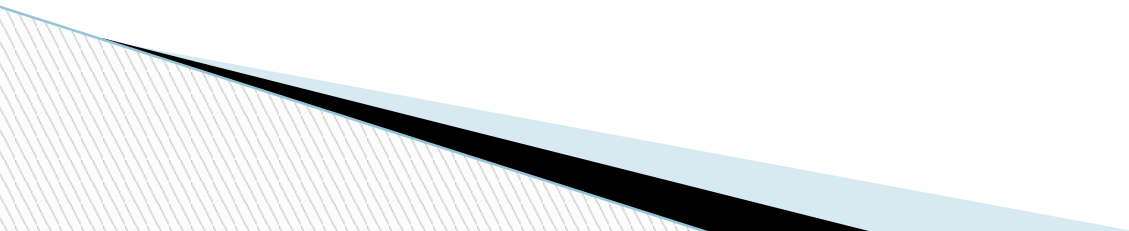


Fundamentals of Algorithm

SYBSC (Computer Science)

UNIT II



Algorithms Design Techniques

What is an algorithm?

An Algorithm is a procedure to solve a particular problem in a finite number of steps for a finite-sized input.

The algorithms can be classified in various ways. They are:

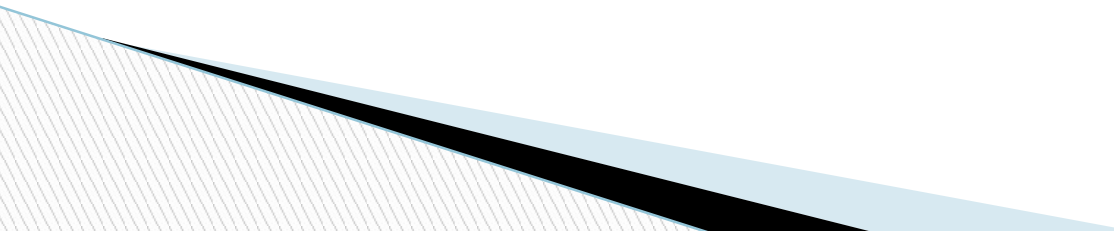
- Implementation Method
- Design Method
- Other Classifications

Classification by Implementation Method: There are primarily three main categories into which an algorithm can be named in this type of classification. They are:

Recursion or Iteration: A recursive algorithm is an algorithm which calls itself again and again until a base condition is achieved whereas iterative algorithms use loops and/or data structures like stacks, queues to solve any problem. Every recursive solution can be implemented as an iterative solution and vice versa.

Exact or Approximate: Algorithms that are capable of finding an optimal solution for any problem are known as the exact algorithm. For all those problems, where it is not possible to find the most optimized solution, an approximation algorithm is used. Approximate algorithms are the type of algorithms that find the result as an average outcome of sub outcomes to a problem.

Serial or Parallel or Distributed Algorithms: In serial algorithms, one instruction is executed at a time while parallel algorithms are those in which we divide the problem into subproblems and execute them on different processors. If parallel algorithms are distributed on different machines, then they are known as distributed algorithms.



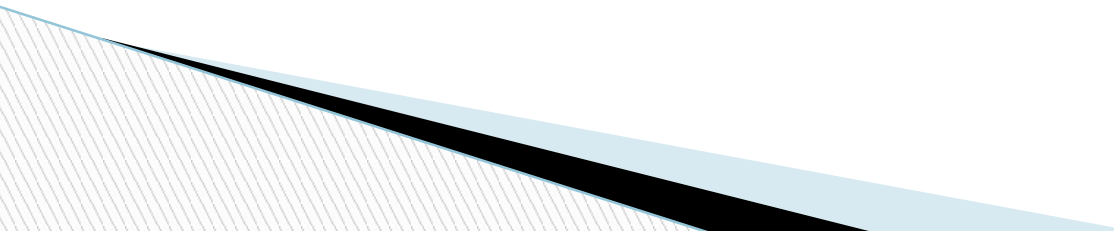
Classification by Design Method: There are primarily three main categories into which an algorithm can be named in this type of classification. They are:

Greedy Method: In the greedy method, at each step, a decision is made to choose the *local optimum*, without thinking about the future consequences.

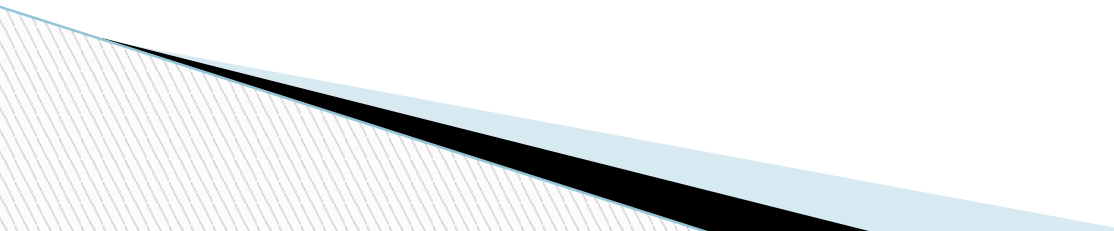
Divide and Conquer: The Divide and Conquer strategy involves dividing the problem into sub-problem, recursively solving them, and then recombining them for the final answer.

Example: Merge sort, Quicksort.

Dynamic Programming: The approach of Dynamic programming is similar to divide and conquer. The difference is that whenever we have recursive function calls with the same result, instead of calling them again we try to store the result in a data structure in the form of a table and retrieve the results from the table. Thus, the overall time complexity is reduced. “Dynamic” means we dynamically decide, whether to call a function or retrieve values from the table.



Reduction(Transform and Conquer): In this method, we solve a difficult problem by transforming it into a known problem for which we have an optimal solution. Basically, the goal is to find a reducing algorithm whose complexity is not dominated by the resulting reduced algorithms.



Other Classifications: Apart from classifying the algorithms into the above broad categories, the algorithm can be classified into other broad categories like:

Randomized Algorithms: Algorithms that make random choices for faster solutions are known as randomized algorithms.

Example: Randomized Quicksort Algorithm.

Classification by complexity: Algorithms that are classified on the basis of time taken to get a solution to any problem for input size. This analysis is known as time complexity analysis.

Example: Some algorithms take $O(n)$, while some take exponential time.

Classification by Research Area: In CS each field has its own problems and needs efficient algorithms.

Example: Sorting Algorithm, Searching Algorithm, Machine Learning etc.

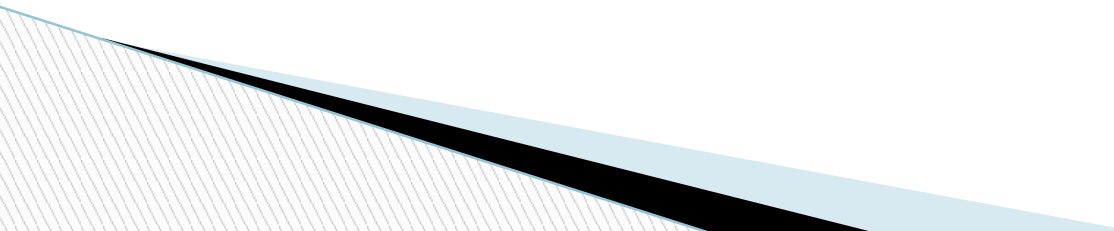
Divide and Conquer Introduction

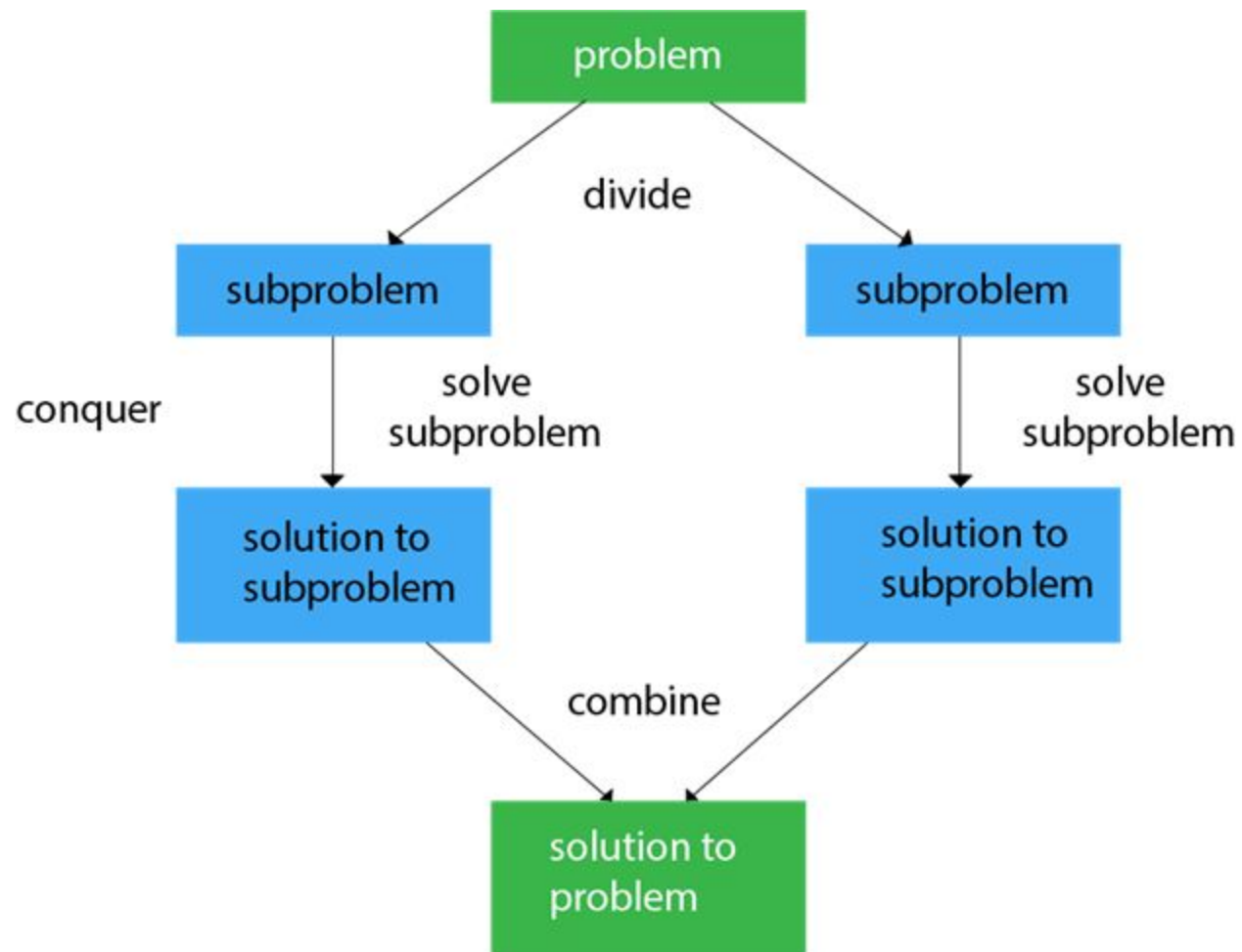
Divide and Conquer is an algorithmic pattern. In algorithmic methods, the design is to take a dispute on a huge input, break the input into minor pieces, decide the problem on each of the small pieces, and then merge the piecewise solutions into a global solution. This mechanism of solving the problem is called the Divide & Conquer Strategy. Divide and Conquer algorithm consists of a dispute using the following three steps.

Divide the original problem into a set of subproblems.

Conquer: Solve every subproblem individually, recursively.

Combine: Put together the solutions of the subproblems to get the solution to the whole problem.

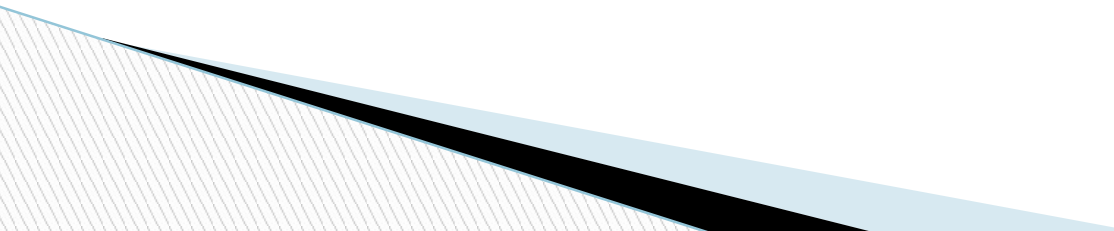




What is a 'Greedy algorithm'?

A greedy algorithm, as the name suggests, **always makes the choice that seems to be the best at that moment**. This means that it makes a locally-optimal choice in the hope that this choice will lead to a globally-optimal solution.

Assume that you have an **objective function** that needs to be optimized (either maximized or minimized) at a given point. A Greedy algorithm makes greedy choices at each step to ensure that the objective function is optimized. The Greedy algorithm has only one shot to compute the optimal solution so that **it never goes back and reverses the decision**.

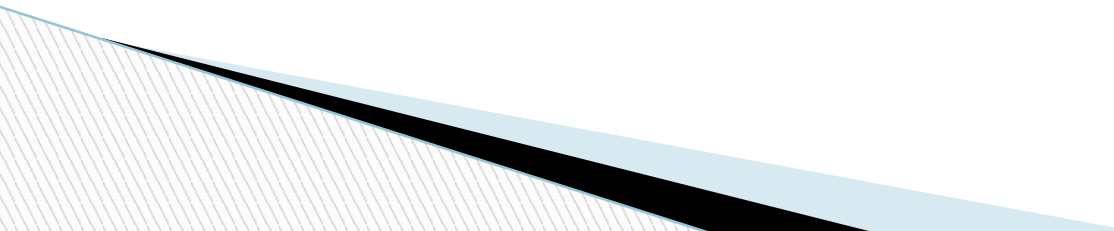


Greedy algorithms have some advantages and disadvantages:

It is quite easy to **come up with a greedy algorithm** (or even multiple greedy algorithms) for a problem.

Analyzing the run time for greedy algorithms will generally be much easier than for other techniques (like Divide and conquer). For the Divide and conquer technique, it is not clear whether the technique is fast or slow. This is because at each level of recursion the size of gets smaller and the number of sub-problems increases.

The difficult part is that for greedy algorithms **you have to work much harder to understand correctness issues**. Even with the correct algorithm, it is hard to prove why it is correct. Proving that a greedy algorithm is correct is more of an art than a science. It involves a lot of creativity.

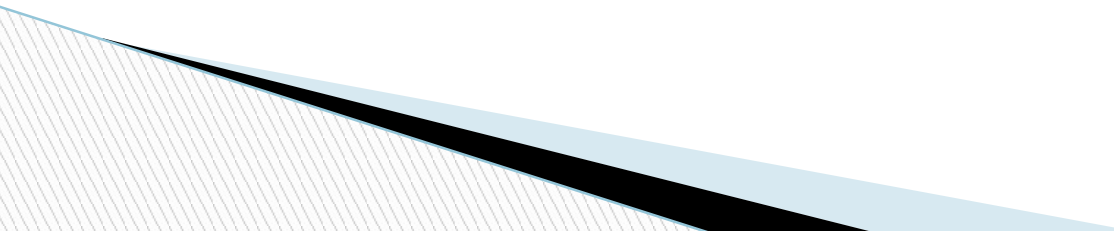


Dynamic Programming

Dynamic Programming is mainly an optimization over plain recursion. Wherever we see a recursive solution that has repeated calls for same inputs, we can optimize it using Dynamic Programming. The idea is to simply store the results of subproblems, so that we do not have to re-compute them when needed later.

1. Top-down with Memoization

In this approach, we try to solve the bigger problem by recursively finding the solution to smaller sub-problems. Whenever we solve a sub-problem, we cache its result so that we don't end up solving it repeatedly if it's called multiple times. Instead, we can just return the saved result. This technique of storing the results of already solved subproblems is called **Memoization**.



2. Bottom-up with Tabulation

Tabulation is the opposite of the top-down approach and avoids recursion. In this approach, we solve the problem “bottom-up” (i.e. by solving all the related sub-problems first). This is typically done by filling up an n-dimensional table. Based on the results in the table, the solution to the top/original problem is then computed.

Tabulation is the opposite of Memoization, as in Memoization we solve the problem and maintain a map of already solved sub-problems.

