What is an algorithm?

- An *algorithm* is a sequence of unambiguous instructions for solving a problem,
- that is, for obtaining a required output for any legitimate input in a finite amount of time.
- Algorithm is a tool for solving any computational problem.
- It may be defined as a sequence of finite, precise and unambiguous instructions which are applied either to perform a computation or to solve a computational problem.
- These instructions are applied on some raw data called the input, and the solution of the problem produced is called the output.

It was named after 9th century (780-850) Persian mathematician **Abu Ja 'far Muhammad ibn-i Musa al-Khwarizmi** and these were originally used in mathematical calculations but they are now widely used in computer programs.

Algorithm: Calculate the average of 'n' numbers	
• Start by initializing a variable called "sum" to zero. • Iterate through each number in the list. • Add the current number to the "sum" variable. • After iterating through all the numbers, divide the "sum" by the total count of numbers in the list. • The result is the average.	**Average(A[], n){** **sum=0** **for(i=0; i<n;i=i+1){** **sum=sum+A[i]** **}** **average=sum/n** **}**

Pseudo Code

- Represent algorithm using a pseudo language
- A combination of the constructs of a programming language together with informal English statements.

Properties of Algorithms?

- **Input**

An algorithm **must be supplied with zero or some finite input values** externally from a specified set-in order to solve the computational problem and generate some output. The input data is transformed during the computation in order to produce the output

- **Output**

The algorithm produces some finite set of outputs after applying some operations on the given set of input values. The output values are the solution. The output can be anything from data returned to the calling algorithm, displaying the message, printing the calculation etc. It is possible to have no output.

- **Finiteness**

The algorithm must be terminated after executing the finite number of steps so an algorithm must be a well-defined, ordered set of instructions.

- **Definiteness**

Each step of an algorithm must be clear and unambiguous so that the actions can be carried out

without any ambiguity. For Example, same symbol should not be used to mean multiplication as well as division in two different places in the algorithm.

- **Effectiveness**

The algorithm must perform each step correctly and in a finite amount of time therefore time tends to be more important in calculating the effectiveness of an algorithm. The space and other resources taken up by algorithm also plays vital role in effectiveness of an algorithm. Effectiveness is precisely measured after translating the algorithm into a computer program.

- **Correctness**

An algorithm must produce the correct output values for all legal input instances of the problem

- **Generality**

The algorithm should be applicable to all problems of a similar form

- **Multiple view**

Same algorithm may be represented in different ways

Analysis of algorithms:

Analysis of Algorithm is the quantitative study of the performance of algorithms, in terms of their run time, memory usage, or other properties.

- Analysis of algorithms is the **process of evaluating and predicting the performance characteristics of an algorithm.**

- It helps in understanding how an algorithm's efficiency scales with the input size and allows for comparison between different algorithms.

Design and Analysis of Algorithms:

✓ **Analysis:** predict the cost of an algorithm in terms of resources and performance

✓ **Design:** design algorithms which minimize the cost.

Performance of Algorithm/Program

- It is the amount of computer memory and time needed to run a program.

- We use two approaches to determine the performance of a program.
    - Analytical
    - Experimental.

- In ***performance analysis*** ***we use*** ***analytical*** methods, while in ***performance measurement*** ***we conduct experiments***.

Performance Analysis:

- Performance analysis **involves assessing the efficiency and effectiveness of an algorithm by using analytical methods**.
- It focuses on understanding and predicting the algorithm's behavior based on mathematical models and theoretical analysis.
- Key techniques used in performance analysis include:

Asymptotic Analysis: This method evaluates how an algorithm's performance scales with the input size. It provides insights into the algorithm's time complexity (how its execution time increases as the input grows) and space complexity (how much additional memory it requires).

Big O Notation: Big O notation is a mathematical notation used to describe the upper bound of an algorithm's time complexity. It helps classify algorithms into different complexity classes (such as O(1), O(log n), O(n), O(n^2), etc.) and compare their efficiency.

Worst-case, Average-case, and Best-case Analysis: Algorithms can have different performance scenarios

depending on the input data. By analyzing the worst-case (maximum input), average-case (average input), and best-case (minimum input) scenarios, we gain insights into how the algorithm performs under different conditions.

Amortized Analysis: Amortized analysis is a technique used to analyze the overall average performance of an algorithm over a sequence of operations, rather than focusing on individual operations. It provides a way to distribute the cost of expensive operations across multiple cheaper operations, resulting in a more balanced analysis.

Performance Measurement:

- Performance measurement involves evaluating the actual performance of an algorithm through experiments and real-world observations.
- It focuses on obtaining empirical data by running the algorithm on specific inputs and measuring various performance metrics.
- Key techniques used in performance measurement include:

Execution Time Measurement: This involves measuring the time taken by an algorithm to execute on a specific input. Techniques such as timing functions or using profiling tools are employed to accurately measure the execution time.

Memory Usage Measurement: Memory consumption is another important aspect of performance measurement. Tools and techniques are used to monitor and measure the amount of memory an algorithm utilizes during execution.

Benchmarking: Benchmarking involves running an algorithm on a predefined set of inputs and measuring its performance metrics (e.g., execution time, memory usage) against other algorithms or established standards. This helps in comparing and evaluating different algorithms based on their real-world performance.

Time Complexity:

- The time complexity of an algorithm is the total amount of time required by an algorithm to complete its execution.

- Generally, running time of an algorithm depends upon,

    - Whether it is running on Single processor machine or Multi processor machine.
    - Whether it is a 32-bit machine or 64-bit machine
    - Read and Write speed of the machine.

- The time it takes to perform Arithmetic operations, logical operations, return value and assignment operations etc.,
- Input data

- When we calculate time complexity of an algorithm, we consider only input data and ignore the remaining things, as they are machine dependent.

- We check only, how our program is behaving for the different input values to perform all the operations like Arithmetic, Logical, Return value and Assignment etc.,

- To calculate time complexity of an algorithm, we need to define a model machine. Let us assume a machine with following configuration...

1. It performs sequential execution.
2. It requires 1 unit of time for Arithmetic and Logical operations.
3. It requires 1 unit of time for Assignment and Return value.
4. It requires 1 unit of time for Read and Write operations.

Example 1:

```
int sum(int a, int b)
{
    return a+b;
}
```

- In above sample code, it requires 1 unit of time to calculate `a+b` and 1 unit of time to return the value.

- That means, totally it takes 2 units of time to complete its execution.

- And it does not change based on the input values of a and b. That means for all input values, it requires same amount of time i.e. 2 units.

- If any program requires fixed amount of time for all input values
- then its time complexity is said to be **Constant Time Complexity**.

Example 2 :

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

Time/Operation	Repeatation	Total
1	1	1
1+1+1	1+(n+1)+n	2n+2
1+1	(1+1)2	2n
1	1	1
		4n+4

- In above calculation Time/Operation Cost is the amount of computer time required for a single operation in each line.

- Repetition is the amount of computer time required by each operation for all its repetitions.

- Total is the amount of computer time required by each operation to execute.

- If the amount of time required by an algorithm is increased with the increase of input value, then that time complexity is said to be **Linear Time Complexity.**

Space Complexity:

- The space complexity of a program is the amount of memory it needs to run to completion.
- The space need by a program has the following components:

  - **Instruction space:** Instruction space is the space needed to store the compiled version of the program instructions.

  - **Data space:** Data space is the space needed to store all constant and variable values. Data space has two components:
    - Space needed by constants and simple variables in program.
    - Space needed by dynamically allocated objects such as arrays and class instances.

  - **Environment stack space:** The environment stack is used to save information needed to resume execution of partially completed functions.

Algorithm Design Goals

- The basic design goals that one should strive for in a program are:

- Try to save Time
- Try to save Space

- A program that runs faster is a better program, so saving time is an obvious goal.
- Likewise, a program that saves space over a competing program is considered desirable.

Asymptotic Notations

- Asymptotic notations are the mathematical notations used to describe the running time of an algorithm when the input tends towards a particular value or a limiting value.
- For example: In bubble sort, when the input array is already sorted, the time taken by the algorithm is linear (The BEST CASE).
- But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements (The WORST CASE).
- When the input array is neither sorted nor in reverse order, then it takes average time (AVERAGE).
- These durations are denoted using asymptotic notations.

There are three main asymptotic notations but five in total:

- Big-O notation (O)
- Big-Omega notation (Ω)
- Big-Theta notation (Θ)
- Small-o notation (o)
- Small-Omega (ω)

Big-O Notation (O-notation)

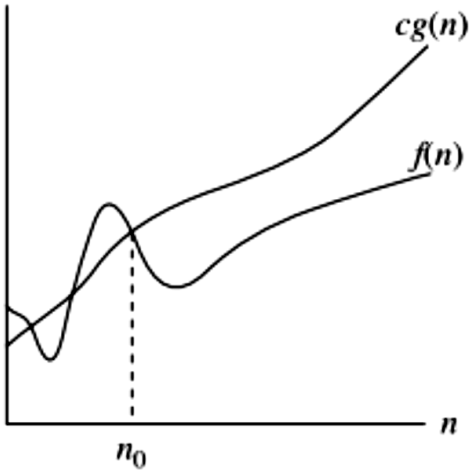
Big - Oh notation is used to define the upper bound of an algorithm in terms of Time Complexity.

That means Big-Oh notation always indicates the maximum time required by an algorithm for all input values. That means Big - Oh notation describes the worst case of an algorithm time complexity.

Big Oh notation can be defined as follow:

O-notation

$O(g(n)) = \{f(n)$: there exist positive constants c and n_0 such that $0 \le f(n) \le cg(n)$ for all $n \ge n_0\}$.



$g(n)$ is an ***asymptotic upper bound*** for $f(n)$.

Big-O gives the upper bound of a function

- ✓ We write $f(n)=O(g(n))$ if $fn \in O(g(n))$.
- ✓ Asymptotic upper bound
- ✓ c is independent of n.

- Consider function $f(n)$ the time complexity of an algorithm and g(n) is the most signification term. If $f(n) \le C\ g(n)$ for all $n \ge n_0$ and C > 0 and $n_0 > 0$, then we represent, $f(n) = O(g(n))$
- The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.

- For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).
- Since it gives the worst-case running time of an algorithm, it is widely used to analyze an algorithm as we are always interested in the worst-case scenario.

Examples:

Example: $2n^2 = O(n^3)$, with $c = 1$ and $n_0 = 2$.

Examples of functions in $O(n^2)$:

n^2
$n^2 + n$
$n^2 + 1000n$
$1000n^2 + 1000n$
Also,
n
$n/1000$
$n^{1.99999}$
$n^2/\lg\lg\lg n$

Consider the following **f(n)** and **g(n)...**

f(n) = 3n + 2 ; g(n) = n

If we want to represent **f(n)** as **O(g(n))** then it must satisfy **f(n) ≤ Cg(n)** for all values of **C > 0** and **n0 ≥ 1**

f(n) ≤ C g(n)

3n + 2 ≤ C n

Above condition is always **TRUE** for all values of **C=4** and **n0 ≥ 2**.

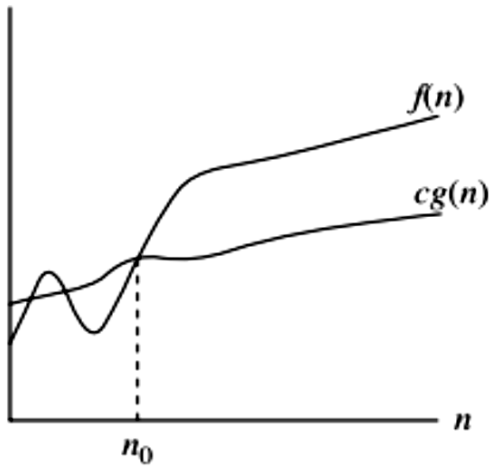
By using Big - Oh notation we can represent the time complexity as follows...

3n + 2 = O(n)

Omega Notation (Ω-notation)

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

$$\Omega(g(n)) = \{f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \le cg(n) \le f(n) \text{ for all } n \ge n_0\} .$$



$g(n)$ is an ***asymptotic lower bound*** for $f(n)$.

Example: $\sqrt{n} = \Omega(\lg n)$, with $c = 1$ and $n_0 = 16$.

Examples of functions in $\Omega(n^2)$:

n^2
$n^2 + n$
$n^2 - n$
$1000n^2 + 1000n$
$1000n^2 - 1000n$
Also,
n^3
$n^{2.00001}$
$n^2 \lg \lg \lg n$
2^{2^n}

Omega gives the lower bound of a function

- The above expression can be described as a function **f(n)** belongs to the set **Ω(g(n))** if there exists a positive constant *C* such that it lies above **Cg(n),** for sufficiently large n.
- For any value of n, the minimum time required by the algorithm is given by Omega Ω(g(n)).

- Big - Omega notation is used to define the lower bound of an algorithm in terms of Time Complexity.
- That means Big - Omega notation always indicates the minimum time
- required by an algorithm for all input values.
- That means Big – Omega notation describes the best case of an algorithm time complexity.
- Big - Omega Notation can be defined as follows.
- Consider function f(n) the time complexity of an algorithm and g(n) is the most significant term. If **f(n) ≥ Cg(n)** for all **n ≥ n0**, **C > 0** and **n0 ≥ 1**. Then we can represent **f(n) as Ω(g(n)).**
- **f(n) = Ω(g(n))**

Example:

Consider the following f(n) and g(n)...

$f(n) = 3n + 2$; $g(n) = n$

If we want to represent f(n) as O(g(n)) then it must satisfy $f(n) \leq Cg(n)$ for all values of $C > 0$ and $n0 \geq 1$

$f(n) \leq C\ g(n)$

$3n + 2 \leq C\ n$

Above condition is always TRUE for all values of $C = 1$ and $n \geq 1$.

By using Big - Omega notation we can represent the time complexity as

follows...

$3n + 2 = \Omega(n)$

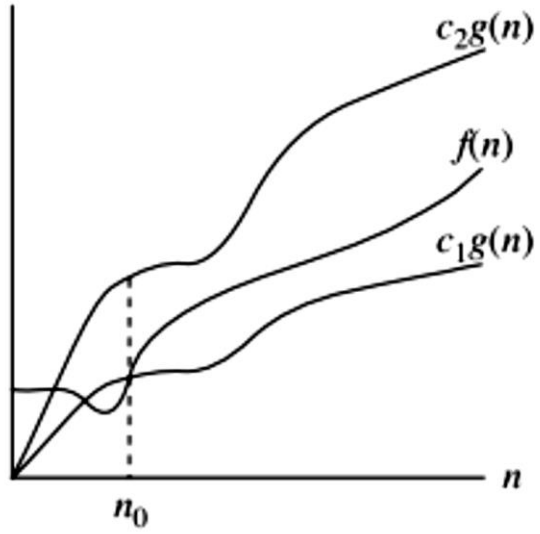
Theta Notation (Θ-notation)

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm.

For a function g(n), Θ(g(n)) is given by the relation:

Θ-notation

$$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2, \text{ and } n_0 \text{ such that } 0 \le c_1 g(n) \le f(n) \le c_2 g(n) \text{ for all } n \ge n_0\}.$$



$g(n)$ is an ***asymptotically tight bound*** for $f(n)$.

Example: $n^2/2 - 2n = \Theta(n^2)$, with $c_1 = 1/4$, $c_2 = 1/2$, and $n_0 = 8$.

- The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c1 and c2 such that it can be sandwiched between c1g(n) and c2g(n), for sufficiently large n.
- If a function f(n) lies anywhere in between c1g(n) and c2g(n) for all n ≥ n0, then f(n) is said to be asymptotically tight bound.
- Big - Theta notation is used to define the **average bound** of an algorithm in
- terms of Time Complexity.

- That means Big - Theta notation always indicates the average time required
- by an algorithm for all input values.
- That means Big - Theta notation describes the average case of an algorithm
- time complexity.
- Big - Theta Notation can be defined as follows...
- Consider function f(n) the time complexity of an algorithm and g(n) is the
- most significant term. If C1*g(n) ≤ f(n) ≥ C2*g(n) for all n ≥ n0, C1,
- C2 > 0 and n0 ≥ 1. Then we can represent f(n) as Θ(g(n)).
- **f(n) = Θ(g(n))**

Example:

Consider the following f(n) and g(n)...

f(n) = 3n + 2 ; g(n) = n

if we want to represent f(n) as Θ(g(n)) then it must satisfy

C1 g(n) ≤ f(n) ≥ C2 g(n)

for all values of C1, C2 > 0 and n0≥ 1

C1 g(n) ≤ f(n) ≥ C2 g(n)

C1 n ≤ 3n + 2 ≥ C2

Above condition is always TRUE for all values of C1 = 1, C2 = 4 and n ≥ 1.

By using Big - Theta notation we can represent the time complexity as follows...

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

Little-O Notation (o)

$o(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq f(n) < cg(n)$ for all $n \geq n_0\}$.

Another view, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$.

$n^{1.9999} = o(n^2)$
$n^2/\lg n = o(n^2)$
$n^2 \neq o(n^2)$ (just like $2 \not< 2$)
$n^2/1000 \neq o(n^2)$

For non-negative functions, f(n) and g(n), f(n) is little o of g(n) if and only if f(n) = O(g(n)), but f(n) ≠ Θ(g(n)). This is denoted as "f(n) = o(g(n))".

This represents a loose bounding version of Big O. g(n) bounds from the top, but it does not bound the bottom.

Little Omega Notation (ω)

$\omega(g(n)) = \{f(n) :$ for all constants $c > 0$, there exists a constant $n_0 > 0$ such that $0 \leq cg(n) < f(n)$ for all $n \geq n_0\}$.

Another view, again, probably easier to use: $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$.

$n^{2.0001} = \omega(n^2)$
$n^2 \lg n = \omega(n^2)$
$n^2 \neq \omega(n^2)$

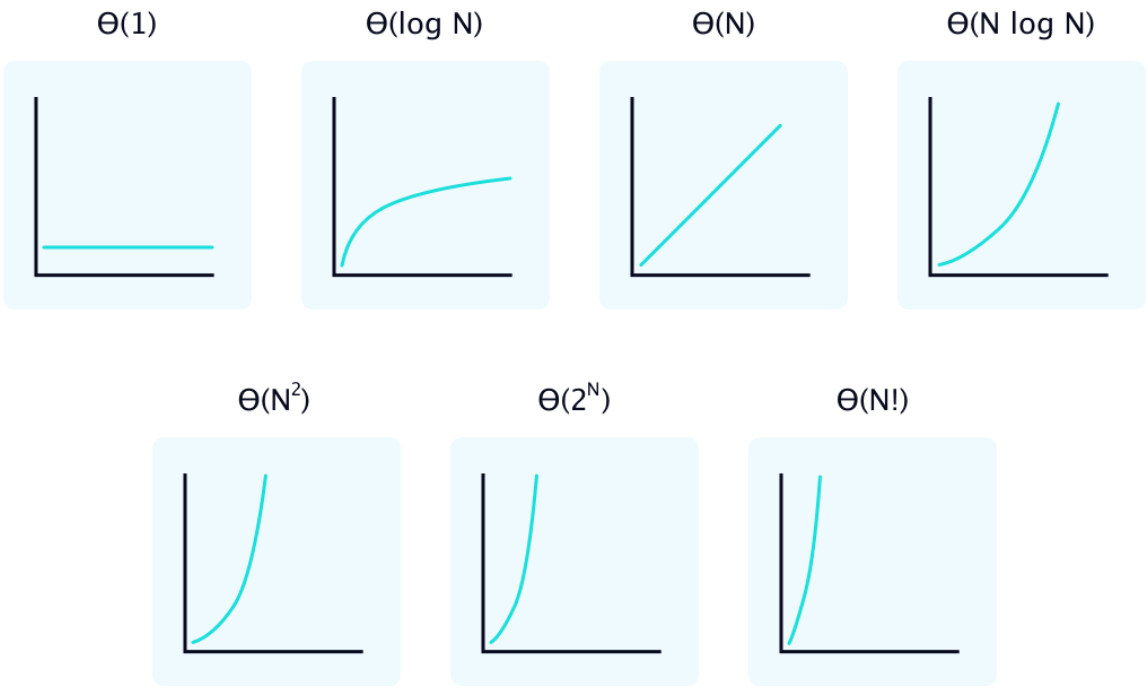
For non-negative functions, f(n) and g(n), f(n) is little omega of g(n) if and only if $f(n) = \Omega(g(n))$, but $f(n) \neq \Theta(g(n))$. This is denoted as "$f(n) = \omega(g(n))$".

Algorithmic Common Runtimes

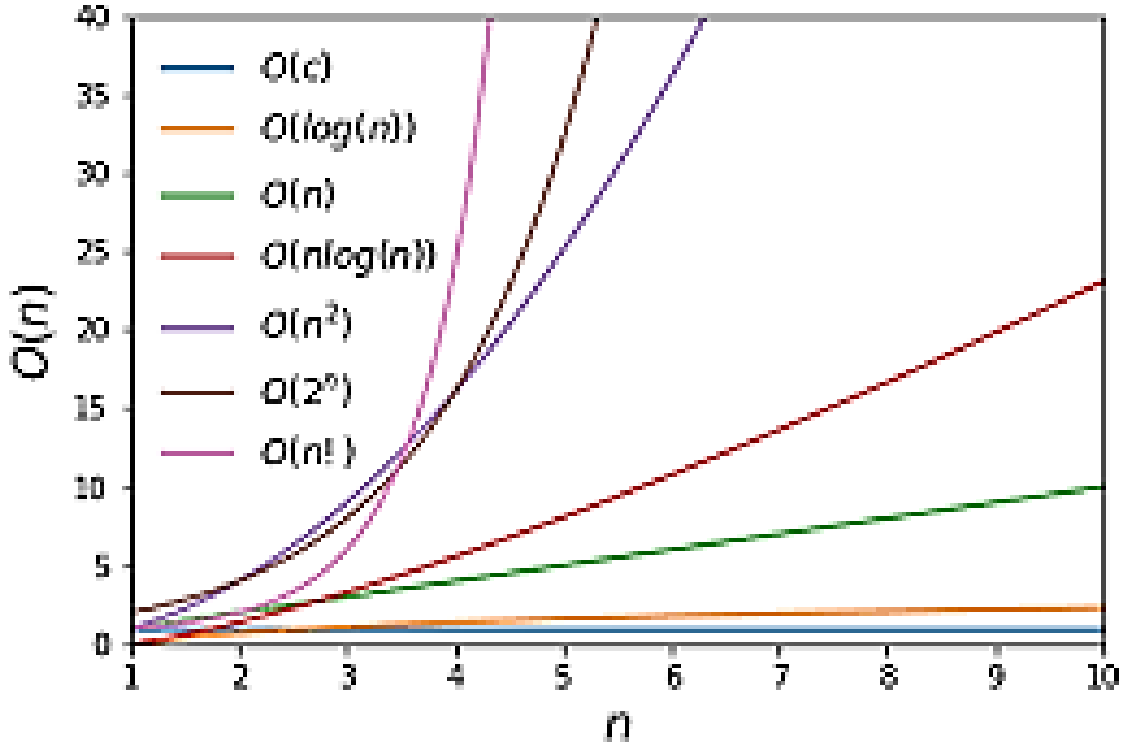
n	$\log_2 n$	$n*\log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65,536
32	5	160	1024	32,768	4,294,967,296
64	6	384	4096	2,62,144	Note 1
128	7	896	16,384	2,097,152	Note 2
256	8	2048	65,536	1,677,216	????????

Note1: The value here is approximately the number of machine instructions executed by a 1 gigaflop computer in 5000 years.

Common Runtimes



$O(1)$, $O(\log_2 n)$, $O(n)$, $O(n. \log_2 n)$, $O(n^2)$, $O(n^3)$, $O(2^n)$, $n!$ and n^n



Properties of Equality

Reflexive Property : $x = x$

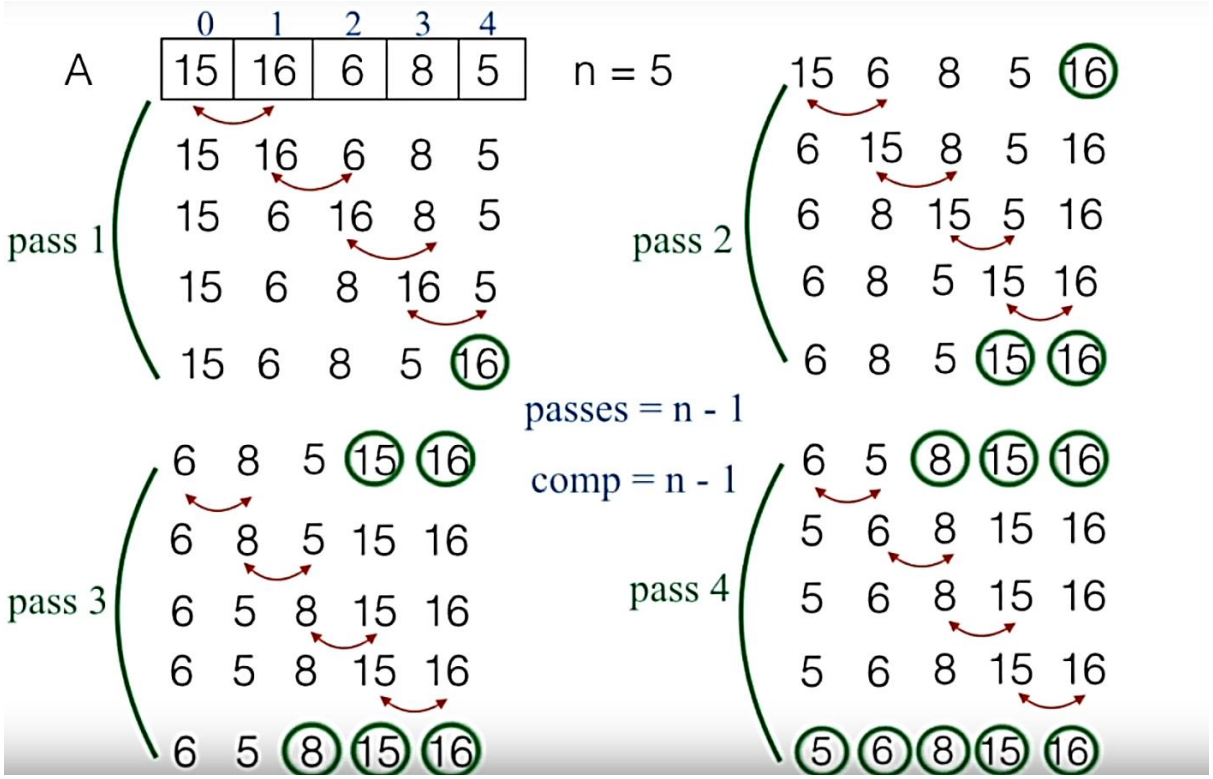
Symmetric Property : If $x = y$, then $y = x$

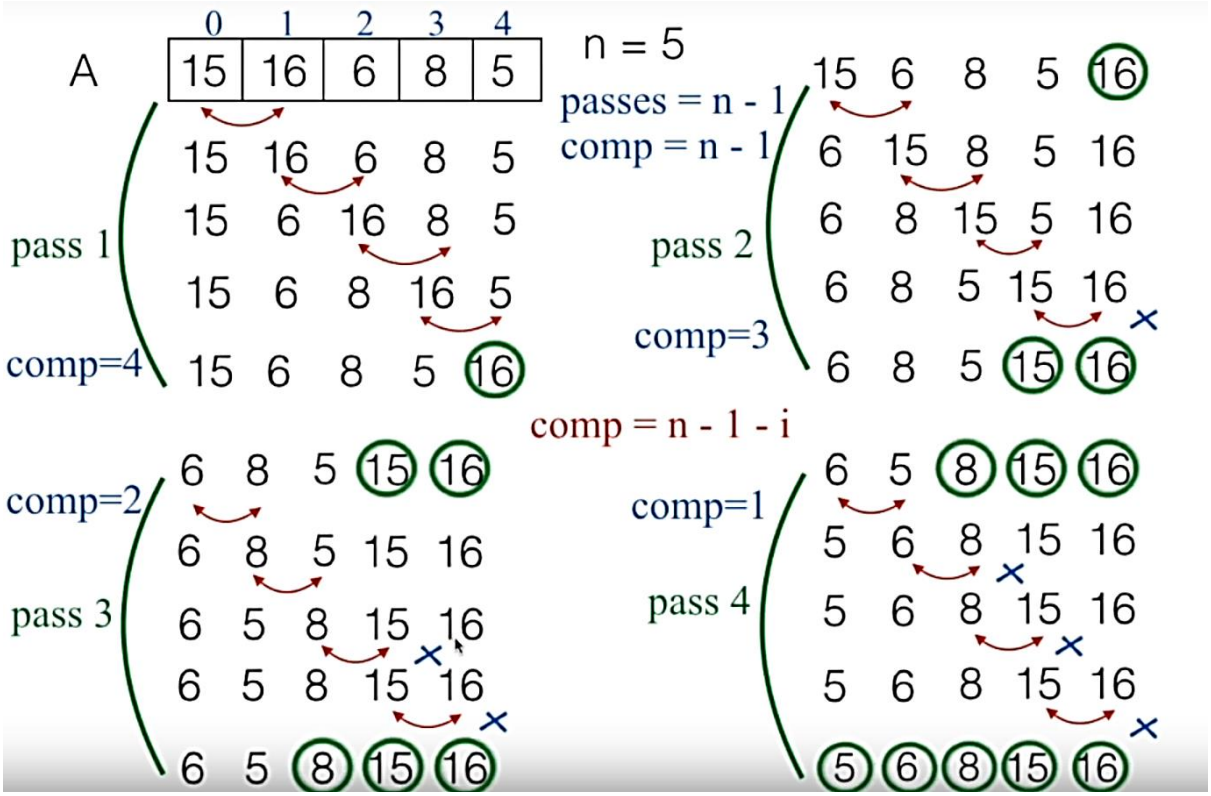
Transitive Property : If $x = y$ and $y = z$, then $x = z$

Notation	Reflexive	Symmetric	Transitive
O (O≈ ≤)	✓	✗	✓
Ω (Ω≈ ≥)	✓	✗	✓
Θ (Θ ≈ =)	✓	✓	✓
o (o≈ <)	✗	✗	✓
ω (ω≈ >)	✗	✗	✓

Best Case Worst Case Average Case Analysis of Bubble Sort

```
for(i = 0; i < n-1; i++)
{
   for(j = 0; j < n-1; j++)
   {
    if(A[j] > A[j+1])
    {
       temp = A[j];
       A[j] = A[j+1];
       A[j+1] = temp;
    }
   }
}
```

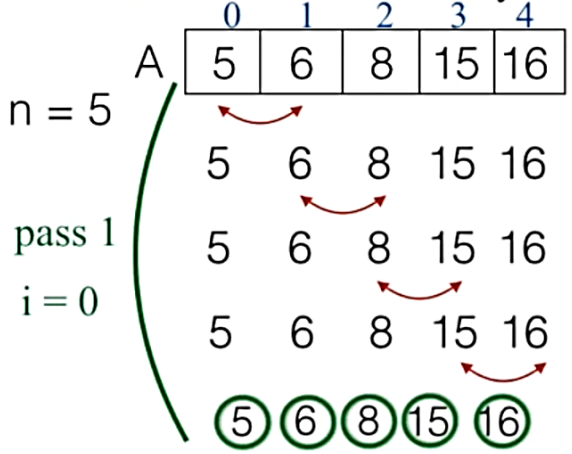




Optimized code:

```
for(i = 0; i < n-1; i++){
    flag = 0
    for(j = 0; j < n-1-i; j++)
    {
     if(A[j] > A[j+1])
     {
        temp = A[j];
        A[j] = A[j+1];
        A[j+1] = temp;
        flag = 1;
        }
    }
   if(flag != 1)
     break;
}
```

Best case : Elements are already sorted.



comparisons: n-1 = 4

Optimized code:

```
for(i = 0; i < n-1; i++){
    flag = 0
    for(j = 0; j < n-1-i; j++)
   {
    if(A[j] > A[j+1])
    {
       temp = A[j];
       A[j] = A[j+1];
       A[j+1] = temp;
       flag = 1;
       }
   }
   if(flag != 1)
     break;
}
```

Worst case: Elements are in descending order.

	0	1	2	3	4	
A	16	15	6	8	5	n = 5

How many passes??

- n -1 passes = 4
- n - 1 - i comparisons

pass 1(i = 0)	pass 2(i = 1)	pass 3(i = 2)	pass 4(i = 3)
4 Comp	3 Comp	2 Comp	1 Comp

Total no.of comparisons: 4 + 3 + 2 + 1 = 10

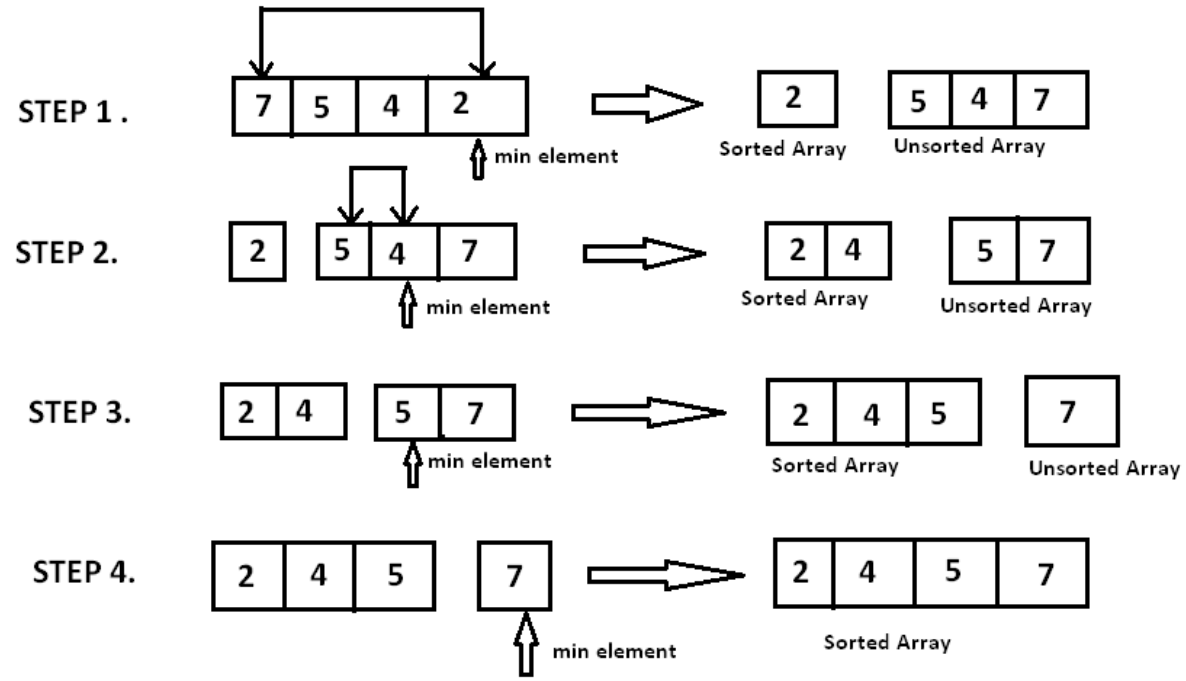
$\frac{n(n-1)}{2} = \frac{1}{2}(n^2 - n) = O(n^2)$ **Average case**

$\frac{n(n-1)}{2} = \frac{5(5-1)}{2} = 10$

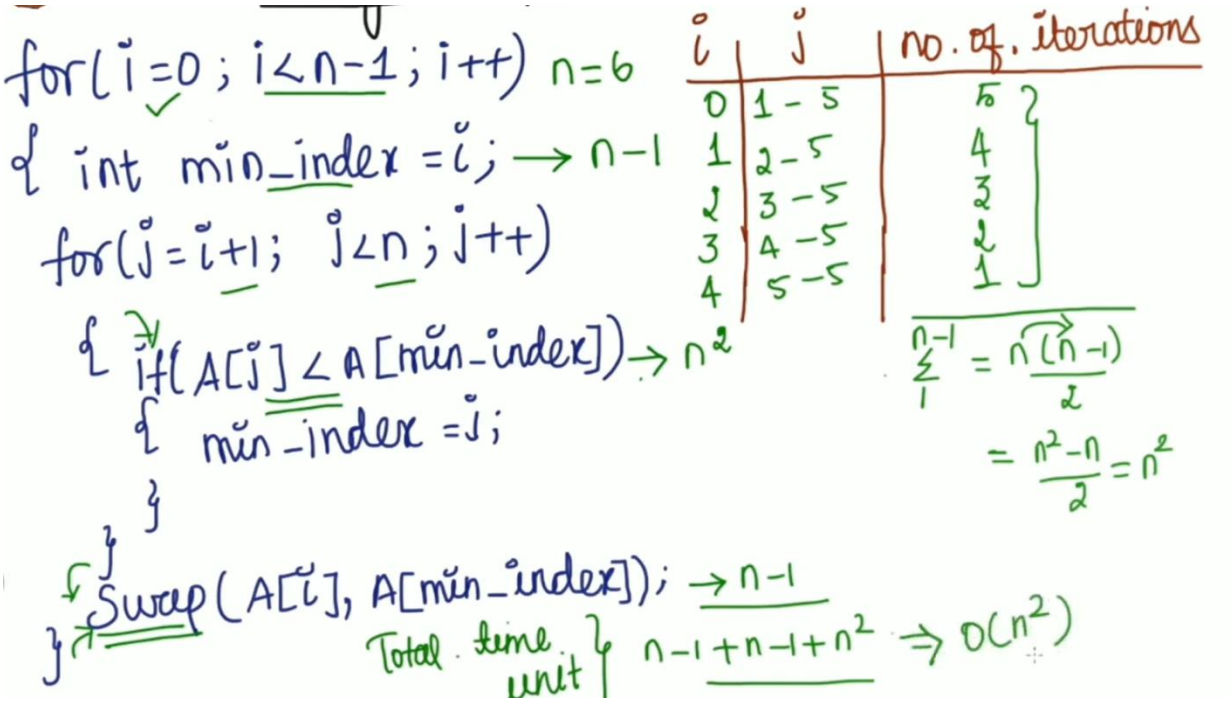
Time complexity:

	Best	Worst	Average
General code	O(n^2)	O(n^2)	O(n^2)
Optimized code	O(n)	O(n^2)	O(n^2)

Best Case Worst Case Average Case Analysis of Selection Sort



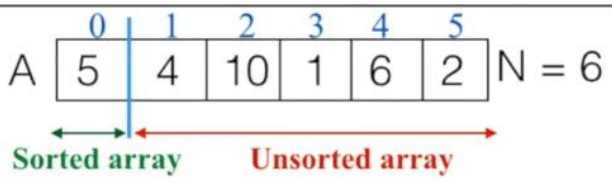

```
void selection_sort (int A[ ], int n) {
        // temporary variable to store the position of minimum element

        int minimum;
        // reduces the effective size of the array by one in  each iteration.

        for(int i = 0; i < n-1 ; i++)  {

           // assuming the first element to be the minimum of the unsorted array .
             minimum = i ;

          // gives the effective size of the unsorted  array .

            for(int j = i+1; j < n ; j++ ) {
                if(A[ j ] < A[ minimum ])  {                //finds the minimum element
                minimum = j ;
                }
            }
          // putting minimum element on its proper position.
          swap ( A[ minimum ], A[ i ]) ;
        }
   }
```



Best Case, Worst Case and Average Case Complexity O(n2)

Best Case Worst Case Average Case Analysis of Insertion Sort

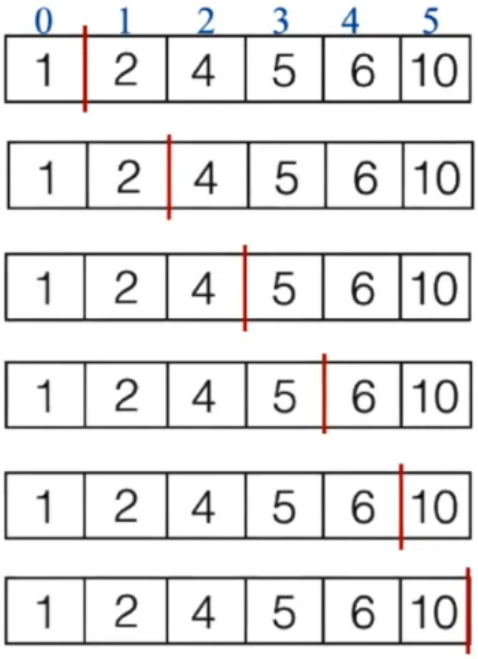
How Insertion sort works?



Program:

```
for(int  i = 1; i < N; i++)
{
   temp = A[i];
   j = i - 1;
   while(j >= 0 && A[j] > temp)
    {
       A[j+1] = A[j];
       j- - ;
    }
    A[j+1] = temp;
}
```

Best Case: Array is already Sorted.

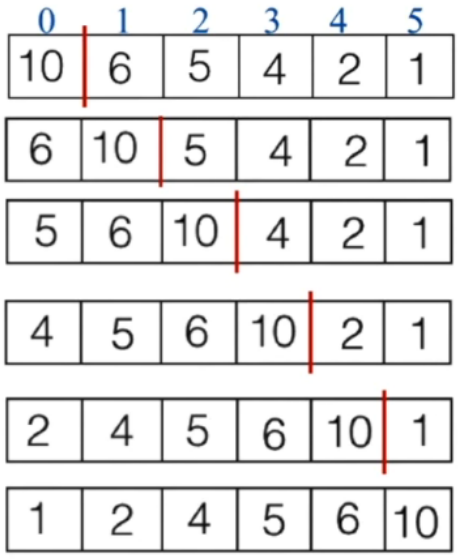


No.of.comparisons

1
1
1 **Best Case: O(N)**
1
1
5 ⟶ **(N-1)**

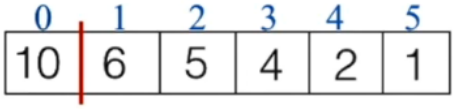
N = 6

Worst Case: Array elements are in descending order.



N = 6

Comparisons:



1 2 3 4 5

Total No.of Comparisons: 1+2+3+4+5

= 1+2+3+4+(N-1)

= N(N-1)/2

= N(N-1) = $N^2 - N$

Worst Case: $O(N^2)$

Time Complexity:

Best Case	Worst Case	Average Case
O(n)	O(n^2)	O(n^2)

Amortized Analysis

- Amortized analysis is ***a technique used*** in computer science and algorithms ***to analyze the average time complexity of an algorithm over a sequence of operations.***
- Rather than analyzing the worst-case time complexity of individual operations. ***It provides a more accurate and realistic understanding of the overall performance of an algorithm.***
- In amortized analysis, the ***cost of a costly operation is spread out over a series of cheaper operations, resulting in a lower average cost per operation***. This approach allows for the possibility of occasional expensive operations as long as they are offset by a sufficient number of inexpensive operations.

Different methods of Amortized Analysis

Aggregate Method

- The Aggregate Method is a straightforward approach to amortized analysis.
- It involves analyzing the total cost of a sequence of operations and dividing it by the number of operations to determine the average cost.
- This method provides an overall perspective of the average performance of the algorithm.

Accounting Method

- The Accounting Method assigns each operation a specific "credit" or "charge" that represents the amortized cost.
- The credits are distributed across operations and used to cover the actual cost of expensive operations.
- This method ensures that the average cost of operations remains bounded and covers the worst-case scenarios.

Potential Method

- The Potential Method assigns a potential function to the data structure being analyzed.
- The potential function measures the state or structure of the data at any given time.
- The amortized cost of an operation is the actual cost of the operation plus the change in potential caused by the operation.
- This method allows for a more fine-grained analysis of the cost associated with specific operations.

Dynamic Array Example

- Let's consider an example of Dynamic Array to illustrate the methods of amortized analysis.
- Dynamic Array are resizable arrays that grow or shrink based on the number of elements.

- We can use the methods discussed to analyze the cost of resizing operations in dynamic arrays.

Aggregate Method Application

- Using the Aggregate Method, we determine the total cost of resizing operations over a sequence of insertions and deletions.
- We divide the total cost by the number of operations to obtain the average cost per operation.
- This allows us to estimate the average time complexity of the operations.

Accounting Method Application

- Applying the Accounting Method, we assign a credit or charge to each resizing operation in the dynamic array.
- We distribute the credits across the insertions and deletions to cover the cost of resizing.
- This ensures that the average cost per operation remains bounded, even in worst-case scenarios.

Potential Method Application

- With the Potential Method, we assign a potential function to the dynamic array.
- The potential function reflects the difference in the size of the array and the number of elements it contains.

- The amortized cost of resizing operations is calculated as the actual cost plus the change in potential.
- This method provides a more detailed analysis of the dynamic array’s state and its impact on the cost of operations.

Bitonic Sorting Network

- Sorting is a fundamental operation in computer science used to arrange elements in a particular order.
- Bitonic Sorting Network is a parallel sorting algorithm that efficiently sorts a sequence of elements in a bitonic order.

Bitonic Sequences:

- A bitonic sequence is a sequence of elements that first monotonically increases and then monotonically decreases or vice versa.
- Formally, a sequence of n elements, a[0], a[1], ..., a[n-1], is bitonic if there exists an index $0 \leq k \leq n-1$ such that $a[0] \leq a[1] \leq ... \leq a[k]$ and $a[k] \geq a[k+1] \geq ... \geq a[n-1]$.
- Bitonic sequences have a distinctive property that facilitates efficient sorting.

Bitonic Sorting Network:

- A Bitonic Sorting Network is a collection of comparators that can sort bitonic sequences.
- The network consists of a series of stages, each stage performing a specific comparison and swapping operation on the input elements.
- The size of the network is determined by the number of elements to be sorted.

Properties of Bitonic Sorting Network:

1. **Bitonicity Preservation:**
    - A Bitonic Sorting Network preserves the bitonicity property of the input sequence.
    - If the input sequence is bitonic, the output sequence will also be bitonic.
    - This property enables the network to efficiently sort bitonic sequences.
2. **Reversal Property:**
    - The Bitonic Sorting Network also possesses the reversal property.
    - If the inputs to a comparator are reversed, i.e., (x, y) becomes (y, x), the output will be the reverse of the original output.
    - This property is crucial for constructing bitonic sequences.

Steps in Bitonic Sorting Network:

1. **Constructing a Bitonic Sequence:**
    - To construct a bitonic sequence, the input sequence is repeatedly halved and sorted recursively in two directions.
    - The sequences are then merged to form a bitonic sequence.
2. **Constructing the Bitonic Sorting Network:**
    - The Bitonic Sorting Network can be constructed using a divide-and-conquer approach.
    - The network is recursively built by combining smaller bitonic sequences and adding comparators at each stage.

3. **Sorting with the Bitonic Sorting Network:**
    - Once the network is constructed, the input elements are fed into the network.
    - The comparators in each stage compare and swap the elements, resulting in a sorted sequence at the output.

Time Complexity:

- The Bitonic Sorting Network has a time complexity of O(log^2 n), where n is the number of elements.
- This complexity makes it suitable for parallel processing and efficient sorting of large datasets.

Conclusion:

- The Bitonic Sorting Network is a parallel sorting algorithm that efficiently sorts bitonic sequences.
- By exploiting the properties of bitonicity and reversal, the network constructs a sorting network that operates in O(log^2 n) time complexity.
- Understanding the concepts and steps involved in the Bitonic Sorting Network provides valuable insights into parallel sorting algorithms and their applications in high-performance computing.