What is an algorithm?

An algorithm is a procedure used for solving a problem or performing a computation. Algorithms act as an exact list of instructions that conduct specified actions step by step in either hardware- or software-based routines.

What are different types of algorithms?

There are several types of algorithms, all designed to accomplish different tasks. For example, algorithms perform the following:

- **Search engine algorithm.** This algorithm takes search stringsof keywords and operators as input, searches its associated database for relevant webpages and returns results.
- **Encryption algorithm.** This computing algorithm transforms data according to specified actions to protect it. A symmetric key algorithm, such as the Data Encryption Standard, for example, uses the same keyto encrypt and decrypt data. As long as the algorithm is sufficiently sophisticated, no one lacking the key can decrypt the data.
- **Greedy algorithm.** This algorithm solves optimization problems by finding the locally optimal solution, hoping it is the optimal solution at the global level. However, it does not guarantee the most optimal solution.
- **Recursive algorithm.** This algorithm calls itself repeatedly until it solves a problem. Recursive algorithms call themselves with a smaller value every time a recursive function is invoked.
- **Backtracking algorithm.** This algorithm finds a solution to a given problem in incremental approaches and solves it one piece at a time.
- **Divide-and-conquer algorithm.** This common algorithm is divided into two parts. One part divides a problem into smaller subproblems. The second part solves these problems and then combines them together to produce a solution.

- **Dynamic programming algorithm.** This algorithm solves problems by dividing them into subproblems. The results are then stored to be applied for future corresponding problems.
- **Brute-force algorithm.** This algorithm iterates all possible solutions to a problem blindly, searching for one or more solutions to a function.
- **Sorting algorithm.** Sorting algorithms are used to rearrange data structure based on a comparison operator, which is used to decide a new order for data.
- **Hashing algorithm.** This algorithm takes data and converts it into a uniform message with a hashing
- **Randomized algorithm.** This algorithm reduces running times and time-based complexities. It uses random elements as part of its logic.
    - **Time Complexity:** The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.
    - In order to calculate time complexity on an algorithm, it is assumed that a **constant time c** is taken to execute one operation, and then the total operations for an input length on **N** are calculated. Consider an example to understand the process of calculation: Suppose a problem is to find whether a pair **(X, Y)** exists in an array, A of **N** elements whose sum is **Z**. The simplest idea is to consider every pair and check if it satisfies the given condition or not.
    - The pseudo-code is as follows:
    - `int a[n];`
    - `for(int i = 0;i < n;i++)`
    - `  cin >> a[i]`
    -
    -
    - `for(int i = 0;i < n;i++)`
    - `  for(int j = 0;j < n;j++)`

```
    if(i!=j && a[i]+a[j] == z)
       return true

return false
```

Data Structure And Algorithms Complexity (Big-O)

Big-O notation is a mathematical representation used to describe the complexity of a **data structure and algorithm**. There are two types of Complexity :

1. **Time Complexity:** Its measure based on steps need to follow for an algorithm.
2. **Space Complexity:** It measures the space required to perform an algorithm and data structure.

Data Structure and Algorithm Decision

Data structure and algorithm decisions are based on the complexity of size and operations need to perform on data. Some algorithms perform better on a small set of data while others perform better for big data sets for certain operations.
These days **Space Complexity** is not big concerns but the main performance of an algorithm measures based on **time complexity**. We can make the decision to choose an algorithm based on below performance criteria with respect to Complexity.

Complexity	Performance
O(1)	Excellent
O(log(n))	Good
O(n)	Fair
O(n log(n))	Bad
O(n^2)	Horrible

O(2^n)	Horrible
O(n!)	Horrible

This post almost covers data structure and algorithms **space and time Big-O complexities**. Here you can also see **time complexities** for types of operations like access, search, insertion, and deletion.

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case, and worst case scenario of an algorithm.

Asymptotic analysis is input bound i.e., if there's no input to the algorithm, it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.

Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, the running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. This means the first operation running time will increase linearly with the increase in **n** and the running time of the second operation will increase exponentially when **n** increases. Similarly, the running time of both operations will be nearly the same if **n** is significantly small.

Usually, the time required by an algorithm falls under three types −

- **Best Case** − Minimum time required for program execution.
- **Average Case** − Average time required for program execution.
- **Worst Case** − Maximum time required for program execution.

1. Worst Case Analysis (Mostly used)

In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum

number of operations to be executed. For Linear Search, the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the search() function compares it with all the elements of arr[] one by one. Therefore, the worst-case time complexity of the linear search would be **O(n)**.

3. Average Case Analysis (Rarely used)

In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs. We must know (or predict) the distribution of cases. For the linear search problem, let us assume that all cases are uniformly distributed (including the case of x not being present in the array). So we sum all the cases and divide the sum by (n+1). Following is the value of average-case time complexity.

Average Case Time = \sum_{i=1}^{n}\frac{\theta (i)}{(n+1)} = \frac{\theta (\frac{(n+1)(n+2)}{2})}{(n+1)} = \theta (n)*

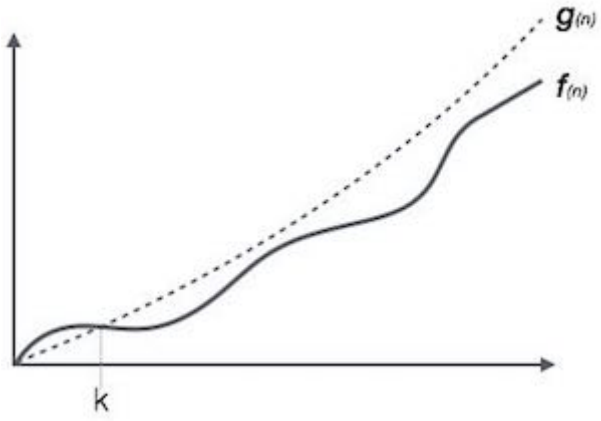
Asymptotic Notations

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- O Notation
- Ω Notation
- θ Notation

Big Oh Notation, O

The notation O(n) is the formal way to express the upper bound of an algorithm's running time. It measures the worst case time complexity or the longest amount of time an algorithm can possibly take to complete.

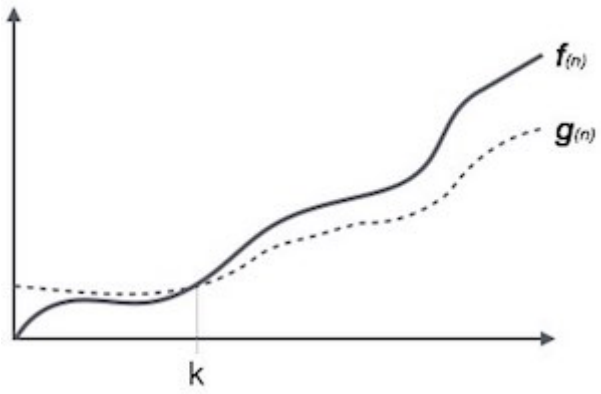


For example, for a function ***f*(n)**

$O(f(n)) = \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } f(n) \leq c.g(n) \text{ for all } n > n_0. \}$

Omega Notation, Ω

The notation Ω(n) is the formal way to express the lower bound of an algorithm's running time. It measures the best case time complexity or the best amount of time an algorithm can possibly take to complete.

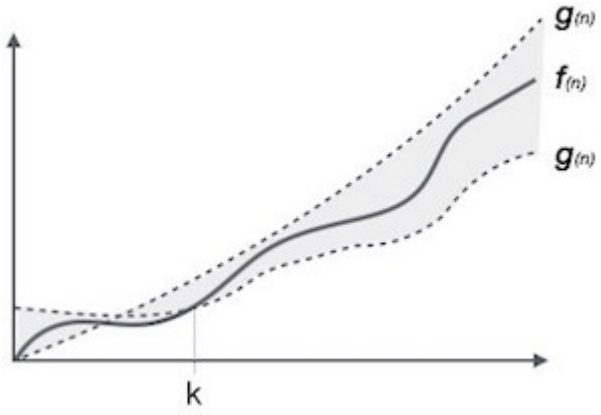


For example, for a function ***f*(n)**

$\Omega(f(n)) \geq \{ g(n) : \text{there exists } c > 0 \text{ and } n_0 \text{ such that } g(n) \leq c.f(n) \text{ for all } n > n_0. \}$

Theta Notation, θ

The notation θ(n) is the formal way to express both the lower bound and the upper bound of an algorithm's running time. It is represented as follows −



$\theta(f(n)) = \{ g(n) \text{ if and only if } g(n) = O(f(n)) \text{ and } g(n) = \Omega(f(n)) \text{ for all } n > n_0. \}$

Common Asymptotic Notations

Following is a list of some common asymptotic notations −

constant	−	O(1)
logarithmic	−	O(log n)
linear	−	O(n)
n log n	−	O(n log n)
quadratic	−	$O(n^2)$
cubic	−	$O(n^3)$
polynomial	−	$n^{O(1)}$
exponential	−	$2^{O(n)}$

Algorithm’s Efficiency and Its Importance.

In computer science, algorithmic efficiency is a characteristic of an algorithm that is related to the number of computational resources required by the algorithm. An algorithm’s resource use must be evaluated, and the efficiency of an algorithm may be assessed based on the use of various resources. For a recurring or continuous process, algorithmic efficiency is comparable to engineering performance. We want to use as few resources as possible to maximize efficiency. However, because various resources, such as time and space complexity, cannot be directly compared, which of the proposed algorithm is judged to be more efficient typically relies on whatever efficiency metric is considered more significant.

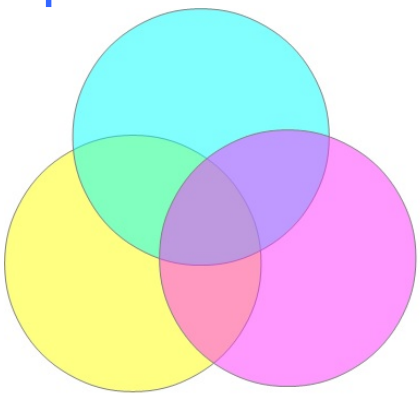
Efficiency Factors

- Space Efficiency
- Time Efficiency

Efficiency Factors

Efficiency is dependent on two factors:

Space Efficiency



Space Efficiency

In some cases, the amount of space/memory consumed has to be examined. For example, in dealing with huge amounts of data or in programming embedded systems, memory consumed must be analyzed. Space/memory usage components are:

- **Space of Instruction:** Compiler, compiler settings, and target machine (CPU), all have an impact on space of instruction.
- **Data Space:** Data size/dynamically allocated memory, static program variables are all factors affecting data space.
- **Stack Space at Runtime:** Compiler, run-time function calls and recursion, local variables, and arguments all have an impact on stack space at runtime.

Time Efficiency

Obviously, the faster a program/function completes its objective, the better the algorithm. The actual running time is determined by a number of factors:

1. **Computer's Speed:** processor (not simply clock speed), I/O, and so on.
1. **Compiler:** The compiler, as well as the compiler parameters.
1. **Amount of Data:** The amount of data, for example, whether to search a lengthy or a short list.

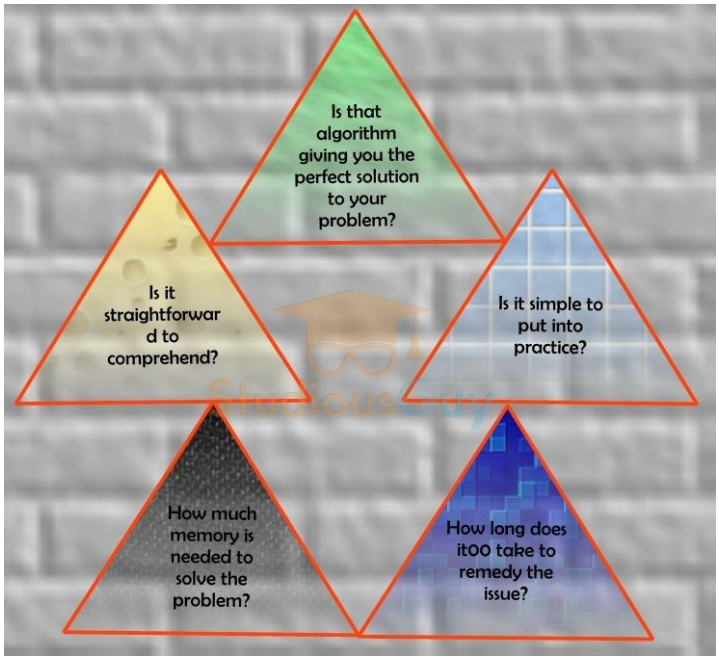
1. **Actual Data:** The actual data, such as whether the name is first or last in a sequential search.

Program Performance Measurement

In computer science, there are numerous algorithms for solving a problem. When there are several different algorithms to solve a problem, we evaluate the performance of all those algorithms. Performance evaluation aids in the selection of the best algorithm from a set of competing algorithms for a given issue. So, we can express algorithm performance as a practice of producing evaluation judgments regarding algorithms.

Index of Article (Click to Jump)

Factors Determining Algorithm's Performance



To compare algorithms, we consider a collection of parameters or components such as the amount of memory required by the algorithm, its execution speed, how easy it is to comprehend and execute, and so on. In general, an algorithm's performance is determined by the following factors:

- Is that algorithm giving you the perfect solution to your problem?
- Is it straightforward to comprehend?
- Is it simple to put into practice?
- How much memory (space) is needed to solve the problem?
- How long does it take to remedy the issue?

When we aim to analyze an algorithm, we just look at space and time requirements of that algorithm and neglect anything else. On the basis of this information, an algorithm's performance may alternatively be described as a technique of determining the space and time requirements of an algorithm.

The following metrics are used to evaluate the performance of an algorithm:

- The amount of space necessary to perform the algorithm's task (Space Complexity). It consists of both program and data space.
- The time necessary to accomplish the algorithm's task (Time Complexity).

Space Complexity

When we create a problem-solving algorithm, it demands the use of computer memory to complete its execution. Memory is necessary for the following purposes in any algorithm:

- To keep track of software instructions.
- To keep track of constant values.
- To keep track of variable values.
- Additionally, for a few additional things such as function calls, jump statements, and so on.

When software is running, it often utilizes computer memory for the following reasons:

- The amount of memory needed to hold compiled versions of instructions, which is referred to as instruction space.
- The amount of memory utilized to hold information about partially run functions at the time of a function call, which is known as the environmental stack.
- The amount of memory needed to hold all of the variables and constants, which is referred to as data space.

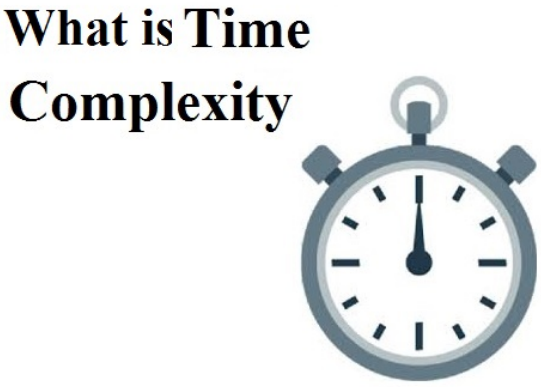
Example

We need to know how much memory is required to store distinct datatype values to compute the space complexity (according to the compiler). Take a look at the following code:

```
int square(int a)

{

return a*a;

}
```

In the preceding piece of code, variable 'a' takes up 2 bytes of memory, and the return value takes up another 2 bytes, i.e., it takes a total of 4 bytes of memory to finish its execution, and for any input value of 'a', this 4 byte memory is fixed. Constant Space Complexity is the name given to this type of space complexity.

Time Complexity



Every algorithm needs a certain amount of computer time to carry out its instructions and complete the operation. The amount of computer time required is referred to as time complexity. In general, an algorithm's execution time is determined by the following:

- It doesn't matter if it's on a single processor or a multi-processor computer.
- It doesn't matter if it's a 32-bit or 64-bit computer.
- The machine's read and write speeds.
- The time taken by an algorithm to complete arithmetic, logical, return value, and assignment operations, among other things.
- Data to be entered

Example

Calculating an algorithm's Time Complexity based on the system configuration is a challenging undertaking since the configuration varies from one system to the next. We must assume a model machine with a certain setup to tackle this challenge. As a result, we can compute generalized time complexity using that model machine. Take a look at the following code:

```
int sum(int a, int b)

{

return a+b;

}
```

In the following example code, calculating a+b takes 1 unit of time, and returning the value takes 1 unit of time, i.e., it takes two units of time to perform the task, and it is unaffected by the input values for a and b. This indicates it takes the same amount of time for all input values, i.e., 2 units.

Recursion

Python also accepts function recursion, which means a defined function can call itself.

Recursion is a common mathematical and programming concept. It means that a function calls itself. This has the benefit of meaning that you can loop through data to reach a result.

Syntax:

```
def func(): <--
               |
               | (recursive call)
               |
    func() ----
```

Example 1: A Fibonacci sequence is the integer sequence of 0, 1, 1, 2, 3, 5, 8….

Substitution Method

In the substitution method, we have a known recurrence, and we use induction to prove that our guess is a good bound for the recurrence's solution. This method works well in providing us with a good upper bound in most recurrences that can't be solved using the Master's Theorem or other more straightforward ways.
Practising similar questions is essential if we try to make a guess that'll probably work for us. We'll cover some examples to get you started in the next section.
The steps to use the Substitution method are as follows.

1. Guess a solution through your experience.
2. Use induction to prove that the guess is an upper bound solution for the given recurrence relation.

Recursion Tree Method

1. Recursion Tree Method is a pictorial representation of an iteration method which is in the form of a tree where at each level nodes are expanded.

2. In general, we consider the second term in recurrence as root.

3. It is useful when the divide & Conquer algorithm is used.

4. It is sometimes difficult to come up with a good guess. In Recursion tree, each root and child represents the cost of a single subproblem.

5. We sum the costs within each of the levels of the tree to obtain a set of pre-level costs and then sum all pre-level costs to determine the total cost of all levels of the recursion.

6. A Recursion Tree is best used to generate a good guess, which can be verified by the Substitution Method.

What is DSA?

The term DSA stands for **Data Structures and Algorithms**. As the name itself suggests, it is a combination of two separate yet interrelated topics – Data Structure and Algorithms.

What is Data Structure?

A data structure is defined as a particular way of storing and organizing data in our devices to use the data efficiently and effectively. The main idea behind using data structures is to minimize the time and space complexities. An efficient data structure takes minimum memory space and requires minimum time to execute the data.

What is Algorithm?

Algorithm is defined as a process or set of well-defined instructions that are typically used to solve a particular group of problems or perform a specific type of calculation. To explain in simpler terms, it is a set of operations performed in a step-by-step manner to execute a task.