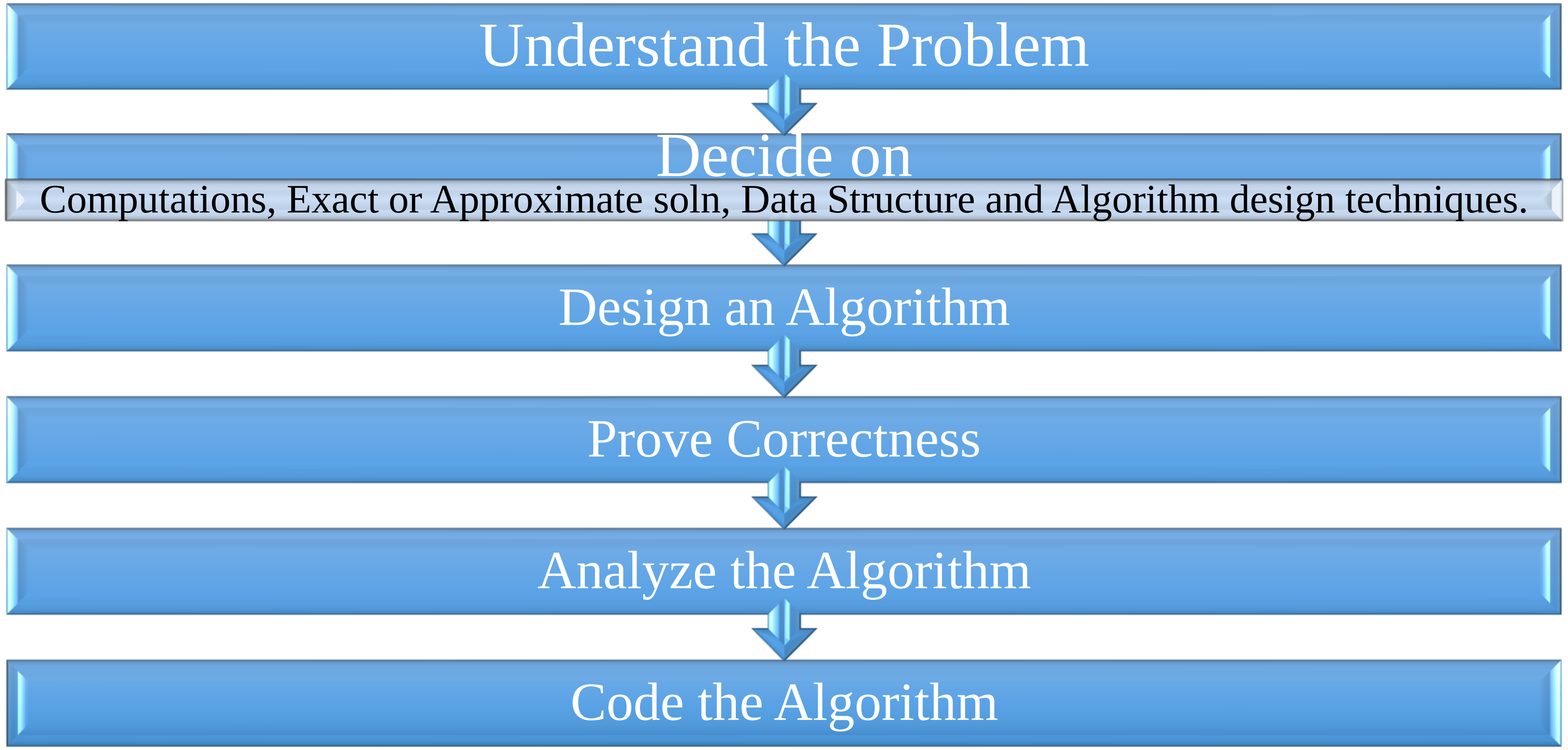
Chapter - 2

Fundamentals of Algorithmic Problem Solving

Algorithm Design Steps



Algorithm Design Steps

<u>Understand the problem:</u>

- Read the problem carefully
- If it falls into common computational problem use already known algorithms.
- Start your design by being specific about the **instances**(inputs) of the problem.

Algorithm Design Steps

<u>**Decide on: Computation**</u>

- need to be ascertain the capabilities of the computational device the algorithm is intended for.
- ***Sequential Algorithms*** use the classic RAM(Random Access Machine) model.
- ***Parallel Algorithms***
- Give some concern about Speed and Memory of the computer only if designing algorithm as a practical tool, otherwise no need to worry in scientific excercises.

Algorithm Design Steps

- **Exact Vs. Approximate Solutions**
- ***Exact algorithms*** can find the optimum solution with precision.
- Exact algorithms apply in "easy" problems. What makes a problem "easy" is that it can be solved in reasonable time and the computation time doesn't scale up exponentially if the problem gets bigger.
- This class of problems is known as **P**(Deterministic Polynomial Time). Problems of this class are used to be optimized using exact algorithms. For every other class of problems approximate algorithms are preferred.

Algorithm Design Steps

- **Exact Vs. Approximate Solutions**
- ***Approximate algorithms*** can find a *near optimum* solution.
- Are efficient algorithms that find **approximate** solutions to optimization problems (in particular **NP-hard** problems) with **provable guarantees** on the distance of the returned solution to the optimal one.
- They naturally arise in the field of **theoretical computer science** as a consequence of the widely believed **P ≠ NP** conjecture.
- The field of approximation algorithms, therefore, tries to understand how closely it is possible to approximate optimal solutions to such problems in polynomial time.

Algorithmic Problem Solving

Class Exercise: ***Old World puzzle***

A peasant finds himself on a riverbank with a **wolf**, a **goat**, and a head of **cabbage**. He needs to transport all three to the other side of the river in his boat. However, **the boat has room for only the peasant himself and one other item** (either the wolf, the goat, or the cabbage). In his absence, the ***wolf would eat the goat***, and ***the goat would eat the cabbage***. Solve this problem for the peasant or prove it has no solution. (Note: The peasant is a vegetarian but does not like cabbage and hence can eat neither the goat nor the cabbage to help him solve the problem. And it goes without saying that the wolf is a protected species.)

Major classes of problems and algorithms

- There is a limitless sea of problems in computing world.
- The problem’s *practical importance* or by *some specific characteristics* making the problem an interesting research subject.
- Those two are the motivating forces to attract researchers in some specific problems.
  - Sorting
  - Searching
  - String processing
  - Graph problems
  - Combinatorial problems
  - Geometric problems
  - Numerical problems

Class Work

Design a simple algorithm for the string-matching problem.

- **Graph problems**
- Some graph problems
  - ***traveling salesman problem (TSP)***
  - ***graph-coloring problem***
- Basic graph algorithms include
  - *graph-traversal* algorithms (how can one reach all the points in a network?)
  - *shortest-path* algorithms (what is the best route between two cities?)
  - *topological sorting* for graphs with directed edges (is a set of courses with their prerequisites consistent or self-contradictory?).

- **Combinatorial problems**
- These are problems that ask, explicitly or implicitly, to find a combinatorial object—such as a permutation, a combination, or a subset—that satisfies certain constraints.
- A desired combinatorial object may also be required to have some additional property such as a maximum value or a minimum cost.
- combinatorial problems are the most difficult problems in computing, from both a theoretical and practical standpoint. because
    1. The number of combinatorial **objects typically grows extremely fast** with a problem's size, reaching unimaginable magnitudes.
    2. There are **no known algorithms** for solving **most** such problems exactly in an acceptable amount of time.

- ***Geometric algorithms*** deal with geometric objects such as points, lines, and polygons.
- Ancient solutions with *rulers* and *compasses*, modern solutions just bits, bytes, and good old human ingenuity.
- The two classic problems
  - Closest-pair problem
  - Convex-hull problem

- **Numerical Problems** involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions, and so on.
- The majority of such mathematical problems can be solved only approximately.
- Recently, numerical analysis has lost its formerly dominating position in both industry and computer science programs because the computing industry has *shifted its focus to business applications*.
- But it is still important to have rudimentary idea about numerical algorithms.

Algorithm Design Approaches

- An ***algorithm design technique*** (or “strategy” or “paradigm”) is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
- Algorithm design techniques make it possible to classify algorithms according to an underlying design idea; therefore, they can serve as a natural way to both categorize and study algorithms.

Algorithm Design Approaches

- **<u>Divide and Conquer</u>**

It is a top-down approach.

The algorithms which follow the divide & conquer techniques involve three steps:

1.**Divide** the original problem into a set of sub problems.

2. **Solve** every sub problem individually, recursively.

3.**Combine** the solution of the sub problems (top level) into a solution of the whole original problem.

- Generally can be written as a *recurrence*.

Problems:

  - Binary Search
  - Merge Sort
  - Quick Sort
  - Matrix Operations

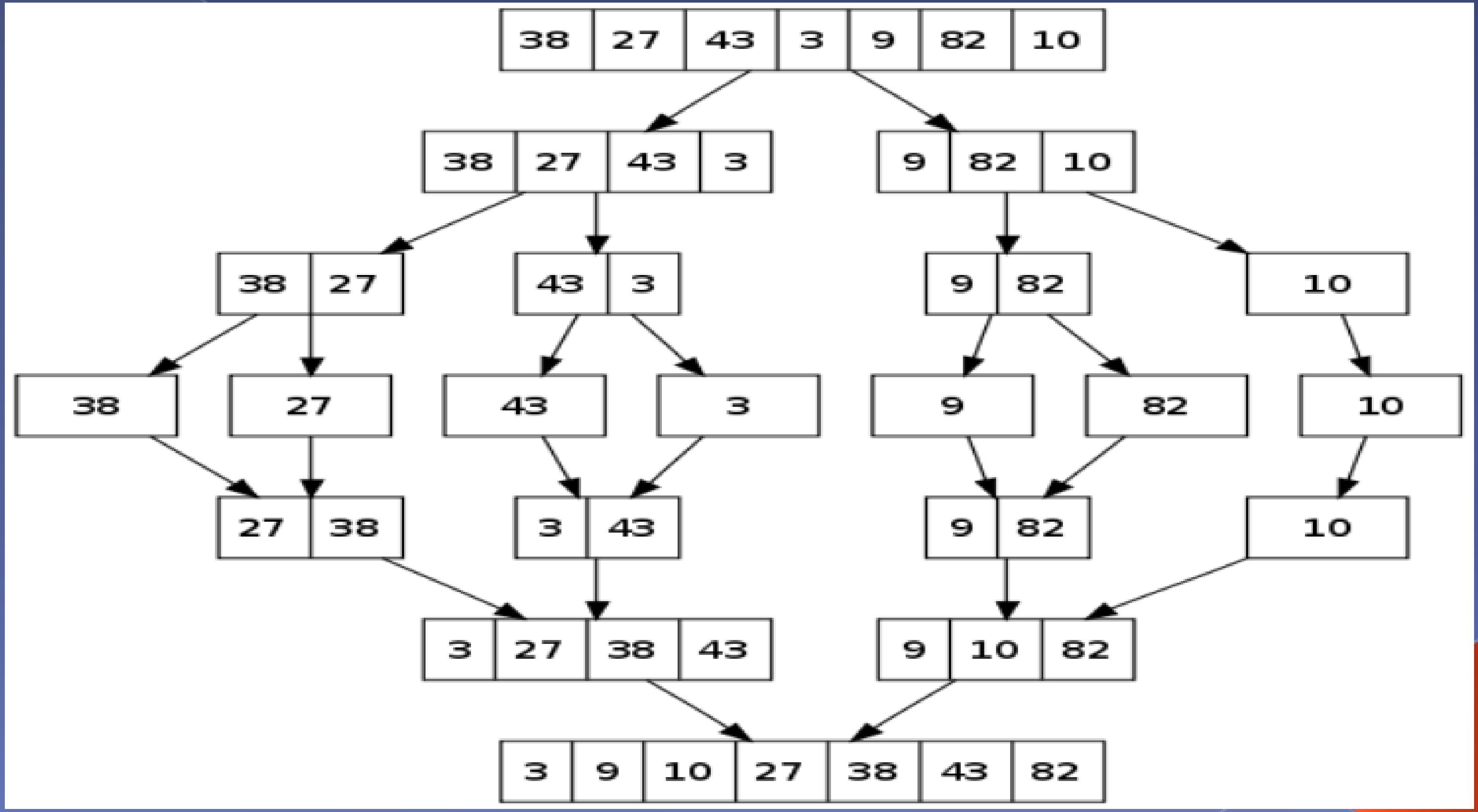
Merge Sort

Divide: divide the unsorted list into two sub lists of about half size.

Conquer: sort each of sub lists recursively until we have list sizes of 1, in which case the list itself is returned.

Combine: merge the two sorted sub lists into one sorted list.

Merge Sort Example



Algorithm Design Approaches

Greedy Approach

- is an approach for solving a problem by **selecting the best option available** at the moment.
- used **for optimization (either maximized or minimized) problems**.
- **Examples**
  - Prim's Minimal Spanning Tree Algorithm. Travelling Salesman Problem. Graph – Map Coloring. Kruskal's Minimal Spanning Tree Algorithm

Algorithm Design Approaches

Dynamic programming

- 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.
- Applications of Dynamic Programming Appoach
  - Matrix Chain Multiplication
  - Longest Common Subsequence
  - Travelling Salesman Problem

Algorithm Design Approaches

Branch and Bound Approach

- A branch-and-bound algorithm consists of a systematic enumeration of candidate solutions by means of **state space search**: the set of candidate solutions is thought of as forming a rooted tree with the full set at the root.
- The algorithm explores *branches* of this tree, which represent subsets of the solution set. Before enumerating the candidate solutions of a branch, the branch is checked against upper and lower estimated *bounds* on the optimal solution, and is discarded if it cannot produce a better solution than the best one found so far by the algorithm.
- Also an Optimization Problem.

- **Application of the Branch and Bound Technique**
    - Applied computing. Enterprise computing.
    - Scheduling Problems
    - Computing methodologies.
    - Artificial intelligence.
    - Search methodologies.
    - Heuristic function construction.
    - Theory of computation.
    - Design and analysis of algorithms.
    - Approximation algorithms analysis

The End!