CHAPTER 1: PRINCIPLES OF ANALYZING ALGORITHMS AND PROBLEMS

By: Ashok Basnet

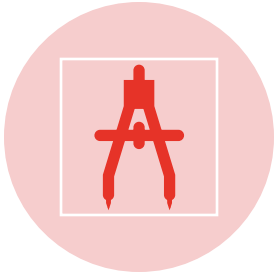
Course Outline

1. Principles of Analyzing Algorithm and Problems: Space and Time Complexity
2. Review of Abstract Data Types: Stack and Queue
3. Sorting, Selection and Sequencing: Merge sort, Binary search, Job sequencing with deadlines.
4. Dynamic Programming: Knapsack Problem
5. Graph Traversal and search techniques: Breadth first search, Depth first search.
6. Backtracking: The 8-queens problem

Chapter 1: Principles of Analyzing Algorithms and Problems

1. Introduction to Algorithms

2. Algorithm Specification



3. Performance Analysis (Space and Time Complexity)

4. Randomized Algorithms

What is Algorithm?

Understanding Algorithms

- In mathematics and computer science, algorithms are powerful tools for solving specific problems and automating tasks.
- **Definition**: An algorithm is a precise sequence of instructions designed to tackle a particular problem or carry out a computation.
- **Problem Solving:** Algorithms serve as blueprints for performing calculations and processing data efficiently.
- **Advanced Capabilities:** Sophisticated algorithms can automate deductions (automated reasoning) and guide code execution using mathematical and logical tests (automated decision-making)

Key Characteristics:

- Algorithms consist of finite, well-defined instructions.
- Each instruction is understandable and achievable in a finite amount of time.
- Regardless of input values, an algorithm always terminates after a finite number of instructions.
- Algorithms are the foundation of computer science and are essential for solving complex real-world challenges.

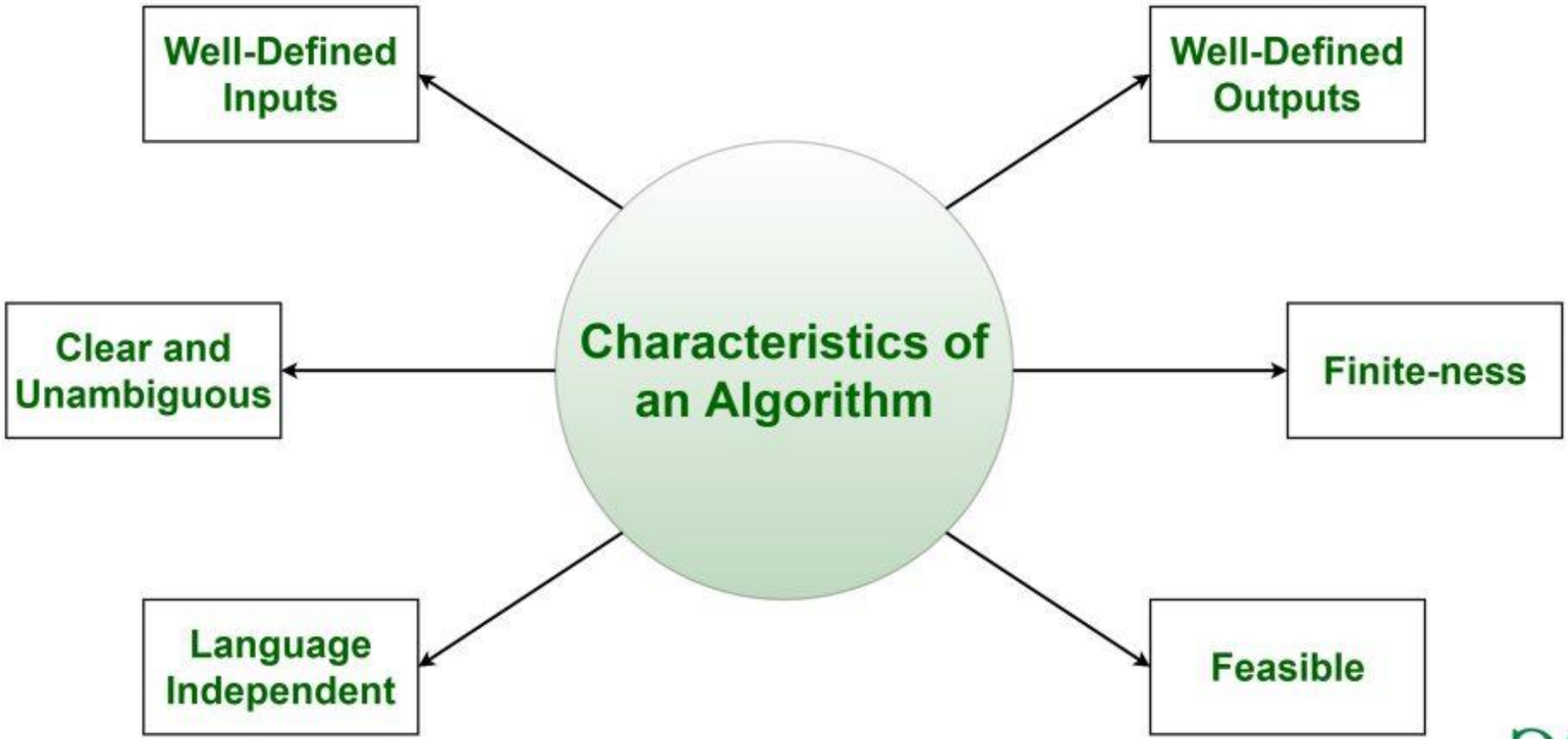
Algorithm Criteria:

- **Input**:
  - An algorithm may accept zero or more externally provided quantities or data.
  - Input should be well-defined and clearly specified for accurate processing.
- **Output**:
  - Every algorithm must produce at least one meaningful output or result.
  - The output should convey relevant information or serve a purpose.
- **Definiteness**:
  - Each instruction within an algorithm must be precise, clear, and free from ambiguity.
  - Clarity ensures that the algorithm's actions are well-understood and correctly executed.

Algorithm Criteria:

- **Finiteness**:
  - An algorithm must exhibit the property of **finiteness**, meaning it will complete its execution within a finite number of well-defined steps.
  - It should **avoid** infinite loops or indeterminate behavior.
- **Effectiveness**:
  - Every instruction in the algorithm must be fundamental and executable.
  - The instructions should be practical and achievable with available resources.

Characteristics of an Algorithm



Example: Multiply two numbers

Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7
Start	Get the knowledge of input. Here we need 3 variables; a and b will be the user input and c will hold the result.	Declare a, b, c variables.	Take input for a and b variable from the user.	Know the problem and find the solution using operators, data structures and logic • We need to multiply a and b variables so we use * operator and assign the result to c. • That is c <- a * b	Check how to give output, Here we need to print the output. So write print c	End

Reason for Analysis

Need to recognize limitations of various algorithms for solving a problem.

Need to understand relationship between problem size and running time when is a running program not good enough?

Need to learn how to analyze an algorithm's running time without coding it.

Need to learn techniques for writing more efficient code.

Need to recognize bottlenecks in code as well as which parts of code are easiest to optimize.

Why do we Analyze Algorithms

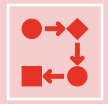
Correctness and Efficiency

Decide whether some problems have no solution in reasonable time

Investigate memory usage as different measure of efficiency.

Performance of a program

The performance of a program is the amount of computer memory and time needed to run a program.



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

One is analytical, and the other experimental.

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

Time Complexity

- The time needed by an algorithm expressed as a function of the size of a problem is called the time complexity of the algorithm.
- The time complexity of a program is the amount of computer time it needs to run to completion.
- The limiting behavior of the complexity as size increases is called the asymptotic time complexity.
- It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

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. The amount of instructions space that is needed depends on factors such as:
  - ✓ The compiler used to complete the program into machine code.
  - ✓ The compiler options in effect at the time of compilation
  - ✓ The target computer.

Space Complexity

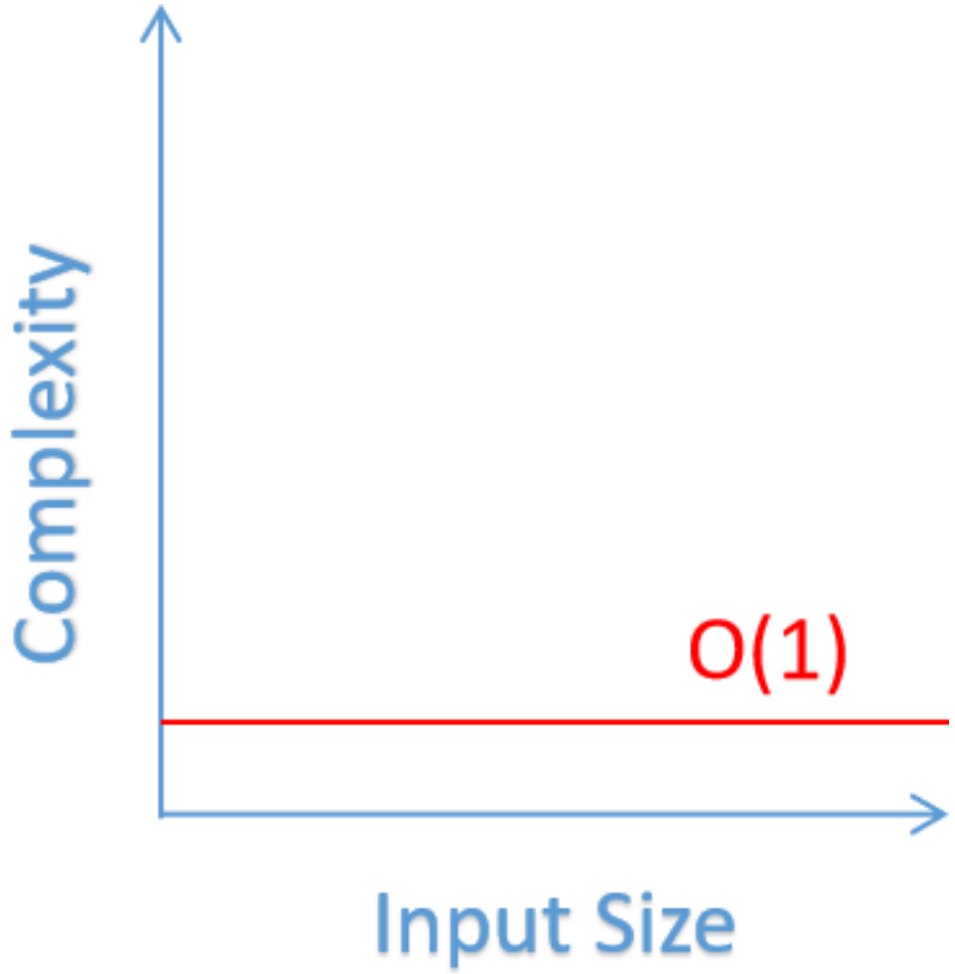
- **Data space:** Data space is the space needed to store all constant and variable values. Data space has two components:
  - ✓ Space needed by constants and simple variables in program.
  - ✓ Space needed by dynamically allocated objects such as arrays and class instances.
- **Environment stack space:** The environment stack is used to save information needed to resume execution of partially completed functions.

Algorithm Design Goals

- The three basic design goals that one should strive for in a program are:
  1. **Try to save Time**
  2. **Try to save Space**
  3. **Try to save Face**
- 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.
- We want to “save face” by preventing the program from locking up or generating reams of garbled data.

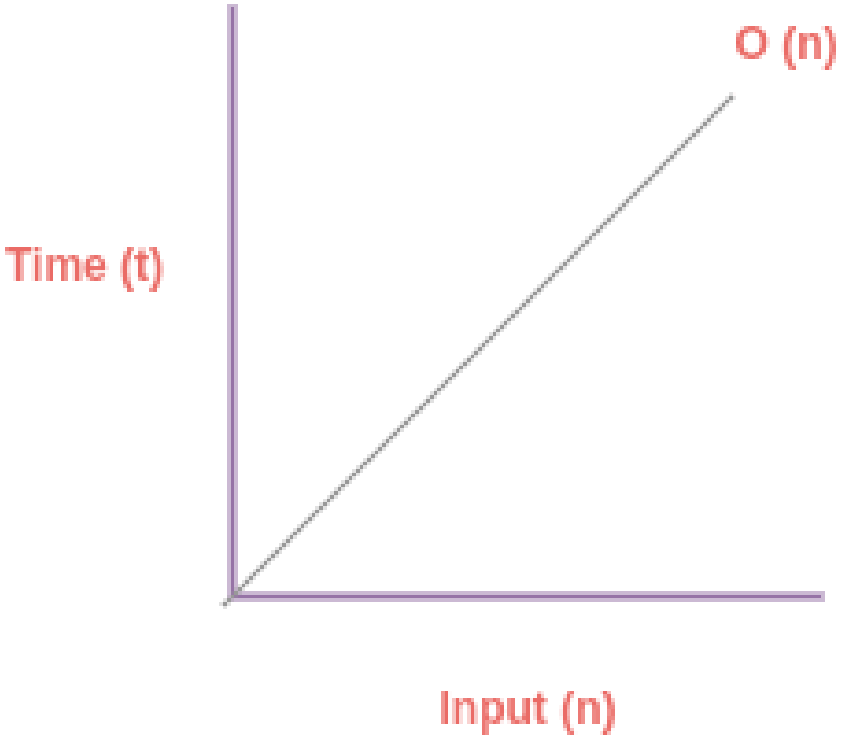
Classification of Algorithms

- If ‘n’ is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.
- **1** : If all the instructions of a program have this property, we say that its running time is a constant.



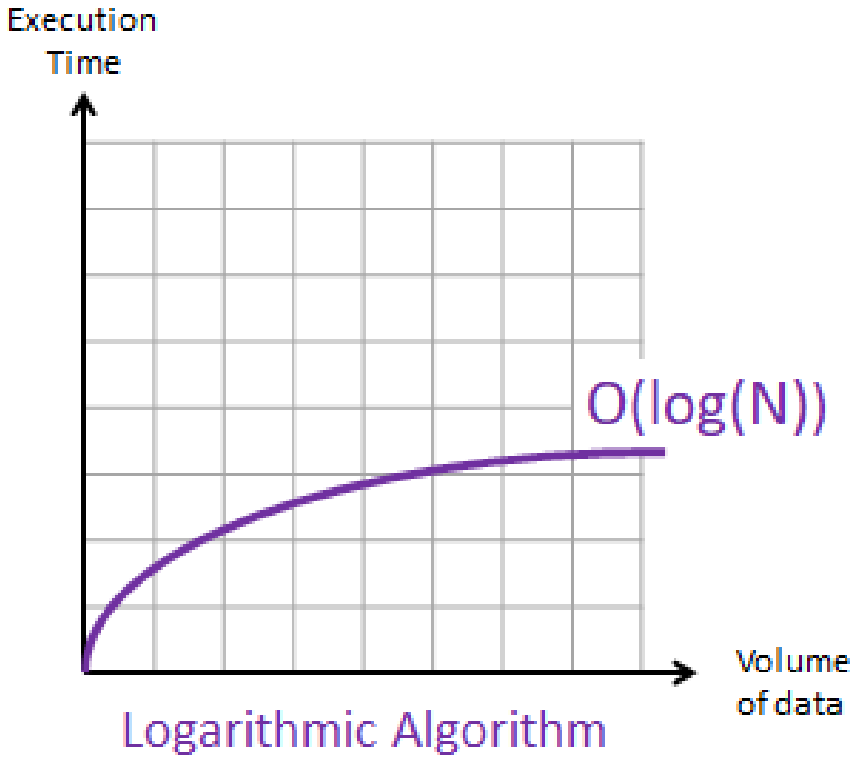
O(n)

- When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element.
- This is the optimal situation for an algorithm that must process n inputs.



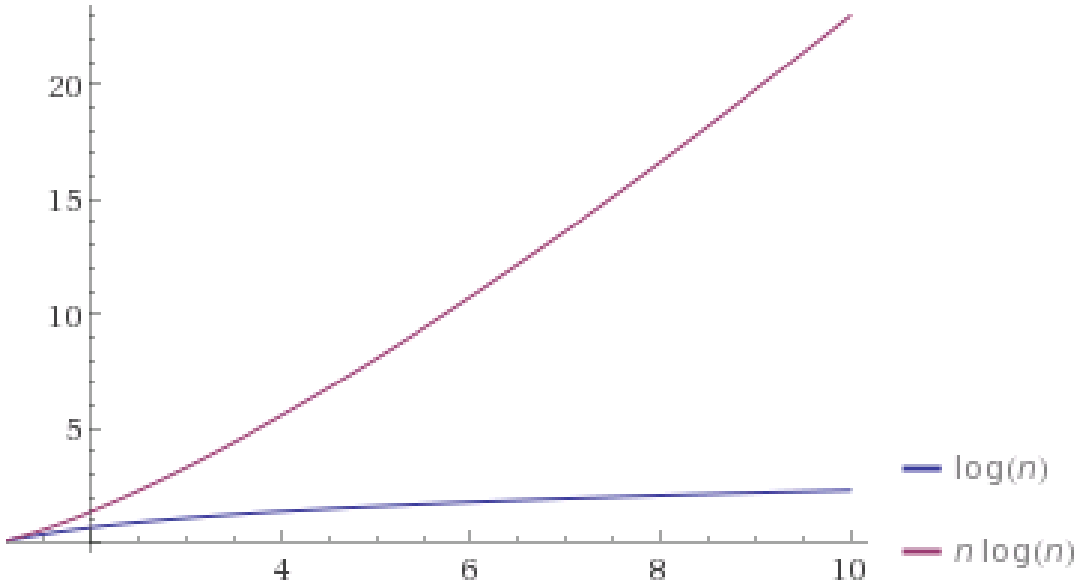
Log n

- When the running time of a program is logarithmic, the program gets slightly slower as n grows.
- This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction.
- When n is a million, log n is a doubled. Whenever n doubles, log n increases by a constant, but log n does not double until n increases to npow(2).



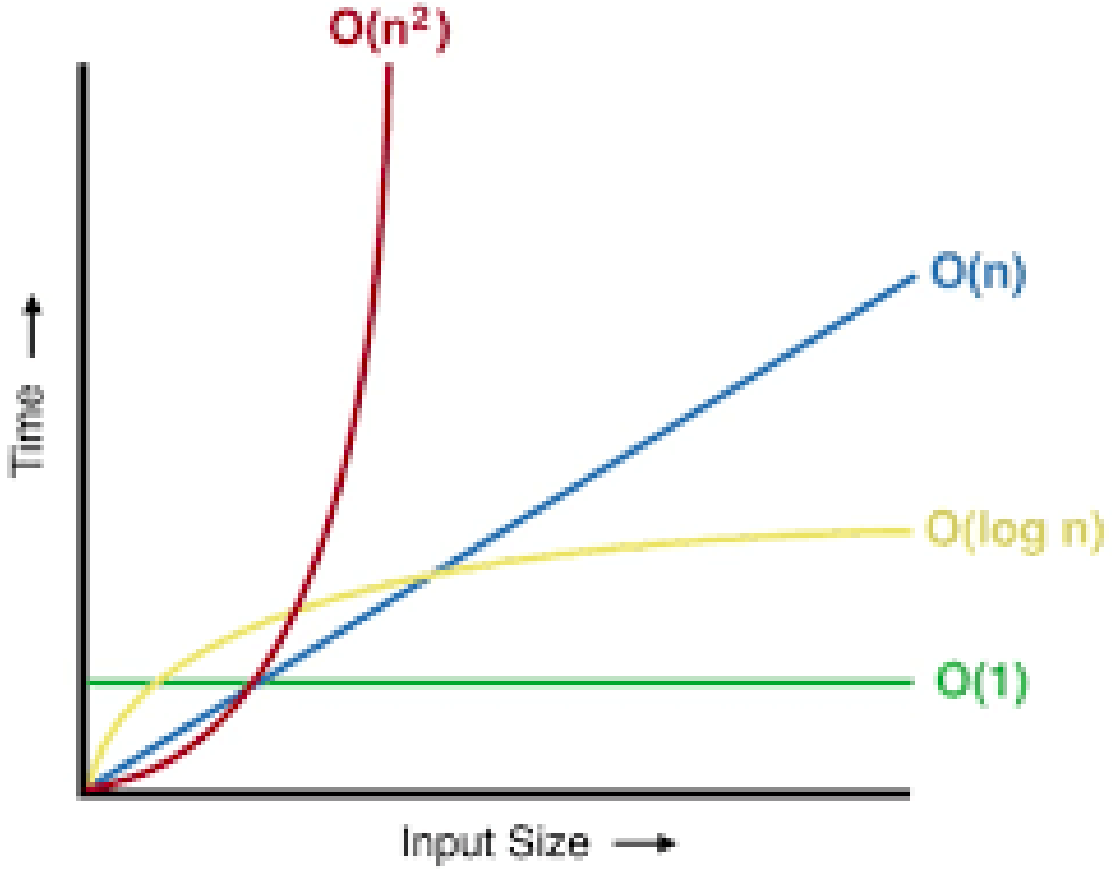
nlog(n)

- This running time arises for algorithms that solve a problem by breaking it up into smaller sub-problems, solving then independently, and then combining the solutions.
- When n doubles, the running time more than doubles.



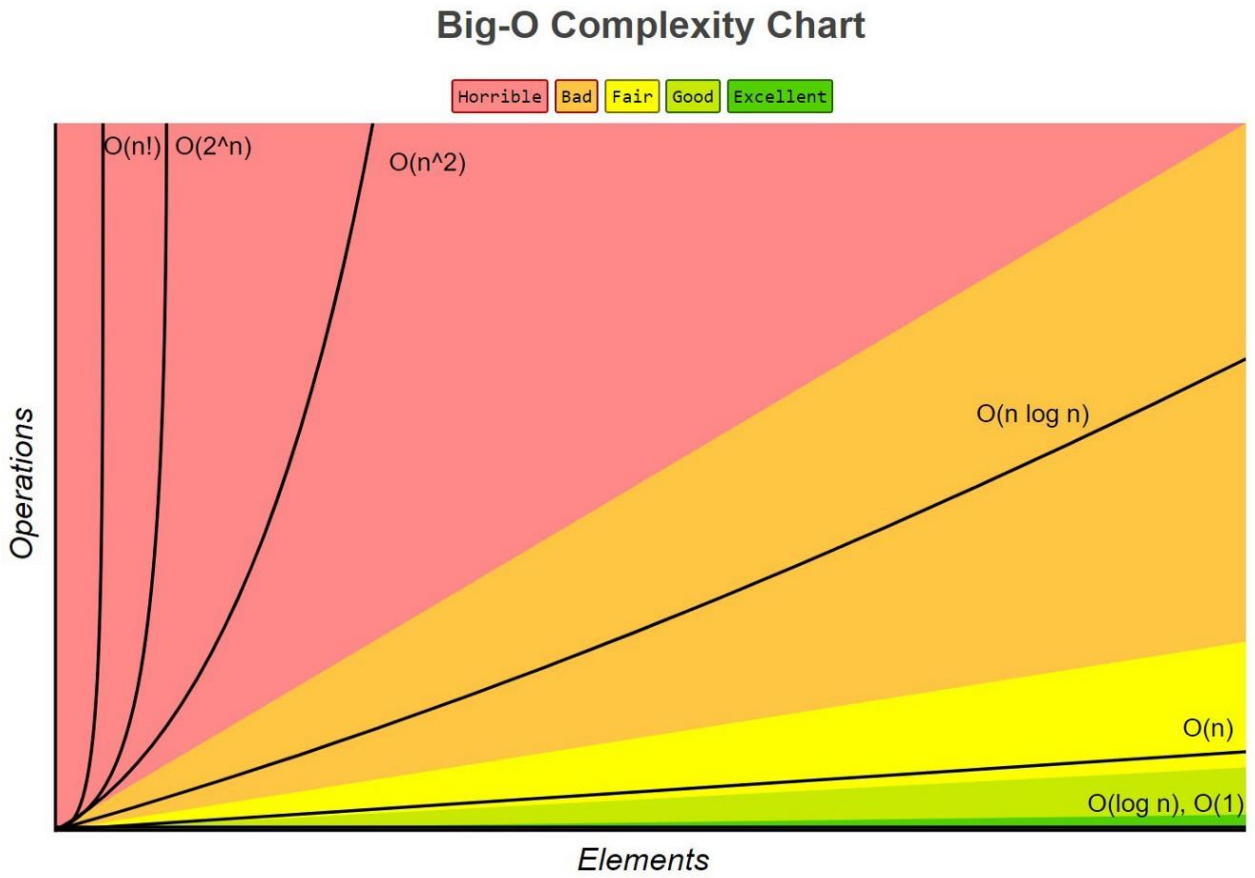
n^2

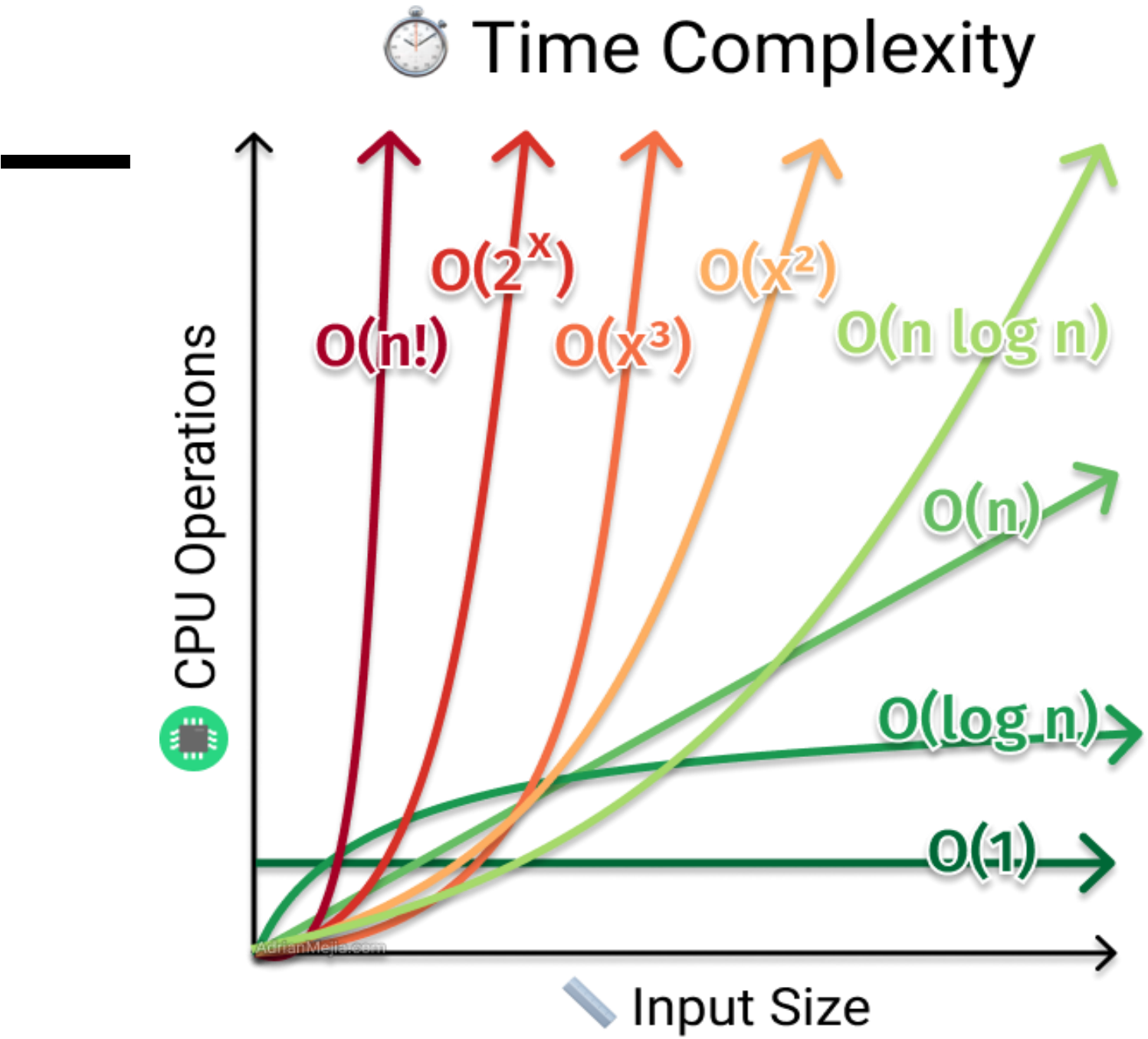
- When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems.
- Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold.

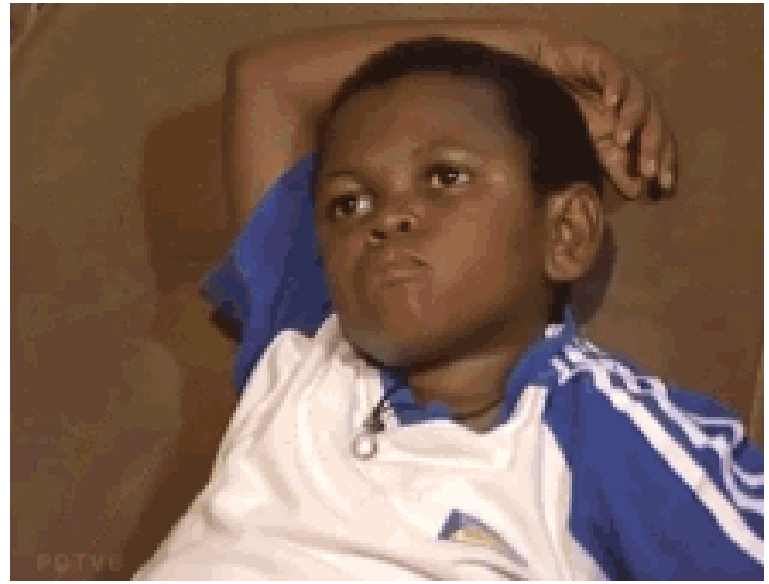


2^n

- Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as “brute–force” solutions to problems.
- Whenever n doubles, tr




Time Complexity
CPU Operations
O(n!)
O(2^x)
O(x^3)
O(x^2)
O(n log n)
O(n)
O(log n)
O(1)
Input Size
AdrianMejia.com



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	????????

Complexity of Algorithms

- The complexity of algorithm **M is defined by the function f(n),** which characterizes the algorithm's performance in terms of its **execution time and storage** space needs relative to the input data size 'n.'
- Typically, the storage space required by an algorithm is **directly proportional** to the size 'n' of the input data.
- It's important to note that the function f(n), which quantifies the algorithm's running time, relies not only on the **input data size 'n'** but also on the specific characteristics of the data being processed.

The complexity function f(n) for certain cases are:

- **Best Case (Infrequently Utilized):** This refers to the minimal conceivable value of f(n) and is referred to as the "best case."
- **Average Case (Seldom Employed):** This pertains to the anticipated value of f(n) and is known as the "average case."
- **Worst Case (Most Commonly Employed):** This signifies the maximum value of f(n) achievable for any conceivable input and is commonly referred to as the "worst case."
- The domain within computer science that delves into the efficiency of algorithms is formally recognized as the "analysis of algorithms."

Best Case:

- This denotes the specific input conditions under which an algorithm exhibits its **most efficient performance,** resulting in the shortest possible execution time.
- The best case provides a lower bound for the algorithm's time complexity.
- For instance, in the case of a linear search, the best case occurs when the sought-after data is found at the very beginning of a large dataset.

Worst Case

- This characterizes the input scenarios in which an **algorithm experiences its most time-consuming** execution, resulting in the longest possible time.
- The worst case offers an upper bound for the algorithm's time complexity.
- As an illustration, in linear search, the worst case transpires when the desired data is entirely absent from the dataset.

Average Case

- In the average case, we consider a **spectrum of random input conditions** and calculate the algorithm's execution time for each of these conditions.
- Subsequently, we determine the average by summing up the execution times for all random inputs and dividing it by the total number of inputs.
- This provides us with a **realistic** estimate of the algorithm's expected performance under typical, random scenarios.
- The formula for average case computation is as follows:
  - **Average Case = (Sum of execution times for all random cases) / (Total number of cases)**

Asymptotic Analysis

- 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 the time (or space) taken by an algorithm increases with the input size.***
- **For example,** let us consider the search problem (searching a given item) in a sorted array.
- The solution to above search problem includes:
  - ***Linear Search (order of growth is linear)***
  - ***Binary Search (order of growth is logarithmic).***

- To understand how Asymptotic Analysis solves the problems mentioned above in analyzing algorithms,
- let us say:
  - We run the **Linear Search** on a fast computer A and
  - **Binary Search** on a slow computer B and
  - pick the constant values for the two computers so that it tells us exactly how long it takes for the given machine to perform the search in seconds.
- Let's say the **constant for A** is 0.2 and the **constant for B** is 1000 which means that A is 5000 times more powerful than B.
- For small values of input array size n, the fast computer may take less time.
- But, after a certain value of input array size, the **Binary Searc**h will definitely start taking less time compared to the **Linear Search** even though the Binary Search is being run on a slow machine.

Input Size	Running time on A	Running time on B
10	2 sec	~ 1 h
100	20 sec	~ 1.8 h
10^6	~ 55.5 h	~ 5.5 h
10^9	~ 6.3 years	~ 8.3 h

- The reason is the order of growth of **Binary Search with respect to input size is logarithmic while the order of growth of Linear Search is linear.**
- So the machine-dependent constants can always be ignored after a certain value of input size.
- Running times for this example:
  - ***Linear Search running time in seconds on A: 0.2 * n***
  - ***Binary Search running time in seconds on B: 1000*log(n)***

Does Asymptotic Analysis always work?

- **Asymptotic Analysis:** A Valuable, Though Simplified, Tool: While not perfect, asymptotic analysis is the most effective method available for analyzing algorithms.
- *It provides valuable insights into how algorithms perform as input sizes increase, simplifying complex performance comparisons.*
- **An Example:** Comparing Sorting Algorithms: Consider two sorting algorithms: one with a time complexity of **1000nlog(n) and another with 2nlog(n).** Both share the same asymptotic complexity **(n*log(n)).** However, in asymptotic analysis, we disregard constant factors.
- **Constant Factors Disregarded:** Asymptotic analysis doesn't consider constant factors. Thus, it doesn't allow us to determine which of the two sorting algorithms is better in practice, as their real-world performance could differ due to these constants.

Big-O Notation

- We define an algorithm’s **worst-case** time complexity by using the Big-O notation.
- It explains the maximum amount of time an algorithm requires to consider all input values.
- **Scalability:** Estimates how an algorithm's performance scales with input size.
- **Comparisons:** Helps in contrasting different algorithms based on their scalability.
- **Constants Ignored:** Simplifies analysis by disregarding constant factors.
- **Practical Considerations:** Real-world performance may vary due to hardware and implementation details.

Omega Notation

- It defines the **best case** of an algorithm’s time complexity.
- It explains the minimum amount of time an algorithm requires to consider all input values.
- Omega notation helps us understand how efficiently an algorithm performs under favorable conditions.
- Unlike Big-O notation, Omega notation acknowledges constant factors and focuses on the lower bounds of an algorithm's performance.

Theta Notation

- It defines the **average case** of an algorithm’s time complexity, the Theta notation defines when the set of functions lies in both **O(expression)** and **Omega(expression)**, then Theta notation is used.
- Theta notation (Θ) is a mathematical notation used to describe the exact and balanced behavior of an algorithm's time complexity.
- It represents both the upper and lower bounds of an algorithm's performance, providing a tight range within which the algorithm operates.
- Theta notation is an essential tool in algorithm analysis and is particularly valuable when the upper and lower bounds of an algorithm's time complexity are the same or very close.

Rate of Growth

The following notations are commonly use notations in performance analysis and used to characterize the complexity of an algorithm:

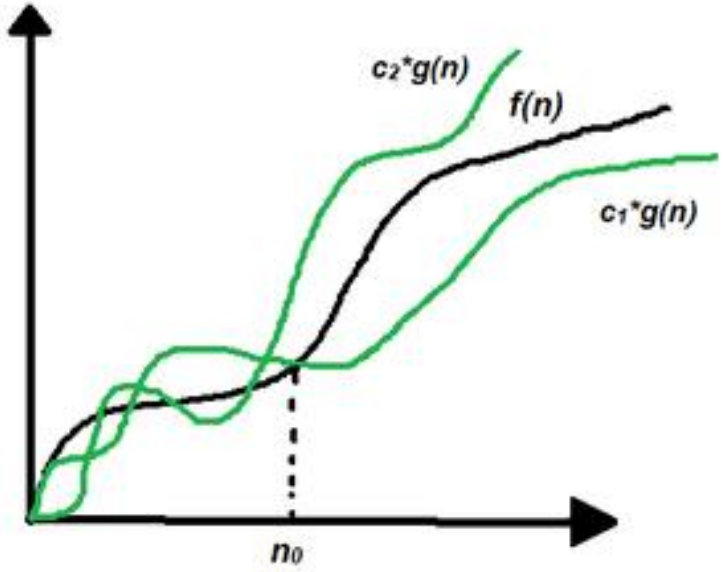
- Big–OH (O)
- Big–OMEGA (Ω)
- Big–THETA (Θ)
- Little–OH (o)
- Little Omega (ω)

The order of complexity

- Linear search is O (n)
- Binary search is O (log n)
- Bubble sort is O (n2)
- Merge sort is O (n log n)

Theta (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.
- 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 ≥ n_0.

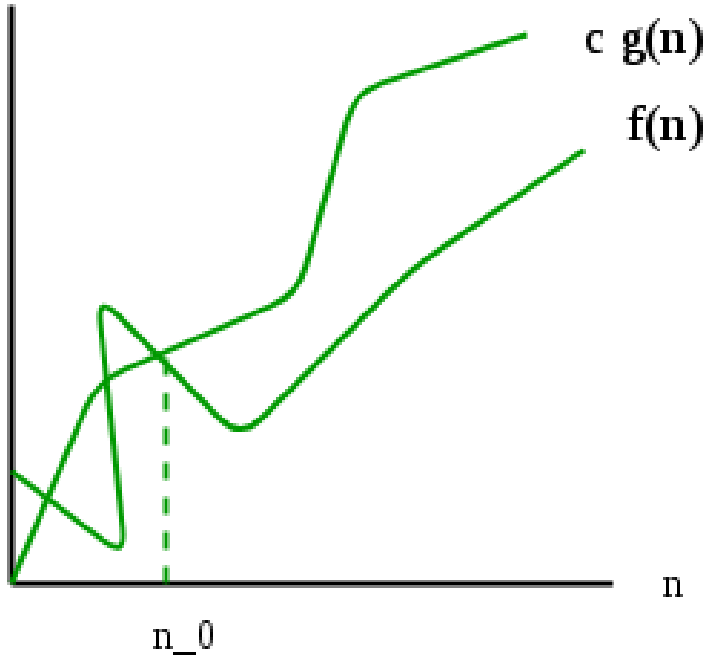


Theta (Θ) Notation

- ***Θ (g(n)) = {f(n): there exist positive constants c1, c2 and n0 such that 0 ≤ c1 * g(n) ≤ f(n) ≤ c2 * g(n) for all n ≥ n_0}***
- The above expression can be described as if f(n) is theta of g(n), then the value f(n) is always between c1 * g(n) and c2 * g(n) for large values of n (n ≥ n0).
- The definition of theta also requires that f(n) must be non-negative for values of n greater than n0.
- A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants.
- For example, Consider the expression **$3n^3 + 6n^2 + 6000 = Θ(n^3)$**

Big-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.
- If f(n) describes the running time of an algorithm, f(n) is O(g(n)) if there exist a positive constant C and n_0 such that, $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$.



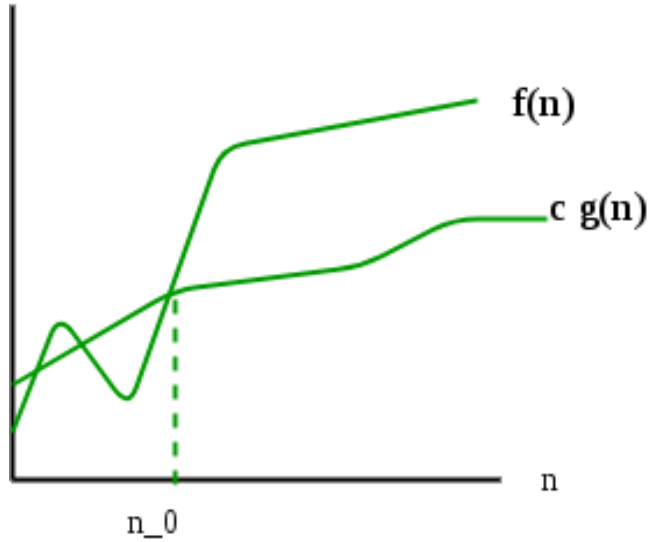
$f(n) = O(g(n))$

Big-O Notation

- O(g(n)) = { f(n): there exist positive constants c and n0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$ }.
- The Big-O notation is useful when we only have an upper bound on the time complexity of an algorithm.
- Many times we easily find an upper bound by simply looking at the algorithm.

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.
- 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 $c*g(n) \leq f(n)$ for all $n \geq n_0$
- $\Omega(g(n)) = \{ f(n)$: there exist positive constants c and n0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$



f(n) = Omega(g(n))

Randomized Algorithms

- An algorithm that uses random numbers to decide what to do next anywhere in its logic is called Randomized Algorithm.
- A **randomized algorithm** is an algorithm that employs a degree of randomness as part of its logic or procedure.
- Makes use of Randomizer (Random number generator).
- Decisions made in the algorithm depends on the output of the randomizer.
- Output and Execution time may vary from run to run for the same input.

Randomized Algorithms

- A randomized algorithm is an algorithm that incorporates randomness or randomization as a fundamental part of its design and operation.
- Unlike traditional algorithms, which produce the same output for a given input every time, randomized algorithms introduce an element of chance into their computations.
- These algorithms use random numbers or random processes to make decisions, perform computations, or achieve specific goals.

Key characteristics

- **Probabilistic Behavior:**
  - Randomized algorithms produce results that are probabilistic or statistical in nature.
  - The outcome of the algorithm may vary each time it is run, but it has a well-defined probability distribution.
- **Efficiency and Approximation:**
  - Randomized algorithms are often used to solve problems that are computationally challenging or difficult to solve deterministically.
  - They may provide approximate solutions or solutions with high confidence.

Key characteristics

- **Random Choices:**
  - Randomized algorithms make random choices at various stages of their execution.
  - These choices may include selecting random elements, flipping coins, or using other sources of randomness.
- **Expected Behavior:**
  - When analyzing randomized algorithms, the focus is on the expected or average behavior over multiple runs.
  - The goal is to achieve desired outcomes with high probability.
- **Applications:**
  - Randomized algorithms find applications in various fields, including computer science, cryptography, optimization, machine learning, and probabilistic data structures.
  - They are particularly useful in situations where deterministic algorithms are too slow or impractical.

Examples

- **Monte Carlo Algorithms:**
  - These algorithms use random sampling to approximate solutions to complex problems, such as estimating the value of mathematical constants or solving optimization problems.
- **Las Vegas Algorithms:**
  - These algorithms use randomness to improve efficiency.
  - They may guarantee that the solution is correct but not bound the running time precisely.
  - Examples include randomized quicksort and some graph algorithms.

Randomized Algorithms

- **Algorithm: Find Repeated Element (a, n)**
- **Description:** This algorithm aims to identify a repeated element within an array a, which spans from index 1 to n.
- **Procedure:**
  - **Initialize Loop:**
    - *Enter an infinite loop.*
  - **Generate Random Indices:**
    - *Generate two random indices, i and j.*
    - *i := Random() mod n + 1*
    - *j := Random() mod n + 1*
  - **Check for Repeated Element:**
    - *If i is not equal to j and a[i] is equal to a[j], indicating the presence of a repeated element:*
  - **Return Index:**
    - *Return the value of i, indicating the index of the repeated element.*


Have a
Good Day!
©WishesMsg.com