1. Algorithm Analysis

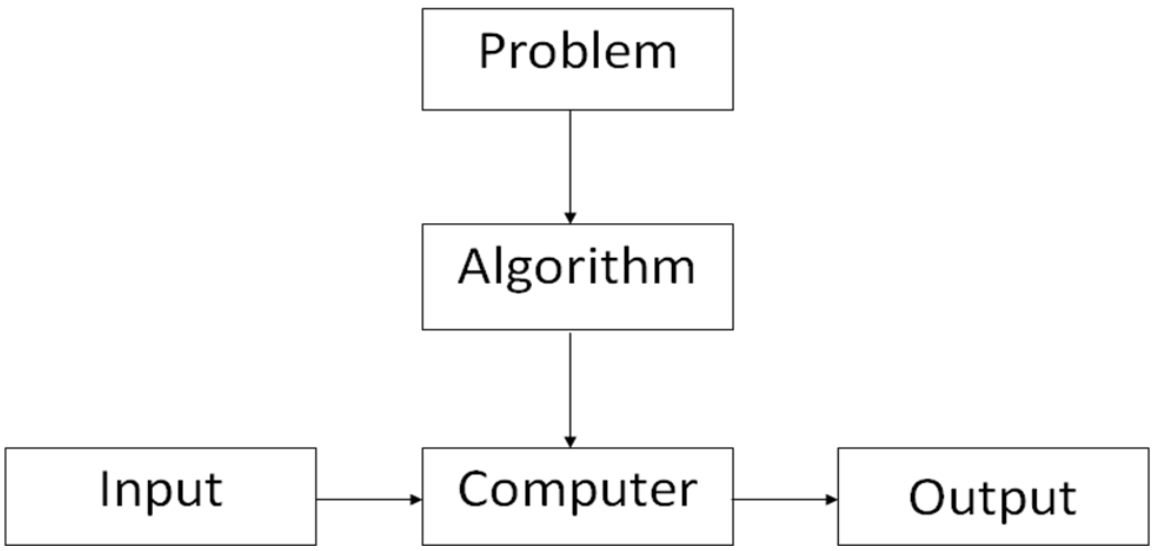
Topics Covered

1. Algorithm
2. Time and space complexities
3. Time-Space Trade off.
4. Classes of Algorithm
5. Algorithm Analysis
6. Asymptoticnotation
   6.1. Big-Oh notation
   6.2. Big-Omega notation

1. Algorithm

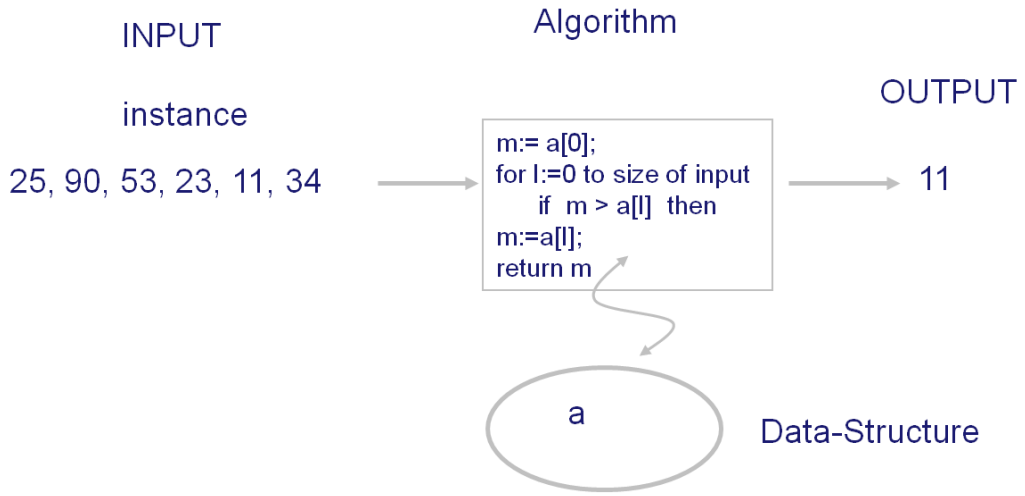
❖ What is Algorithm?

- "A step-by-step procedure to solve a problem."
- Definition:"An algorithm is a sequence of unambiguous instructions for solving the problem and to obtain a required output for any legitimate (Valid) input in finite (fix) amount of time".
- Is any well-defined computational procedure that takes some value, or set of values, as input and produces some value, or set of values, as output?
- An algorithm takes the input to a problem (function) and transforms it to the output.



❖ Example: What is an Algorithm?

- **Problem:** Input is a sequence of integers stored in an array. Output the minimum.



❖ Characteristics (Properties) of algorithm

- **Input**: An algorithm can take zero or more inputs.
- **Output**: It should give one or more outputs.
- **Definiteness**: Each instruction should be clear and unambiguous.
- **Finiteness**: All cases the algorithm must terminate in finite number of steps.
- **Effectiveness**: It must be simple and feasible.
- **Termination**: It must be terminate after finite (fix) number of steps.

❖ Why we Analyzing an Algorithm?

- **Time efficiency**: How fast the algorithm runs.
- **Space efficiency**: How much extra space the algorithm needs.
- **Simplicity**: Simpler algorithms are easier to understand.
- Correctness:Does the input/output relation match algorithm requirement?
- **Generality**: It is easier to design an algorithm for a problem posed in more general terms.
- The “complexity of an algorithm” is simply the amount of work the algorithm performs to complete its task.
- Analyzing an algorithm means predicts the resource that the algorithm requires such as memory, communication bandwidth, logic gates and time.
- So. We can say that , Complexity refers to the rate which they required storage or consumed time grows as a function of the problem size.
- The running time of program is describe as a function of the size of its input .on particular input It is traditionally measured as the number of primitive operations or steps executed
- Let ‘n’ denotes size

- The time require of a specific algorithm for solving this program is expressed by a function **f:R->R**
  Such that , **f(n)** is the largest amount of time needed by the algorithm to solve the problem of size n .'f' is usually called the time complexity function.
- The analysis of the program requires two main consideration :
  - Space Complexity.
  - Time Complexity.

2. Time and space complexities

- The time complexity of a program /algorithm is the amount of computer time that it need to run to completion.
- The space complexity of a program/algorithm is the amount of memory that it needs to run completion.
- **Space Complexity**
  - The **amount of memory is** requiring to run and completion of an algorithm or program is known as *Space complexity*.
  - Measure of an algorithm's total memory requirements during runtime.
    - Data structures,
    - Temporary variables.
  - Its analysis is known as analysis of space complexity of an algorithm.
  - This is essentially the number of memory cells which an algorithm needs. A good algorithm keeps this number as small as possible, too.
  - The space needed by a program consists of following components
    - *Instruction space*: space needed to store the executable version of program and it is fixed.
    - *Data space:* space needed to store all variable ,constants
    - *Environment stack space:* this space is needed to store the information to resume the suspended function. *Each* time a function is invoked the following data is saved on environment stack.
      --Return address.
      --Value of formal parameter.
- **Time Complexity**
  - The time complexity of a program /algorithm is the amount of computer time that it need to run to completion.
  - The number of (machine) instructions which a program executes during its running time is called its time complexity in computer science
  - How much time does it take to run the algorithm
  - Many criteria affect the running time of an algorithm, including
    - speed of CPU, bus and peripheral hardware
    - design think time, programming time and debugging time
    - language used and coding efficiency of the programmer
    - quality of input (good, bad or average)
  - To find out exact time complexity , we need to know the exact instruction executed by the hardware and time require for the instruction .
  - And it's also depending on input given to algorithm.
  - Example:
    - **ALGORITHM 1:**a=a+1

- in algorithm 1 we see that only one statement a=a+1 is independent and is not contain any loop so number of time shell be executed is 1.
- We can say that frequency counter of algorithm is 1.
- **ALGORITHM 2:**for i=1 to n step 1

a=a+1

- In above algorithm ,the key statement out of two is the a=a+1 and contains loop ,the number of times it is executed is n .
- We can say that frequency counter of algorithm is n
- **ALGORITHM 3:**for i=1 to n step 1
  for j=1 to n step 1

a=a+1

- We can say that frequency counter of algorithm is n^2**.**

3. Time-Space Trade off

- The analysis of algorithm focuses on time complexity and space complexity .as compared to time analysis , the analysis of space requirement for an algorithm is generally easier ,but wherever necessary , both the techniques are used .
- The space refers to as storage require in addition to the space require store inputted data.
- For an algorithm, time complexity depends upon the size of the input ,thus it is a function of input size 'n' .
- The best algorithm to solve a given problem is one that require less memory and takes less time to complete its execution.
- But in practice it is not always possible to achieve both of these objectives.
- There may be a more than one approach to solve a problem
- One approach may require more time but less space while other require less time but more space.
- That is ,we may be able to reduce space requirement by increasing running time or reducing running time by allocating more space .
- This situation where we compromise one to batter the other is known as **Time-space tradeoff.**

4. Classes of Algorithm

- Algorithm can be classified into two categories.
  - By Implementation way.
  - By design paradigm(pattern).
- **By Implementation way**
  - **Recursion or Iteration**
    - A recursive algorithm means that ,it invokes itself repeatedly until a certain condition matches . A best example of recursive algorithm is Quick Sort.
    - Iterative algorithms use repetitive constructs like loops and sometimes additional data structures like stacks to solve the given problems. Some problems are naturally suited for one implementation or the other.
    - Every recursive version has an equivalent (but possibly more or less complex) iterative version, and vice versa.
  - **Logical**

- In pure logic programming languages the control component is fixed and algorithms are specified by supplying only the logic component. The appeal of this approach is the elegant semantics: a change in the structure has a well-defined change in the algorithm.

- **Serial or parallel or distributed**
  - Algorithms are usually discussed with the assumption that computers execute one instruction of an algorithm at a time. Those computers are sometimes called serial computers. An algorithm designed for such an environment is called a serial algorithm,
  - Parallel algorithms take advantage of computer architectures where several processors can work on a problem at the same time, whereas distributed algorithms utilize multiple machines connected with a network.
- **Deterministic or non-deterministic**:
  - Deterministic algorithms solve the problem with exact decision at every step of the algorithm whereas non-deterministic algorithms solve problems via guessing although typical guesses are made more accurate through the use of heuristics
- **Exact or approximate**:
  - While many algorithms reach an exact solution, approximation algorithms seek an approximation that is close to the true solution. Approximation may use either a deterministic or a random strategy. Such algorithms have practical value for many hard problems.

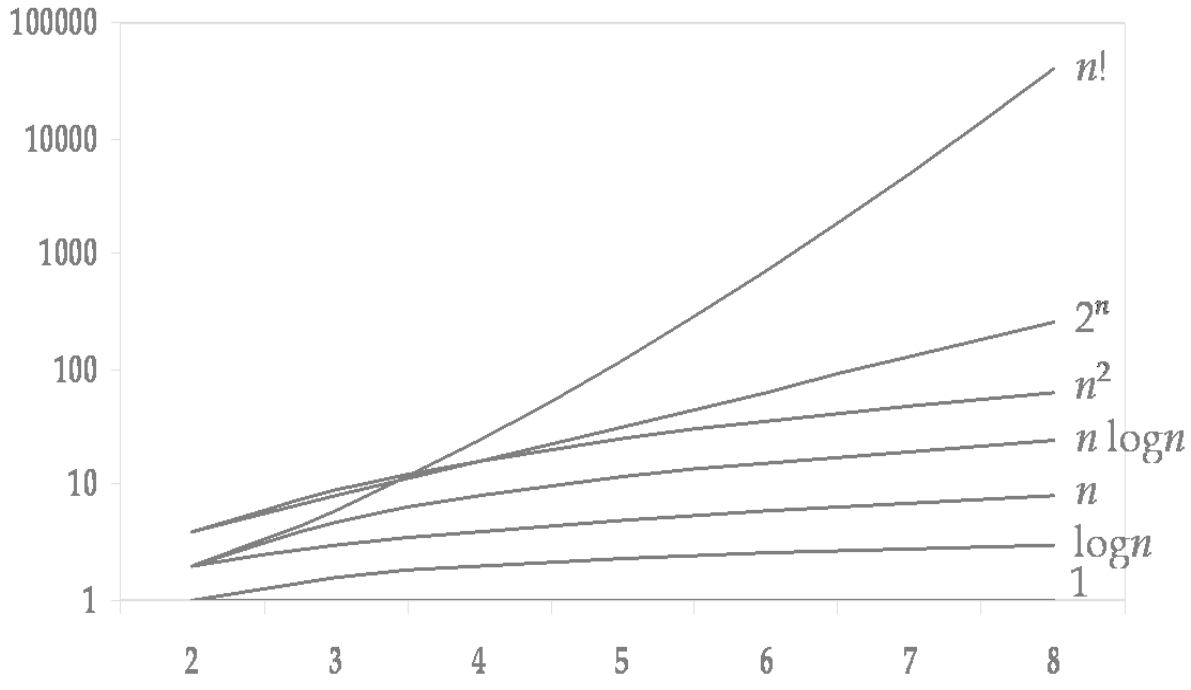
- **By design paradigm**
  - **Divide and Conquer**
    - A divide and conquer algorithm consists of two parts:
    - Divide the problem into smaller sub problems of the same type, and solve these sub problems recursively
    - Combine the solutions to the sub problems into a solution to the original problem.
    - **Examples**
    - *Quicksort*:
    - Partition the array into two parts, and quicksort each of the parts
    - No additional work is required to combine the two sorted parts
    - *Mergesort:*
    - Cut the array in half, and mergesort each half
    - Combine the two sorted arrays into a single sorted array by merging them
  - **Greedy algorithms**
    - An optimization problem is one in which you want to find, not just *a* solution, but the *best* solution
    - A “greedy algorithm” sometimes works well for optimization problems
  - **Branch and bound algorithms**
    - Branch and bound algorithms are generally used for optimization problems
    - As the algorithm progresses, a tree of sub problems is formed
    - The original problem is considered the “root problem”
    - A method is used to construct an upper and lower bound for a given problem

- Travelling salesman problem: A salesman has to visit each of n cities (at least) once each, and wants to minimize total distance travelled

- **Brute force algorithm**
  - A brute force algorithm simply tries *all* possibilities until a satisfactory solution is found
- **Randomized algorithms**
  - A randomized algorithm uses a random number at least once during the computation to make a decision
  - Example: In Quicksort, using a random number to choose a pivot.
- **Dynamic programming algorithms**
  - A dynamic programming algorithm remembers past results and uses them to find new results
  - Dynamic programming is generally used for optimization problems
  - Multiple solutions exist, need to find the "best" one
- **Backtracking algorithms**
  - Backtracking algorithms are based on a depth-first recursive search.
  - A backtracking algorithm tests to see if a solution has been found, and if so, returns it; otherwise.

5. Algorithm Analysis

- How to estimate the time required for an algorithm.
- Many criteria affect the running time of an algorithm, including
  - speed of CPU, bus and peripheral hardware
  - design think time, programming time and debugging time
  - language used and coding efficiency of the programmer
  - quality of input (good, bad or average)
- Concept of growth rate allows us to compare running time of two algorithms without writing two programs and running them on the same computer



Function	Name
c	Constant
$\log N$	Logarithmic
$\log^2 N$	Log-squared
N	Linear
$N \log N$	N log N
N^2	Quadratic
N^3	Cubic
2^N	Exponential

Functions in order of increasing growth rate

- When we analyze an algorithm it depend is a function of the input size. There are three different type
- **Best case** fastest time to complete, with optimal inputs chosen. For example, the best case for a sorting algorithm would be data that's already sorted.
- **Worst case** slowest time to complete, with incorrect inputs chosen. For example, the worst case for a sorting algorithm might be data that's sorted in reverse order (but it depends on the particular algorithm).
- **Average case** arithmetic mean. Run the algorithm many times, using many different inputs, compute the total running time (by adding the individual times), and divide by the number of trials. You may also need to normalize the results based on the size of the input sets.

6. Asymptoticnotation

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 i.e. the best case.

But, when the input array is in reverse condition, the algorithm takes the maximum time (quadratic) to sort the elements i.e. the worst case.

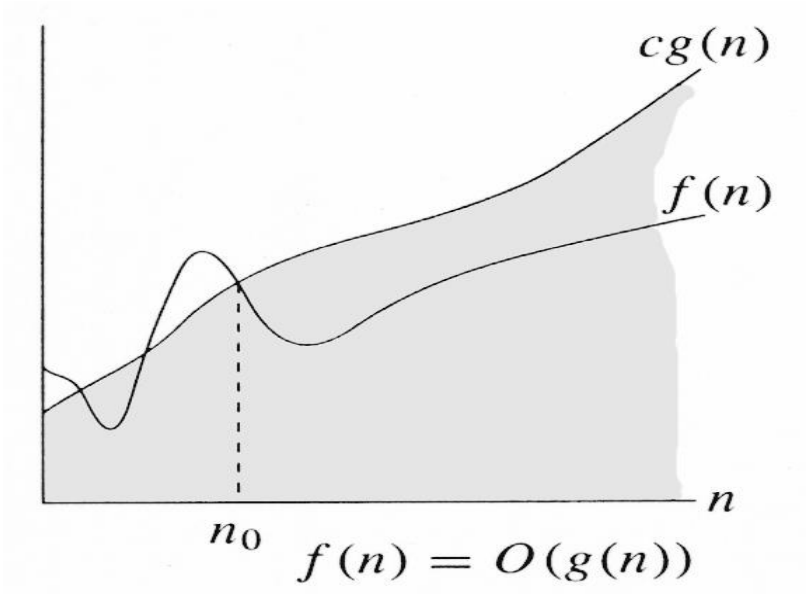
When the input array is neither sorted nor in reverse order, then it takes average time. These durations are denoted using asymptotic notations.

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

6.1 Big-Oh notation(Big O)

- Time complexity of algorithms is expressed using the O-notation (Big-Oh).
- Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm.



O(g(n)) = { f(n): there exist positive constants c and n_0 such that 0 ≤ f(n) ≤ cg(n) for all n ≥ n_0 }

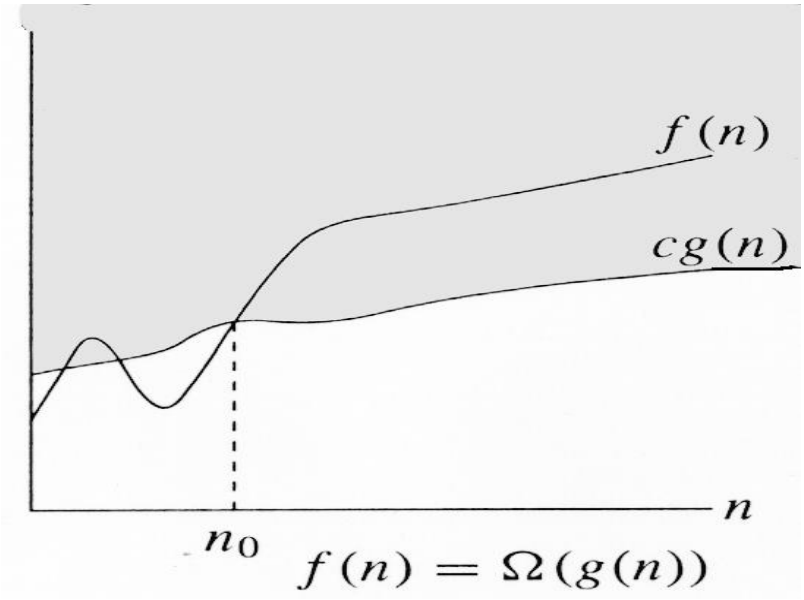
- The above expression can be described as a function f(n) belongs to the set O(g(n)) if there exists a positive constant c such that it lies between 0 and cg(n), for sufficiently large n.
- For any value of n, the running time of an algorithm does not cross the time provided by O(g(n)).
- Since it gives 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.

Complexity	Term
***O*(1)**	**constant**
***O*(log *n*)**	**logarithmic**
***O*(*n*)**	**linear**
***O*(*n* log *n*)**	***n* log *n***
***O*(*n*!)**	**factorial**

- **O(1):**
  - O(1) describes an algorithm that will always execute in the same time (or space) regardless of the size of the input data set.
- **O(n):**
  - O(N) describes an algorithm whose performance will grow linearly and in direct proportion to the size of the input data set.
- **O(N^2)**
  - O(N^2) represents an algorithm whose performance is directly proportional to the square of the size of the input data set.
- **O(2^n)**
  - O(2^n) denotes an algorithm whose growth will double with each additional element in the input data set. The execution time of an O(2N) function will quickly become very large.
- **O(logN)**
  - This type of algorithm is described as **O(log N)**. The iterative halving of data sets described in the binary search example produces a growth curve that peaks at the beginning and slowly flattens out as the size of the data sets increase

6.2 Big-Omega notation(Big Ω)

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



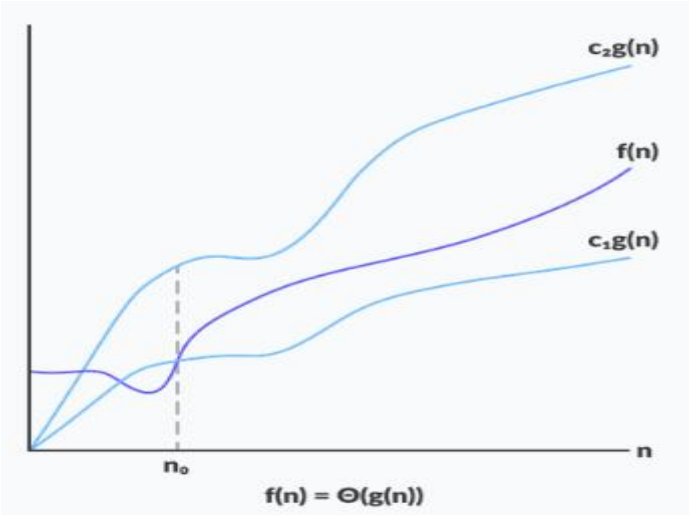
Ω(g(n)) = { f(n): there exist positive constants c and n_0 such that $0 \le cg(n) \le f(n)$ for all $n \ge n_0$ }

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)).

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:

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

The above expression can be described as a function f(n) belongs to the set Θ(g(n)) if there exist positive constants c_1 and c_2 such that it can be sandwiched between $c_1g(n)$ and $c_2g(n)$, for sufficiently large n.

If a function f(n) lies anywhere in between $c_1g(n)$ and $c_2g(n)$ for all $n \ge n0$, then f(n) is said to be asymptotically tight bound.

Quick Summary of Various sorting algorithms time complexities

Name	Best Case	Average Case	Worst Case	Method (Algorithm)
Quick Sort	*n logn*	*n logn*	n^2	Divide and Conquer
Merge Sort	*n logn*	*n logn*	*n logn*	Divide and Conquer
Heap Sort	*n logn*	*n logn*	*n logn*	Transform and Conquer
Insertion Sort	*n*	n^2	n^2	Decrease and Conquer
Selection Sort	n^2	n^2	n^2	Brute Force
Shell Sort	*n*	$n(logn)^2$	$n(logn)^2$	Decrease and Conquer
Bubble Sort	*n*	n^2	n^2	Brute Force