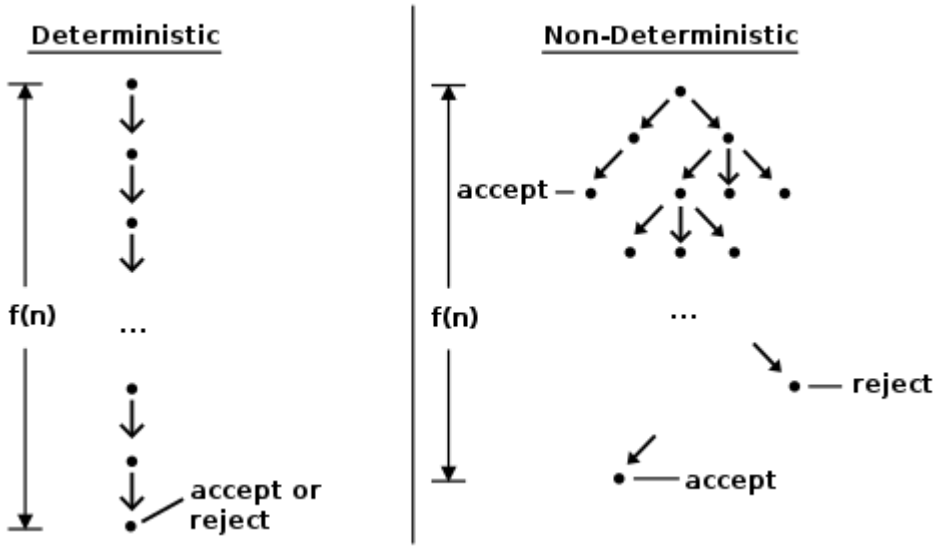
Chapter 10

1. Deterministic and Non-deterministic Algorithms

In **deterministic algorithm**, for a given particular input, the computer will always produce the same output going through the same states but in case of **non-deterministic algorithm**, for the same input, the compiler may produce different output in different runs. In fact non-deterministic algorithms can't solve the problem in polynomial time and can't determine what is the next step. The non-deterministic algorithms can show different behaviors for the same input on different execution and there is a degree of randomness to it.



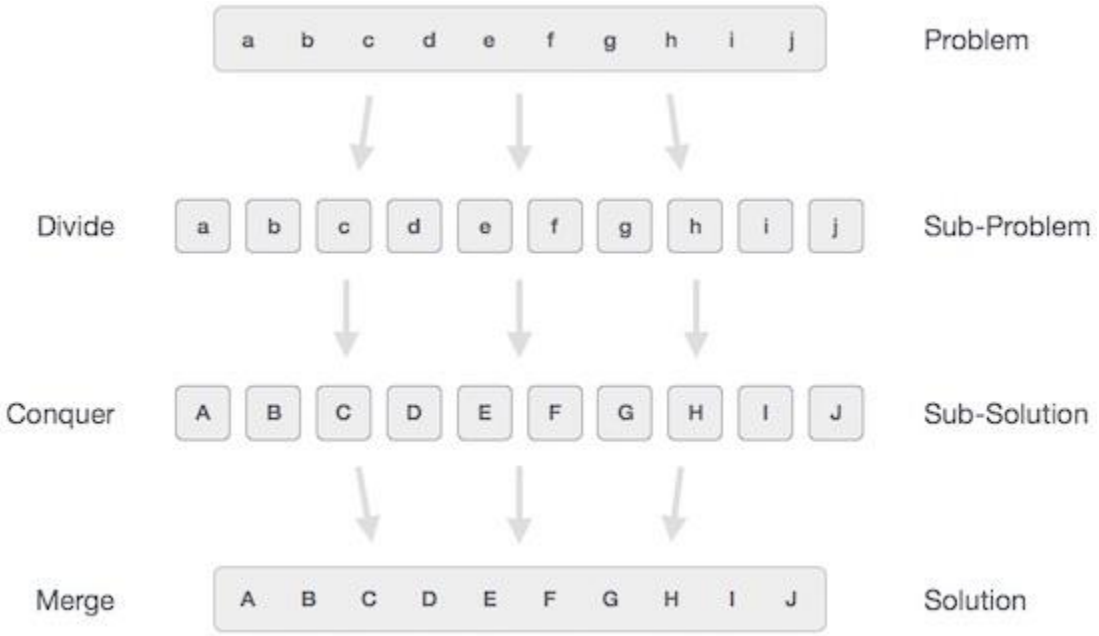
A deterministic algorithm that performs $f(n)$ steps always finishes in $f(n)$ steps and always returns the same result. A non deterministic algorithm that has $f(n)$ levels might not return the same result on different runs. A non deterministic algorithm may never finish due to the potentially infinite size of the fixed height tree.

A **nondeterministic algorithm** is an algorithm that, even for the same input, can exhibit different behaviors on different runs, as opposed to a deterministic algorithm. There are several ways an algorithm may behave differently from run to run. A concurrent algorithm can perform differently on different runs due to a race condition. A probabilistic algorithm's behaviors depends on a random number generator.

DETERMINISTIC ALGORITHM	NON-DETERMINISTIC ALGORITHM
For a particular input the computer will give always same output.	For a particular input the computer will give different output on different execution.
Can solve the problem in polynomial time.	Can't solve the problem in polynomial time.
Can determine the next step of execution.	Cannot determine the next step of execution due to more than one path the algorithm can take.

2. Divide and Conquer

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 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.



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 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.

3. Series and Parallel algorithm

Algorithms in which operations must be executed step by step are called serial or sequential algorithms. Algorithms in which several operations may be executed simultaneously are referred to as parallel algorithms. A parallel algorithm for a parallel computer can be defined as set of processes that may be executed simultaneously and may communicate with each other in order to solve a given problem. The term process may be defined as a part of a program that can be run on a processor.

In designing a parallel algorithm, it is important to determine the efficiency of its use of available resources. Once a parallel algorithm has been developed, a measurement should be used for evaluating its performance (or efficiency) on a parallel machine. A common measurement often used is run time. Run time (also referred to as elapsed time or completion time) refers to the time the algorithm takes on a parallel machine in order to solve a problem. More specifically, it is the

elapsed time between the start of the first processor (or the first set of processors) and the termination of the last processor (or the last set of processors).

Various approaches may be used to design a parallel algorithm for a given problem. One approach is to attempt to convert a sequential algorithm to a parallel algorithm. If a sequential algorithm already exists for the problem, then inherent parallelism in that algorithm may be recognized and implemented in parallel. Inherent parallelism is parallelism that occurs naturally within an algorithm, not as a result of any special effort on the part of the algorithm or machine designer. It should be noted that exploiting inherent parallelism in a sequential algorithm might not always lead to an efficient parallel algorithm. It turns out that for certain types of problems a better approach is to adopt a parallel algorithm that solves a problem similar to, but different from, the given problem. Another approach is to design a totally new parallel algorithm that is more efficient than the existing one.

In either case, in the development of a parallel algorithm, a few important considerations cannot be ignored. The cost of communication between processes has to be considered, for instance. Communication aspects are important since, for a given algorithm, communication time may be greater than the actual computation time. Another consideration is that the algorithm should take into account the architecture of the computer on which it is to be executed. This is particularly important, since the same algorithm may be very efficient on one architecture and very inefficient on another architecture.

A **parallel algorithm**, as opposed to a traditional serial algorithm, is an algorithm which can do multiple operations in a given time. Many parallel algorithms are executed concurrently – though in general concurrent algorithms are a distinct concept – and thus these concepts are often conflated, with which aspect of an algorithm is parallel and which is concurrent not being clearly distinguished. Further, non-parallel, non-concurrent algorithms are often referred to as "sequential algorithms", by contrast with concurrent algorithms.

Motivation to Parallel algorithm

Parallel algorithms on individual devices have become more common since the early 2000s because of substantial improvements in multiprocessing systems and the rise of multi-core processors. Up until the end of 2004, single-core processor performance rapidly increased via frequency scaling, and thus it was easier to construct a computer with a single fast core than one with many slower cores with the same throughput, so multicore systems were of more limited use. Since 2004 however, frequency scaling hit a wall, and thus multicore systems have become more widespread, making parallel algorithms of more general use.

Issues to parallel algorithm

Communication

The cost or complexity of serial algorithms is estimated in terms of the space (memory) and time (processor cycles) that they take. Parallel algorithms need to optimize one more resource, the communication between different processors. There are two ways parallel processors communicate, shared memory or message passing.

Shared memory processing needs additional locking for the data, imposes the overhead of additional processor and bus cycles, and also serializes some portion of the algorithm.

Message passing processing uses channels and message boxes but this **communication adds transfer overhead on the bus**, additional memory need for queues and message boxes and latency in the messages. Designs of parallel processors use special buses like crossbar so that the communication overhead will be small but it is the parallel algorithm that decides the volume of the traffic.

If the communication overhead of additional processors outweighs the benefit of adding another processor, one encounters parallel slowdown.

Load balancing

Another problem with parallel algorithms is ensuring that they are suitably load balanced, by ensuring that *load* (overall work) is balanced, rather than input size being balanced. For example, checking all numbers from one to a hundred thousand for primality is easy to split amongst processors; however, if the numbers are simply divided out evenly (1–1,000, 1,001–2,000, etc.), the amount of work will be unbalanced, as smaller numbers are easier to process by this algorithm (easier to test for primality), and thus some processors will get more work to do than the others, which will sit idle until the loaded processors complete.

4. Heuristic and Approximate algorithm

Heuristic algorithm

A heuristic algorithm is one that is designed to solve a problem in a faster and more efficient fashion than traditional methods by sacrificing optimality, accuracy, precision, or completeness for speed. Heuristic algorithms often times used to solve NP-complete problems, a class of decision problems. In these problems, there is no known efficient way to find a solution quickly and accurately although solutions can be verified when given. Heuristics can produce a solution individually or be used to provide a good baseline and are supplemented with optimization algorithms. Heuristic algorithms are most often employed when approximate solutions are sufficient and exact solutions are necessarily computationally expensive.

A heuristic or approximation algorithm may be used when

1. A feasible solution is required rapidly; within a few seconds or minutes
2. An instance is so large, it cannot be formulated as an IP or MIP of reasonable size
3. In branch-and-bound algorithm, looking to find good/quick primal bounds on subproblems solutions sufficiently quickly
4. Provide a feasible solution without guarantee on its optimality, or even quality, or algorithm running time.
5. Well-designed heuristics for particular problem classes have been empirically shown to find good solutions fast.
6. Examples of heuristic algorithms:
    a. Lagrangian: Solve the Lagrangian Relaxation with a good value of u. If the solution x(u) is not feasible, make adjustments while keeping objective function deteriorations small.
    b. Greedy: Construct a feasible solution from scratch, choosing at each step the decision bringing the "best" immediate reward.
    c. Local Search Start with a feasible solution x, and compare it with "neighboring" feasible solutions. If a better neighbor y is found, move to y, and repeat the same procedure. If no such neighbor is found, the process stops — a local optimum is found.
7. In practice, often used in combination: a greedy procedure or other heuristic to construct a feasible solution, followed by a local search to improve it.

Local Search Heuristics: Start with a feasible solution x, and compare it with "neighboring" feasible solutions. If a better neighbor y is found, move to y, and repeat the same procedure. If no such neighbor is found, the process stops — a local optimum is found. To define a local search, need a starting solution and a definition of a neighborhood.

The following are well-known examples of "intelligent" algorithms that use clever simplifications and methods to solve computationally complex problems.

Examples

Traveling Salesmen Problem

A well-known example of a heuristic algorithm is used to solve the common Traveling Salesmen Problem. The problem is as follows: given a list of cities and the distances between each city, what is the shortest possible route that visits each city exactly once? A heuristic algorithm used to quickly solve this problem is the nearest neighbor (NN) algorithm (also known as the Greedy Algorithm). Starting from a randomly chosen city, the algorithm finds the closest city. The remaining cities are analyzed again, and the closest city is found.

These are the steps of the NN algorithm:

1. Start at a random vertex
2. Determine the shortest distance connecting the current vertex and an unvisited vertex *V*
3. Make the current vertex the unvisited vertex *V*
4. Make *V* visited
5. Record the distance traveled
6. Terminate if no other unvisited vertices remain
7. Repeat step 2.

This algorithm is heuristic in that it does not take into account the possibility of better steps being excluded due to the selection process. For n cities, the NN algorithm creates a path that is approximately 25% longer than the most optimal solution.

Knapsack Problem

Another common use of heuristics is to solve the Knapsack Problem, in which a given set of items (each with a mass and a value) are grouped to have a maximum value while being under a certain mass limit. The heuristic algorithm for this problem is called the Greedy Approximation Algorithm which sorts the items based on their value per unit mass and adds the items with the highest v/m as long as there is still space remaining.

Approximation algorithms

Essentially, heuristics with a provable guarantee on the quality of the obtained solution.

Approximation algorithms are efficient algorithms that find approximate solutions to NP-hard optimization problems with **provable guarantees** on the distance of the returned solution to the optimal one.[1] Approximation algorithms naturally arise in the field of theoretical computer science as a consequence of the widely believed $P \neq NP$ conjecture. Under this conjecture, a wide class of optimization problems cannot be solved exactly in polynomial time. The field of approximation algorithms, therefore, tries to understand how closely it is possible to approximate optimal solutions to such problems in polynomial time. In an overwhelming majority of the cases, the guarantee of such algorithms is a multiplicative one expressed as an approximation ratio or approximation factor i.e., the optimal solution is always guaranteed to be within a (predetermined) multiplicative factor of the returned solution. However, there are also many approximation algorithms that provide an additive guarantee on the quality of the returned solution. A notable example of an approximation algorithm that provides *both* is the classic approximation algorithm of Lenstra, Shmoys and Tardos[2] for Scheduling on Unrelated Parallel Machines.

The design and analysis of approximation algorithms crucially involves a mathematical proof certifying the quality of the returned solutions in the worst case.[1] This distinguishes them from heuristics such as annealing or genetic algorithms, which find reasonably good solutions on some inputs, but provide no clear indication at the outset on when they may succeed or fail.

There is widespread interest in theoretical computer science to better understand the limits to which we can approximate certain famous optimization problems. For example, one of the long-standing open questions in computer science is to determine whether there is an algorithm that outperforms the 1.5 approximation algorithm of Christofides to the Metric Traveling Salesman Problem. The desire to understand hard optimization problems from the perspective of approximability is motivated by the discovery of surprising mathematical connections and broadly applicable techniques to design algorithms for hard optimization problems. One well-known example of the former is the Goemans-Williamson algorithm for Maximum Cut which solves a graph theoretic problem using high dimensional geometry.

A simple example of an approximation algorithm is one for the Minimum Vertex Cover problem, where the goal is to choose the smallest set of vertices such that every edge in the input graph contains at least one chosen vertex. One way to find a vertex cover is to repeat the following process: find an uncovered edge, add both its endpoints to the cover, and remove all edges incident to either vertex from the graph. As any vertex cover of the input graph must use a distinct vertex to cover each edge that was considered in the process (since it forms a matching), the vertex cover produced, therefore, is at most twice as large as the optimal one. In other words, this is a constant factor approximation algorithm with an approximation factor of 2. Under the recent Unique Games Conjecture, this factor is even the best possible one.[4]

NP-hard problems vary greatly in their approximability; some, such as the Knapsack Problem, can

be approximated within a multiplicative factor , for any fixed , and therefore produce solutions arbitrarily close to the optimum (such a family of approximation algorithms is called a polynomial time approximation scheme or PTAS). Others are impossible to approximate within any constant, or even polynomial, factor unless P = NP, as in the case of the Maximum Clique Problem. Therefore, an important benefit of studying approximation algorithms is a fine-grained classification of the difficulty of various NP-hard problems beyond the one afforded by the theory of NP-completeness. In other words, although NP-complete problems may be equivalent (under polynomial time reductions) to each other from the perspective of exact solutions, the corresponding optimization problems behave very differently from the perspective of approximate solutions.

By now there are several established techniques to design approximation algorithms. These include the following ones.

1. Greedy algorithm
2. Local search
3. Enumeration and dynamic programming
4. Solving a convex programming relaxation to get a fractional solution. Then converting this fractional solution into a feasible solution by some appropriate rounding. The popular relaxations include the following.

- Linear programming relaxations
- Semidefinite programming relaxations

5. Primal-Dual Methods
6. Dual Fitting
7. Embedding the problem in some metric and then solving the problem on the metric. This is also known as metric embedding.
8. Random sampling and the use of randomness in general in conjunction with the methods above.

5. Asymptotic Notations

Asymptotic notations are mathematical tools to represent time complexity of algorithms for asymptotic analysis. Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can conclude the best case, average case, and worst case scenario of an algorithm. **Asymptotic Notations** are the expressions that are used to represent the complexity of an algorithm.

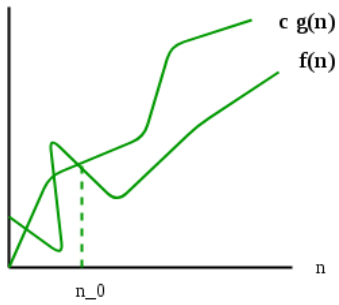
There are three **types of analysis** that we perform on a particular algorithm.

1. **Best Case**: In which we analyse the performance of an algorithm for the input, for which the algorithm takes less time or space.
2. **Worst Case**: In which we analyse the performance of an algorithm for the input, for which the algorithm takes long time or space.
3. **Average Case**: In which we analyse the performance of an algorithm for the input, for which the algorithm takes time or space that lies between best and worst case.

Types of Asymptotic Notation

1. Big-O Notation (O) – Big O notation specifically describes worst case scenario.
2. Omega Notation (Ω) – Omega(Ω) notation specifically describes best case scenario.
3. Theta Notation (θ) – This notation represents the average complexity of an algorithm.

1) Big O Notation: The Big O notation defines an upper bound of an algorithm, it bounds a function only from above. For example, consider the case of Insertion Sort. It takes linear time in best case and quadratic time in worst case. We can safely say that the time complexity of Insertion sort is O(n^2). Note that O(n^2) also covers linear time.



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

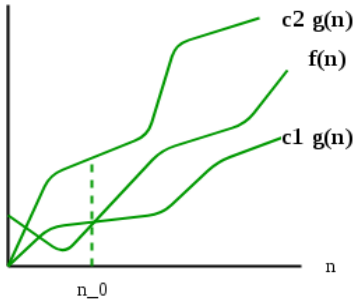
If we use Θ notation to represent time complexity of Insertion sort, we have to use two statements for best and worst cases:

1. The worst case time complexity of Insertion Sort is Θ(n^2).
2. The best case time complexity of Insertion Sort is Θ(n).

The Big O notation is useful when we only have upper bound on time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

O(g(n)) = { f(n): there exist positive constants c and n_0 such that 0 <= f(n) <= c*g(n) for all n >= n_0}

1) Θ Notation: The theta notation bounds a functions from above and below, so it defines exact asymptotic behavior.



f(n) = theta(g(n))

A simple way to get Theta notation of an expression is to drop low order terms and ignore leading constants. For example, consider the following expression.

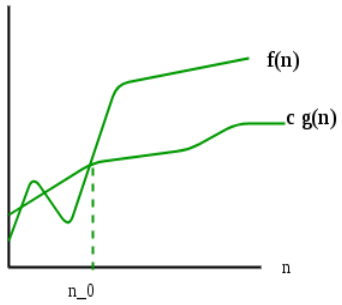
$3n^3 + 6n^2 + 6000 = \Theta(n^3)$

Dropping lower order terms is always fine because there will always be a n0 after which $\Theta(n^3)$ has higher values than Θn^2) irrespective of the constants involved.

For a given function g(n), we denote Θ(g(n)) is following set of functions.

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 >= n0}

3) Ω Notation: Just as Big O notation provides an asymptotic upper bound on a function, Ω notation provides an asymptotic lower bound.



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

Ω Notation can be useful when we have lower bound on time complexity of an algorithm. As discussed in the previous post, the best case performance of an algorithm is generally not useful, the Omega notation is the least used notation among all three.
For a given function g(n), we denote by Ω(g(n)) the set of functions.

Ω (g(n)) = {f(n): there exist positive constants c and n0 such that 0 <= c*g(n) <= f(n) for all n >= n0}.

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as Ω(n), but it is not a very useful information about insertion sort, as we are generally interested in worst case and sometimes in average case.