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.

Time Factor − The time is measured by counting the number of key operations such as comparisons in sorting algorithm

Space Factor − The 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 storage space required by the algorithm in terms of n as the size of input data.

Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. 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 & constant 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.

Time Complexity

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, 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 addition of two bits. Here, we observe that **T(n)** grows linearly as input size increases.

Asymptotic analysis of an algorithm, refers to defining the mathematical boundary/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.

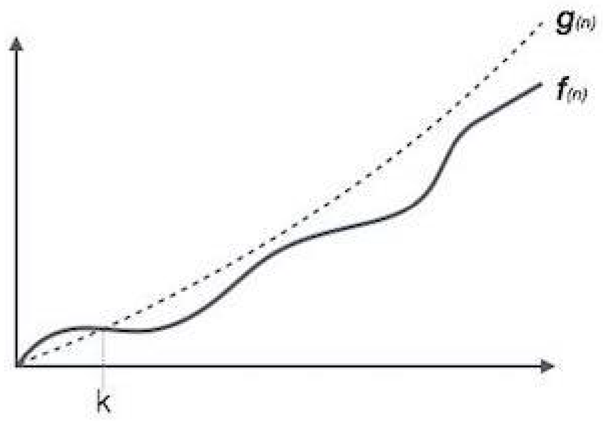
Asymptotic analysis is input bound i.e., if there's no input to the algorithm it is concluded to work in a constant time. Other than the "input" all other factors are considered constant.
Asymptotic analysis refers to computing the running time of any operation in mathematical units of computation. For example, running time of one operation is computed as $f(n)$ and may be for another operation it is computed as $g(n^2)$. Which means first operation running time will increase linearly with the increase in n and running time of second operation will increase exponentially when n increases. Similarly the running time of both operations will be nearly same if n is significantly small.
Usually, 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.

Asymptotic Notations
Following are commonly used asymptotic notations used in calculating running time complexity of an algorithm.

i. O Notation
ii. Ω Notation
iii. θ Notation

Big Oh Notation, O
The 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 longest amount of time an algorithm can possibly take to complete.

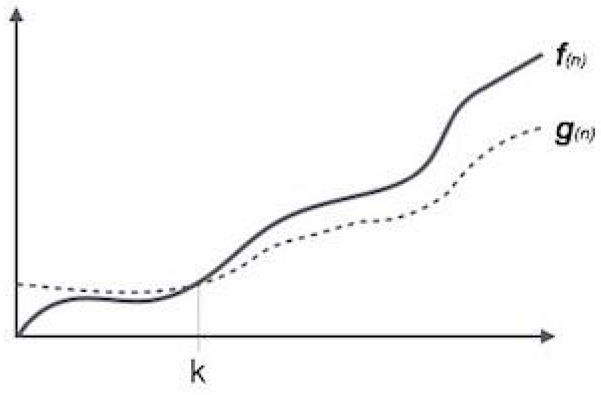


For example, for a function $f(n)$

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

Omega Notation, Ω

The 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 best amount of time an algorithm can possibly take to complete.

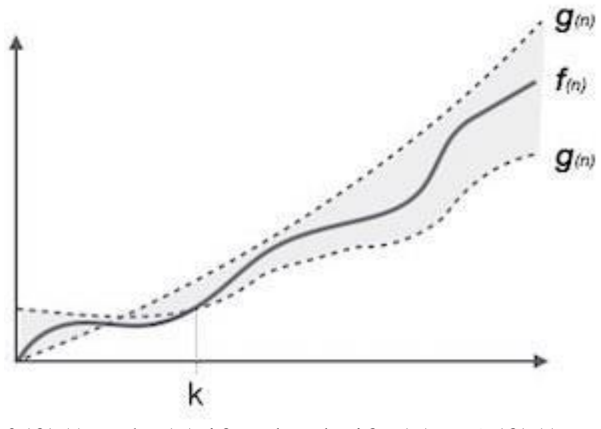


For example, for a function *f*(n)

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

Theta Notation, θ

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



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

Common Asymptotic Notations

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	−	$2^{O(n)}$

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

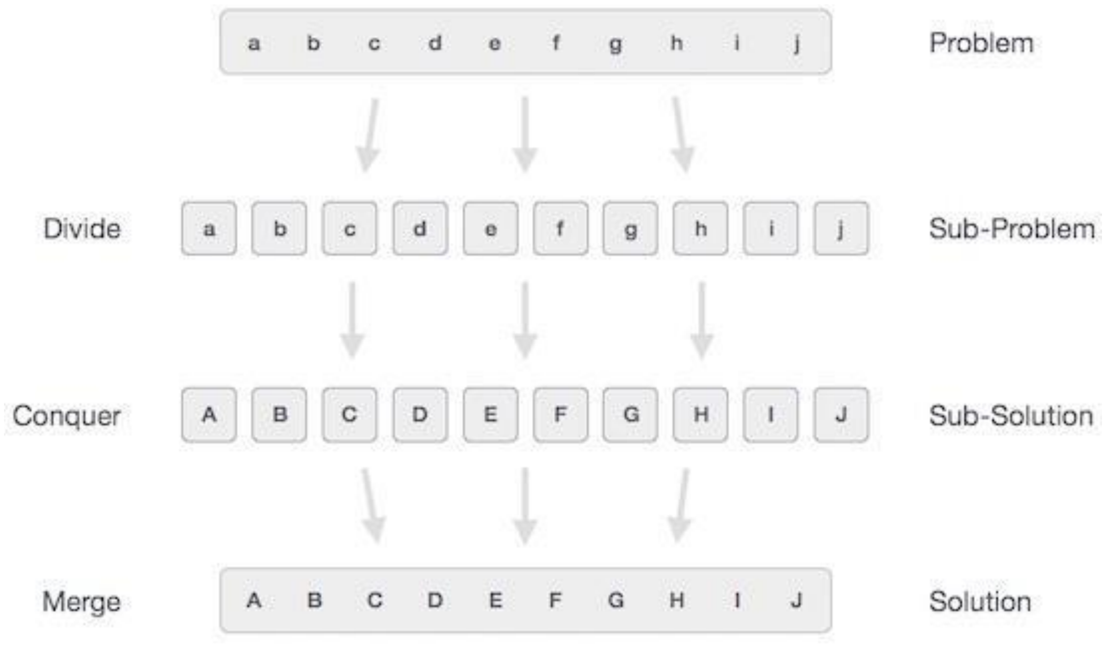
Greedy algorithms tries to find localized optimum solution which may eventually land in globally optimized solutions. But generally greedy algorithms do not provide globally optimized solutions.

Counting Coins
This problem is to count to a desired value by choosing least possible coins and 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, remaining count is 8
2 − Then select one € 5 coin, remaining count is 3
3 − Then select one € 2 coin, remaining count is 1
4 − And finally 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 currency system, where we have coins of 1, 7, 10 value, counting coins for value 18 will be absolutely optimum but for count like 15, it may use more coins than necessary. For example − greedy approach will use 10 + 1 + 1 + 1 + 1 + 1 total 6 coins. Where the same problem could be solved by using only 3 coins (7 + 7 + 1)
Hence, we may conclude that greedy approach picks immediate optimized solution and may fail where global optimization is major concern.

Examples
Most networking algorithms uses greedy approach. Here is the list of few of them −
- ✓ Travelling Salesman Problem
- ✓ Prim's Minimal Spanning Tree Algorithm
- ✓ Kruskal's Minimal Spanning Tree Algorithm
- ✓ Dijkstra's Minimal Spanning Tree Algorithm
- ✓ Graph - Map Coloring
- ✓ Graph - Vertex Cover
- ✓ Knapsack Problem
- ✓ Job Scheduling Problem

These and there are lots of similar problems which uses greedy approach to find an optimum solution.
In 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 sub-problems into even smaller sub-problems, we may eventually reach at 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 original problem.



Broadly, we can understand **divide-and-conquer** approach as three step process.

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

Conquer/Solve
This step receives lots of smaller sub-problem to be solved. Generally at this level, problems are considered 'solved' on their own.

Merge/Combine
When the smaller sub-problems are solved, this stage recursively combines them until they formulate solution of the original problem.

This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Examples

The following computer algorithms are based on **divide-and-conquer** programming approach −

- ✓ Merge Sort
- ✓ Quick Sort
- ✓ Binary Search
- ✓ Strassen's Matrix Multiplication
- ✓ Closest pair (points)

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

Dynamic programming approach is similar to divide and conquer in breaking down the problem in 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 in 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, dynamic algorithm will try to examine the results of previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best solution.

So we can say that −

- The problem should be able to be divided in to smaller overlapping sub-problem.
- The optimum solution can be achieved by using optimum solution of smaller sub-problems.
- Dynamic algorithms use memoization.

Comparison

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

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

Example

The following computer problems can be solved using dynamic programming approach −

- ✓ Fibonacci number series
- ✓ Knapsack problem
- ✓ Tower of Hanoi
- ✓ All pair shortest path by Floyd-Warshall
- ✓ Shortest path by Dijkstra
- ✓ Project scheduling

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

Data Structure is a way to organized data in such a way that it can be used efficiently. This tutorial explains basic terms related to data structure.

Data Definition

Data Definition defines a particular data with following characteristics.

- **Atomic** − Definition should define a single concept
- **Traceable** − Definition should be able to be mapped to some data element.
- **Accurate** − Definition should be unambiguous.
- **Clear and Concise** − Definition should be understandable.

Data Object

Data Object represents an object having a data.

Data Type

Data type is way to classify various types of data such as integer, string etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data. Data type of two types −

- Built-in Data Type
- Derived Data Type

Built-in Data Type

Those data types for which a language has built-in support are known as Built-in Data types. For example, most of the languages provides following built-in data types.

- Integers
- Boolean (true, false)
- Floating (Decimal numbers)
- Character and Strings

Derived Data Type

Those data types which are implementation independent as they can be implemented in one or other way are known as derived data types. These data types are normally built by combination of primary or built-in data types and associated operations on them. For example −

- List
- Array
- Stack
- Queue

Basic Operations

The data in the data structures are processed by certain operations. The particular data structure chosen largely depends on the frequency of the operation that needs to be performed on the data structure.

- Traversing
- Searching
- Insertion
- Deletion
- Sorting
- Merging