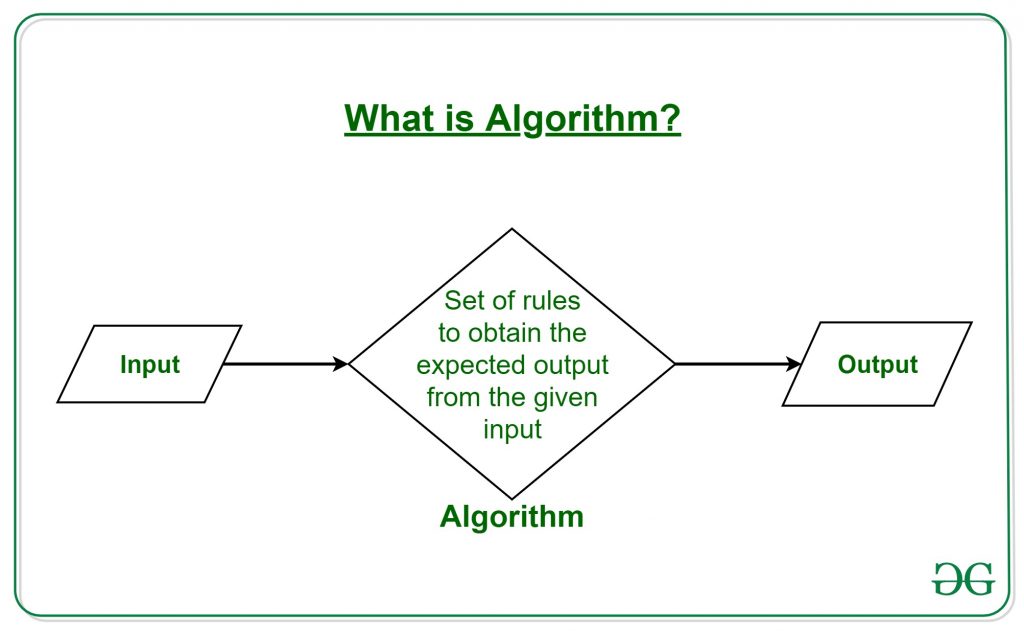
What is an Algorithm?

* The word Algorithm means “a process or set of rules to be followed in calculations or other problem-solving operations”.
* Therefore Algorithm refers to a set of rules/instructions that step-by-step define how a work is to be executed upon in order to get the expected results.
* Flowchart:



* It can be understood by taking an example of cooking a new recipe.
  + To cook a new recipe, one reads the instructions and steps and execute them one by one, in the given sequence.
  + The result thus obtained is the new dish cooked perfectly.
  + Similarly, algorithms help to do a task in programming to get the expected output.
* The Algorithm designed are language-independent, i.e. they are just plain instructions that can be implemented in any language, and yet the output will be the same, as expected.

Analysis of Algorithm.

* In computer science, the analysis of algorithms is the process of finding the computational complexity of algorithms—the amount of time, storage, or other resources needed to execute them.
* Usually, this involves determining a function that relates the size of an algorithm's input to the number of steps it takes (its time complexity) or the number of storage locations it uses (its space complexity).
* An algorithm is said to be efficient when this function's values are small, or grow slowly compared to a growth in the size of the input.
* Different inputs of the same size may cause the algorithm to have different behavior, so best, worst and average case descriptions might all be of practical interest.
* When not otherwise specified, the function describing the performance of an algorithm is usually an upper bound, determined from the worst case inputs to the algorithm.
* The term "analysis of algorithms" was coined by Donald Knuth.
* Algorithm analysis is an important part of a broader computational complexity theory, which provides theoretical estimates for the resources needed by any algorithm which solves a given computational problem.
* These estimates provide an insight into reasonable directions of search for efficient algorithms.
* In theoretical analysis of algorithms it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big O notation, Big-omega notation and Big-theta notation are used to this end.

Complexity of the algorithm!

**Complexities of an Algorithm**

* The complexity of an algorithm computes the amount of time and spaces required by an algorithm for an input of size (n).
* The complexity of an algorithm can be divided into two types. The time complexity and the space complexity.

**Time Complexity of an Algorithm**

* The time complexity is defined as the process of determining a formula for total time required towards the execution of that algorithm.
* This calculation is totally independent of implementation and programming language.

**Space Complexity of an Algorithm**

* Space complexity is defining as the process of defining a formula for prediction of how much memory space is required for the successful execution of the algorithm.
* The memory space is generally considered as the primary memory.

Running time analysis

* 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*(n2). 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.

How to compare algorithm

“Quality” of an Algorithm

When it comes to different outfits or gadgets we have some criteria upon which we evaluate them. So certainly, we want to define some parameters upon which we can compare two algorithms. The two most important factors are:

Time: Ever since the invention of computers, programs are written to reduce the time taken to achieve a job. Hence, an algorithm is considered better if it runs faster than the other one.

Space: Memory is cheap now days, but everyone loves an application that takes the least amount of RAM. Don’t we all hate when you have 70 tabs open, and a web browser slows down your entire PC. Also, when writing system level code, every extra byte of memory matters. Thus, an algorithm which takes up less RAM would be considered to be a better one.

Experimental:

A fool proof way to compare 2 different algorithms would be to actually run them and observe the results. The one which gives you the output in less time would said to be the better one.

But, when running these algorithms, you need to ensure that we are using the same hardware for both. A faster processor can give better results for a poorly written algorithm. Also, you cannot rely on a single run. Your computer might be running other workloads in the background and that can affect the performance. Hence, it is recommended that you several iterations of both the techniques and then get an average value before comparing.

We also need to make sure that the input workload for the algorithms remain the same. Using random input data cannot give us reliable results. It could be possible that certain input loads work unreasonably fast. Example: Trying to sort an already sorted list.

Even with all these conditions in mind, you cannot certainly reach an exact result, as it is scientifically impossible to run 2 algorithms in the exact same conditions. You wouldn’t be able to compare two algorithms which work nearly the same. Thus, you need to do a little deep dive in the actual flow of the algorithm.

Analytical:

Your algorithm may contain a lot of conditional blocks, and control blocks that determine the runtime of your algorithm. Suppose you are given with 2 code blocks. Forget about the implementation details, and assume that both of them give the correct answer.

**Algorithm 1:**

**for**(i = 0; i < 10; i++) {

**for**(j = 0; j< 5; j++) {

// do something...

}

}

**Algorithm 2:**

**for**(i = 0; i < 10; i++) {

// do something...

}

**for**(j = 0; j < 5; j++) {

// do something...

}

Let us analyze both the algorithms:

Algorithm 1:

We are starting a loop i that runs for 10 times.

Inside the loop we are starting one more loop j that runs for 5 times.

This means that for each iteration of i, we run j 5 times.

Hence, the total number of times we do something is 10 \* 5 = 5010∗5=50 times.

Algorithm 2:

We start a loop i that runs for 10 times.

Now we do something for 10 times.

Exit the loop.

We start another loop j that runs for 5 times.

Now we do something for 5 times.

Hence, the total number of times we do something was 10 + 5 = 1510+5=15 times

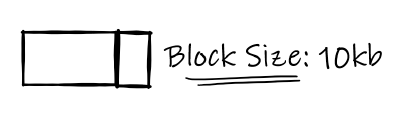
What did we infer from this?

At the first glance it may seem that both the codes have 2 loops. One runs for 10 times and the other runs for 5 times. So, it is natural to feel that both will take the same time to run. But when we compare both of them analytically, we see that algorithm 1 is doing something for 50 times, while algorithm 2 is doing something for just 15 times. This means that algorithm 1 would be a better choice.

This phenomenon is also known as Runtime Complexity.

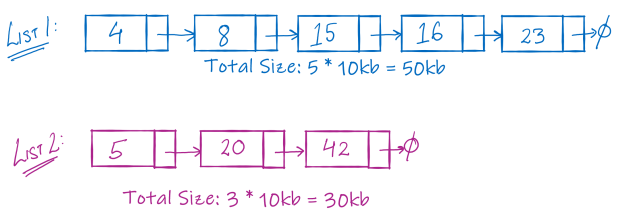
**Space Consumption:**

We just discussed in our quality criteria that ‘space’ is an important factory to determine which algorithm performs better. What do you mean by that? Let us understand this with a very basic example. To set up some context, we would consider the block size of a single node in a [Linked List](https://studyalgorithms.com/link_list/what-is-a-linked-list/) as 10kb.

Fig: Block size of a Linked List node.

**Question**: You are given with 2 linked lists, that are already sorted. Merge the two lists and return a single sorted list.

According to the question, we are given with 2 lists. Each node in the linked list would have a block size of 10kb, and hence the setup would look something like this:

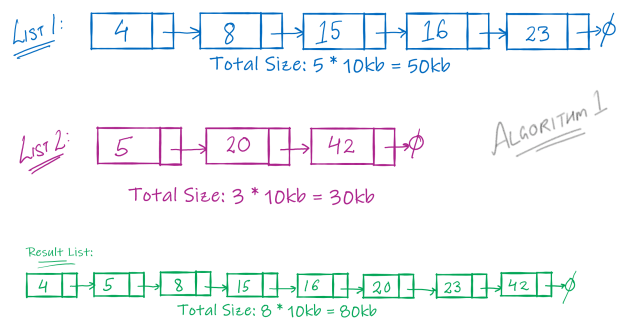
Fig: 2 lists and their total sizes

The size of first list would be 5 \* 10 = 50kb5∗10=50*kb*, and the size of the second list would be 3 \* 10 = 30kb3∗10=30*kb*. Hence, currently you are using 80kb of memory. Let us skip the implementation details and look at 2 algorithms to solve this problem.

**Algorithm 1:**

* Create a new list of 5 + 3 = 85+3=8 nodes
* Sort the two lists
* Fill up the new list
* Return the result.

This algorithm would look something like:

Fig: Merging 2 sorted linked lists with extra space

In the above algorithm, we create a new list that takes up **extra** 80kb of space. Hence, the total space required to run this algorithm would be:

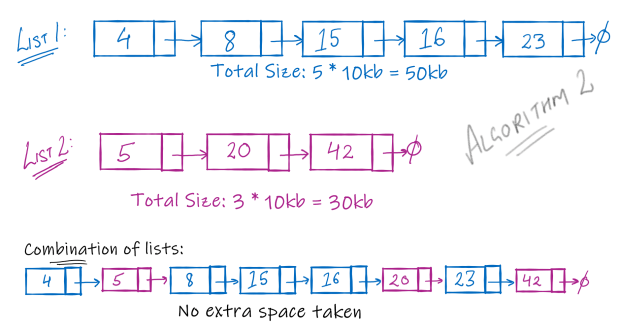
50 (list 1) + 30 (list 2) + 80 (new list) = 160kb

Thus, we need a total of 160kb memory for algorithm 1 to run successfully. Let us look at another approach to solve the same problem.

**Algorithm 2:**

* Scan the shorter list one by one.
* Take up an element.
* Scan the longer list and **insert** it at the right place.
* Repeat the process for all the elements of the smaller list.

This algorithm when completed would give us a result something like:

Fig: Merging 2 sorted linked lists without taking extra space.

You see, that in the above algorithm, we do not consume any extra space. We were clever enough and we saved space. This algorithm would just need 80kb of space to run. That is just the space required by the two already present lists.

**What did we infer from this?**

When you have to compare these 2 algorithms, we can easily say that algorithm 2 is the clear winner when it comes to saving space. 80kb of extra space may seem very small, but consider writing codes for gigantic lists. You should always compare your methods to write the code keeping space consumed in mind.

This is also known as the **Space Complexity**.

What is Data Structures

What are Data Structures?

Data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

Depending on your requirement and project, it is important to choose the right data structure for your project. For example, if you want to store data sequentially in the memory, then you can go for the Array data structure.

A data structure is a specialized format for organizing, processing, retrieving and storing data. There are several basic and advanced types of data structures, all designed to arrange data to suit a specific purpose. Data structures make it easy for users to access and work with the data they need in appropriate ways. Most importantly, data structures frame the organization of information so that machines and humans can better understand it.

In computer science and computer programming, a data structure may be selected or designed to store data for the purpose of using it with various algorithms. In some cases, the algorithm's basic operations are tightly coupled to the data structure's design. Each data structure contains information about the data values, relationships between the data and -- in some cases -- functions that can be applied to the data.

For instance, in an object-oriented programming language, the data structure and its associated methods are bound together as part of a class definition. In non-object-oriented languages, there may be functions defined to work with the data structure, but they are not technically part of the data structure.

Big O Notation

Big O Notation is a way to measure an algorithm’s efficiency. It measures the time it takes to run your function as the input grows. Or in other words, how well does the function scale.

There are two parts to measuring efficiency — time complexity and space complexity. Time complexity is a measure of how long the function takes to run in terms of its computational steps. Space complexity has to do with the amount of memory used by the function. This blog will illustrate time complexity with two search algorithms.

Big O is sometimes referred to as the algorithm’s upper bound, meaning that it deals with the worst-case scenario. The best-case scenario doesn’t really tell us anything — it will be finding our item in the first pass. We use worst-case to remove uncertainty — the algorithm will never perform worse than we expect.

There are often many choices of writing code to get to a solution. Understanding Big O notation is important in writing algorithms. It helps you determine when your algorithm is getting faster or slower. You can also compare different methods and choose the most efficient.