byte^XL **Name of Faculty: *Suraj Mourya***

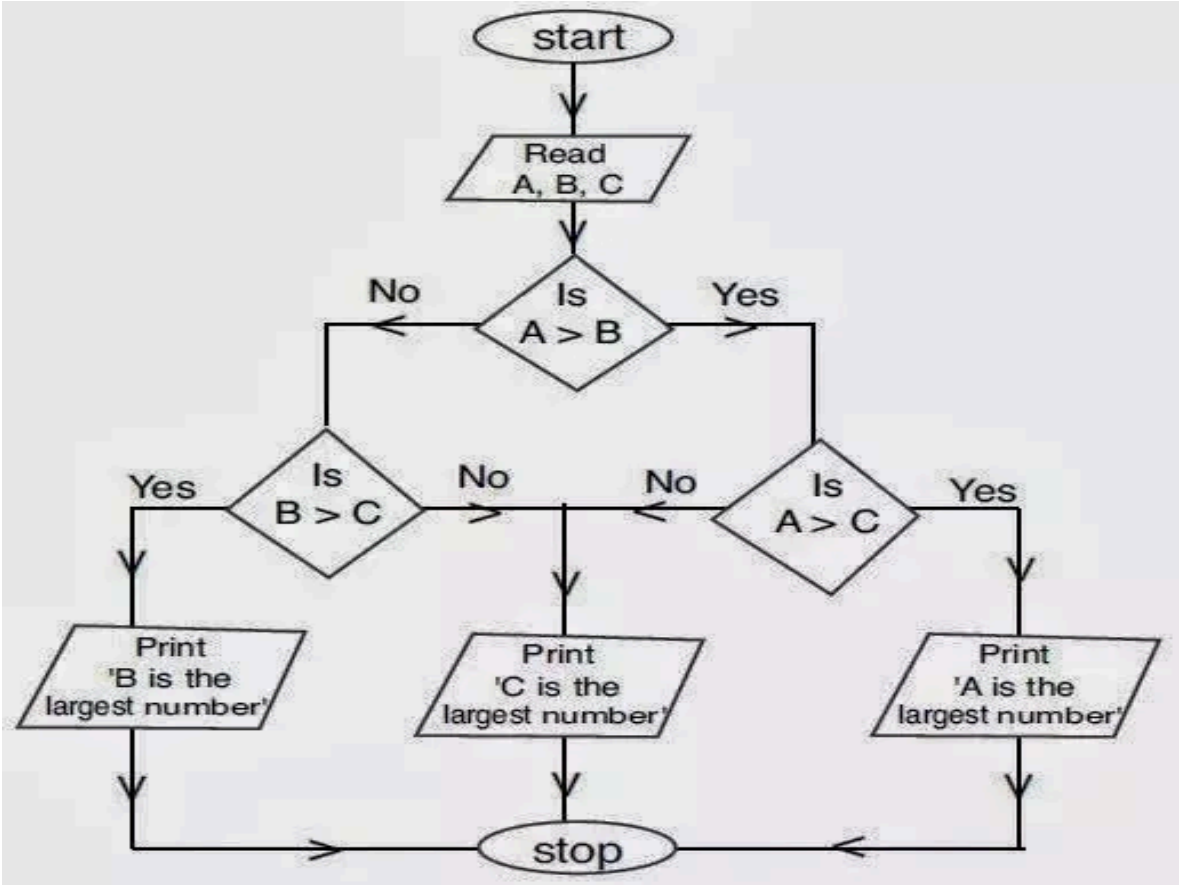
FACULTY OF ENGG. & TECH. - **PIET**	Sem: **5th**
LESSON PLAN Academic Year: **2024**	Name of Department: **CSE**

Module 1

Introduction and Analysis of Algorithms:
Algorithm: Definition, Properties, Types of Algorithms, Writing an AlgoritAlgorithm Analysis: Parameters, Design Techniques of Algorithms
Asymptotic Analysis: Big Oh, Big Omega & Big Theta Notations, Lower Bound, Upper Bound and Tight Bound, Best Case, Worst Case, Average Case
Analyzing control statement, Loop invariant and the correctness of the algorithm, Recurrences- substitution method, recursion tree method, master method. Sorting Techniques with analysis: Bubble Sort, Selection Sort, Insertion sort.

Find the Largest Number Among Three Numbers ?

1. Start
2. Read the three numbers to be compared, as A, B and C.
3. Check if A is greater than B.
    - 3.1 If true, then check if A is greater than C.
        - 3.1.1 If true, print 'A' as the greatest number.
        - 3.1.2 If false, print 'C' as the greatest number.
    - 3.2 If false, then check if B is greater than C.
        - 3.2.1 If true, print 'B' as the greatest number.
        - 3.2.2 If false, print 'C' as the greatest number.
4. End




```
# Python program to find the largest number among three numbers

# Taking input from the user
A = int(input("Enter the number A: "))
B = int(input("Enter the number B: "))
C = int(input("Enter the number C: "))

# Finding the largest number
if A >= B and A >= C:
    print(f"{A} is the largest number.")
elif B >= A and B >= C:
    print(f"{B} is the largest number.")
else:
    print(f"{C} is the largest number.")
```

Introduction to Algorithm

- In mathematics and computer science, an algorithm is a finite sequence of well-defined, computer-implementable instructions, typically to solve a class of problems or to perform a computation. A step by step Procedure.
- Or

  ” A set of finite rules or instructions to be followed in calculations or other problem-solving operations ”

Use of the Algorithms:

Algorithms play a crucial role in various fields and have many applications. Some of the key areas where algorithms are used include:

- **Computer Science:** Algorithms form the basis of computer programming and are used to solve problems ranging from simple sorting and searching to complex tasks such as artificial intelligence and machine learning.
- **Mathematics:** Algorithms are used to solve mathematical problems, such as finding the optimal solution to a system of linear equations or finding the shortest path in a graph.
- **Operations Research:** Algorithms are used to optimize and make decisions in fields such as transportation, logistics, and resource allocation.
- **Artificial Intelligence:** Algorithms are the foundation of artificial intelligence and machine learning, and are used to develop intelligent systems that can perform tasks such as image recognition, natural language processing, and decision-making.
- **Data Science:** Algorithms are used to analyze, process, and extract insights from large amounts of data in fields such as marketing, finance, and healthcare.

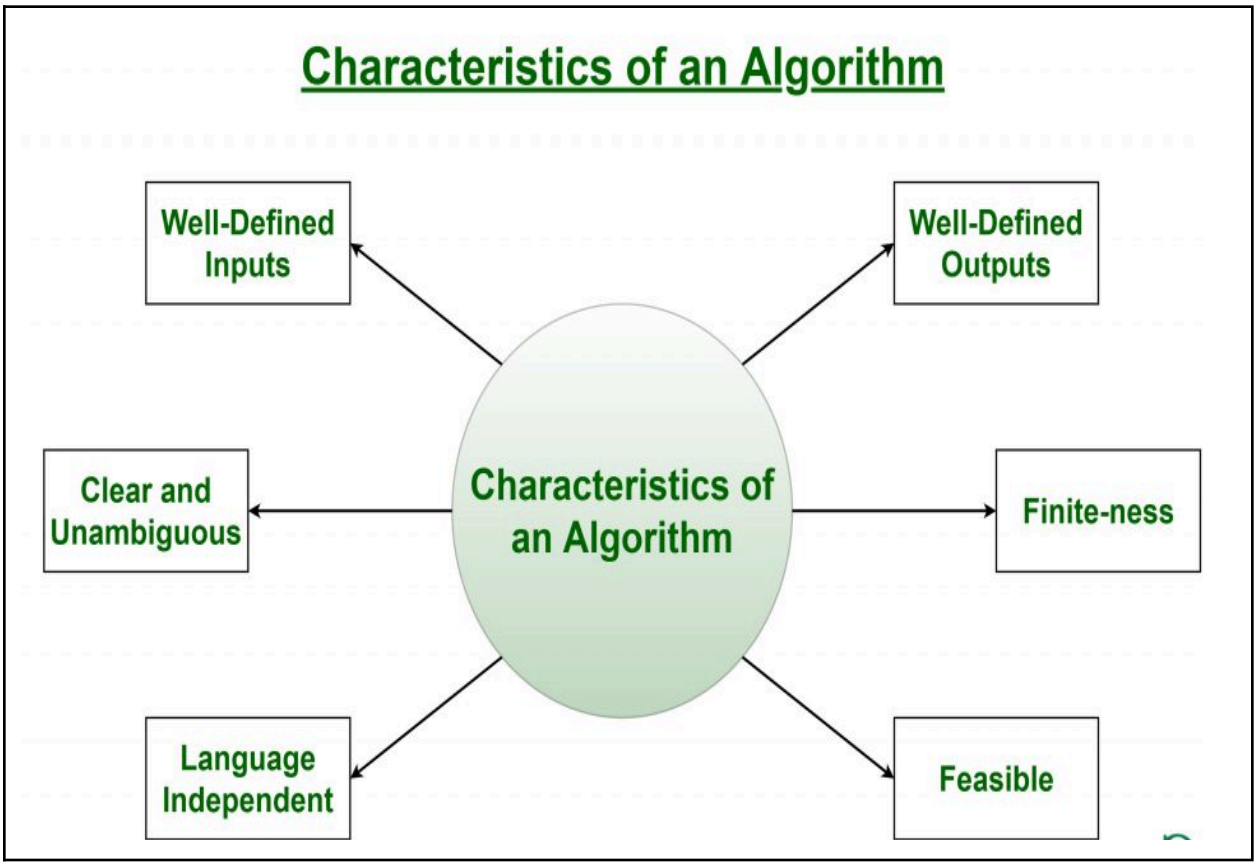
These are just a few examples of the many applications of algorithms. The use of algorithms is continually expanding as new technologies and fields emerge, making it a vital component of modern society.

What are the Characteristics of an Algorithm?

- **Clear and Unambiguous:** The algorithm should be unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.

Beta, go & Burn the book, It’s not useful for you

- **Finite-ness:** The algorithm must be finite, i.e. it should terminate after a finite time.
- **Feasible:** The algorithm must be simple, generic, and practical, such that it can be executed with the available resources. It must not contain some future technology or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.



- **Definiteness:** All instructions in an algorithm must be unambiguous, precise, and easy to interpret. By referring to any of the instructions in an algorithm one can clearly understand what is to be done. Every fundamental operator in instruction must be defined without any ambiguity.
- **Finiteness:** An algorithm must terminate after a finite number of steps in all test cases. Every instruction which contains a fundamental operator must be terminated within a finite amount of time. Infinite loops or recursive functions without base conditions do not possess finiteness.
- **Effectiveness:** An algorithm must be developed by using very basic, simple, and feasible operations so that one can trace it out by using just paper and pencil.

Properties of Algorithm:

1. It should terminate after a finite time.
2. It should produce at least one output.
3. It should take zero or more input.
4. It should be deterministic means giving the same output for the same input case.
5. Every step in the algorithm must be effective i.e. every step should do some work.

Difference between Algorithm and Program

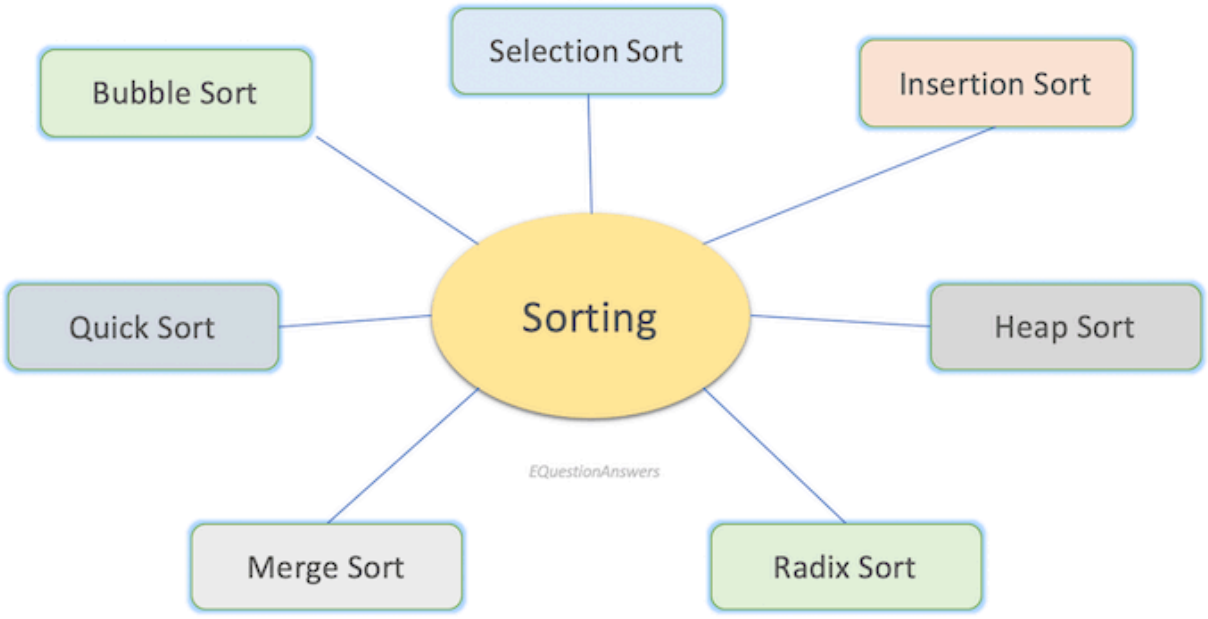
Algorithm	Program
Algorithms are written at design phase	Programs are written at Implementation Stage
Algorithms are programming syntax independent	Programs are programming syntax dependent
Algorithms are not dependant on operating system architecture and hardware	Programs are dependent on operating system architecture and hardware
Algorithms are analysed on efficiency in terms of completion time and the space used.	We just test the program to see if it will scale in production

Problem Solving Cycle

- Problem Definition: Understand Problem
- Constraints & Conditions: Understand constraints if any
- Design Strategies (Algorithmic Strategy)
- Express & Develop the algo
- Validation (Dry run)
- Analysis (Space and Time analysis)
- Coding
- Testing & Debugging
- Installation
- Maintenance

Need for Analysis

We do analysis of algorithm to do a performance comparison between different algorithm to figure out which one is best possible option.



What parameters can be considered for comparison between cars?

For Algo?

Following are the parameters which can be considered while analysis of an algorithm

- **Time**
- **Space**
- Bandwidth
- Register
- Battery power

Out of all **time** is the most important Criteria for analysis of algorithm

to analyze time?

Types of Analysis

• **Experimental or** • **Apostrium or** • **Relative analysis**	• **Apriori Analysis or** • **Independent analysis or** • **Absolute analysis**
	

Experimental or Apostrium or relative analysis : Means analysis of algorithm after it is converted to code. Implement both the algorithms and run the two programs on your computer for different inputs and see which one takes less time.

- **Advantage:**

  Exact values no rough

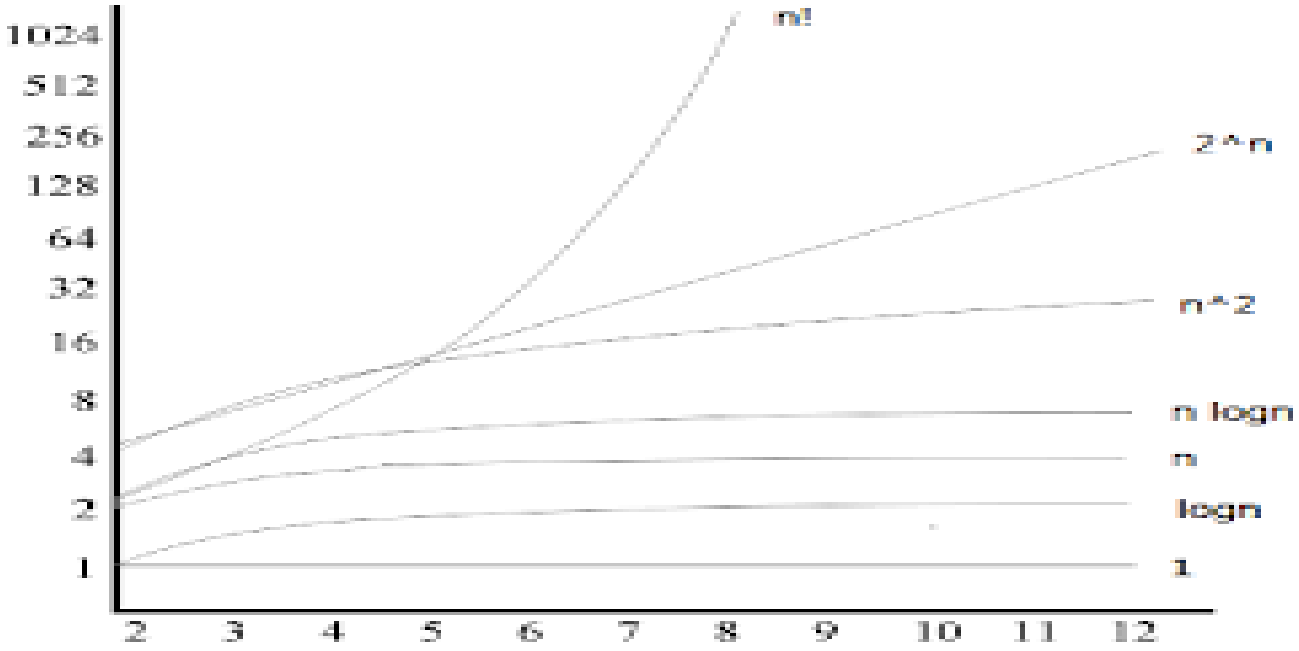
- **Disadvantage:**

  Final result instead of depending only algorithm depends on many other factors like background software & hardware, programming language, even the temperature of the room.

Apriori Analysis or Independent analysis or Absolute analysis:

- We do analysis using asymptotic notations and mathematical tools of only algorithms, i.e. before converting it into a program of a particular programming language.
- It is a determination of the order of magnitude of a statement.

In Asymptotic Analysis, we evaluate the performance of an algorithm in terms of input size (we don't measure the actual running time). We calculate, how does the time (or space) taken by an algorithm increases with the input size.



- Asymptotic Analysis is not perfect, but that's the best way available for analyzing algorithms.
- It might be possible that those large inputs are never given to your software and an algorithm which is asymptotically slower, always performs better for your particular situation. So, you may end up choosing an algorithm that is Asymptotically slower but faster for your software.

• **Advantage:** Uniform result depends only on algorithm.

• **Disadvantage:** Estimated or approximate value no accurate and precise value.

Types of Asymptotic Notations in Complexity Analysis of Algorithms

Asymptotic Notations:

- Asymptotic Notations are mathematical tools used to analyze the performance of algorithms by understanding how their efficiency changes as the input size grows.
- These notations provide a concise way to express the behavior of an algorithm's time or space complexity as the input size approaches infinity.
- Rather than comparing algorithms directly, asymptotic analysis focuses on understanding the relative growth rates of algorithms' complexities.
- It enables comparisons of algorithms' efficiency by abstracting away machine-specific constants and implementation details, focusing instead on fundamental trends.
- Asymptotic analysis allows for the comparison of algorithms' space and time complexities by examining their performance characteristics as the input size varies.
- By using asymptotic notations, such as Big O, Big Omega, and Big Theta, we can categorize algorithms based on their worst-case, best-case, or average-case time or space complexities, providing valuable insights into their efficiency.

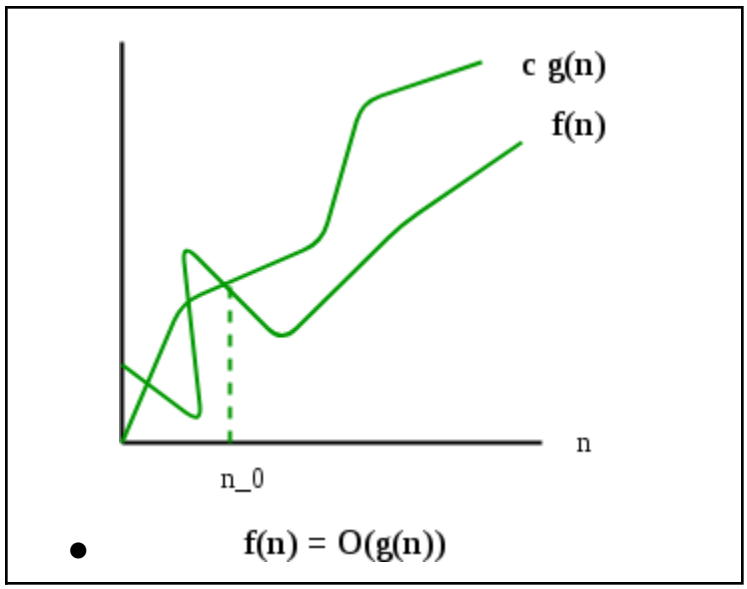
There are mainly three asymptotic notations:

1. ***Big-O Notation (O-notation)***
2. ***Omega Notation (Ω-notation)***
3. ***Theta Notation (Θ-notation)***

1. Big-O Notation (O-notation):

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm.

- It is the most widely used notation for Asymptotic analysis.
- It specifies the upper bound of a function.
- The maximum time required by an algorithm or the worst-case time complexity.
- It returns the highest possible output value(big-O) for a given input.
- Big-Oh(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.



If f(n) describes the running time of an algorithm, f(n) is O(g(n)) if there exist a positive constant C and n0 such that, f(n) ≤ c*g(n) for all n ≥ n0

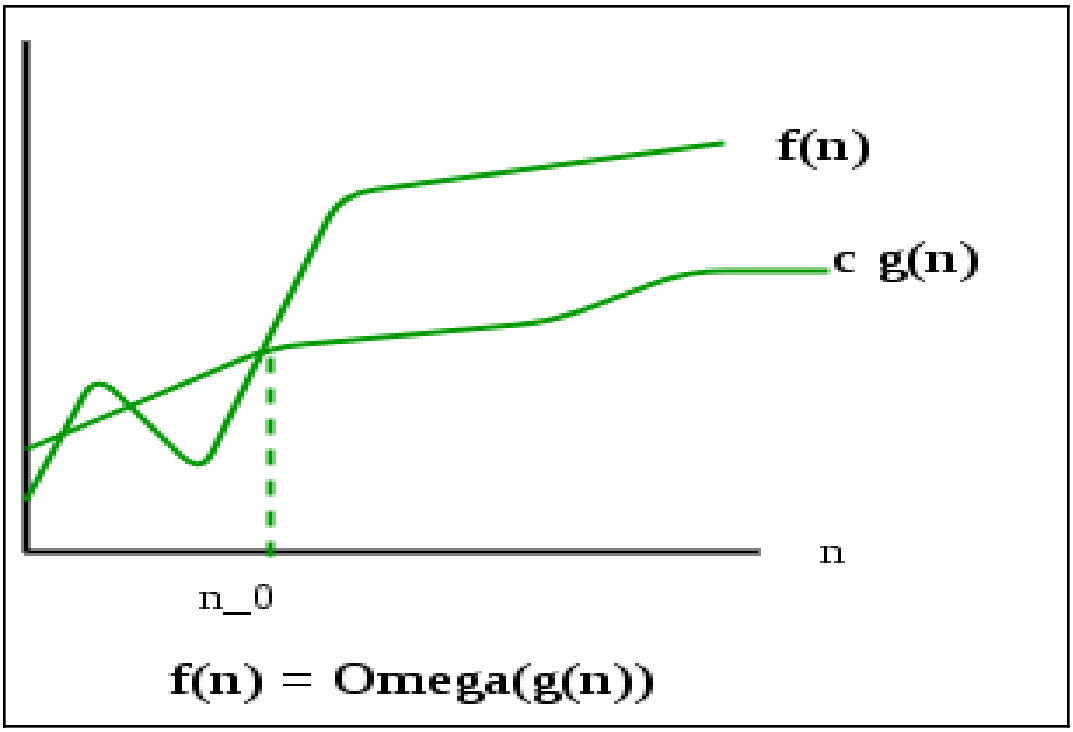
It returns the highest possible output value (big-O)for a given input.

The execution time serves as an upper bound on the algorithm's time complexity.

2. 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.
- The execution time serves as a lower bound on the algorithm's time complexity.
- It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.

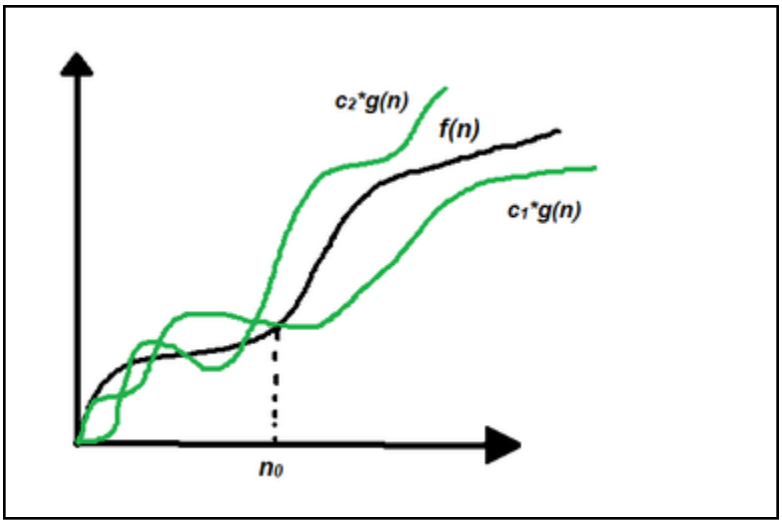
Let g and f be the function from the set of natural numbers to itself. The function f is said to be Ω(g), if there is a constant c > 0 and a natural number n0 such that c*g(n) ≤ f(n) for all n ≥ n0



3. 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.
- Theta (Average Case) You add the running times for each possible input combination and take the average in the average case.

Let g and f be the function from the set of natural numbers to itself. The function f is said to be Θ(g), if there are constants c1, c2 > 0 and a natural number n0 such that c1* g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n0



Example:

For any algorithm, the given function expression, F(n) = 2n+3, then How will you show in all three notations?

For O(big oh):

f(n) ≤ c*g(n)

2n + 3 <= 2n+3n

2n + 3 <= 5n (true for all n>=1)

For **Ω** (big omega):

f(n) >= c*g(n)

2n + 3 >= 1*n

2n + 3 >= n

For **Θ** (theta):

c1* g(n) ≤ f(n) ≤ c2 * g(n)

1*n <= 2n+3 <= 5*n

n <= 2n+3 <= 5n

Properties of Asymptotic Notations:

1. General Properties:

If f(n) is O(g(n)) then a*f(n) is also O(g(n)), where a is a constant.

Example:

f(n) = $2n^2+5$ is $O(n^2)$
then, 7*f(n) = $7(2n^2+5) = 14n^2+35$ is also $O(n^2)$.

Similarly, this property satisfies both Θ and Ω notation.

We can say,

If f(n) is Θ(g(n)) then a*f(n) is also Θ(g(n)), where a is a constant.
If f(n) is Ω (g(n)) then a*f(n) is also Ω (g(n)), where a is a constant.

2. Transitive Properties:

If f(n) is O(g(n)) and g(n) is O(h(n)) then f(n) = O(h(n)).

Example:

If f(n) = n, g(n) = n^2 and h(n)=n^3
n is $O(n^2)$ and n^2 is $O(n^3)$ then, n is $O(n^3)$

Similarly, this property satisfies both Θ and Ω notation.

We can say,

If f(n) is Θ(g(n)) and g(n) is Θ(h(n)) then f(n) = Θ(h(n)) .
If f(n) is Ω (g(n)) and g(n) is Ω (h(n)) then f(n) = Ω (h(n))

3. Reflexive Properties:

Reflexive properties are always easy to understand after transitive.
If f(n) is given then f(n) is O(f(n)). Since MAXIMUM VALUE OF f(n) will be f(n) ITSELF!
Hence x = f(n) and y = O(f(n) tie themselves in reflexive relation always.

Example:

f(n) = n^2 ; $O(n^2)$ i.e O(f(n))
Similarly, this property satisfies both Θ and Ω notation.

We can say that,

If f(n) is given then f(n) is Θ(f(n)).
If f(n) is given then f(n) is Ω (f(n)).

4. Symmetric Properties:

If f(n) is Θ(g(n)) then g(n) is Θ(f(n)).

Example:

If$(n) = n^2$ and $g(n) = n^2$
then, $f(n) = \Theta(n^2)$ and $g(n) = \Theta(n^2)$

This property only satisfies for Θ notation.

There are two more notations called little o and little omega. Little o provides a strict upper bound (equality condition is removed from Big O) and little omega provides strict lower bound (equality condition removed from big omega)

Which Complexity analysis is generally used?

Below is the ranked mention of complexity analysis notation based on popularity:

1. Worst Case Analysis:

Most of the time, we do worst-case analyses to analyze algorithms. In the worst analysis, we guarantee an upper bound on the running time of an algorithm which is good information.

2. Average Case Analysis

The average case analysis is not easy to do in most practical cases and it is rarely done. In the average case analysis, we must know (or predict) the mathematical distribution of all possible inputs.

3. Best Case Analysis

The Best Case analysis is bogus. Guaranteeing a lower bound on an algorithm doesn't provide any information as in the worst case, an algorithm may take years to run.

Interesting information about asymptotic notations:

A) For some algorithms, all the cases (worst, best, average) are asymptotically the same. i.e., there are no worst and best cases.

Example: Merge Sort does ?(n log(n)) operations in all cases.

B) Where as most of the other sorting algorithms have worst and best cases.

Example 1: In the typical implementation of Quick Sort (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occurs when the pivot elements always divide the array into two halves.

Example 2: For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the

same order as output.

Important points:

1. The worst case analysis of an algorithm provides an upper bound on the running time of the algorithm for any input size.
2. The average case analysis of an algorithm provides an estimate of the running time of the algorithm for a random input.
3. The best case analysis of an algorithm provides a lower bound on the running time of the algorithm for any input size.
4. The big O notation is commonly used to express the worst case running time of an algorithm.
5. Different algorithms may have different best, average, and worst case running times.

Analyzing control statement, Loop invariant and the correctness of the algorithm

Calculate Time Complexity:-

(i)
```
for (i=0; i<n; i++)
    stmnt;
```
→ O(n)

(ii)
```
for (i=0; i<n; i++)
    for (j=0; j<i; j++)
        stmnt;
```
⇒ $O(n^2)$

i	j	no. of time
0	0×	0
1	0✓ 1×	1
2	0✓ 1✓ 2×	2
3	0✓ 1✓ 2✓ 3×	3
.	.	.
.	.	.
n	—	n

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

$= \frac{n^2+1}{2}$

(iii)
```
p=0
For(i=1; p<=n; i++)
{ p=p+i;
}
```

i	p
1	0+1
2	0+1+2
3	1+2+3
4	1+2+3+4
.	.
.	.
k	1+2+3+4+---+k

Assume $p > n$

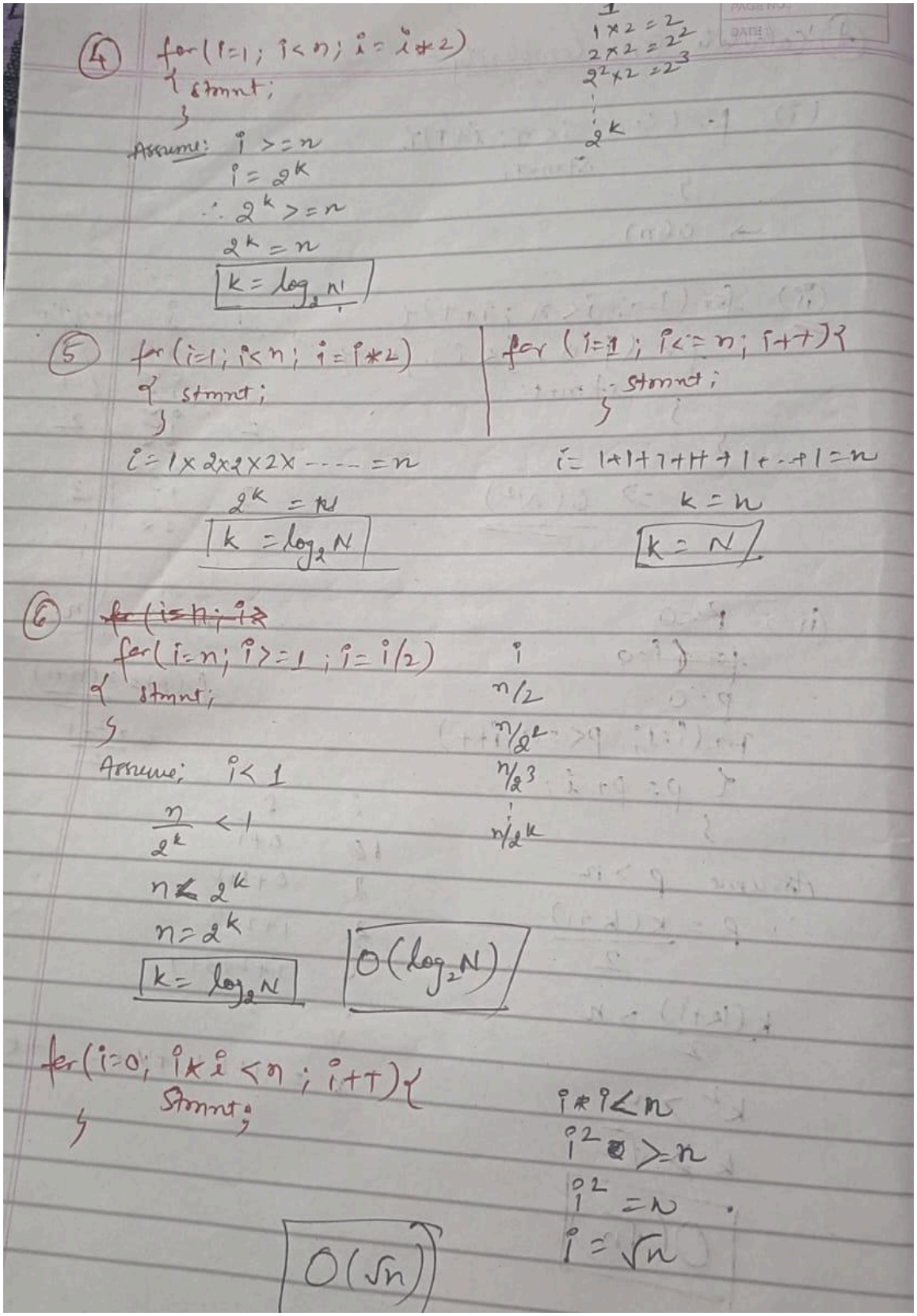
$\therefore p = \frac{k(k+1)}{2}$

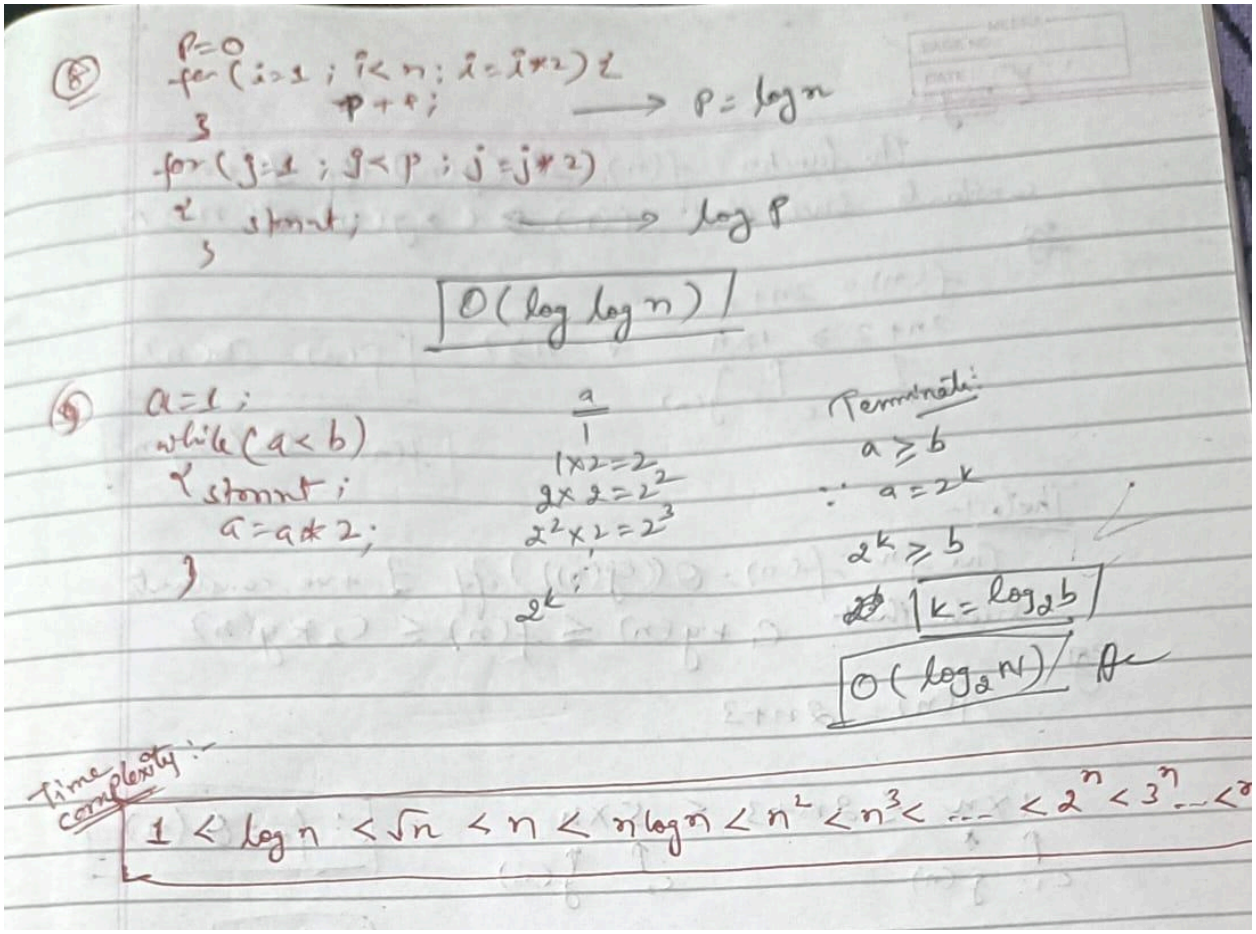
$\frac{k(k+1)}{2} > n$

$k^2 > n$

$k \geq \sqrt{n}$

$\boxed{O(\sqrt{n})}$


④ for(i=1; i<n; i=i*2)
{ stmnt;
}
Assume: i >= n
i = 2^k
∴ 2^k >= n
2^k = n
k = log2 n
1
1×2 = 2
2×2 = 2^2
2^2×2 = 2^3
2^k
⑤ for(i=1; i<n; i=i*2)
{ stmnt;
}
i = 1×2×2×2×---- = n
2^k = n
k = log2 N
for(i=1; i<=n; i++){
stmnt;
}
i = 1+1+1+1+1+---+1 = n
k = n
k = N
⑥ for(i=n; i>=1; i=i/2)
{ stmnt;
}
Assume; i < 1
n/2^k < 1
n < 2^k
n = 2^k
k = log2 N
O(log2 N)
i
n/2
n/2^2
n/2^3
n/2^k
for(i=0; i*i<n; i++){
Stmnt;
}
i*i < n
i^2 >= n
i^2 = n
i = √n
O(√n)



Recurrence Relation

A recurrence is an equation or inequality that describes a function in terms of its values on smaller inputs. To solve a Recurrence Relation means to obtain a function defined on the natural numbers that satisfy the recurrence.

For Example, the Worst Case Running Time T(n) of the MERGE SORT Procedures is described by the recurrence.

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

$2T\left(\frac{n}{2}\right) + \theta(n)$ if $n>1$

There are three methods for solving Recurrence:

1. Substitution Method
2. Recursion Tree Method
3. Master Method

1. Substitution Method:

The Substitution Method Consists of two main steps:

1. Guess the Solution.
2. Use the mathematical induction to find the boundary condition and show that the guess is correct.

For Example1 Solve the equation by Substitution Method.

(1) Substitution method:-

$$T(n) = \begin{cases} T(n/2) + c & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

$T(n) = T(n/2) + c$ ——— (i)

$T(n/2) = T(n/2^2) + c$ ——— (ii)

$T(n/4) = T(n/2^3) + c$ ——— (iii)

then,

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

$= T(n/4) + c + c$

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

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

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

So,

.... k times:

$= T\left(\frac{n}{2^k}\right) + kc$

∵ we want answer in "n"

so, $n = 2^k \longrightarrow k = \log_2 n$

then,

$T(1) + kc$

$= 1 + kc$

$= 1 + \log_2 n$

hence, $\boxed{O(\log_2 n)}$ Ans.

(2) $T(n) = \begin{cases} 3T(n-1), & \text{if } n > 0 \\ 1, & \text{otherwise} \end{cases}$

$T(n) = 3T(n-1)$ — (I)

$T(n-1) = 3T(n-1-1) = 3T(n-2)$ — (II)

$T(n-2) = 3T(n-3)$ — (III)

Q1.

$$T(n) = 3T(n-1)$$
$$= 3[3T(n-2)]$$
$$= 3^2[T(n-2)]$$
$$= 3^2[3T(n-3)]$$
$$= 3^3[T(n-3)]$$

⋮ k times

$$= 3^k[T(n-k)]$$

and, here, $k = n$

$$= 3^k[T(n-n)]$$
$$= 3^k[T(0)]$$
$$= 3^k \cdot 1$$
$$= 3^n \Rightarrow \boxed{O(3^n)}$$ Ans

Q2.

$$T(n) = \begin{cases} 2T(n-1)-1, & \text{if } n>0 \\ 1 & \text{otherwise} \end{cases}$$

$T(n) = 2T(n-1) - 1$ — (I)

$T(n-1) = 2T(n-2) - 1$ — (II)

$T(n-2) = 2T(n-3) - 1$ — (III)

and,

$$T(n) = 2T(n-1) - 1$$
$$= 2[2T(n-2) - 1] - 1$$
$$= 2[2T(n-2) - 1] - 1$$
$$= 2^2[T(n-2)] - 2 - 1$$

$a(r^n-1)/(r-1)$

$= 2^2 [2T(n-3)-1] - 2 - 1$

$= 2^3 [T(n-3)] - 2^2 - 2 - 1$

⋮

k times:

$= 2^k [T(n-k)] - 2^{k-1} - 2^{k-2} - 2^{k-3} \cdots - 2 - 1$

$= 2^k [T(n-k)] - (2^{k-1} + 2^{k-2} + 2^{k-3} + \cdots + 2 + 2^0)$

take $k = n$

hence

$= 2^n [T(n-n)] - (2^{n-1} + 2^{n-2} + 2^{n-3} + \cdots + 2 + 2^0)$

~~$= 2^n - [2^{n-1}(2^{n-1}-1)/(2-1)-1]$~~

formula:

$[2^0 + 2^1 + 2^2 + 2^3 + \cdots + 2^{n-3} + 2^{n-2} + 2^{n-1}]$

$= \text{Sum} = \frac{a(r^n-1)}{r-1} = \frac{2^0(2^n-1)}{2-1} = 2^n - 1$

and,

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

$= \boxed{O(1)}$

④ $T(n) = \begin{cases} 1 & \text{if } n = 1 \\ 2T(\frac{n}{2}) + n & \text{otherwise} \end{cases}$

Sol^n:

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

$T(n/2) = 2T(\frac{n}{2^2}) + n$

$T(n/4) = 2T(n/8) + n$

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

$= 2[T(n/2)] + n$

$= 2[2T(n/4) + \frac{n}{2}] + n$

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

and,

$= 2^2 (2T(n/8) + \frac{n}{4}) + 2n$

$= 2^3 T(n/8) + 3n$

k times

$= 2^k T(n/2^k) + 8kn$

and

$n/2^k = 1 \Rightarrow n = 2^k$

$\Rightarrow \boxed{k = \log_2 n}$

and,

$= 2^k T(1) + kn$

$\Rightarrow n \cdot 1 + \log_2 n \times n$

$= n + n\log n \Rightarrow \boxed{O(n\log n)}$ Ans

Recursion Tree Method

What is a Recursion Tree?

- A recursion tree is a graphical representation that illustrates the execution flow of a recursive function.
- It provides a visual breakdown of recursive calls, showcasing the progression of the algorithm as it branches out and eventually reaches a base case.
- The tree structure helps in analyzing the time complexity and understanding the recursive process involved.

How to Use a Recursion Tree to Solve Recurrence Relations?

- The cost of the sub problem in the recursion tree technique is the amount of time needed to solve the sub problem. Therefore, if you notice the phrase "cost" linked with the recursion tree, it simply refers to the amount of time needed to solve a certain sub problem.

Let's understand all of these steps with a few examples.

Example

Consider the recurrence relation,

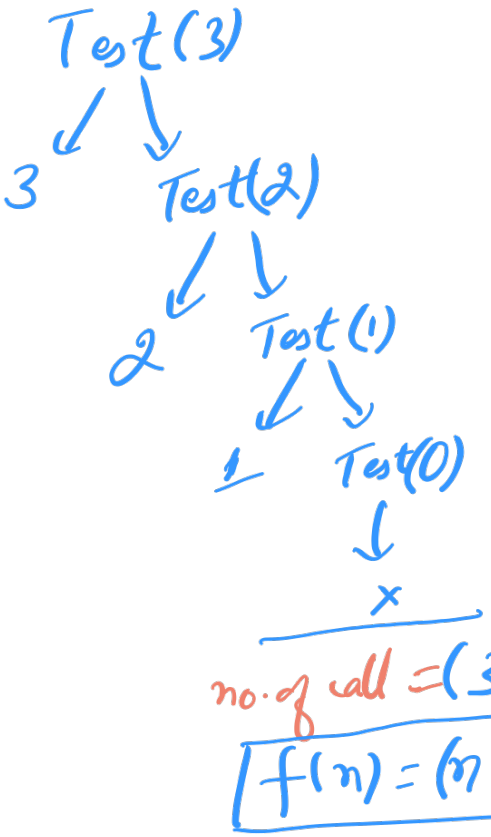
$T(n) = 2T(n/2) + K$

Solution

A problem size n is divided into two sub-problems each of size n/2. The cost of combining the solutions to these sub-problems is K.

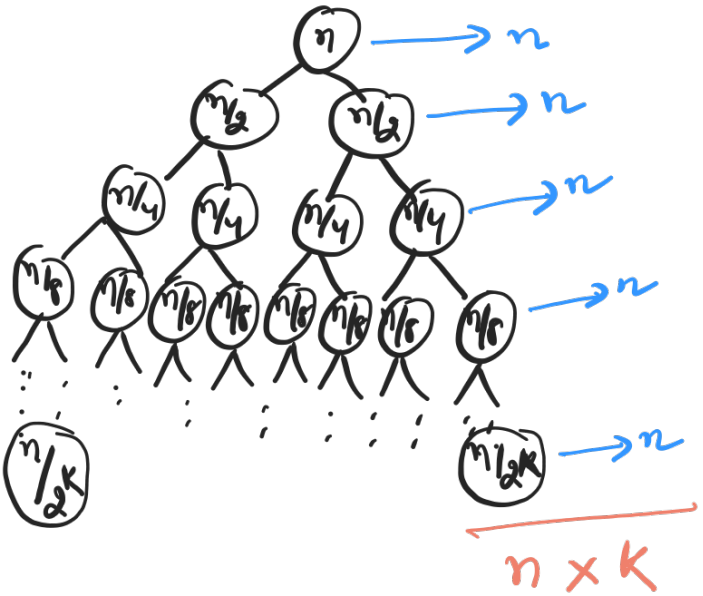
Each problem size of n/2 is divided into two sub-problems each of size n/4 and so on.

At the last level, the sub-problem size will be reduced to 1. In other words, we finally hit the base case.




```
void Test(int n)
{
  if (n>0)
  {
    printf("%d", n);
    Test(n-1);
  }
}
```

Question: $T(n) = 2T(n/2) + n$

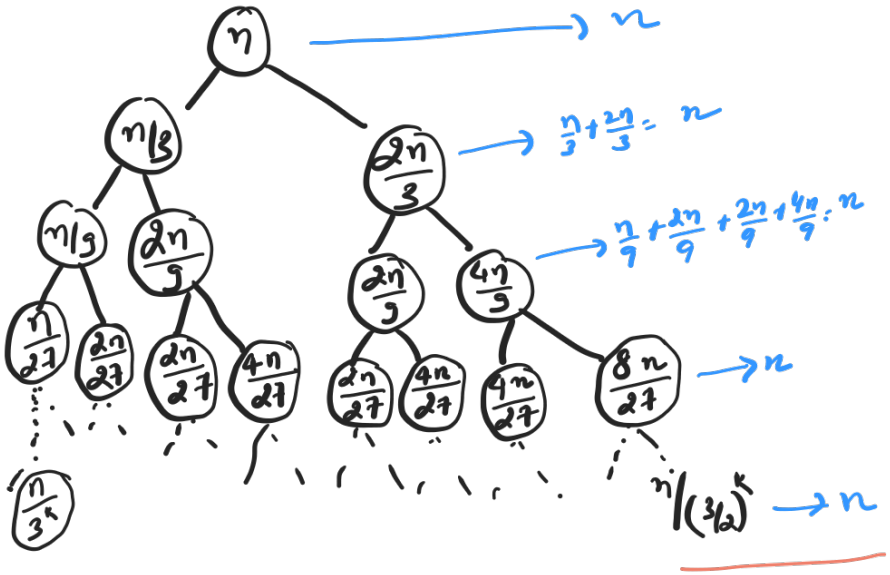


Let $\frac{n}{2^k} = 1 \Rightarrow n = 2^k$

$\Rightarrow k = \log_2 n$

$\Rightarrow n \times \log_2 n \Rightarrow O(n \log n)$

Question 2:- $T(n) = T(n/3) + T(2n/3) + n$



$n \times k$

Let
$\frac{n}{3^k} = 1$
$n = 3^k$
$k = \log_3 n$
and,
$\frac{n}{(3/2)^k} = 1$
$\Rightarrow n = (\frac{3}{2})^k$
$\Rightarrow k = \log_{3/2} n$

$\Rightarrow n \times \log_{3/2} n \Rightarrow O(n \log_{3/2} n)$

MASTER THEOREM

$T(n) = a\,T(n/b) + f(n)$

$a \geq 1$
$b > 1$

$f(n) = \theta(n^k \log^p n)$

we need to find:
(i) $\log_b a$
(ii) k

Case 1 : if $\log_b a > k$ then $\theta(n^{\log_b a})$

Case 2: if $\log_b a = k$

if $p > -1 \longrightarrow \Theta(n^k \log^{p+1} n)$

if $p = -1 \longrightarrow \Theta(n^k \log\log n)$

if $p < -1 \longrightarrow \Theta(n^k)$

Case 3: if $\log_b a < k$

if $p \geq 0 \Rightarrow \Theta(n^k \log^p n)$

if $p < 0 \Rightarrow \Theta(n^k)$

① $T(n) = 2T(n/2) + 1$

Soln: $a = 2$, $b = 2$

$f(n) = \Theta(1) = \Theta(n^0 \log^0 n)$

$\log_2^2 = 1 > k = 0$

Case - I

hence: $\Theta(n)$

② $T(n) = 4T(n/2) + n$

Soln: $\log_2^4 = 2 > k = 1$

Case - I

hence, $\Theta(n^2)$

③ $T(n) = 8T(n/2) + n$

Soln: $\log_2^8 = 3 > k = 1$

Case - I: $\Theta(n^3)$

④ $T(n) = 2T(n/2) + n$

Soln: $(\log_2^2 = 1) = (k = 1)$ (case II)

$\Theta(n \log n)$

⑤ $T(n) = 4T(n/2) + n^2 \log^2 n$

Soln: $(\log_2 4 = 2) = (k = 2)$ (case II)

$\Theta(n^2 \log^3 n)$

⑥ $T(n) = 2T(n/2) + \frac{n}{\log n}$, $p = -1$

Soln: $(\log_2^2 = 1) = (k = 1)$ (case - II)

$O(n \log\log n)$

(8) $T(n) = 2T(n/2) + \frac{n}{\log^2 n}$

Soln: $(\log_2^2 = 1) = (k = 1)$ $p = -2$

(Case-II)

$\Theta(n)$

(9) $T(n) = 2T(\frac{n}{2}) + n^2 \log n$

Soln: $\log_2^2 = 1 < k = 2$ (Case-III)

$\Theta(n^2 \log n)$

(7) $T(n) = 4T(n/2) + n^3$

Soln: $(\log_2^2 = 2) < (k = 3)$

(case-III)

$\Theta(n^3)$

Exercise Question:- (Solve by using Master theorem)

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

(2) $T(n) = 4T(n/2) + 1$

(3) $T(n) = 16T(n/2) + n^2$

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

(5) $T(n) = 2T(n/2) + n^2 \log n$

(6) $T(n) = 2T(\frac{n}{2}) + \frac{n^2}{\log n}$

(7) $T(n) = T(n/2) + 1$

(8) $T(n) = 2T(\frac{n}{2}) + n \log n$

(9) $T(n) = 4T(n/2) + n^2$

(10) $T(n) = 4T(n/2) + (n \log n)^2$

Answer Key ⇒

(1) $\Theta(n)$
(2) $\Theta(n^2)$
(3) $\Theta(n^4)$
(4) $\Theta(n)$
(5) $\Theta(n^2 \log n)$
(6) $\Theta(n^2)$
(7) $\Theta(\log n)$
(8) $\Theta(n \log^2 n)$
(9) $\Theta(n^2 \log n)$
(10) $\Theta(n^2 \log^3 n)$

Limitation of Master theorem :-

(1) 'a' must be constant

$$T(n) = \eta T\left(\frac{n}{2}\right) + n^2$$

↑ a is not constant.

(2) 'a' must be greater than or equal to 1 ($a \geq 1$)

$$T(n) = \frac{1}{2}\left(\frac{n}{2}\right) + n^2$$

(3) 'b' must be greater than 1. ($b > 1$)

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

(4) f(n) should be positive.

$$T(n) = 2T\left(\frac{n}{2}\right) - n^2$$

↑ function is negative

*Imp

(5) f(n) must be polynomially greater than $\log_b a$

(a) $T(n) = 2T\left(\frac{n}{2}\right) + n \log n$

$n \log_b a = n \log_2 2 = n$

$f(n) > n$

$n \log n > n$ ⇐ Since f(n) is logarithmic greater not polynomially

(b) $T(n) = 2T(n/2) + n^2 \log n$

$\Rightarrow n^2 \log n > n$ ⇐ here it is valid

This is polynomially greater.

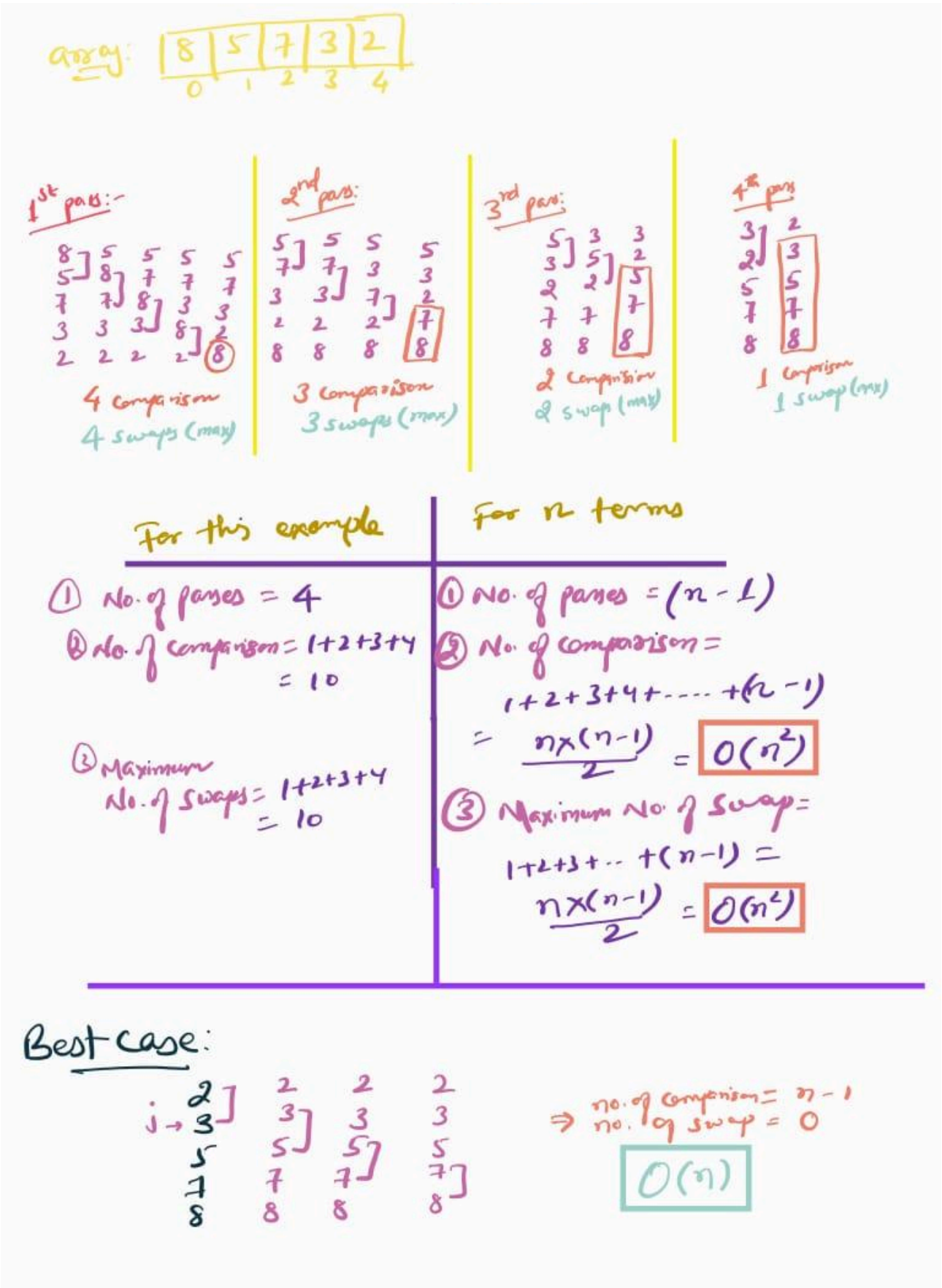
Sorting Techniques with analysis: Bubble Sort, Selection Sort, Insertion sort

Bubble sort Algorithm

- Bubble sort works on the repeatedly swapping of adjacent elements until they are not in the intended order.
- It is called bubble sort because the movement of array elements is just like the movement of air bubbles in the water.
- Bubbles in water rise up to the surface; similarly, the array elements in bubble sort move to the end in each iteration.
- Although it is simple to use, it is primarily used as an educational tool because the performance of bubble sort is poor in the real world.
- It is not suitable for large data sets. The average and worst-case complexity of Bubble sort is O(n2), where n is a number of items.
- Bubble short is majorly used where -
    - complexity does not matter
    - simple and shortcode is preferred

Algorithm

```
begin BubbleSort(arr)
  for all array elements
    if arr[i] > arr[i+1]
      swap(arr[i], arr[i+1])
    end if
  end for
  return arr
end BubbleSort
```


array: 8 5 7 3 2
0 1 2 3 4
1st pass:-
4 comparison
4 swaps (max)
2nd pass:
3 comparison
3 swaps (max)
3rd pass:
2 comparison
2 swap (max)
4th pass
1 comparison
1 swap (max)
For this example
For n terms
① No. of passes = 4
② No. of comparison = 1+2+3+4 = 10
③ Maximum No. of swaps = 1+2+3+4 = 10
① No. of passes = (n-1)
② No. of comparison = 1+2+3+4+.... +(n-1) = n×(n-1)/2 = O(n²)
③ Maximum No. of swap = 1+2+3+.. +(n-1) = n×(n-1)/2 = O(n²)
Best case:
i → 2 3 5 7 8
⇒ no. of comparison = n-1
no. of swap = 0
O(n)

Bubble sort complexity

Now, let's see the time complexity of bubble sort in the best case, average case, and worst case. We will also see the space complexity of bubble sort.

1. Time Complexity

Case	Time Complexity
Best Case	**O(n)**
Average Case	**$O(n^2)$**
Worst Case	**$O(n^2)$**

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of bubble sort is O(n).
- **Average Case Complexity** - It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of bubble sort is $O(n^2)$.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of bubble sort is $O(n^2)$.

2. Space Complexity

Space Complexity	**O(1)**
Stable	**YES**

- The space complexity of bubble sort is O(1). It is because, in bubble sort, an extra variable is required for swapping.

Selection Sort Algorithm

In selection sort, the first smallest element is selected from the unsorted array and placed at the first position.

After that second smallest element is selected and placed in the second position. The process continues until the array is entirely sorted.

The average and worst-case complexity of selection sort is O(n2), where n is the number of items. Due to this, it is not suitable for large data sets.

Selection sort is generally used when -

- A small array is to be sorted
- Swapping cost doesn't matter
- It is compulsory to check all elements

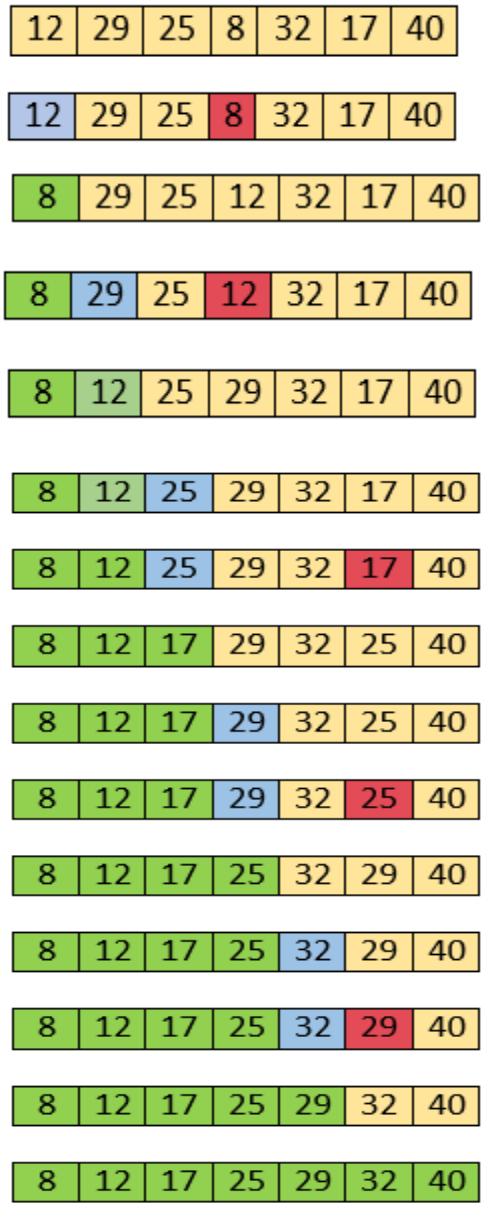
Algorithm

```
SELECTION SORT(arr, n)

Step 1: Repeat Steps 2 and 3 for i = 0 to n-1
Step 2: CALL SMALLEST(arr, i, n, pos)
Step 3: SWAP arr[i] with arr[pos]
[END OF LOOP]
Step 4: EXIT

SMALLEST (arr, i, n, pos)
Step 1: [INITIALIZE] SET SMALL = arr[i]
Step 2: [INITIALIZE] SET pos = i
Step 3: Repeat for j = i+1 to n
if (SMALL > arr[j])
    SET SMALL = arr[j]
SET pos = j
[END OF if]
[END OF LOOP]
Step 4: RETURN pos
```

Working of Selection sort Algorithm

Now, let's see the working of the Selection sort Algorithm.

Let the elements of array are -



Now, the array is completely sorted.

Selection sort complexity

Now, let's see the time complexity of selection sort in best case, average case, and in worst case. We will also see the space complexity of the selection sort.

1. Time Complexity

Case	Time Complexity
Best Case	**$O(n^2)$**
Average Case	**$O(n^2)$**
Worst Case	**$O(n^2)$**

- **Best Case Complexity -** It occurs when there is no sorting required, i.e. the array is already sorted. The best-case time complexity of selection sort is **$O(n^2)$**.
- **Average Case Complexity -** It occurs when the array elements are in jumbled order that is not properly ascending and not properly descending. The average case time complexity of selection sort is **$O(n^2)$**.
- **Worst Case Complexity -** It occurs when the array elements are required to be sorted in reverse order. That means suppose you have to sort the array elements in ascending order, but its elements are in descending order. The worst-case time complexity of selection sort is **$O(n^2)$**.

2. Space Complexity

Space Complexity	O(1)
Stable	YES

- The space complexity of selection sort is O(1). It is because, in selection sort, an extra variable is required for swapping.

Insertion Sort Algorithm

- It is a **stable sorting algorithm**, meaning that elements with equal values maintain their relative order in the sorted output.
- Insertion sort is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.
- Insertion sort is a simple sorting algorithm that works by building a sorted array one element at a time.
- It is considered an **" in-place "** sorting algorithm, meaning it doesn't require any additional memory space beyond the original array.

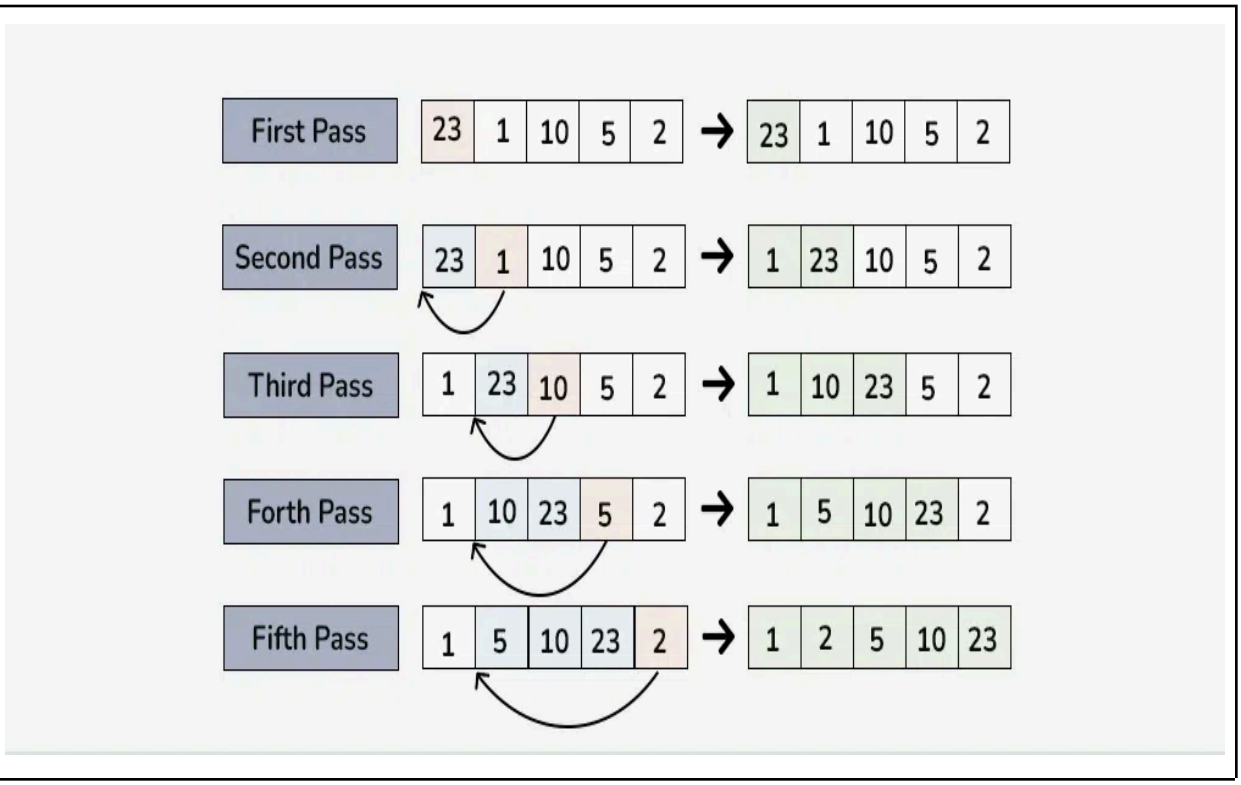
Algorithm:

To achieve insertion sort, follow these steps:

- We have to start with second element of the array as first element in the array is assumed to be sorted.
- Compare second element with the first element and check if the second element is smaller then swap them.
- Move to the third element and compare it with the second element, then the first element and swap as necessary to put it in the correct position among the first three elements.
- Continue this process, comparing each element with the ones before it and swapping as needed to place it in the correct position among the sorted elements.
- Repeat until the entire array is sorted.

Working of Insertion Sort Algorithm:

Consider an array having elements : {23, 1, 10, 5, 2}



- Time Complexity of Insertion Sort

1. Time Complexity

Case	Time Complexity
Best Case	**O(n)**
Average Case	**$O(n^2)$**
Worst Case	**$O(n^2)$**

- Space Complexity of Insertion Sort
  - Auxiliary Space: O(1), Insertion sort requires O(1) additional space, making it a space-efficient sorting algorithm.