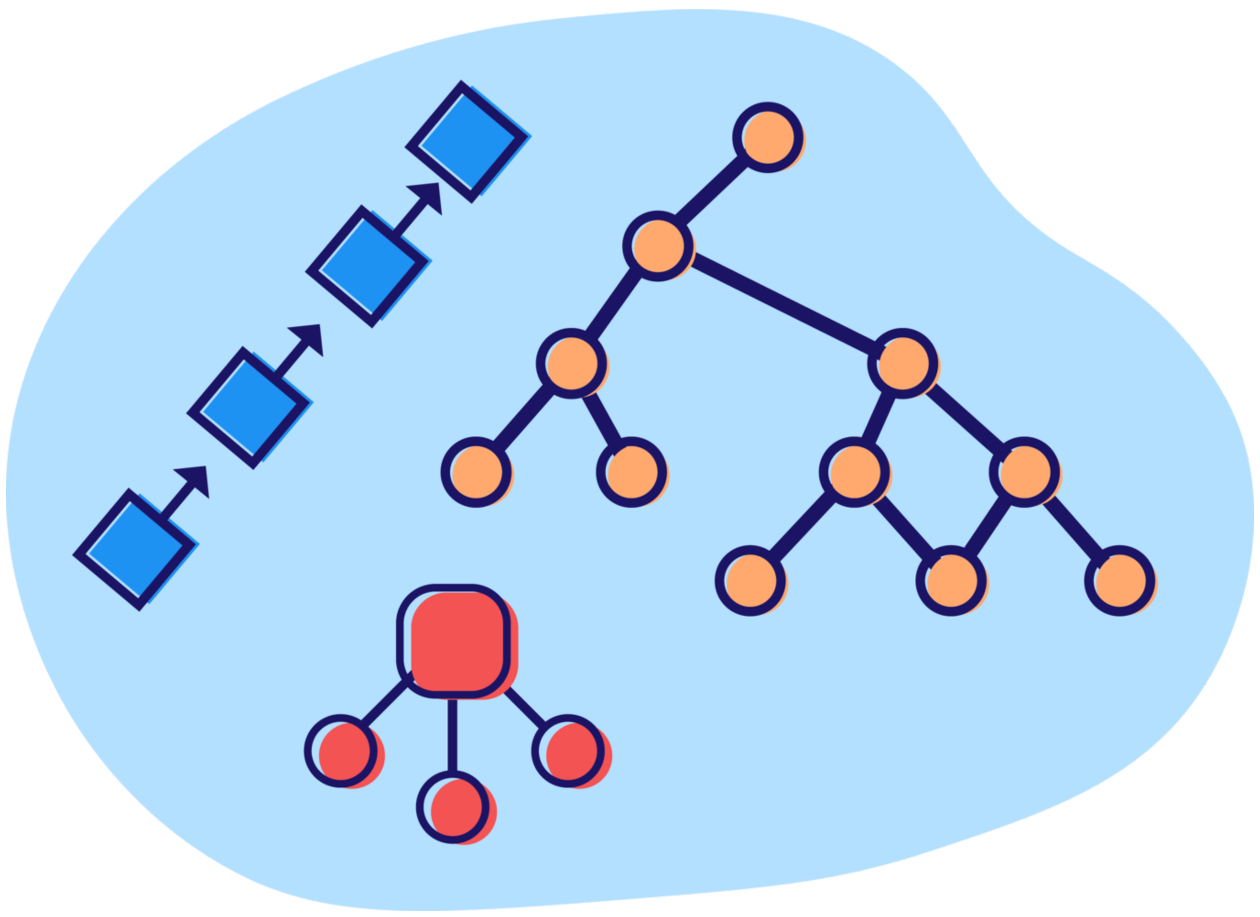


Data Structures & Algorithms

Chapter - 02

Basics Of Data Structures & Algorithms

Rakin Mohammad Sifullah
Computer Science Engineer

Overview

In this chapter, we will cover the basic understanding of Algorithms.

We will cover -

1. What is an Algorithm?
2. Categories of Algorithms
3. What are the characteristics of Algorithms
4. How to write Algorithms?
5. How to analyze Algorithms
6. What is Algorithm Complexity
7. What is Time Complexity?
8. What is Space Complexity
9. Asymptotic analysis
10. Greedy algorithms
11. Divide and Conquer

What is an Algorithm?

An algorithm is a step-by-step procedure that defines a set of instructions to be executed in a particular order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

Categories of Algorithms

From the data structure point of view, the following are some important categories of algorithms:

1. **Search** − Algorithm to search an item in a data structure.
2. **Sort** − Algorithm to sort items in a certain order.
3. **Insert** − Algorithm to insert an item in a data structure.
4. **Update** − Algorithm to update an existing item in a data structure.
5. **Delete** − Algorithm to delete an existing item from a data structure.

Characteristics of an Algorithm

Not all procedures can be called an algorithm. An algorithm should have the following characteristics:

1. **Unambiguous** − The algorithm should be clear and unambiguous. Each of its steps (or phases), and their inputs/outputs should be clear and must lead to only one meaning.
2. **Input** − An algorithm should have 0 or more well-defined inputs.
3. **Output** − An algorithm should have 1 or more well-defined outputs and should match the desired output.
4. **Finiteness** − Algorithms must terminate after a finite number of steps.
5. **Feasibility** − This should be feasible with the available resources.
6. **Independent** − An algorithm should have step-by-step directions, which should be independent of any programming code.

How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource-dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

Example

Let's try to learn algorithm writing by using an example.

Problem − Design an algorithm to add two numbers and display the result.

```
Step 1 − START

Step 2 − declare three integers a, b & c

Step 3 − define values of a & b

Step 4 − add values of a & b

Step 5 − store output of step 4 to c

Step 6 − print c

Step 7 − STOP
```

In the design and analysis of algorithms, usually, the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm by ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing step numbers is optional.

We design an algorithm to get a solution to a given problem. A problem can be solved in more than one way.

How to analyze Algorithms

The efficiency of an algorithm can be analyzed at two different stages, before implementation, and after implementation. They are the following −

1. ***A Priori* Analysis** − This is a theoretical analysis of an algorithm. The efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
2. ***A Posterior* Analysis** − This is an empirical analysis of an algorithm. The selected algorithm is implemented using a programming language. This is then executed on the target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about *a priori* algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

What is Algorithm Complexity

Suppose X is an algorithm and n is the size of input data, the time and space used by the algorithm X are the two main factors, which decide the efficiency of X.

1. **Time Factor** − Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
2. **Space Factor** − Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm f(n) gives the running time and/or the storage space required by the algorithm in terms of n as the size of input data.

What is Time Complexity

The time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function T(n), where T(n) can be measured as the number of steps, provided each step consumes constant time.

For example, the addition of two n-bit integers takes n steps. Consequently, the total computational time is $T(n) = c * n$, where c is the time taken for the addition of two bits. Here, we observe that T(n) grows linearly as the input size increases.

What is Space Complexity

The space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components −

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity S(P) of any algorithm P is S(P) = C + SP(I), where C is the fixed part and S(I) is the variable part of the algorithm, which depends on instance characteristic I. Following is a simple example that tries to explain the concept −

Algorithm: SUM(A, B)

```
Step 1 -  START
Step 2 -  C ← A + B + 10
Step 3 -  Stop
```

Here we have three variables A, B, and C and one constant. Hence S(P) = 1 + 3. Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

Asymptotic analysis:

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

1. **Best Case** − Minimum time required for program execution.
2. **Average Case** − Average time required for program execution.
3. **Worst Case** − Maximum time required for program execution.

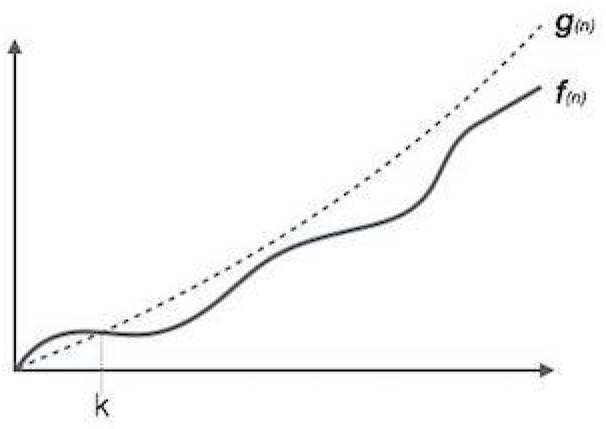
Asymptotic Notations

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

1. **O Notation**
2. **Ω Notation**
3. **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.

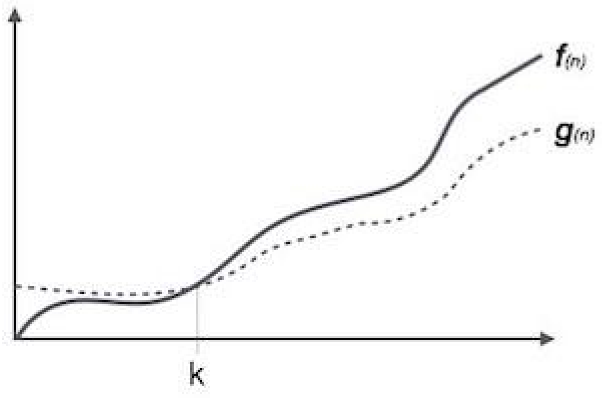


For example, for a function $f(n)$

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

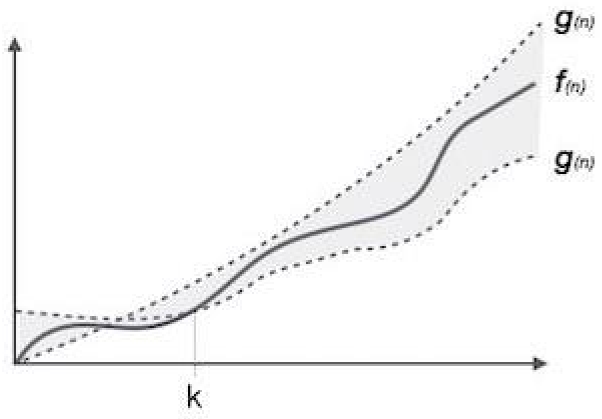


For example, for a function *f*(n)

Ω(*f*(n)) ≥ { *g*(n) : there exists c > 0 and n_0 such that *g*(n) ≤ c.*f*(n) 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 −



θ(*f*(n)) = { *g*(n) if and only if *g*(n) = O(*f*(n)) and *g*(n) = Ω(*f*(n)) 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	−	$2^{O(n)}$

Greedy Algorithms

An algorithm is designed to achieve the optimum solution for a given problem. In the greedy algorithm approach, decisions are made from the given solution domain. As greedy, the closest solution that seems to provide an optimum solution is chosen.

Greedy algorithms try to find a localized optimum solution, which may eventually lead to globally optimized solutions. However, generally greedy algorithms do not provide globally optimized solutions.

Counting Coins

This problem is to count to the desired value by choosing the least possible coins and the greedy approach forces the algorithm to pick the largest possible coin. If we are provided coins of 1, 2, 5, and 10 and we are asked to count ₹ 18 then the greedy procedure will be −

- 1 − Select one 10 coin, and the remaining count is 8
- 2 − Then select one 5 coins, the remaining count is 3
- 3 − Then select one 2 coins, the remaining count is 1
- 4 − And finally, the selection of one 1 coins solves the problem

Though, it seems to be working fine, for this count we need to pick only 4 coins. But if we slightly change the problem then the same approach may not be able to produce the same optimum result.

For the currency system, where we have coins of 1, 7, and 10 value, counting coins for value 18 will be absolutely optimum but for counting like 15, it may use more coins than necessary. For example, the greedy approach will use 10 + 1 + 1 + 1 + 1 + 1, total of 6 coins. Whereas the same problem could be solved by using only 3 coins (7 + 7 + 1)

Hence, we may conclude that the greedy approach picks an immediate optimized solution and may fail where global optimization is a major concern.

Examples

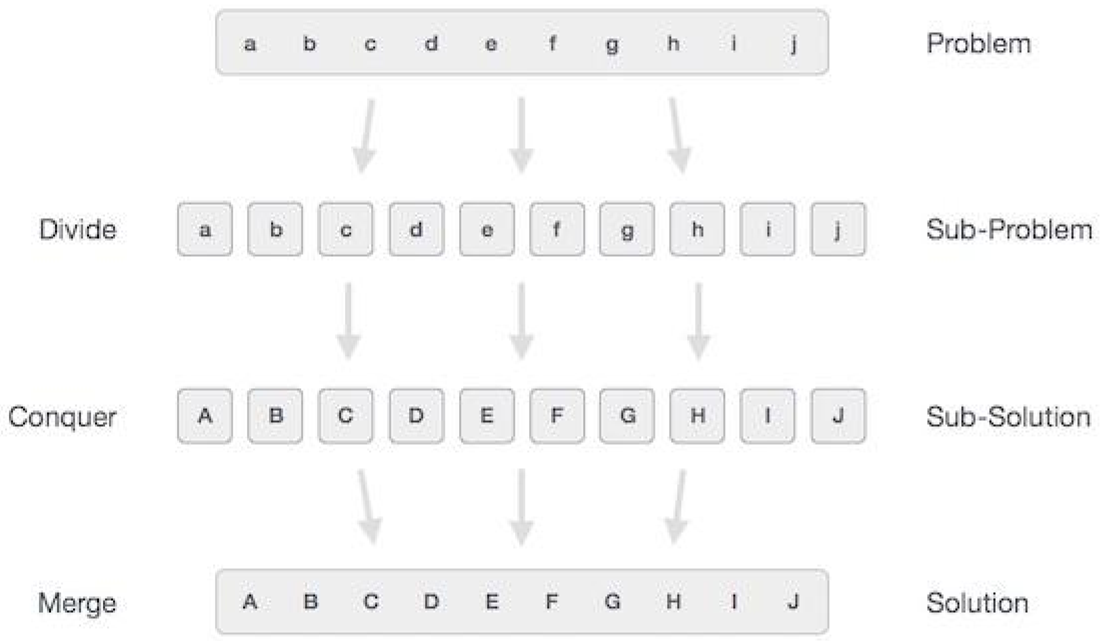
Most networking algorithms use the greedy approach. Here is a list of a few of them −

1. Travelling Salesman Problem
2. Prim's Minimal Spanning Tree Algorithm
3. Kruskal's Minimal Spanning Tree Algorithm
4. Dijkstra's Minimal Spanning Tree Algorithm
5. Graph - Map Coloring
6. Graph - Vertex Cover
7. Knapsack Problem
8. Job Scheduling Problem

There are lots of similar problems that use the greedy approach to find an optimum solution.

Divide and Conquer

In the divide and conquer approach, the problem in hand is divided into smaller sub-problems, and then each problem is solved independently. When we keep on dividing the subproblems into even smaller sub-problems, we may eventually reach a stage where no more division is possible. Those "atomic" smallest possible sub-problem (fractions) are solved. The solution of all sub-problems is finally merged in order to obtain the solution of an original problem.



Broadly, we can understand the divide-and-conquer approach in a three-step process.

Divide/Break

This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.

Conquer/Solve

This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.

Merge/Combine

When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution to the original problem. This algorithmic approach works recursively and conquers & merges steps works so close that they appear as one.

Examples

The following computer algorithms are based on the divide-and-conquer programming approach:

1. **Merge Sort**
2. **Quick Sort**
3. **Binary Search**
4. **Strassen's Matrix Multiplication**
5. **Closest pair (points)**

There are various ways available to solve any computer problem, but the mentioned are good examples of the divide and conquer approach.

Dynamic Programming

Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike, divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

Dynamic programming is used where we have problems, which can be divided into similar sub-problems so that their results can be re-used. Mostly, these algorithms are used for optimization. Before solving the in-hand sub-problem, the dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So we can say that −

1. The problem should be able to be divided into smaller overlapping sub-problem.
2. An optimum solution can be achieved by using an optimum solution of smaller sub-problems.
3. Dynamic algorithms use Memoization.

Comparison

In contrast to greedy algorithms, where local optimization is addressed, dynamic algorithms are motivated for the overall optimization of the problem.

In contrast, to divide and conquer algorithms, where solutions are combined to achieve an overall solution, dynamic algorithms use the output of a smaller sub-problem and then try to optimize a bigger sub-problem. Dynamic algorithms use Memoization to remember the output of already solved sub-problems.

Example

The following computer problems can be solved using a dynamic programming approach-

1. Fibonacci number series
2. Knapsack problem
3. Tower of Hanoi
4. All pair shortest path by Floyd-Warshall
5. Shortest path by Dijkstra
6. Project scheduling

Dynamic programming can be used in both top-down and bottom-up manner. And of course, most of the time, referring to the previous solution output is cheaper than recomputing in terms of CPU cycles.