Introduction to AOA

- Algorithm → Persian mathematician al-Khowarizmi → Ninth Century

Def^n - The algorithm is set of rules defined in specific order to do certain computation and carry out some predefined task.
It is the step by step procedure to solve the Input problem

Problem → Descriptive Solution (Natural Language) → Algorithm (Pseudo Code) → Program (Programming Language) → Output (Correct Result / Error if any)

Properties of AOA

i] Input
- Algorithm may take zero or more input arguments.

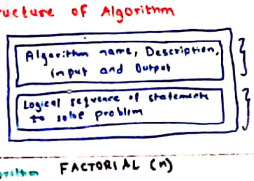
ii] Output
- Algorithm needs input, process it and produces at least one output.

iii] Definiteness
- All instructions in the algorithm should be unambiguous and simple to interpret
- Instruction should be precise and concise.

iv] Finiteness
- Every algorithm must terminate after a finite number of steps
- If algorithm contains a loop, the upper bound of the loop must be finite.
- Recursive calls should have a well defined base case to terminate the algorithm

v] Effectiveness
- The algorithm should be written with a basic set of instruction.
- Complex operations should be performed using a combination of basic instruction.

Structure of Algorithm




```
Algorithm FACTORIAL (n)
// Description: Find factorial of given number
// Input: Number n whose factorial is to be computed
// Output: Factorial of n = n x (n-1) x ... x 2 x 1
if (n == 1) then
    return 1
else
    return n * FACTORIAL (n-1)
end
```

Space Complexity

Def^n - Problem solving using computer requires memory to hold temporary data or final result while the program is in execution.
The amount of memory required by the algorithm to solve given problem is called Space Complexity of the algorithm

- Controlling components of space complexity

→ **i] Fixed Size Components**
- Includes the programming part whose memory requirement does not alter on program execution
  Ex - Instructions
  - Variables
  - Constants
  - Arrays

→ **ii] Variable Size Components**
- Includes the part of program whose size depends on the problem being solved.
  Ex - Size of loop
  - Stack required to handle recursive call
  - Dynamic data structure like linked list

Time Complexity

Def^n - The valid algorithm takes a finite amount of time for execution.
The time required by the algorithm to solve given problem is called Time Complexity of the algorithm.
It is very useful measure in algorithm analysis

Growth of Function

Def^n - The efficiency of the algorithm is expressed in term of input size n.

The relationship between input size and performance of the algorithm is called order of growth

Efficiency class	Order of growth rate	Example
- Constant	1	- Delete the 1st node from LL. - Remove the max element from max heap - Add 2 no.
- Logarithmic	log n	- Binary Search - Insert / Delete element
- Linear	n	- Linear Search - Insert node at the end of LL
- n log n	n log n	- Find min/max from array - Merge Sort - Binary Search - Quick sort - Heap sort
- Quadratic	n^2	- Selection sort - Bubble sort - Input 2D Array - Find max from 2D matrix
- Cubic	n^3	- Matrix multiplication
- Exponential	2^n	- Finding power set of any set - Optimal soln for knapsack pb
- Factorial	n!	- TSP using dynamic programming - Generating permutation of given set - TSP pb using brute force approach.

Efficiency classes are sorted as -

$O(1) < O(\log^k n) < O(\log(\log n)) < O(\log n) < O((\log n)^k) < O(n) < O(n \log^k n) < O(n \log n) < O(n^2) < O(n^3) < O(2^n) < O(n!)$.

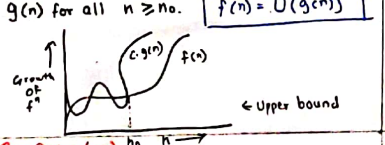
Asymptotic Notation

Def^n - It is a mathematical tool to find a time and space complexity of an algorithm without implementing it in programming language. This measure is independent of machine-specific constants.
It is a way of describing a major component of the cost of the entire algorithm.

Asymptotic Notation
→ Big Oh (O)
→ Big Omega (Ω)
→ Big Theta (θ)

Big Oh (O)

Def^n Let $f(n)$ & $g(n)$ are 2 non-negative f^n → Indicates running time of 2 Algorithms.
$g(n)$ is upper bound of $f(n)$; if $\exists$ +ve const c & n_0 such that $0 \le f(n) \le c$ $g(n)$ for all $n \ge n_0$. $f(n) = O(g(n))$



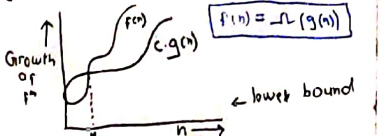
Framework for analysis of non-recursive Algo

- Finding complexity in non-recursive is simpler than in recursive algorithm

STEPS -
i] Determine size of problem / input
ii] Find out primitive / elementary operation
iii] Find count of primitive operations for best, worst, average case
iv] Simplify the summation by dropping multiplicative and divisive constants of highest degree polynomial term in sum

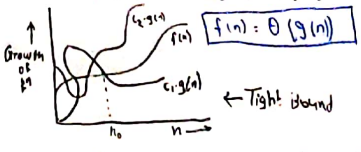
Big Omega (Ω)

Def^n Let $f(n)$ & $g(n)$ are 2 non-negative f^n → Indicates running time of 2 Algorithms
$g(n)$ is lower bound of $f(n)$; if $\exists$ +ve const c & n_0 such that $0 \le c\ g(n) \le f(n)$, $\forall\ n \ge n_0$. $f(n) = \Omega(g(n))$



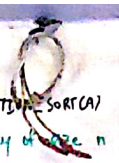
Big Theta (θ)

Def^n - Let $f(n)$ & $g(n)$ are 2 non-negative f^n indicating running time of 2 Algorithm.
$g(n)$ is tight bound of $f(n)$; if $\exists$ +ve const c_1, c_2, n_0 such that $0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n)$; $\forall\ n \ge n_0$



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

Selection Sort



Algorithm: SELECTION_SORT(A)
// A is an array of size n

```
for i ← 1 to n-1 do
    min ← i
    for j ← i+1 to n do
        if (A[j] < A[min]) do
            min ← j
        end
    end
    swap(A[i], A[min])
end
```

No. of Iterations

	0	1	2	3	4	No. of elements (n)	No. of comparisons
①	6	8	3	1	7		=4
②	1	8	3	6	7	(n-1)	=3 (n-2)
③	1	3	8	6	7		=2 (n-3)
④	1	3	6	8	7		=1 (n-4)
n-1	1	3	6	7	8		

$\sum_{i=1}^{K} i = \frac{K(K+1)}{2}$

In selection sort, $K = n-1$. Hence, total no. of comparisons required are

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

- $\frac{(n-1) \cdot n}{2}$
- $\frac{n^2}{2} - \frac{n}{2}$
- $= \frac{1}{2} n^2 - \frac{n}{2}$ ← lower order terms

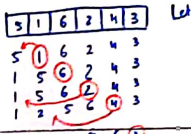
C eliminating constant & lower order terms $= n^2$

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

Best case	Average-case	Worst Case
$O(n^2)$	$O(n^2)$	$O(n^2)$

Insertion Sort

```
Algorithm Insertion Sort(A)                    Cost    Time
// A is an array of size n
for j = 2 to n do                              c1      n
    Key ← A[j]                                 c2      n-1
    i ← j-1                                    c3      n-1
    while (i > 0 && A[i] > Key)                c4      Σ_{j=2}^{n} t_j
        A[i+1] ← A[i]                          c5      Σ_{j=2}^{n} (t_j - 1)
        i ← i-1                                c6      Σ_{j=2}^{n} (t_j - 1)
    end                                        -       -
    A[i+1] ← Key                               c7      n-1
end                                            -       -
```



Lets take this array

(Always we start with the second element as key)

Best case	Average case	Worst case
$O(n)$	$O(n^2)$	$O(n^2)$