 5n^3$	$O(n^3)$

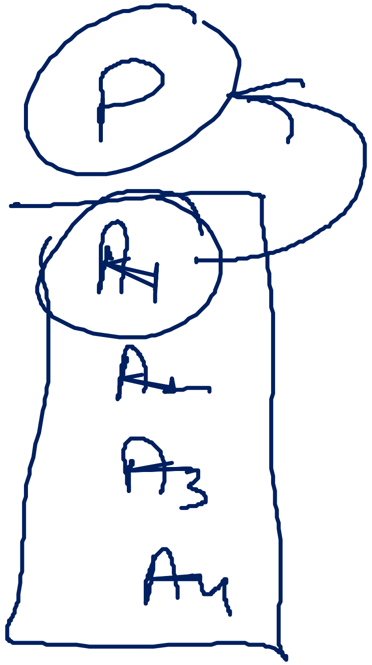
$n = 1000$

$$O(1) < O(n^{1/2}) < O(n \log n) < O(n^3) < O(n!)$$

Catagories of Time complexity.

Category	Big O	Example	Description
→ constant	$\boxed{O(1)}$	Find value of 5th ele. of array	no change in I/p.
→ Logarithmic	$O(\log n)$	Binary Search.	
→ Linear	$O(n)$	Linear Search.	
→ Linear-Logarithmic	$O(n \log n)$	Merge Sort, Quick Sort	
→ Quadratic	$O(n^2)$	selection Sort	
→ Polynomial	$O(n^k)$	Matrix mult. $n=3$	
→ Exponential	$O(2^n)$	Travelling Salesman prob.	
→ Factorial	$O(n!)$	Brute-Force Search.	Huge.





$i = i/2$

Example: $A_1 \rightarrow \boxed{2n\log n}$ $O(2n\log n)$

Time : $\boxed{1 \text{ nano sec}}$ $(10^{-9}$ sec$)$

Input size = 1000 $(n = 1000)$

$f(n) \times$ Time each op

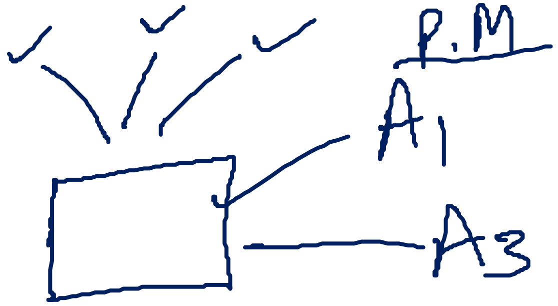
$$f(n) = 2n\log n$$
$$= 2(1000)\log(1000)$$
$$T.T = \boxed{2000\log 1000} \times 10^{-9} \text{ sec}$$
$$= 2\log 1000 \times 10^{-6} \text{ sec}$$
$$= 2 \times 10 \times 10^{-6} \text{ sec}$$

Ex:

Algo-No	Time Taken (in milli sec): $n=10$		$n=100$	
Algo 1	$1/100$	Best	$1/10$	Best
Algo 2	1	W	10	W
Algo 3	$1/100$	Best	1	

$O(1/n^2)$

P.M $n=10$ $n=100$



$O(1)$

$A_1 > A_3 > A_2$.

Best = A_1 Worst = A_2.

1. Big-O (worst case)

$f(n) = O(g(n))$

$f(n) \leq c \cdot g(n)$

~~where~~

$c > 0$

$n \geq n_0$

2. Big-Ω (Best case)

$f(n) = \Omega(g(n))$

$f(n) \geq c \cdot g(n)$

where $c > 0$

$n \geq n_0$

3. Big Θ (Avg. case)

$f(n) = \Theta(g(n))$

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

where $c_1, c_2 > 0$

$n \geq n_0$.

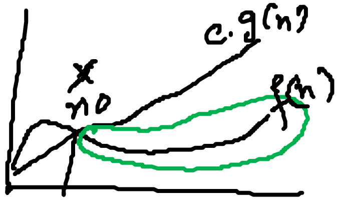
4. Small-o

$f(n) = o(g(n))$

$f(n) < c \cdot g(n)$

$c > 0$

$n > n_0$



5. small omega.

$f(n) = \omega(g(n))$

$f(n) > c \cdot g(n)$

$n > n_0$

$c > 0$