Problem Solving with Computer and Algorithm Development

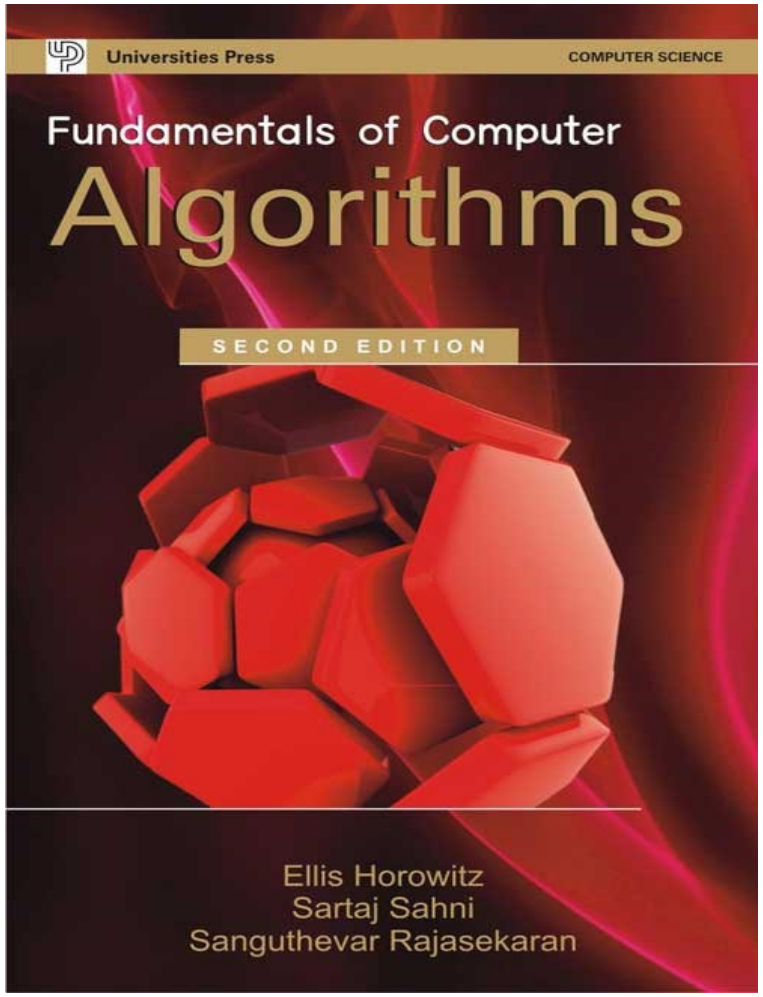
Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Text Book


Universities Press
COMPUTER SCIENCE
Fundamentals of Computer
Algorithms
SECOND EDITION
Ellis Horowitz
Sartaj Sahni
Sanguthevar Rajasekaran

Reference: The Art of Computer Programming (TAOCP)

Volume 5 (Estimated to be ready in 2025)

Syntactic Algorithms, in preparation.

9. Lexical scanning (includes also string search and data compression)

10. Parsing techniques

The Art of Computer Programming (TAOCP)

- *Donald E. Knuth*

  Computer Science Department
  Gates Building 4B
  Stanford University
  Stanford, CA 94305-9045 USA.

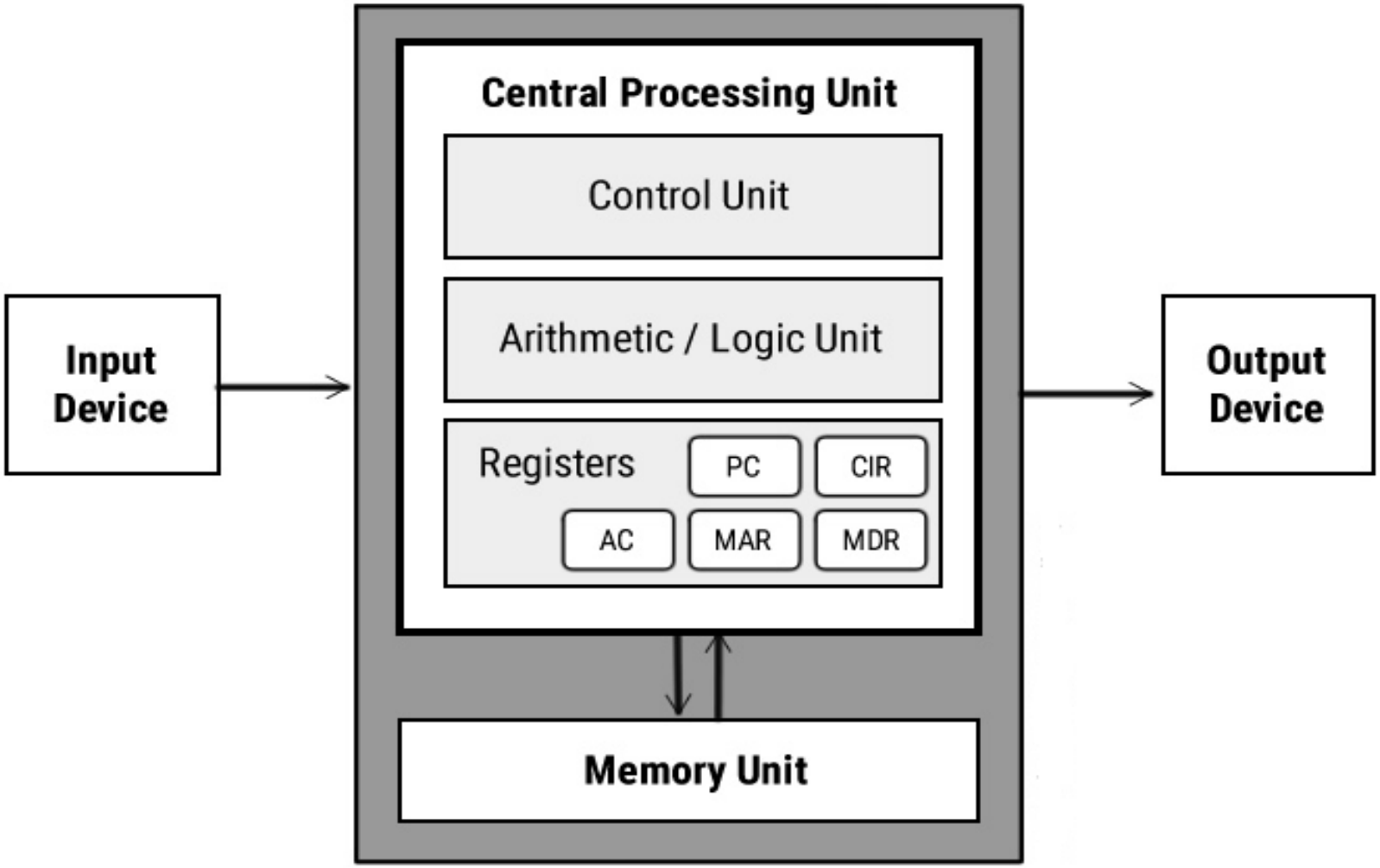
- http://www-cs-faculty.stanford.edu/~uno/

- **Donald E. Knuth** is known throughout the world for his pioneering work on algorithms and programming techniques, for his invention of the Tex and Metafont systems for computer typesetting, and for his prolific and influential writing. Professor Emeritus of The Art of Computer Programming at Stanford University, he currently devotes full time to the completion of these fascicles and the seven volumes to which they belong.

The Art of Computer Programming (TAOCP)

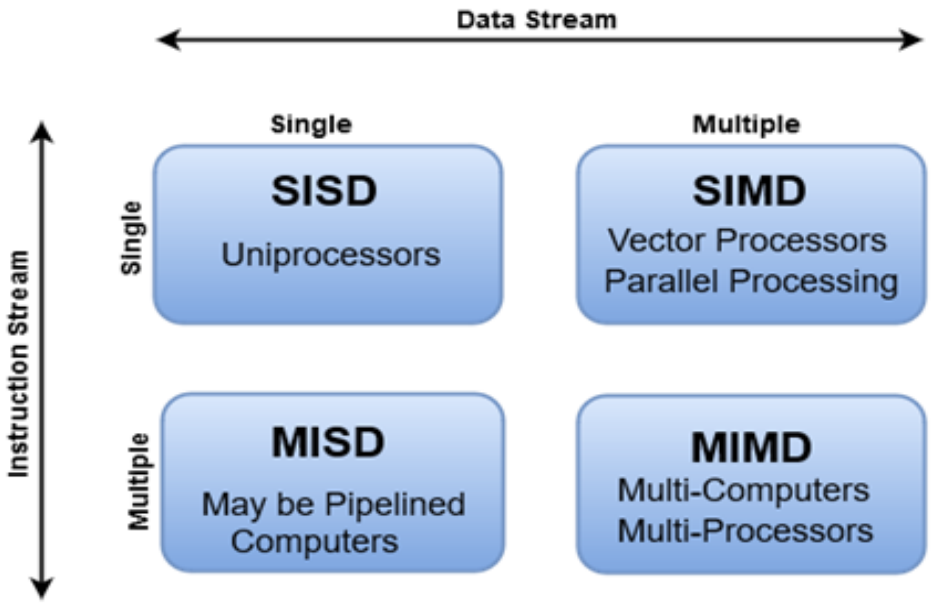
- *www-cs-faculty.stanford.edu/~uno/**books**.html*
- ***Fundamental Algorithms***, Third Edition (Reading, Massachusetts: Addison-Wesley, 1997), xx+650pp. ISBN 0-201-89683-4
- ***Seminumerical Algorithms***, Third Edition (Reading, Massachusetts: Addison-Wesley, 1997), xiv+762pp. ISBN 0-201-89684-2
- ***Sorting and Searching***, Second Edition (Reading, Massachusetts: Addison-Wesley, 1998), xiv+780pp.+foldout. ISBN 0-201-89685-0
- ***Combinatorial Algorithms**, Part 1* (Upper Saddle River, New Jersey: Addison-Wesley, 2011), xvi+883pp. ISBN 0-201-03804-8

Von Neumann Architecture



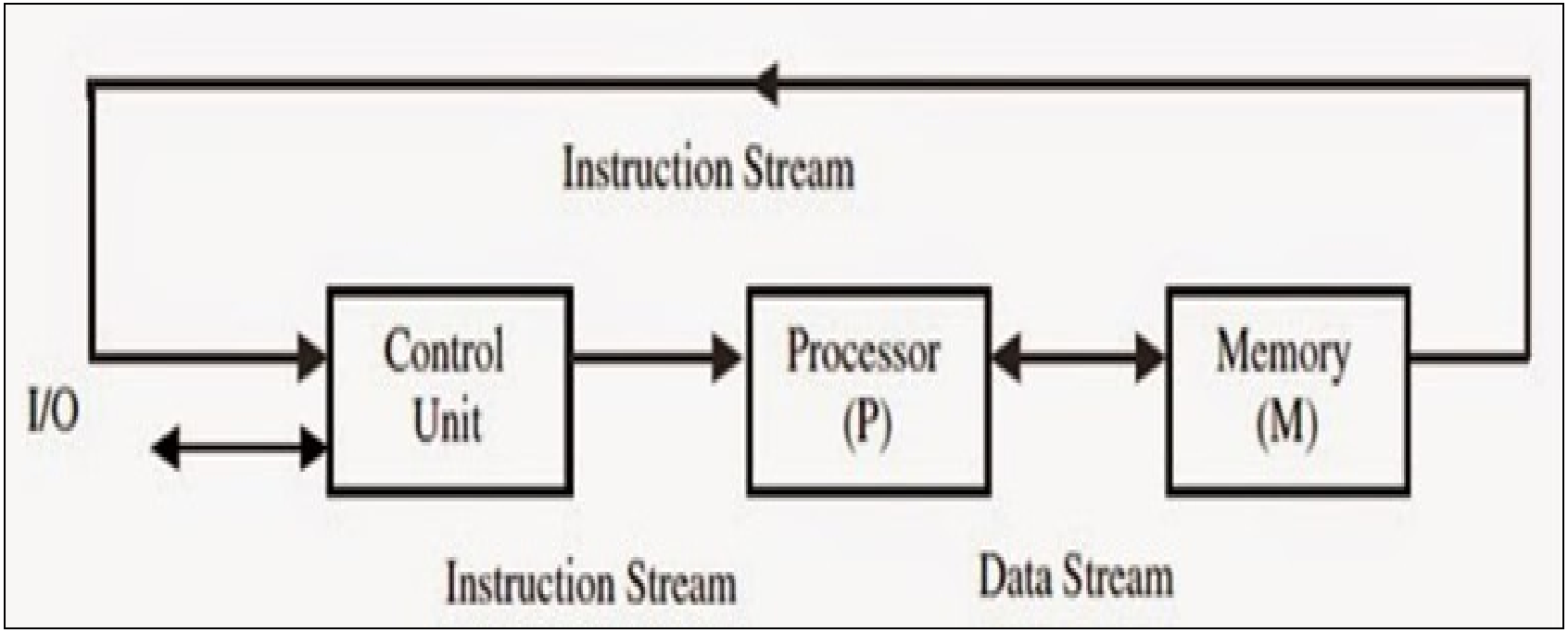
Flynn's Classification of Computers

- M.J. Flynn proposed a classification for the organization of a computer system by the number of instructions and data items that are manipulated simultaneously.
- The sequence of instructions read from memory constitutes an **instruction stream**.
- The operations performed on the data in the processor constitute a **data stream**.



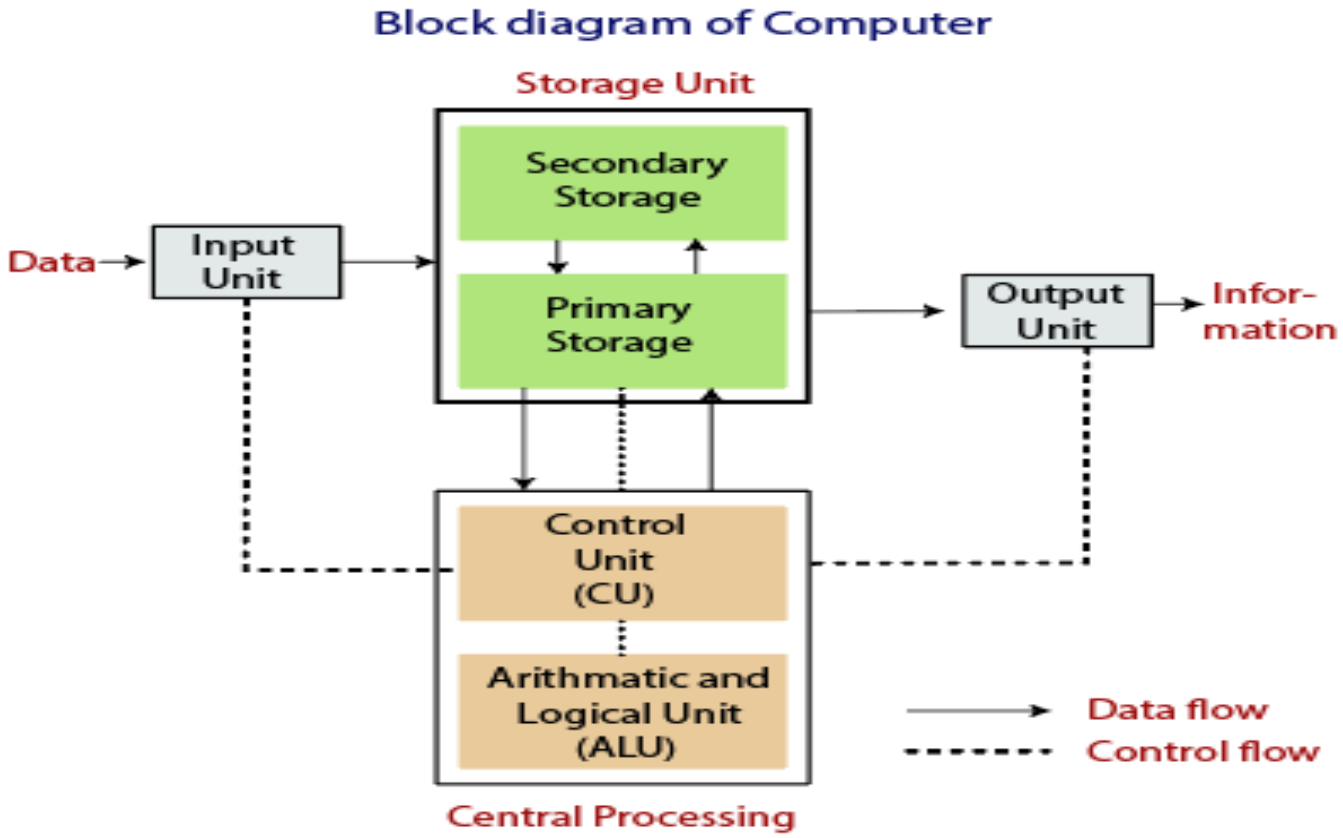
Von Neumann Architecture (SISD)

SISD (**Single Instruction, Single Data**) refers to the traditional von Neumann architecture where a single sequential processing element (PE) operates on a single stream of data. SIMD (Single Instruction, Multiple Data) performs the same operation on multiple data items simultaneously.

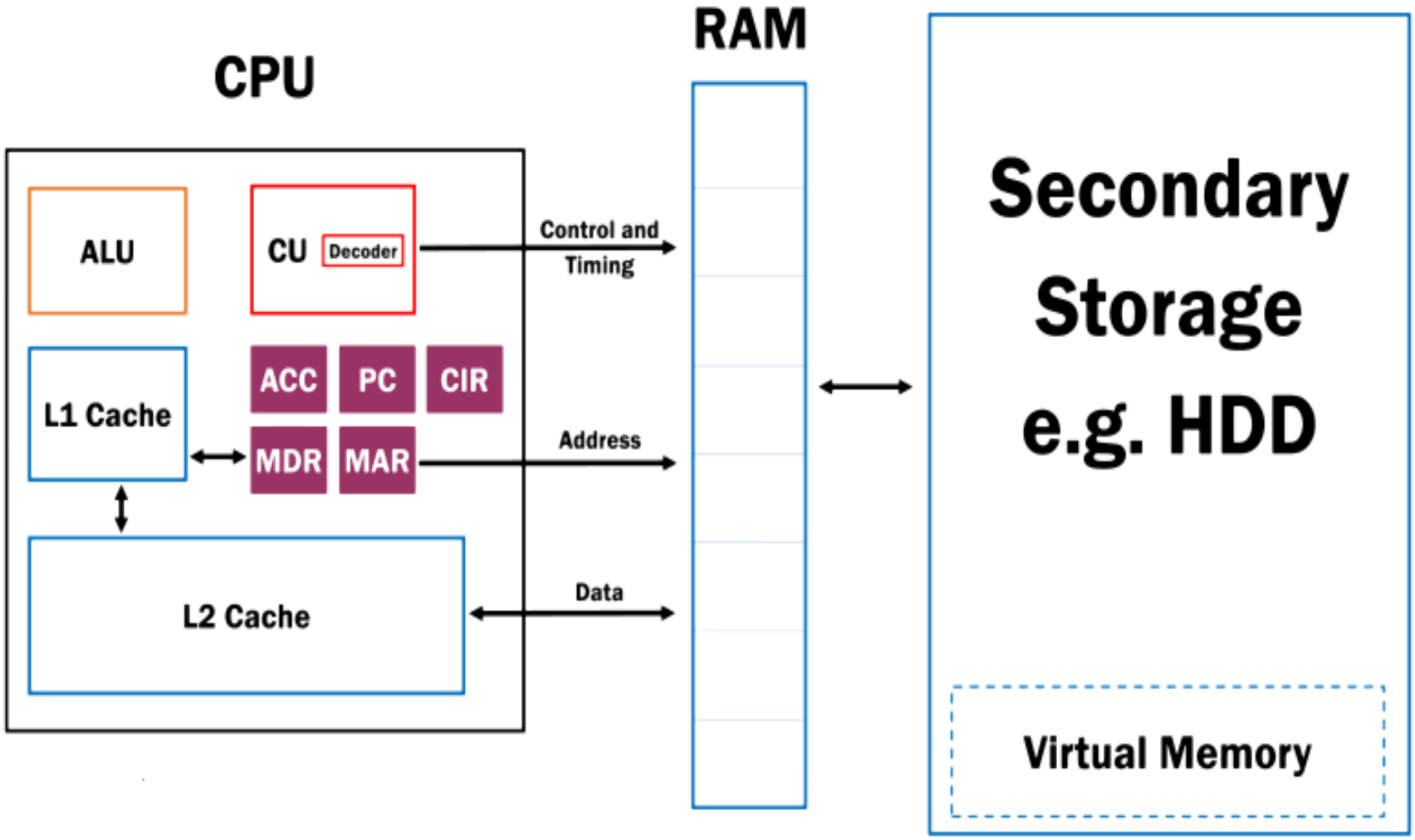


Von Neumann Architecture

- Von Neumann architecture is based on the stored-program computer concept, where instruction data and program data are stored in the same memory.

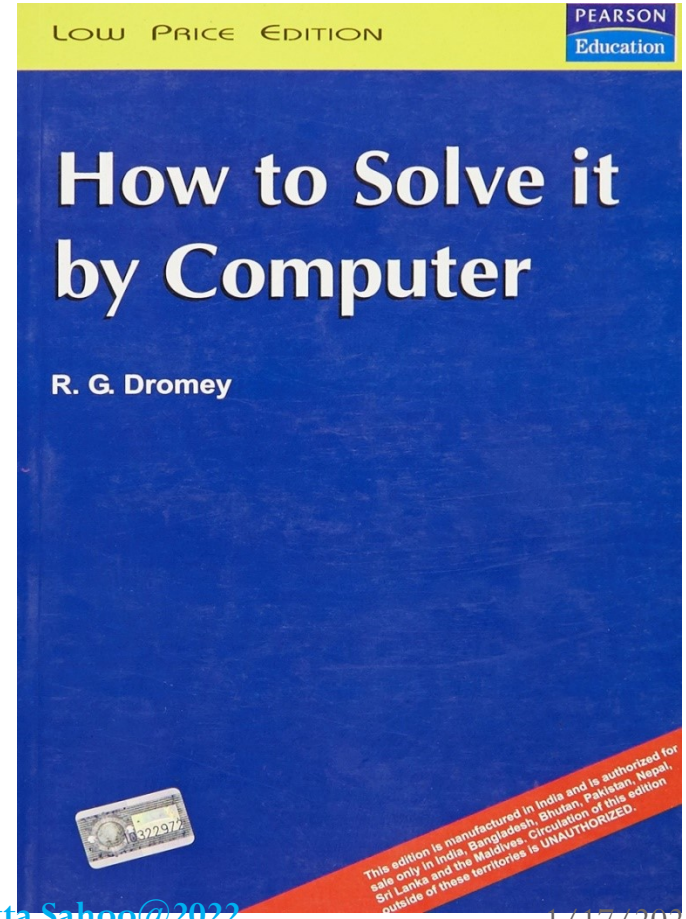


Von Neumann Architecture



Problem

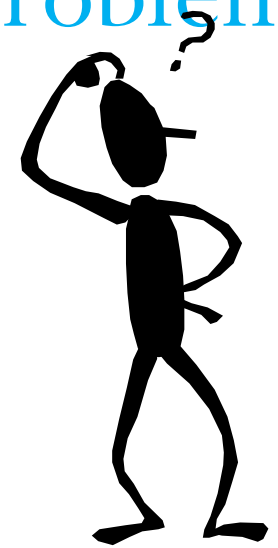
- Problem is a question to which we seek an answer?



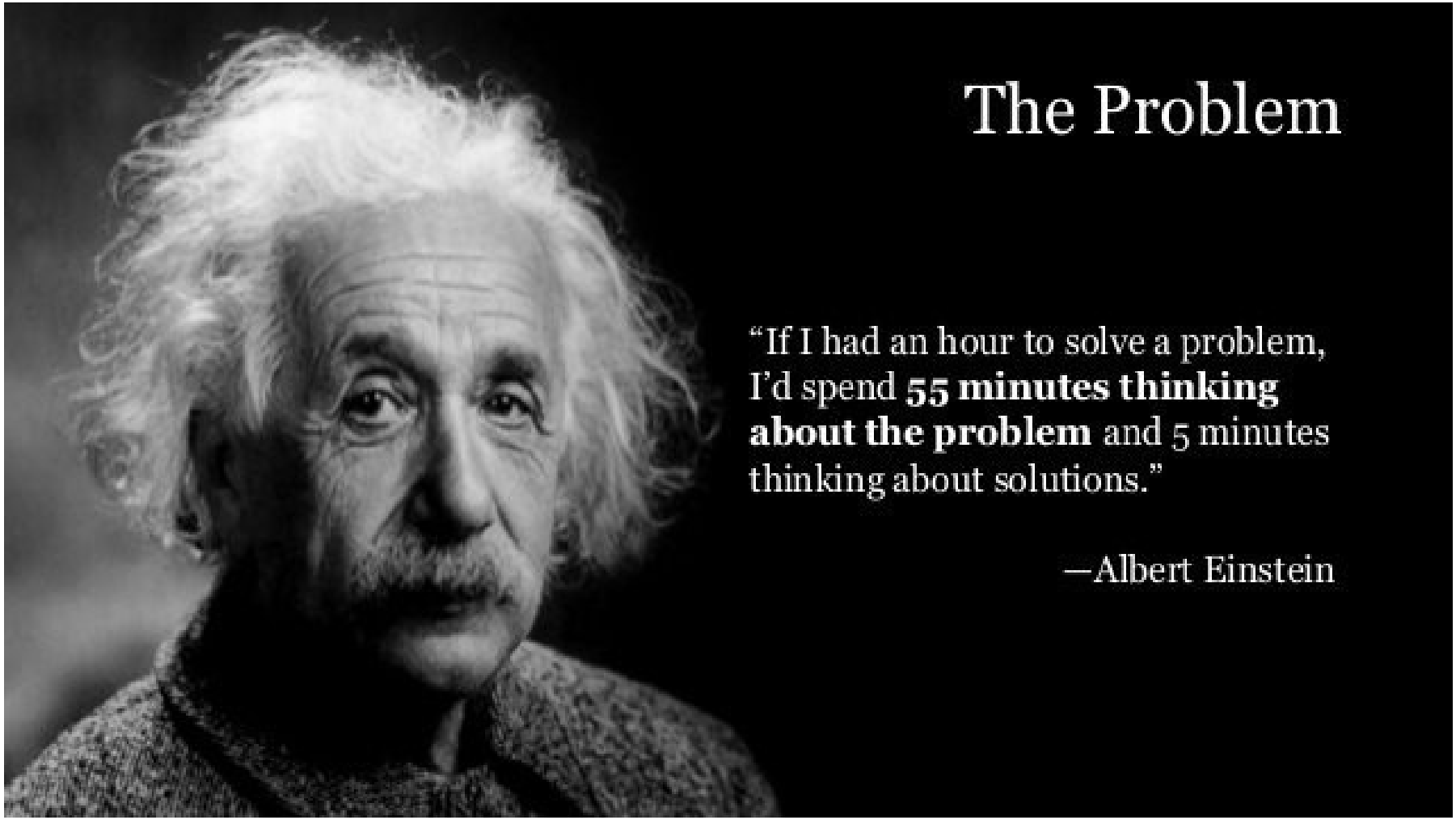
Problem : Selection problem

- Suppose you have a group of N numbers and would like to determine the k^{th} largest. This is known as the ***selection problem***.

Problem solving ?



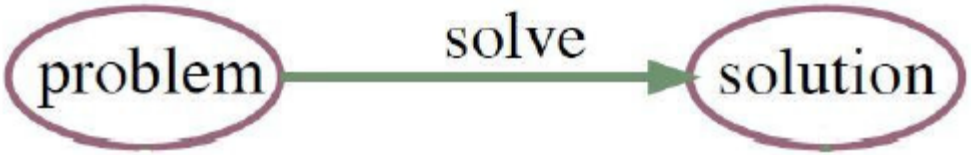
If we are solving some problem, we are usually looking for some solution, which will be the best among others.


The Problem
"If I had an hour to solve a problem, I'd spend **55 minutes thinking about the problem** and 5 minutes thinking about solutions."
—Albert Einstein

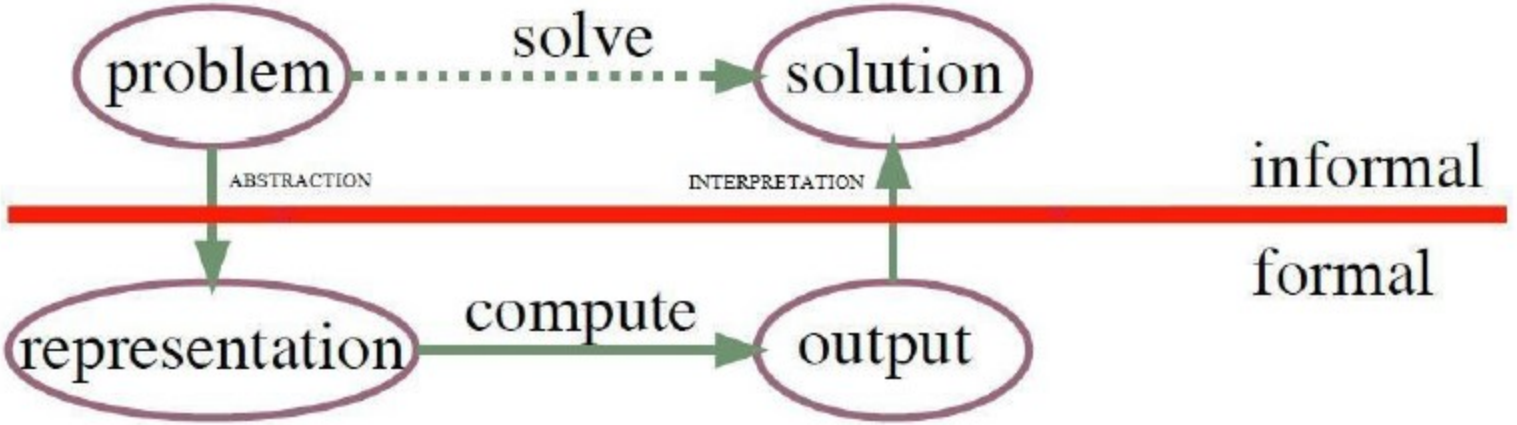
What is Problem solving ?

We have a problem and want to find a solution !

Ideally

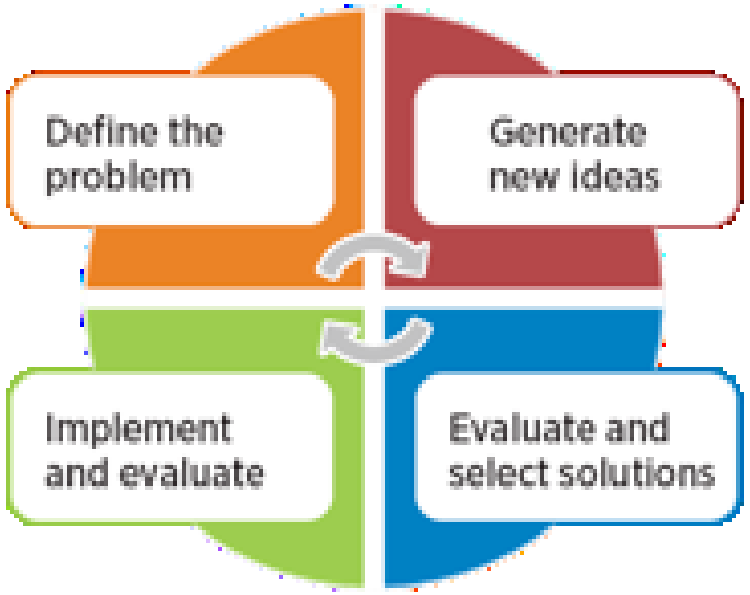


In Practice



What is problem solving?

- Problem solving is the act of **defining a problem**; determining the cause of the problem; identifying, prioritizing, and selecting alternatives for a solution; and implementing a solution.



The problem-solving process

Step	Characteristics
1. Define the problem	•Differentiate fact from opinion •Specify underlying causes •Consult each faction involved for information •State the problem specifically •Identify what standard or expectation is violated •Determine in which process the problem lies •Avoid trying to solve the problem without data
2. Generate alternative solutions	•Postpone evaluating alternatives initially •Include all involved individuals in the generating of alternatives •Specify alternatives consistent with organizational goals •Specify short- and long-term alternatives •Brainstorm on others' ideas •Seek alternatives that may solve the problem
3. Evaluate and select an alternative	•Evaluate alternatives relative to a target standard •Evaluate all alternatives without bias •Evaluate alternatives relative to established goals •Evaluate both proven and possible outcomes •State the selected alternative explicitly
4. Implement and follow up on the solution	•Plan and implement a pilot test of the chosen alternative •Gather feedback from all affected parties •Seek acceptance or consensus by all those affected •Establish ongoing measures and monitoring •Evaluate long-term results based on final solution

5-steps to Problem Solving

1. **Define the problem.**

- In understanding and communicating the problem effectively, we have to be clear about what the issue is. Remember to focus on the behaviour instead of attacking the person.

2. **Gather information.**

- What were the circumstances? What are the non-verbal messages being sent?
- Who does it affect? What behaviour is typical for the child's age? Are your expectations reasonable?

3. **Generate possible solutions.**

- Work together to brainstorm on all possible solutions. Do not judge whether the ideas are good or bad at this point.

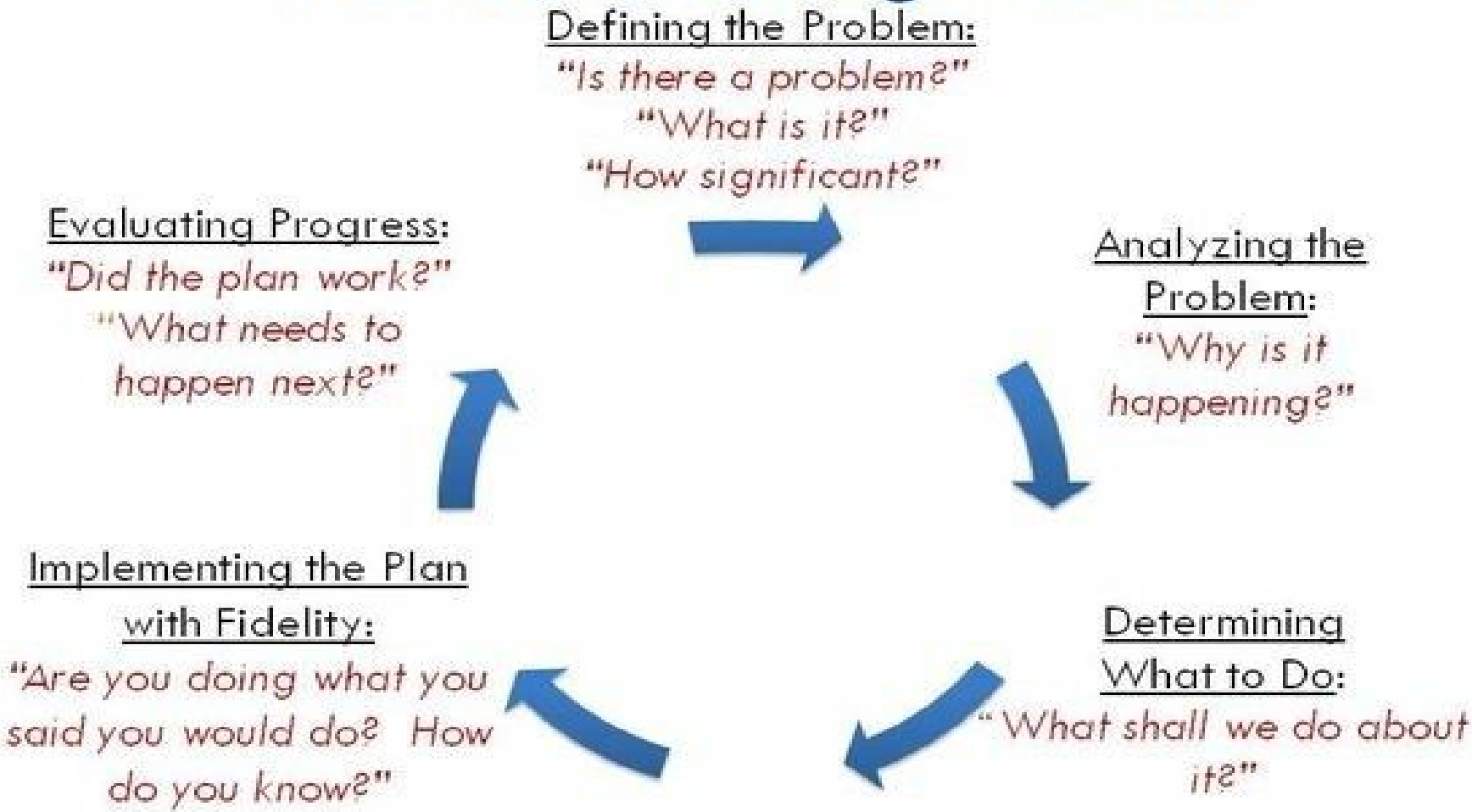
4. **Evaluate ideas and then choose one.**

- Decide which options you like and which you don't like. After weighing the pros and cons of each, choose an option that you both feel comfortable with. Once a solution has been found to meet everyone's needs while keeping everyone's self-respect and self-esteem intact, then make a plan to follow through and do it. Put down a time-frame for action.

5. **Evaluate.**

- Did it work? If yes, great! Consider how your solution may be applicable to other different problems. Ask how the problem can be prevented from happening again.
- What if it didn't work? Go back to step one or try out the other possible solutions that you and your child made in step 3.

Problem Solving Method



Problem solving

Real-World Problem

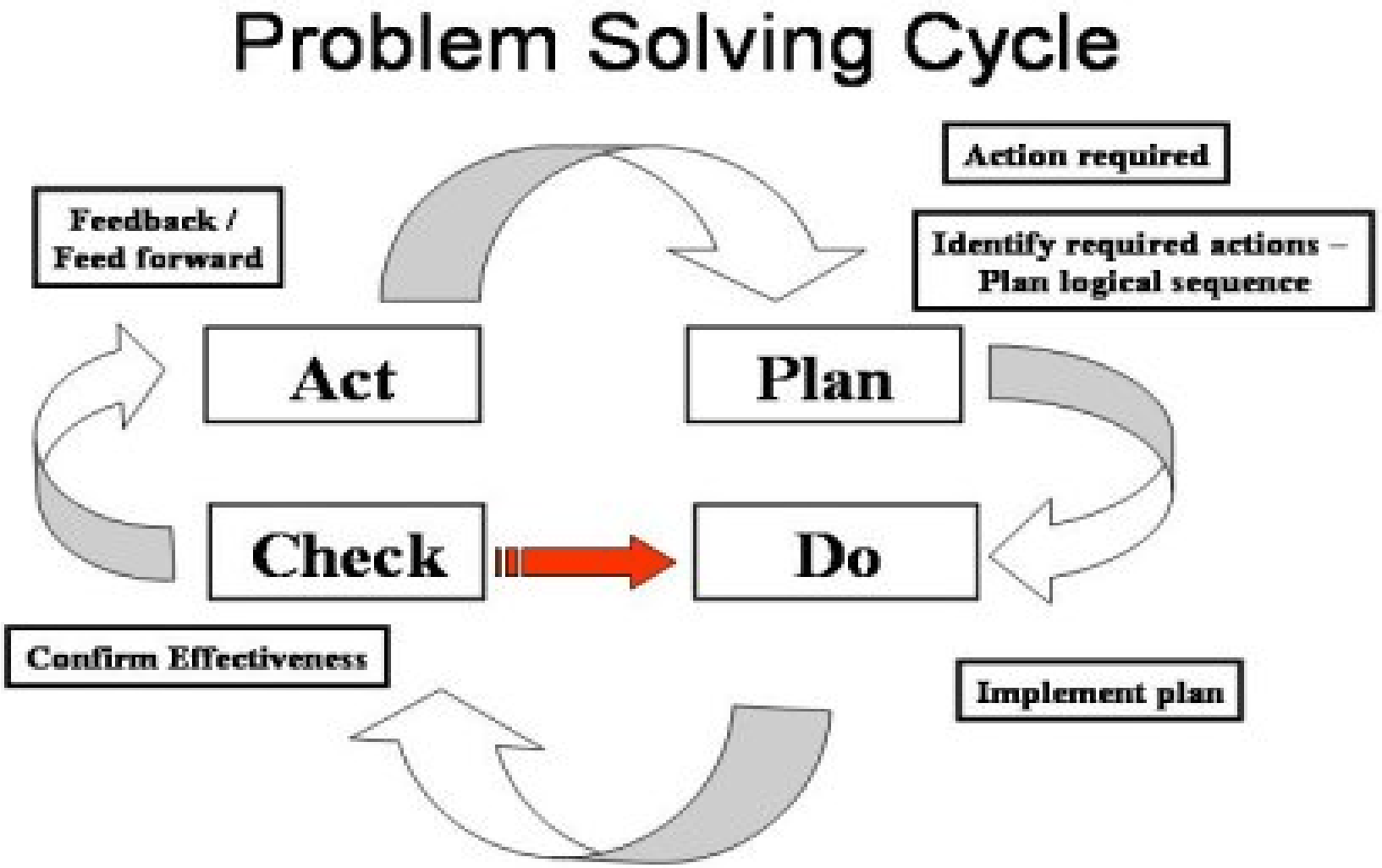
Formulation of Abstract Problem

Solve the Abstract Problem

Interpret the Solution

Implement the Solution

Problem Solving Cycle



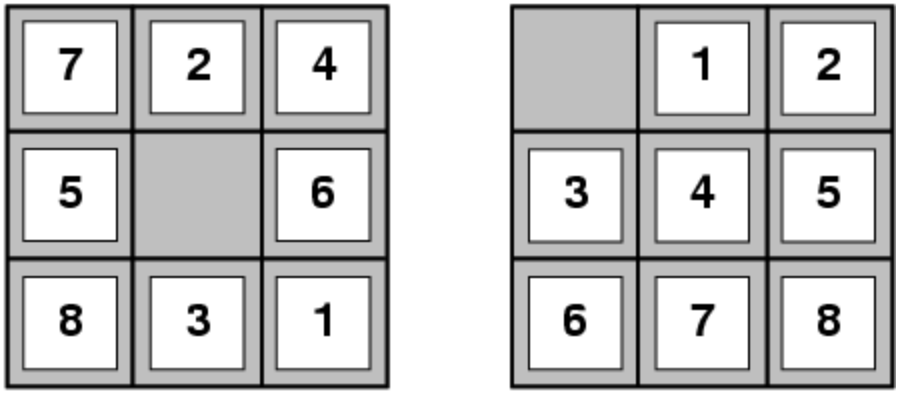
Two types of problem approaches

- **Toy Problem:** It is a concise and exact description of the problem which is used by the researchers to compare the performance of algorithms.

- **Real-world Problem:** It is real-world based problems which require solutions. Unlike a toy problem, it does not depend on descriptions, but we can have a **general formulation** of the problem.

Example: Toy Problem

- **8 Puzzle Problem:** Here, we have a 3×3 matrix with movable tiles numbered from 1 to 8 with a blank space. The tile adjacent to the blank space can slide into that space.
- In the figure, our task is to convert the current state into goal state by sliding digits into the blank space.



8 Puzzle Problem formulation

1. **States:** It describes the location of each numbered tiles and the blank tile.
2. **Initial State:** We can start from any state as the initial state.
3. **Actions:** Here, actions of the blank space is defined, i.e., either **left, right, up or down**
4. **Transition Model:** It returns the resulting state as per the given state and actions.
5. **Goal test:** It identifies whether we have reached the correct goal-state.
6. **Path cost:** The path cost is the number of steps in the path where the cost of each step is 1.

- **Note:** The 8-puzzle problem is a type of **sliding-block problem** which is used for testing **new search algorithms** in **artificial intelligence.**

4-queens problem

- **4-queens problem:** The aim of this problem is to place four queens on a chessboard in an order where no queen may attack another. A queen can attack other queens either **diagonally or in same row and column.**

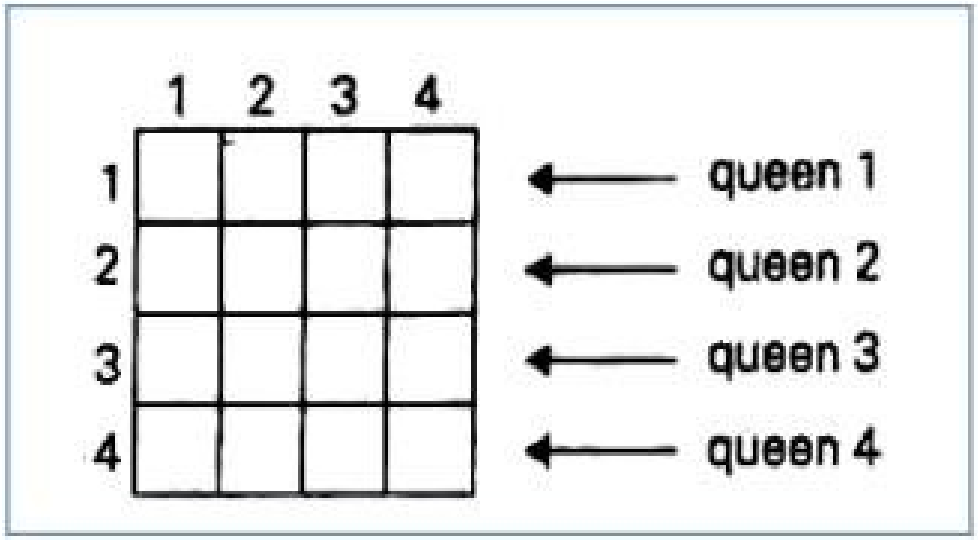
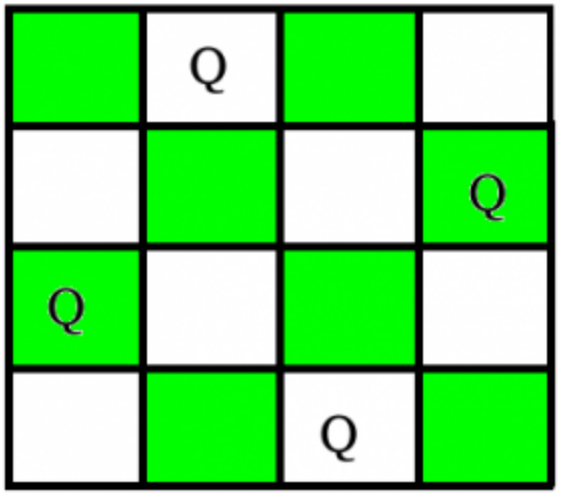
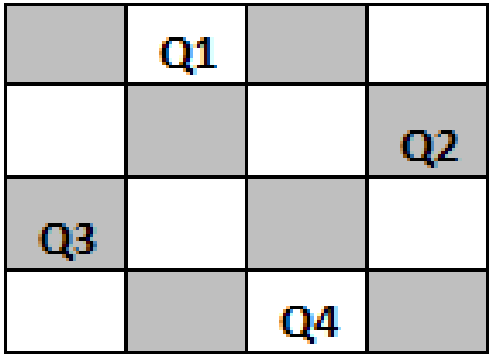


Fig: Board for the Four-queens problem

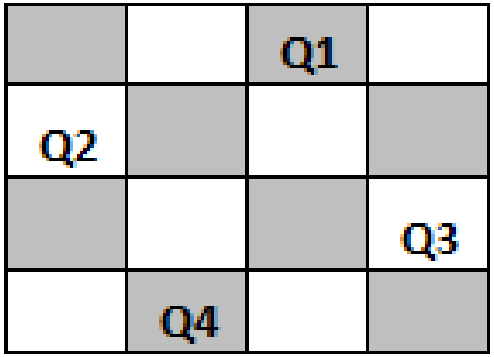


4 Queen Problem

Two solutions to 4-queen problem is written as [2 4 1 3] & [3 1 4 2]

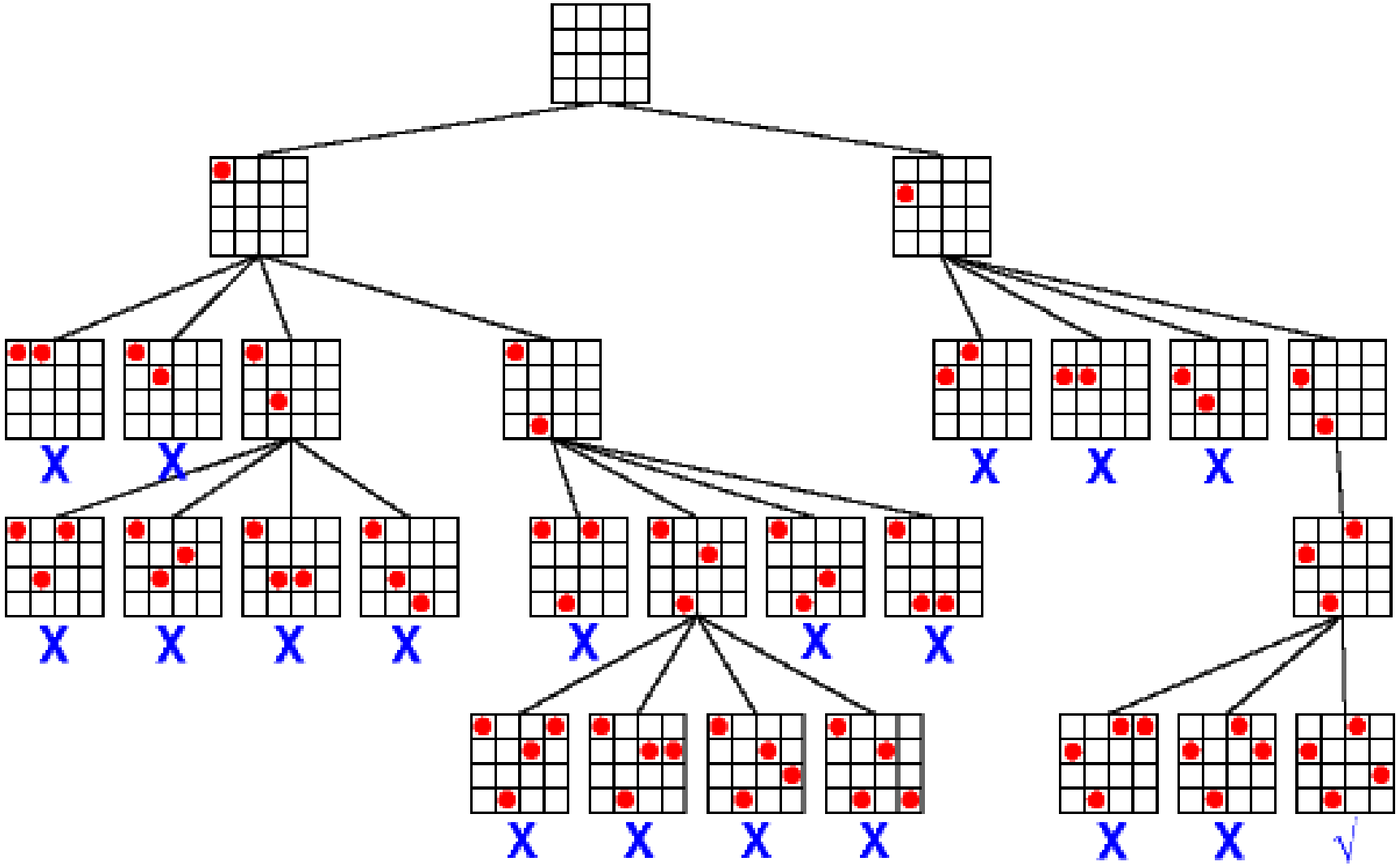


Solution 1

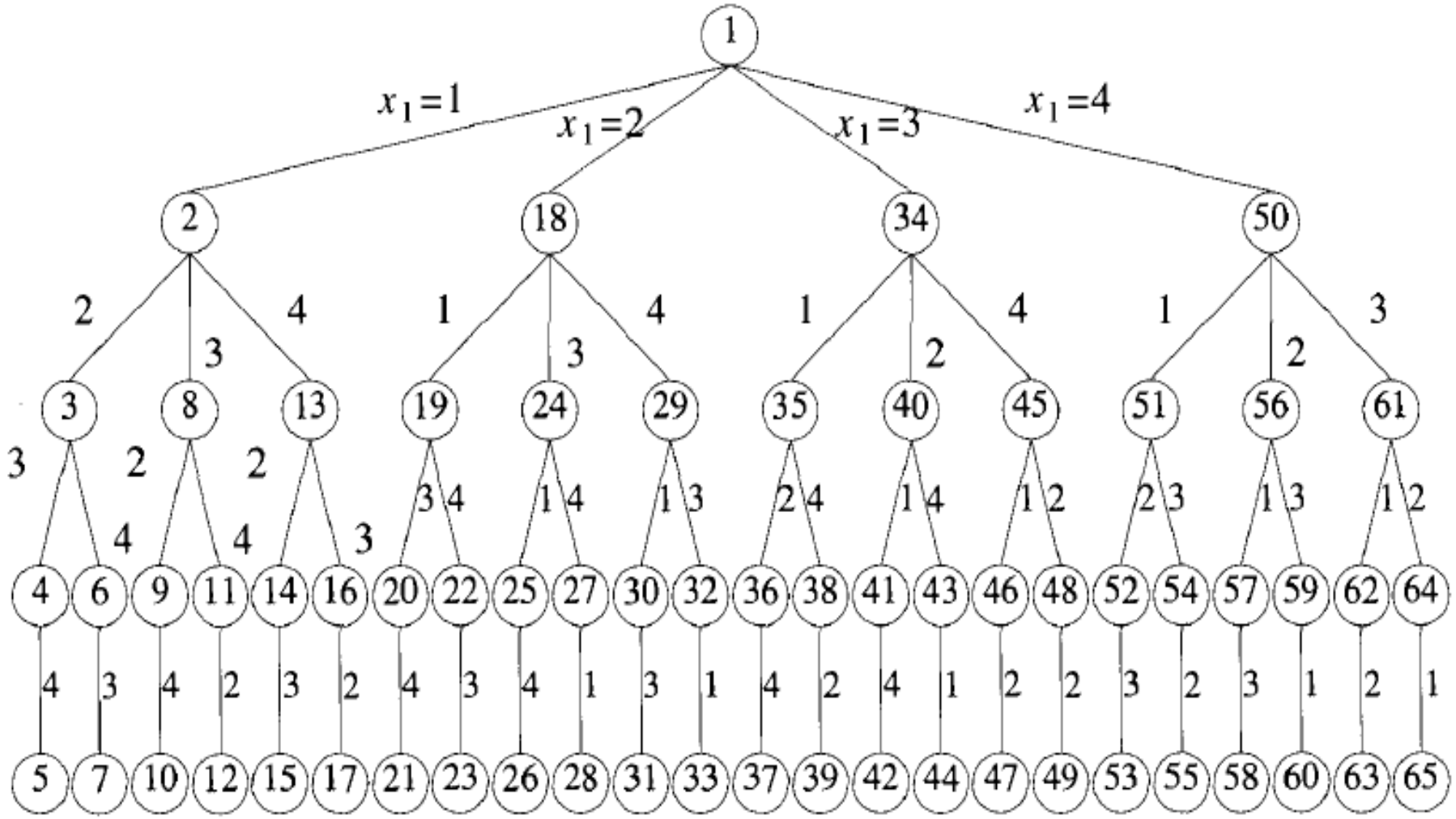


Solution 2

4 Queen Problem



Tree organization of the 4-queenssolutionspace :DFS

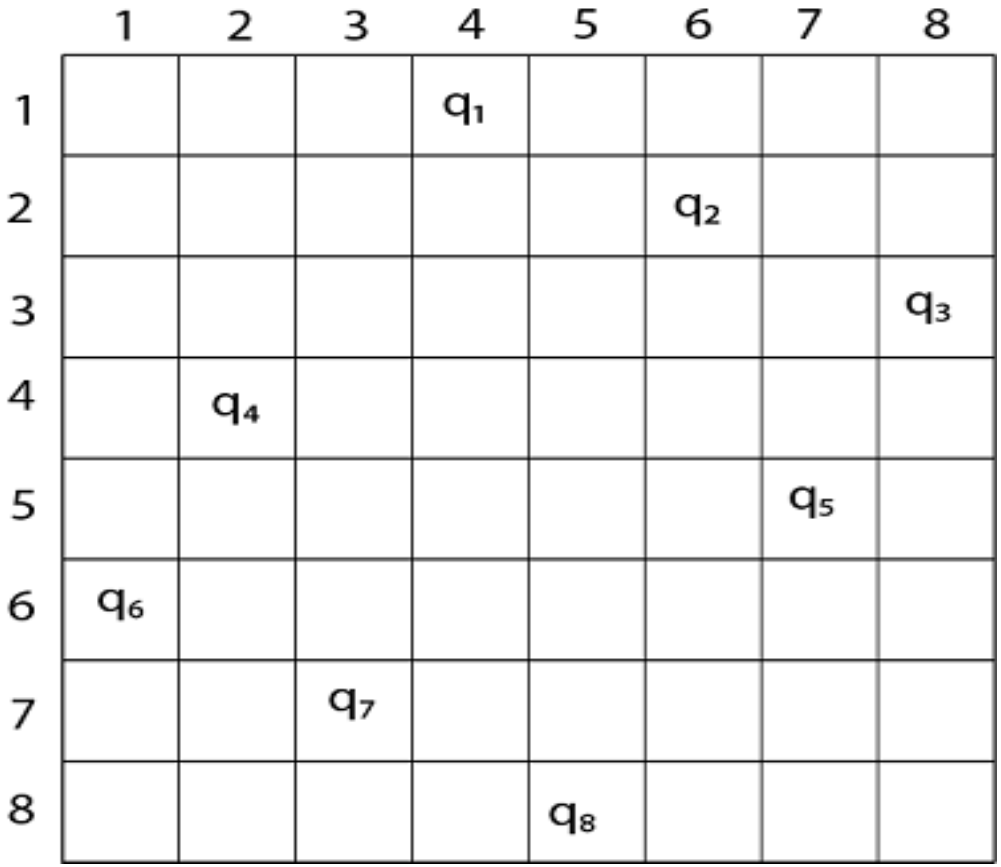


8-queens problem

- **8-queens problem:** The aim of this problem is to place eight queens on a chessboard in an order where no queen may attack another. A queen can attack other queens either **diagonally or in same row and column.**

A solution **for** 8 -queen problem

(4, 6, 8, 2, 7, 1, 3, 5)



Problem formulation : [two main kinds of formulation]

[1] Incremental formulation: It starts from an empty state where the operator augments a queen at each step.

Following steps are involved in this formulation:

1. **States:** Arrangement of any 0 to 8 queens on the chessboard.
2. **Initial State:** An empty chessboard
3. **Actions:** Add a queen to any empty box.
4. **Transition model:** Returns the chessboard with the queen added in a box.
5. **Goal test:** Checks whether 8-queens are placed on the chessboard without any attack.
6. **Path cost:** There is no need for path cost because only final states are counted.

- In this formulation, there is approximately **1.8×10^{14}** possible sequence to investigate.

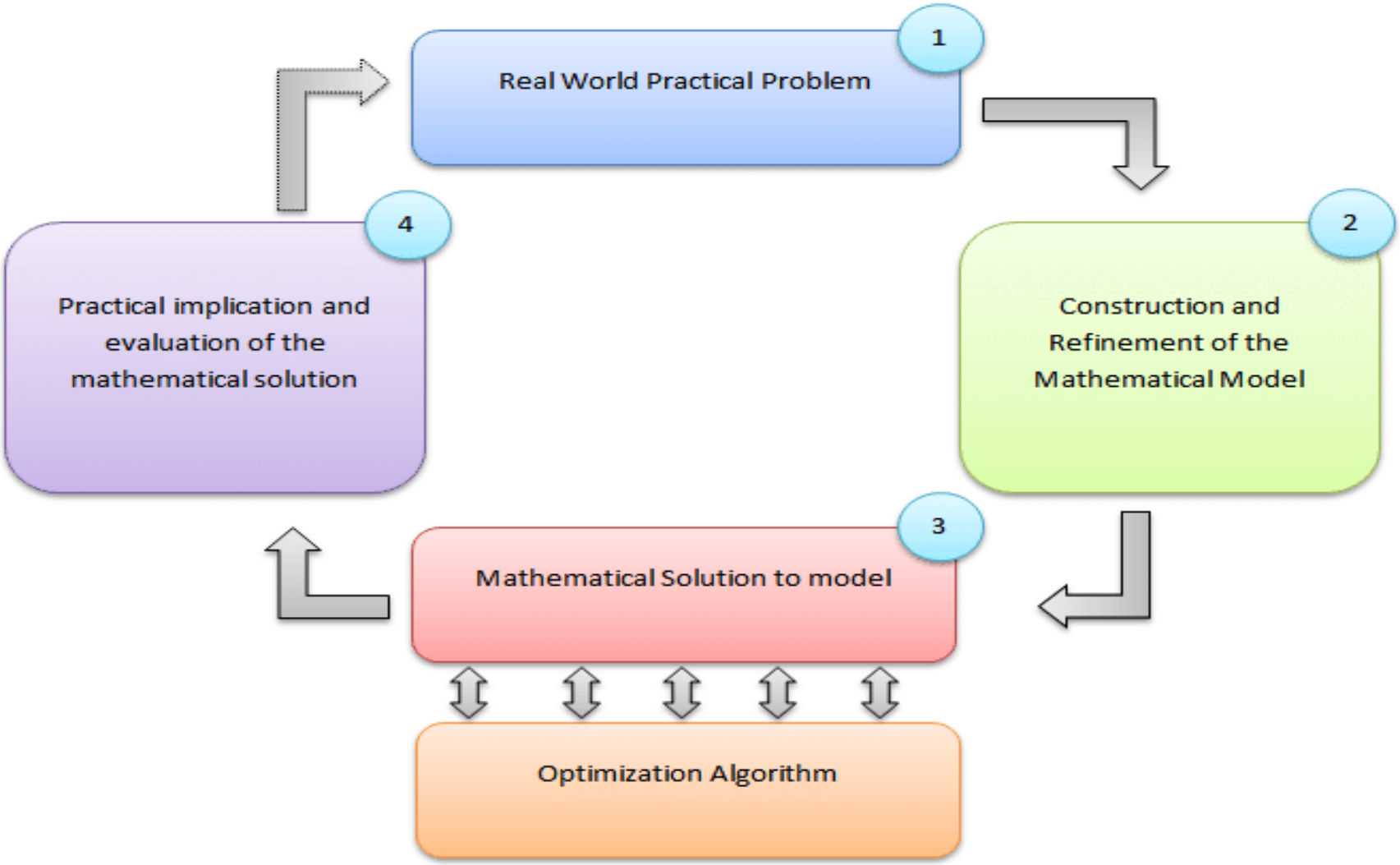
Problem formulation : [two main kinds of formulation]

[2] Complete-state formulation: It starts with all the 8-queens on the chessboard and moves them around, saving from the attacks.

Following steps are involved in this formulation

- **States:** Arrangement of all the 8 queens one per column with no queen attacking the other queen.
- **Actions:** Move the queen at the location where it is safe from the attacks.
- This formulation is better than the incremental formulation as it reduces the state space from **1.8×10^{14}** to **2057**, and it is easy to find the solutions.

Solution to Real world problem



Some Real-world problems

1. **Traveling salesperson problem(TSP):** It is a **touring problem** where the salesman can visit each city only once. The objective is to find the shortest tour and sell-out the stuff in each city.
2. **VLSI Layout problem:** In this problem, millions of components and connections are positioned on a chip in order to minimize the area, circuit-delays, stray-capacitances, and maximizing the manufacturing yield. The layout problem is split into two parts:
   - **Cell layout:** Here, the primitive components of the circuit are grouped into cells, each performing its specific function. Each cell has a fixed shape and size. The task is to place the cells on the chip without overlapping each other.
   - **Channel routing:** It finds a specific route for each wire through the gaps between the cells.

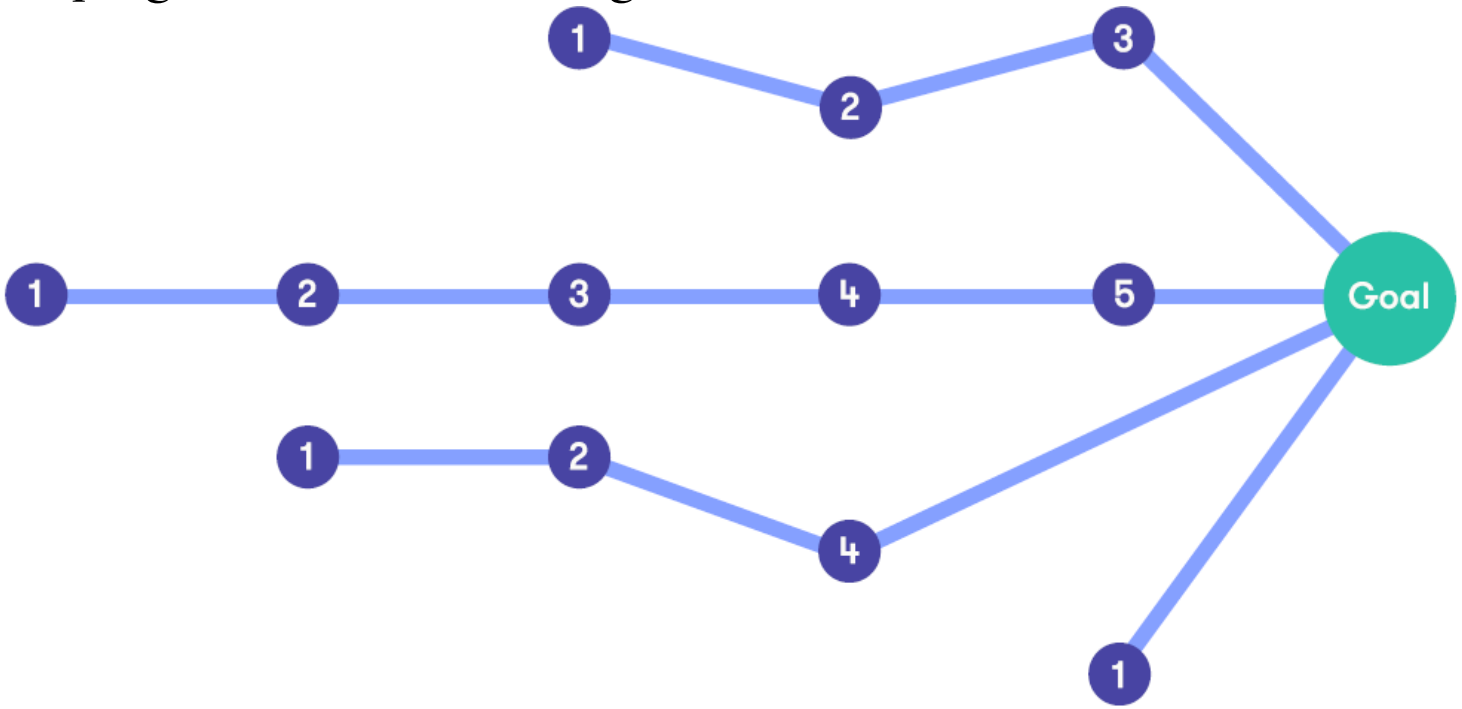
Search and problem solving

- Many problems can be phrased as search problems. This requires that we start by formulating the alternative choices and their consequences.
- **Search in practice: getting from A to B ?**

- This question belongs to the class of search and planning problems. Similar problems need to be solved by self-driving cars, and (perhaps less obviously) AI for playing games.

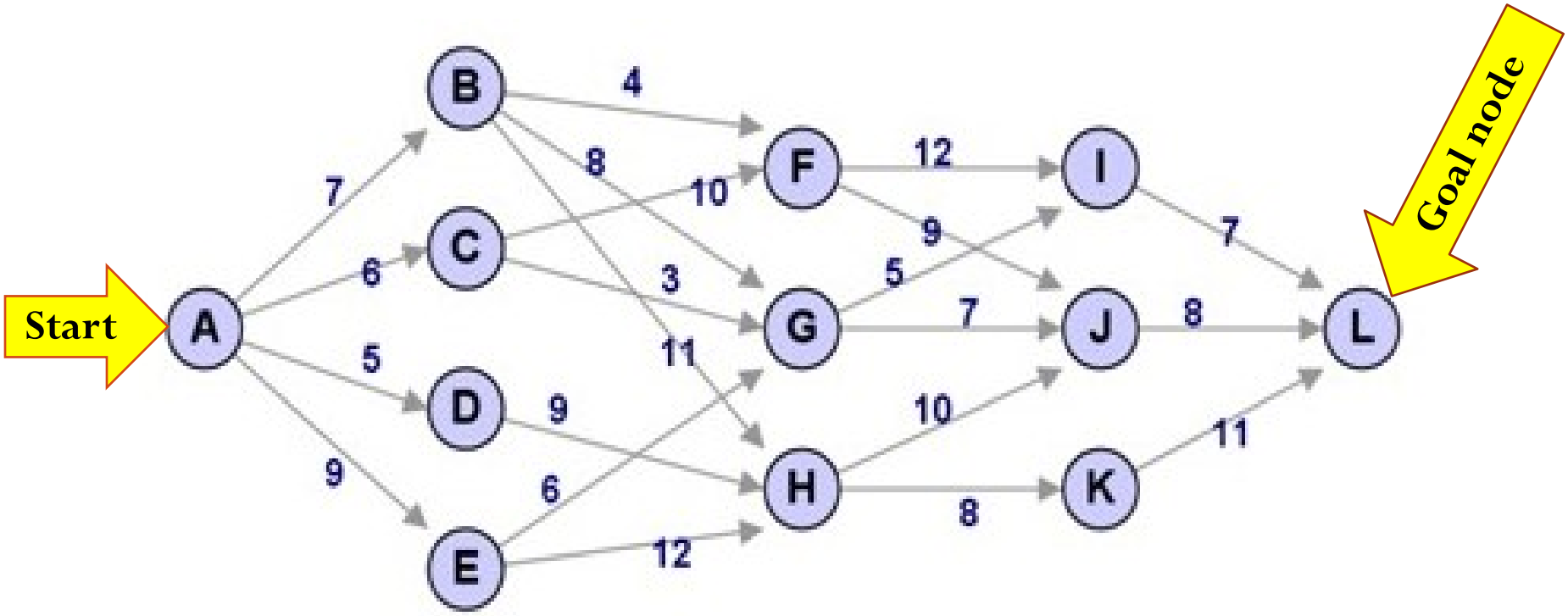
Many different ways to solve the problem

- Often there are many different ways to solve the problem, some of which may be more preferable in terms of time, effort, cost or other criteria.
- Different **search techniques** may lead to different solutions, and developing advanced search algorithms is an established research area.



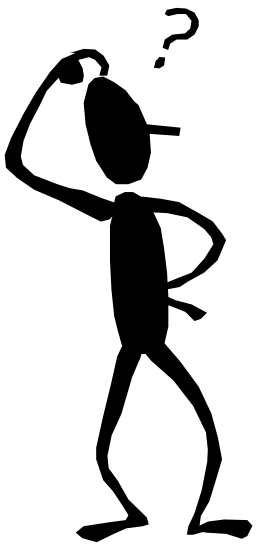
Finding solutions in a Multistage Graph

- The main objective is to select those path which have minimum cost. So we can say that it is an minimization problem.



Searching for solutions

- There may be many solutions to a particular problem. If you can think of the task you want your agent to perform in these terms, then you will need to write a problem solving **agent** which uses search.
- In Artificial Intelligence, Search techniques are universal problem-solving methods. Rational agents or **Problem-solving agents** in AI mostly used these search strategies or algorithms to solve a specific problem and provide the best result.
- **Problem-solving agents** are the goal-based agents and use atomic representation.



- Problem SOLVING ?

If we are solving some problem, we are usually looking for some solution, which will be the best among others.

Problem Taxonomy

Problem Taxonomy

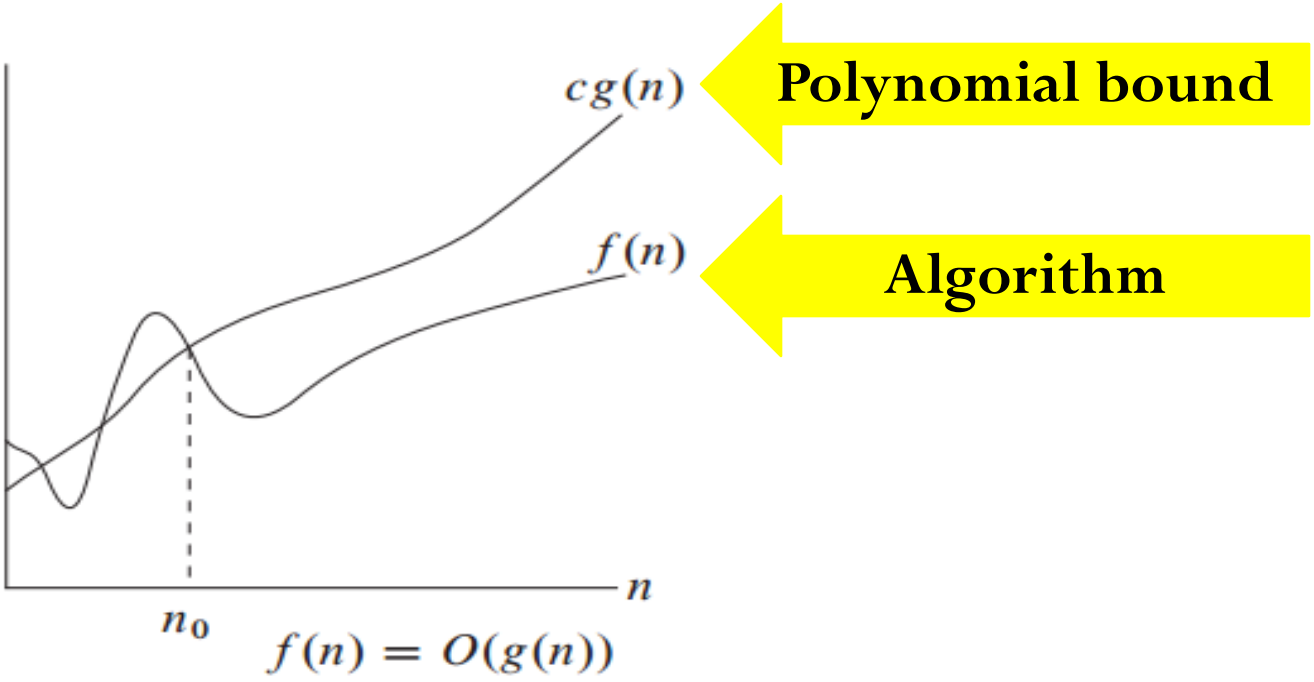
- **Un-decidable Problem:** Do not have algorithms of any known complexity (polynomial or super-polynomial)

- **Intractable problem:** Have algorithms with super polynomial time complexity.

- **Tractable problem:** Have good algorithms with polynomial time complexity (irrespective of the degree)

Polynomial bound

- The function f(n) represents that, how running time of the algorithm/program is increasing when giving larger inputs to the problem.
- Let another function g(n) which is always greater than f(n) after some limit $n = n_0$.
- Therefore we say f(n) = O(g(n)), in the condition that, f(n) <= c g(n), where $n >= n_0$., c > 0, $n_0 >= 1$. This says that f(n) is smaller than g(n).



Different type of decidable Problem

[1] Decision Problems: The class of problems, the output is either yes or no.

- **Example** : Whether a given number is prime?

[2] Counting Problem: The class of problem, the output is a natural number

- **Example** :How many distinct factor are there for a given number

[3] Optimization Problem: The class of problem with some **objective function** based on the problem instance

- **Example** :**Finding a minimal spanning tree** for a weighted graph

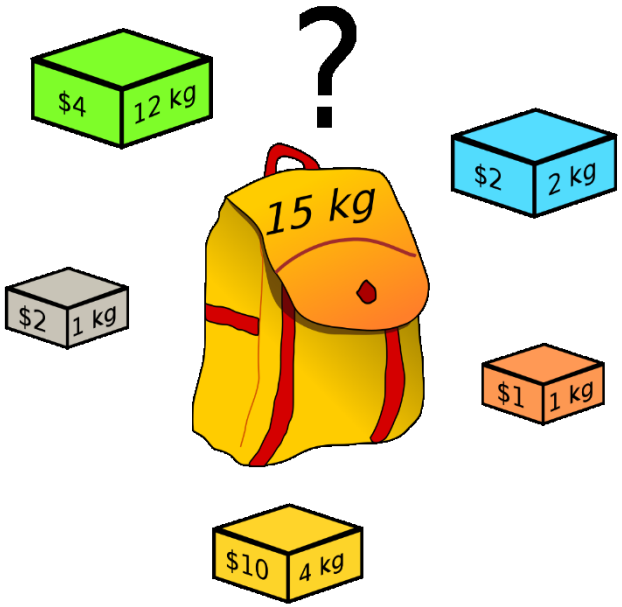
Three general categories of intractable problems

1. Problems for which polynomial-time algorithms have been found .
2. Problems that have been proven to be intractable.
3. Problems that have not been proven to be intractable, but for which polynomial-time algorithms have never been found .

- It is a surprising phenomenon that most problems in computer sciences seem to fall into either the first or third category.

0-1 Knapsack Problem

0-1 Knapsack Problem
Given a set of items, each with a weight and benefit, determine the items to include in a collection so that the total weight is less than or equal to a given weight limit and the total benefit is maximized



Knapsack problem; introduction

- The classical knapsack problem assumes that each item must be put entirely in the knapsack or not included at all. It is this 0/1 property that makes the knapsack problem difficult.
- In the **0/1 Knapsack problem**, you are given a bag which can hold W kg. There is an array of items each with a different weight and price value assigned to them.
- The objective of the 0/1 Knapsack is to **maximize the value** you put into the bag. You are allowed to only take all or nothing of a given item.

Knapsack problem

- *Knapsack* problem : Suppose we have n integers $a_1, a_2, \ldots, a_n$ and a constant W. We want to find a subset of integers so that their sum is less than or equal to but is as close to W as possible. There are 2^n subsets. ($n = 100$, $2^n = 1.26765 \times 10^{30}$, it takes **$4.01969 \times 10^{12}$** years if our computer can examine **10^{10}** subsets per second.)

- A problem is considered *tractable* (computationally easy, $O(n^k)$) if it can be solved by an efficient algorithm and *intractable* (computationally difficult, the lower bound grows fast than n^k) if there is no efficient algorithm for solving it.

- The class of *NP-complete* problems: There is a class of problems, including TSP and Knapsack, for which no efficient algorithm is currently known.

Knapsack problem

There are two types of knapsack problem.

1. **0-1 knapsack problem**: In 0-1 knapsack problem each item either be taken or left behind.

2. **Fractional knapsack problem**: In fractional knapsack problem fractions of items are allowed to choose.

Knapsack problem; introduction

- The classical knapsack problem assumes that each item must be put entirely in the knapsack or not included at all. It is this 0/1 property that makes the knapsack problem difficult.
- In the **0/1 Knapsack problem**, you are given a bag which can hold W kg. There is an array of items each with a different weight and price value assigned to them.
- The objective of the 0/1 Knapsack is to **maximize the value** you put into the bag. You are allowed to only take all or nothing of a given item.

0-1 Knapsack Problem

Given a knapsack of capacity C and n objects of volumes $\{w_1, w_2 \ldots w_n\}$ and profits $\{p_1, p_2 \ldots p_n\}$, the objective is to choose a subset of n objects that fits into the knapsack and that maximizes the total profit.

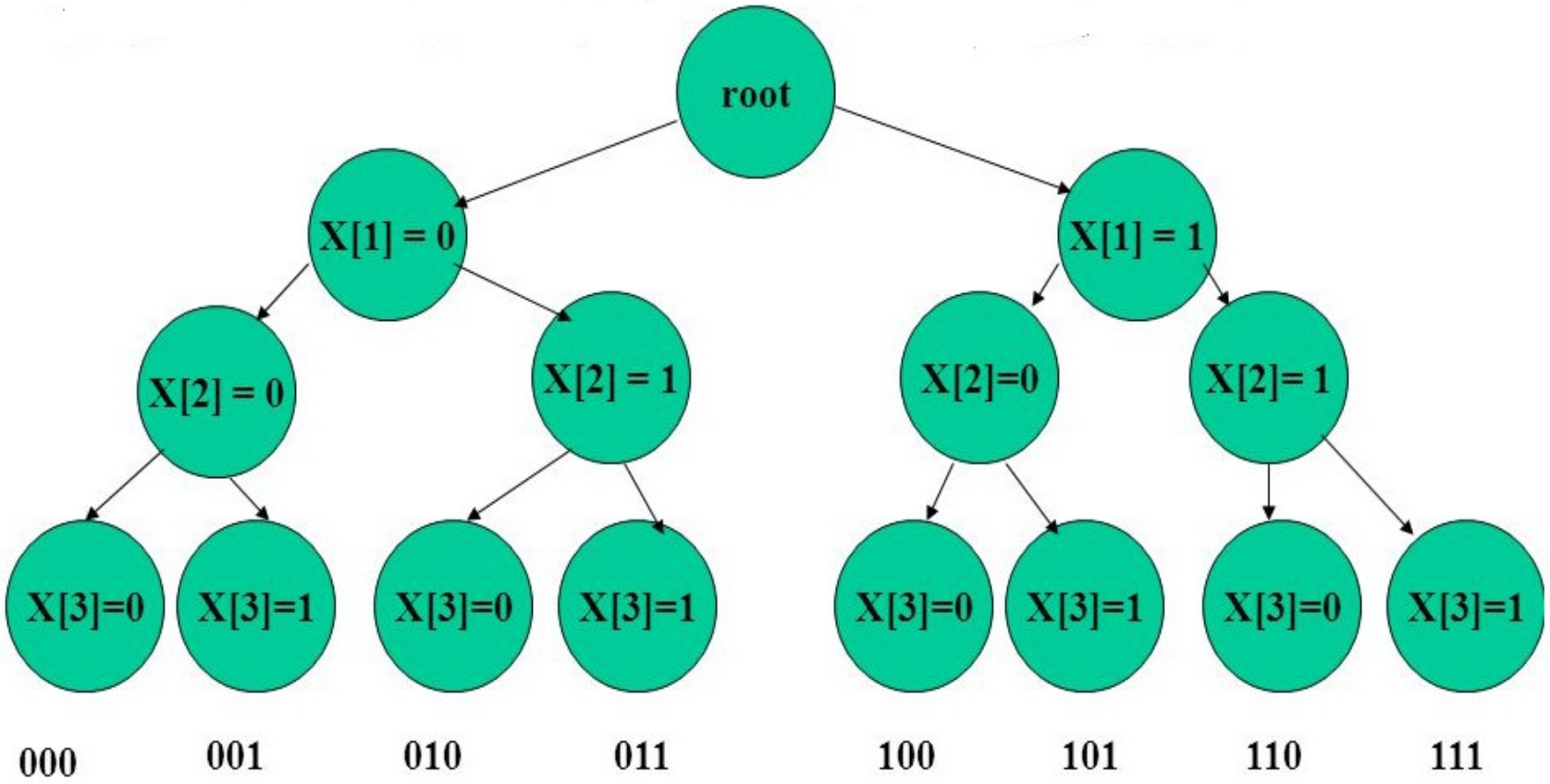
In more formal terms, let x_i be 1 if object i is present in the subset and 0 otherwise.

The knapsack problem can be stated as

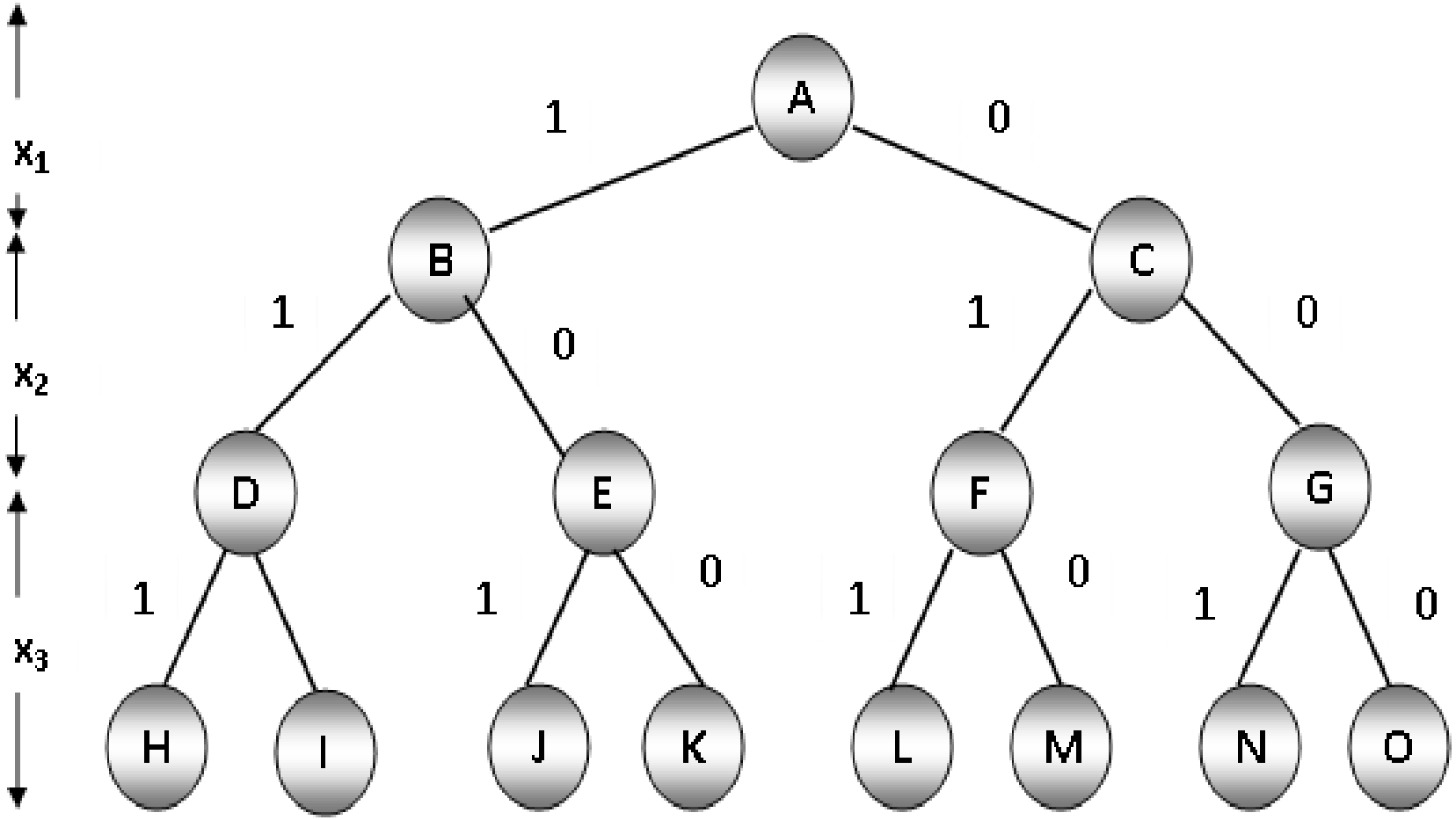
$$\text{Maximize} \sum_{i=0}^{n} x_i \cdot p_i \quad \text{subject to} \quad \sum_{i=0}^{n} x_i \cdot w_i \leq C$$

Note that the constraint $x_i \in \{0, 1\}$ is not linear. A simplistic approach will be to enumerate all subsets and select the one that satisfies the constraints and maximizes the profits. Any solution that satisfies the capacity constraint is called a *feasible* solution. The obvious problem with this strategy is the running time which is at least 2^n corresponding to the power-set of n objects.

Solution tree for 3 item 0/1 knapsack problem



Solution tree for 3 item 0/1 knapsack problem



Knapsack problem

- *Knapsack* problem : Suppose we have n integers $a_1, a_2, \ldots, a_n$ and a constant W. We want to find a subset of integers so that their sum is less than or equal to but is as close to W as possible. There are 2^n subsets. ($n = 100$, $2^n = 1.26765 \times 10^{30}$, it takes 4.01969×10^{12} years if our computer can examine 10^{10} subsets per second.)

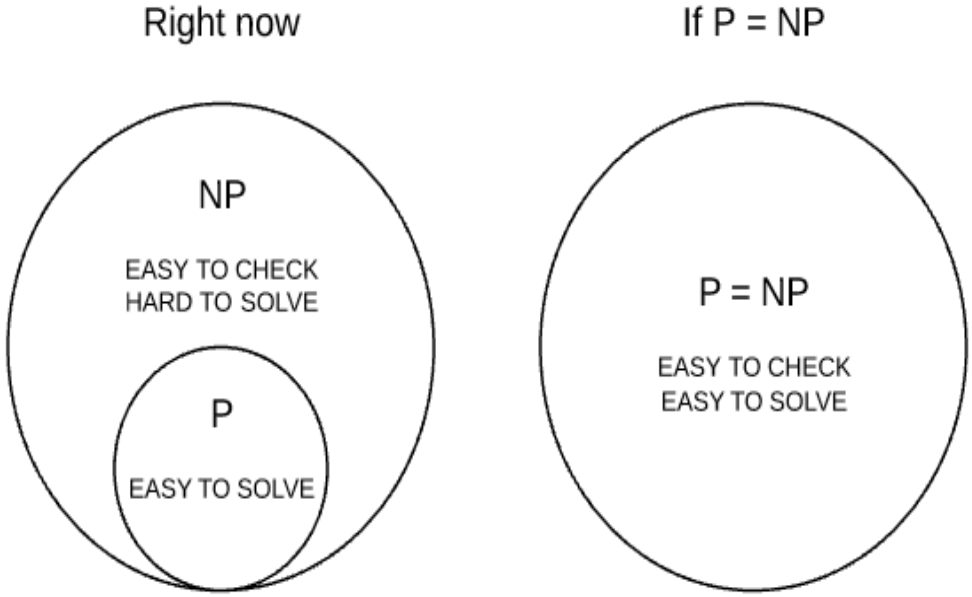
- A problem is considered *tractable* (computationally easy, $O(n^k)$) if it can be solved by an efficient algorithm and *intractable* (computationally difficult, the lower bound grows fast than n^k) if there is no efficient algorithm for solving it.

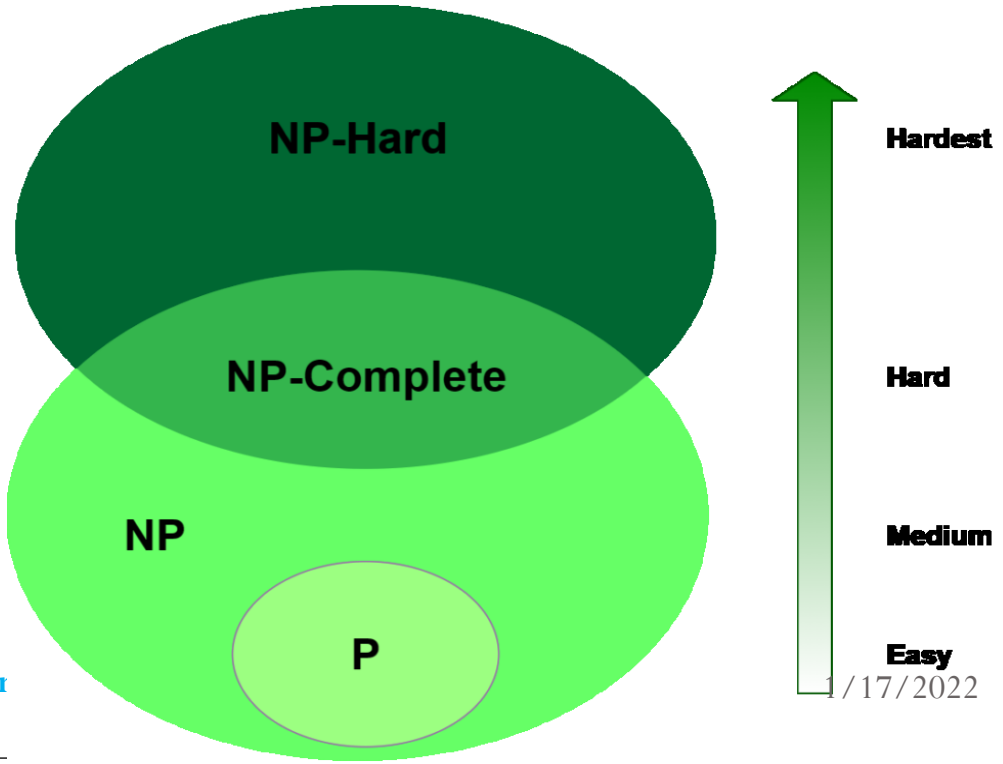
- The class of *NP-complete* problems: There is a class of problems, including TSP and Knapsack, for which no efficient algorithm is currently known.

knapsack problem is NP-complete

- The knapsack problem is NP-complete because the known NP-complete problem **subset-sum** is polynomially reducible to the **knapsack problem**, hence every problem in NP is reducible to the knapsack problem.
- The fact that we know an O(nW) algorithm for knapsack doesn't contradict the possibility that P≠NP because O(nW) is not polynomial in the length of the input to the algorithm, which is O(nlogW). If we ever did find an algorithm that ran in time polynomial in n and logW that would prove that P=NP.
- Algorithms that are polynomial in the magnitudes of coefficients in the input instead of their sizes are called *pseudo polynomial*. An NP-complete problem might or might not have a pseudo polynomial algorithm. Those that don't (assuming P≠NPP≠NP) are *strongly NP-complete*. The knapsack problem is not strongly NP-complete, but it is still NP-complete.

P vs NP Problem





0-1 knapsack problem is pseudo-polynomial

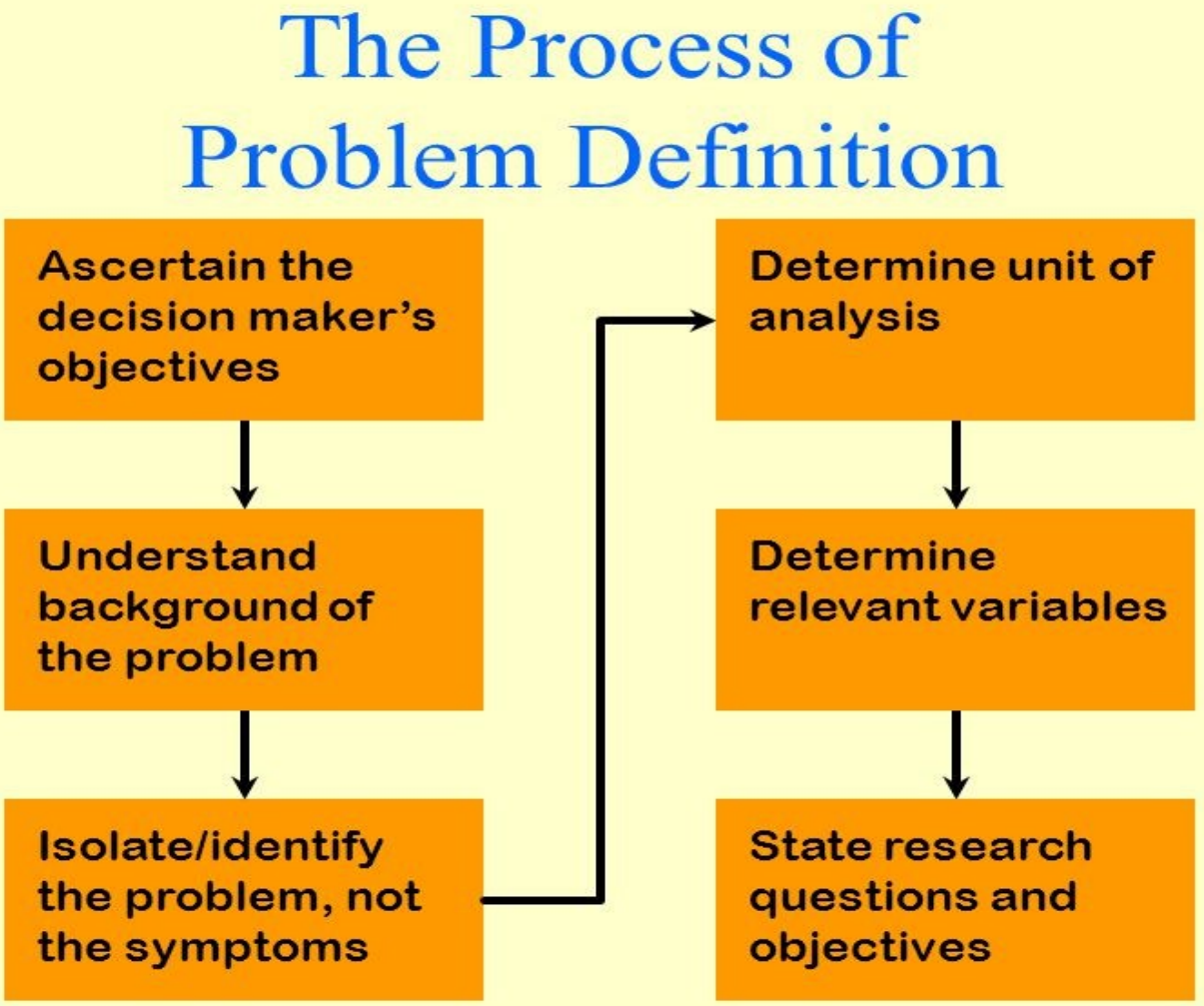
- knapsack problem is NP-complete while it can be solved by DP. They say that the DP solution is pseudo-polynomial, since it is exponential in the "length of input" (i.e. the numbers of bits required to encode the input).
- The running time is O(NW) for an unbounded knapsack problem with N items and knapsack of size W. W is not polynomial in the length of the input though, which is what makes it *pseudo*-polynomial.
- Consider W = 1,000,000,000,000. It only takes 40 bits to represent this number, so input size = 40, but the computational runtime uses the factor 1,000,000,000,000 which is $O(2^{40})$.
- So the runtime is more accurately said to be $O(N.2^{\text{bits in W}})$, which is **exponential**.

Running times for different sizes of input.

n	$\log n$	n	$n \log n$	n^2	n^3	2^n
8	3 nsec	0.01 μ	0.02 μ	0.06 μ	0.51 μ	0.26 μ
16	4 nsec	0.02 μ	0.06 μ	0.26 μ	4.10 μ	65.5 μ
32	5 nsec	0.03 μ	0.16 μ	1.02 μ	32.7 μ	4.29 sec
64	6 nsec	0.06 μ	0.38 μ	4.10 μ	262 μ	5.85 cent
128	0.01 μ	0.13 μ	0.90 μ	16.38 μ	0.01 sec	10^{20} cent
256	0.01 μ	0.26 μ	2.05 μ	65.54 μ	0.02 sec	10^{58} cent
512	0.01 μ	0.51 μ	4.61 μ	262.14 μ	0.13 sec	10^{135} cent
2048	0.01 μ	2.05 μ	22.53 μ	0.01 sec	1.07 sec	10^{598} cent
4096	0.01 μ	4.10 μ	49.15 μ	0.02 sec	8.40 sec	10^{1214} cent
8192	0.01 μ	8.19 μ	106.50 μ	0.07 sec	1.15 min	10^{2447} cent
16384	0.01 μ	16.38 μ	229.38 μ	0.27 sec	1.22 hrs	10^{4913} cent
32768	0.02 μ	32.77 μ	491.52 μ	1.07 sec	9.77 hrs	10^{9845} cent
65536	0.02 μ	65.54 μ	1048.6 μ	0.07 min	3.3 days	10^{19709} cent
131072	0.02 μ	131.07 μ	2228.2 μ	0.29 min	26 days	10^{39438} cent
262144	0.02 μ	262.14 μ	4718.6 μ	1.15 min	7 mnths	10^{78894} cent
524288	0.02 μ	524.29 μ	9961.5 μ	4.58 min	4.6 years	10^{157808} cent
1048576	0.02 μ	1048.60 μ	20972 μ	18.3 min	37 years	10^{315634} cent

Note: “nsec" stands for nanoseconds, “ μ”is one microsecond and “cent" stands for centuries. The explosive running time (measured in centuries) when it is of the order 2^n.

Problem definition



Need for Programming Language

- We have some problem.
- We have to solve the problem.
- We can solve the problem, but we don't want to repeatedly solve the similar problems.
- We want to use the computer to do this task for us whenever required.
- We have to tell the computer, "How to solve the problem."

Problem and Problem Solving

- **What is Problem?**

❑ A problem is an obstacle which makes it difficult to achieve a desired goal, objective or purpose.

❑ It refers to a situation, condition, or issue that is yet unresolved.

❑ In a broad sense, a problem exists when an individual becomes aware of a significant difference between what actually is and what is desired.

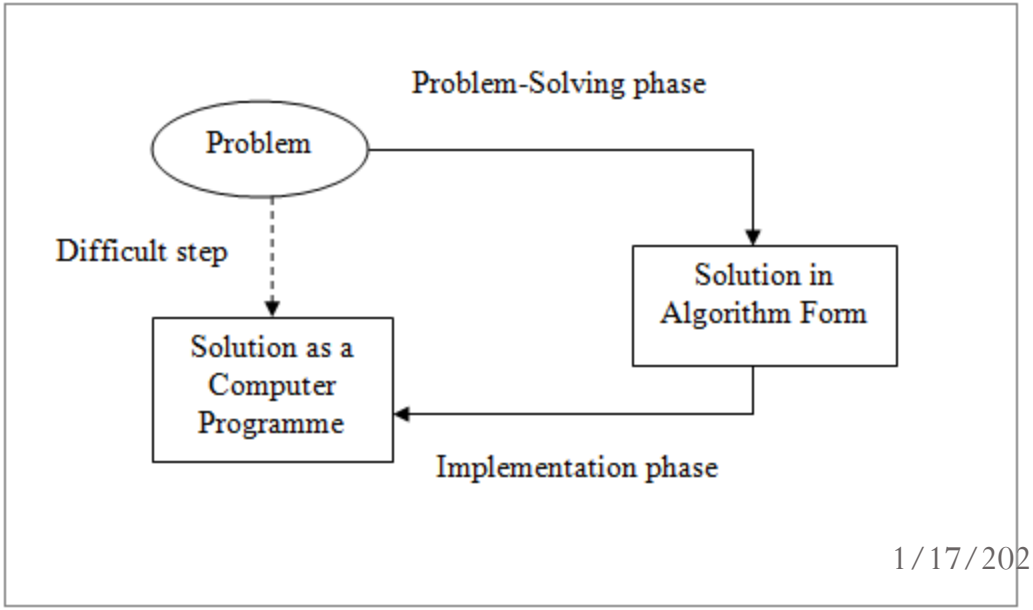
- **What is Problem Solving?**

❑ Every problem in the real world needs to be identified and solved. Trying to find a solution to a problem is known as problem solving.

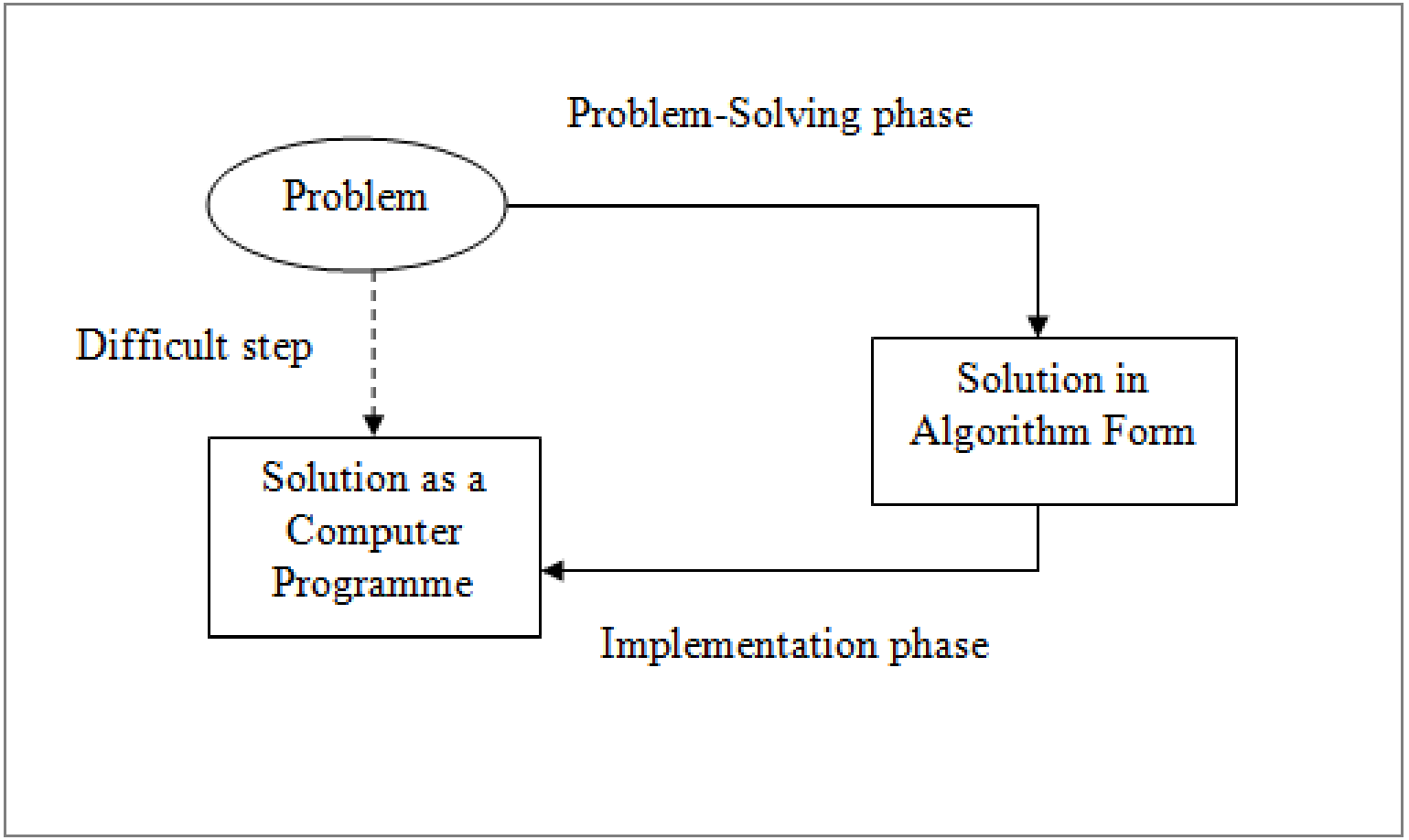
❑ Problem solving becomes a part of our day to day activity. Especially, when we use computers for performing our day to day activity.

How to solve a problem?

1. If you have a problem, either you can solve it manually or using computer.
2. If the problem is easy enough, solve it manually or else use computers.
3. Bigger problems can be sub-divided into smaller problems (sub-problems) and start solving them one by one.
4. After completing solving sub-problems, the entire big problem has been solved easily.



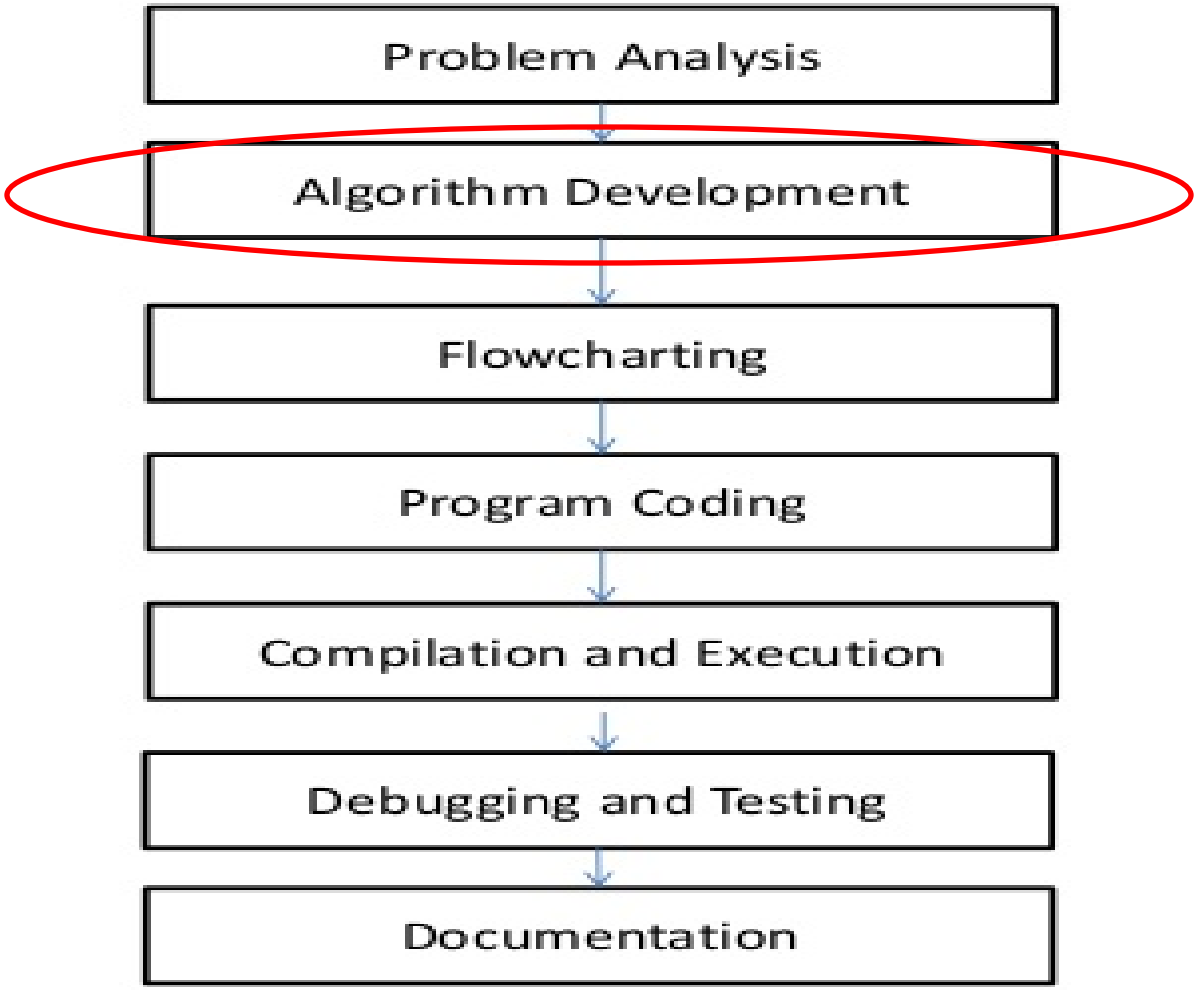
Problem solving with computer



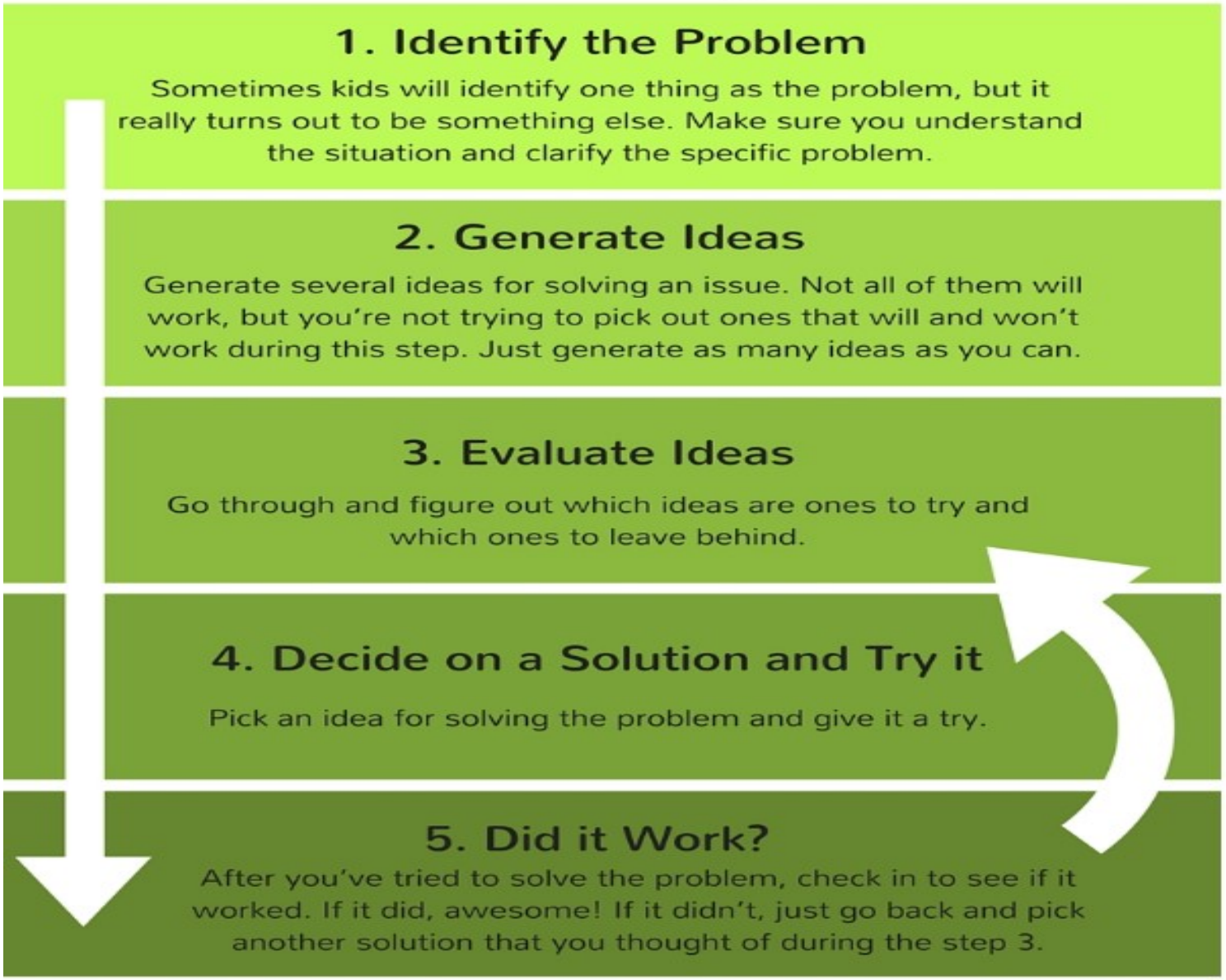
Problem Solving

- According to psychology, "*a problem-solving refers to a state where we wish to reach to a definite goal from a present state or condition.*"
- According to computer science, *a problem-solving is a part of artificial intelligence which encompasses a number of techniques such as algorithms, heuristics to solve a problem.*

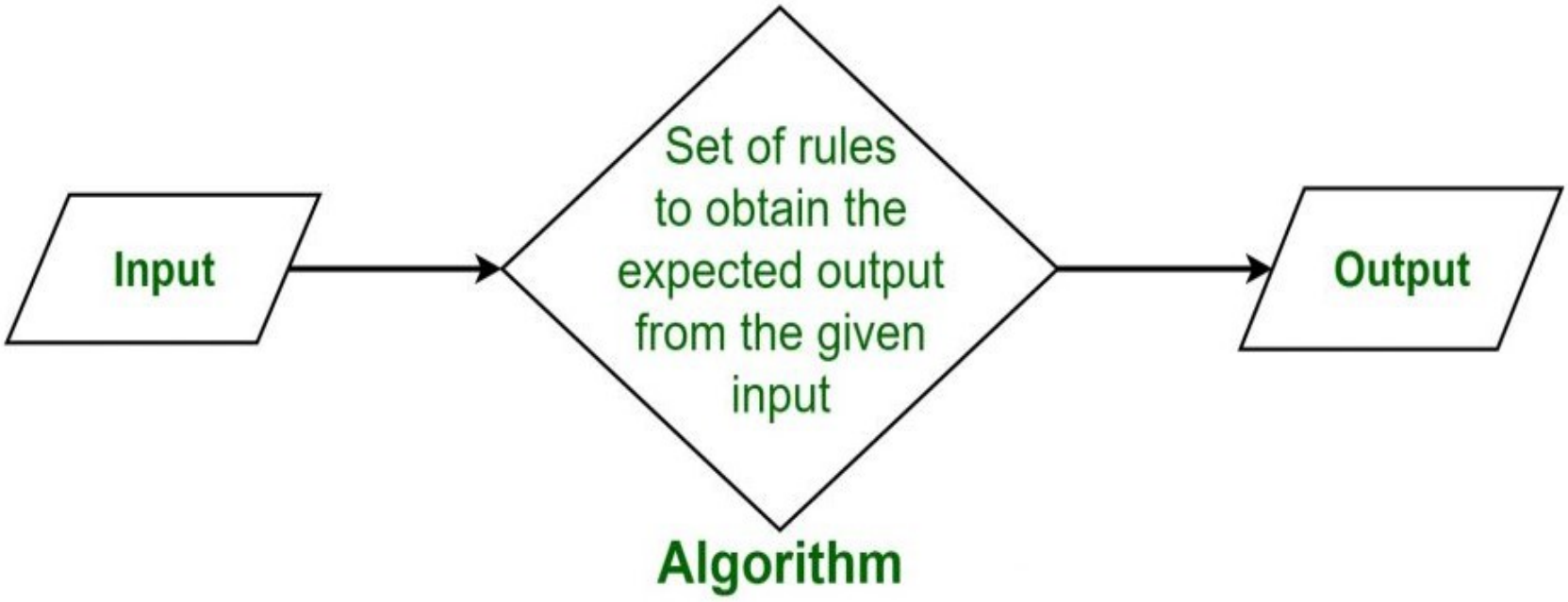
Steps in Problem Solving



5 Steps to Solving a Problem



What is Algorithm?



Algorithm

- Algorithm: is a procedure that consists of a ***finite set of instructions*** which, given an ***input*** from some set of possible inputs, enables us to obtain an *output* if such an output exists or else obtain nothing at all if there is no output for that particular input through a ***systematic execution*** of the ***instructions.***

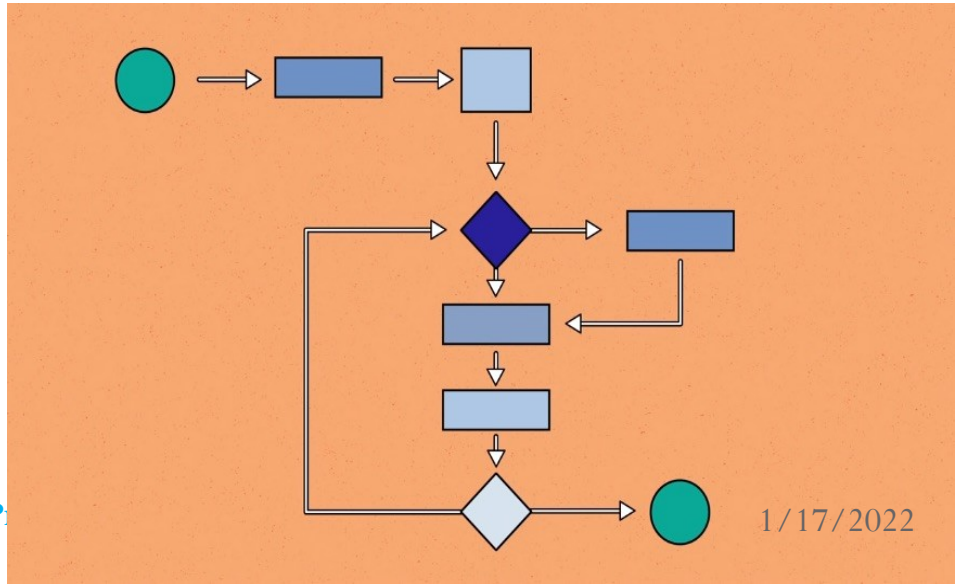
- An algorithm is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

- As a computer professional, it is useful to know a standard set of important algorithms and to be able to design new algorithms as well as to analyze their efficiency.

Algorithm

A clearly specified set of instructions to solve a problem.

- Characteristics:
  - **Input:** Zero or more quantities are externally supplied
  - **Definiteness:** Each instruction is clear and unambiguous
  - **Finiteness:** The algorithm terminates in a finite number of steps.
  - **Effectiveness:** Each instruction must be primitive and feasible
  - **Output**: At least one quantity is produced

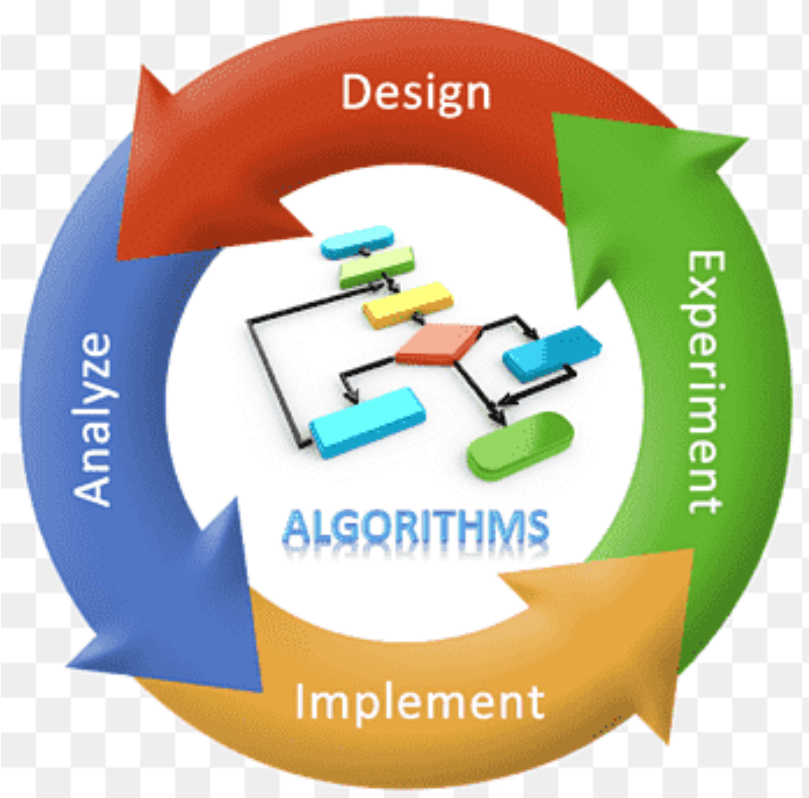


What is an algorithm?

- An algorithm is a procedure to accomplish a specific task.
- An algorithm is the idea behind any reasonable computer program.
- To be interesting, an algorithm must solve a general, well-specified *problem*.
- *An* algorithmic problem is specified by describing the complete set of *instances it must* work on and of its output after running on one of these instances.
- This distinction, between a problem and an instance of a problem, is fundamental.
- **An *algorithm is a procedure that takes any of the possible input instances* and transforms it to the desired output.**

Design and Analysis of Algorithms

- ***Analysis:*** predict the cost of an algorithm in terms of resources and performance
- ***Design:*** design algorithms which minimize the cost

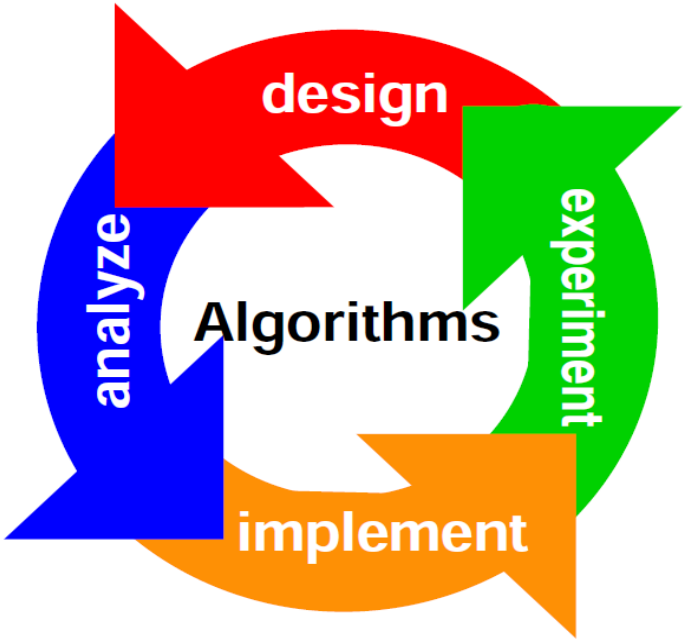


Algorithm development

- **Algorithm development** is the act of designing the steps that solve a particular problem for a computer or **any other device** to follow not excluding human being, but in this case computers only and computer like devices.

Steps in development of Algorithms

1. Problem definition
2. Development of a model
3. Specification of Algorithm
4. **Designing an Algorithm**
5. Checking the correctness of Algorithm
6. **Analysis of Algorithm**
7. Implementation of Algorithm
8. Program testing
9. Documentation Preparation



Defining Algorithm

- An algorithm is a finite set of instructions that, is followed, accomplishes a particular task. In addition, all algorithms must satisfy the following criteria:

1. Input: Zero or more quantities
2. Output: At least one
3. Definiteness : Each instruction is clear and unambiguous
4. Finiteness: Algorithm terminates in finite number of steps
5. Effectiveness: Every instruction must be very basic and feasible.

* *Horowitz, Sahni @ Rajasekaran : Fundamentals of Algorithms*

Defining Algorithm

- Criteria1and 2 require that an algorithm produce one or more outputs and have zero or more inputs that are externally supplied.
- According to criterion 3, each operation must be definite, meaning that it must be perfectly clear what should be done.
- The fourth criterion for algorithms is that it terminate after a finite number of operations.
- Criterion 5 requires that each operation be effective; each step must be such that it can, at least in principle, be done by a person using pencil and paper in a finite amount of time.

Defining Algorithm

- An algorithm produces one or more outputs and may have zero or more inputs which are externally supplied.
- Algorithms that are definite and effective are also called computational procedure.
- Example of a computational procedure is the **operating system** of a digital computer.
- A **program** is the expression of an algorithm in a programming language.
- In order to help us achieve the criterion of **definiteness**, algorithm will be written in a programming language.

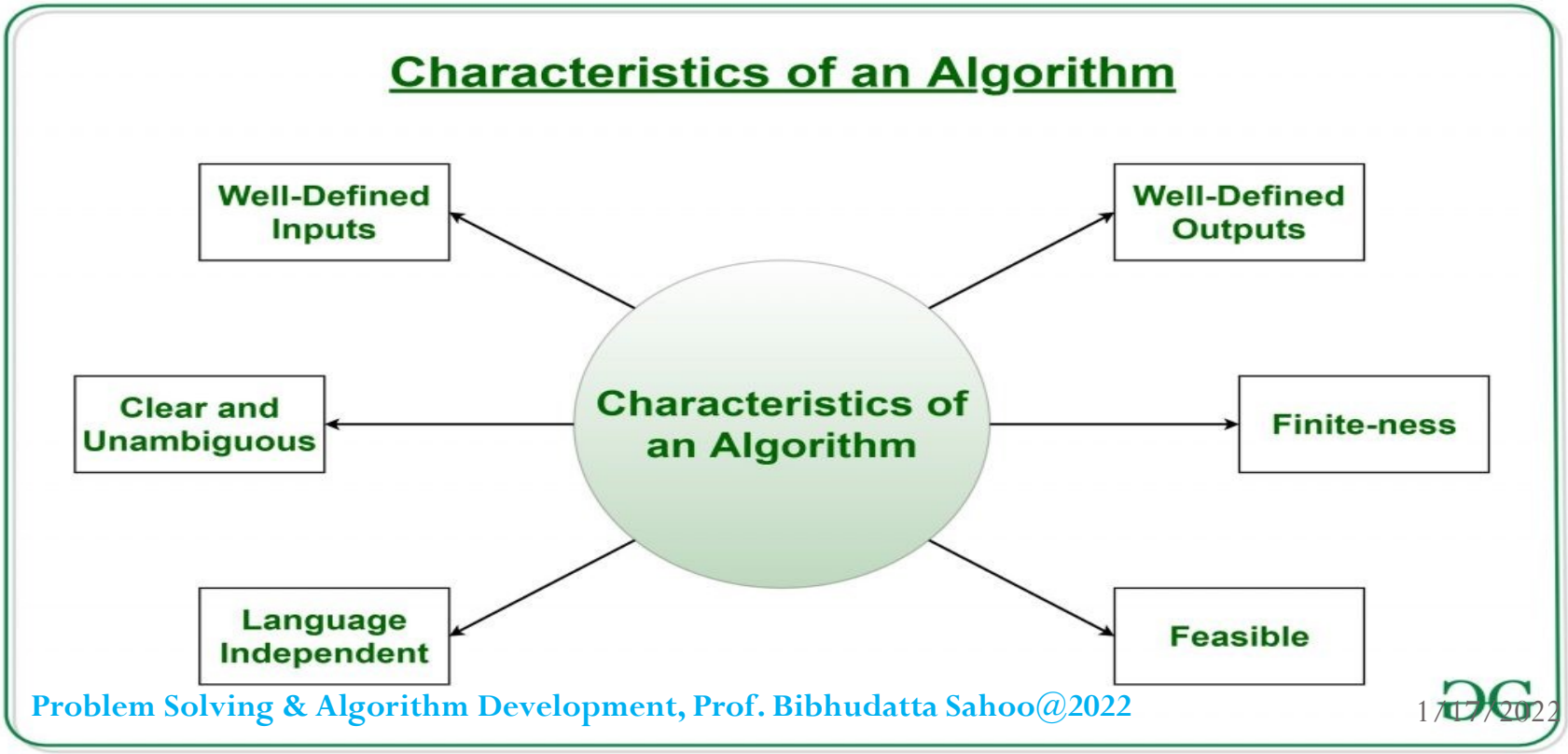
Algorithm Properties

- An algorithm possesses the following properties:
  - It must be correct.
  - It must be composed of a series of concrete steps.
  - There can be no ambiguity as to which step will be performed next.
  - It must be composed of a finite number of steps.
  - It must terminate.

- A **computer program** is an instance, or concrete representation, for an algorithm in some programming language.

Defining algorithm ??

- An algorithm is a *well-ordered* collection of *unambiguous* and *effectively computable* operations that, when executed, *produces a result* and *halts in a finite amount of time*.



Active area of algorithm research

The study of algorithms includes many important and active areas of research.

1. How to devise algorithms
2. How to analyze algorithms
3. How to express algorithms
4. How to validate algorithms
5. How to test algorithms

1. How to devise algorithms

- The act of creating an algorithm is an art which may never be fully automated.
- A major goal of this course is to study various **design techniques** which have proven to be useful in that they have often yielded good algorithms.
- A knowledge of design will certainly help one to create a **good algorithms**, yet without the tools of analysis there is no way to determine the quality of the result.

Design strategies/ techniques:

- divide-and-conquer,
- the greedy method,
- dynamic programming,
- search- and-traversal,
- Backtracking
- & branch-and-bound.

2. How to analyze algorithms

- As an algorithm is executed, it makes use of the computer's central processing unit (CPU) to perform operations and it uses the memory to hold the program and its data.
- **Analysis of algorithms** refers to the process of determining how much computing time and storage an algorithm will require.
- **Analysis of algorithm** is a challenging one which sometimes requires great mathematical skill.
- Analyzing even a simple algorithm can be a challenge. The mathematical tools required may include discrete combinatorics, elementary probability theory, algebraic dexterity, and the ability to identify the most significant terms in a formula.

3. How to express algorithms

- A Program is the expression of an algorithm in a programming language.
- The three most common forms of algorithmic notation are:
  I. **English,**
  II. **Pseudocode,**
  III. **A real programming language.**

4. How to validate algorithms

- Once an algorithm is devised it is necessary to show that it computes the correct answer for all possible legal inputs. We refer to this process as algorithm validation. T
- The purpose of the validation is to assure us that this algorithm will work correctly **independent** of the issues concerning the **programming lang**uage it will eventually be written in.
- Once the validity of the method has been shown, a program can be written and a second phase begins. This phase is referred to as program **verification**.

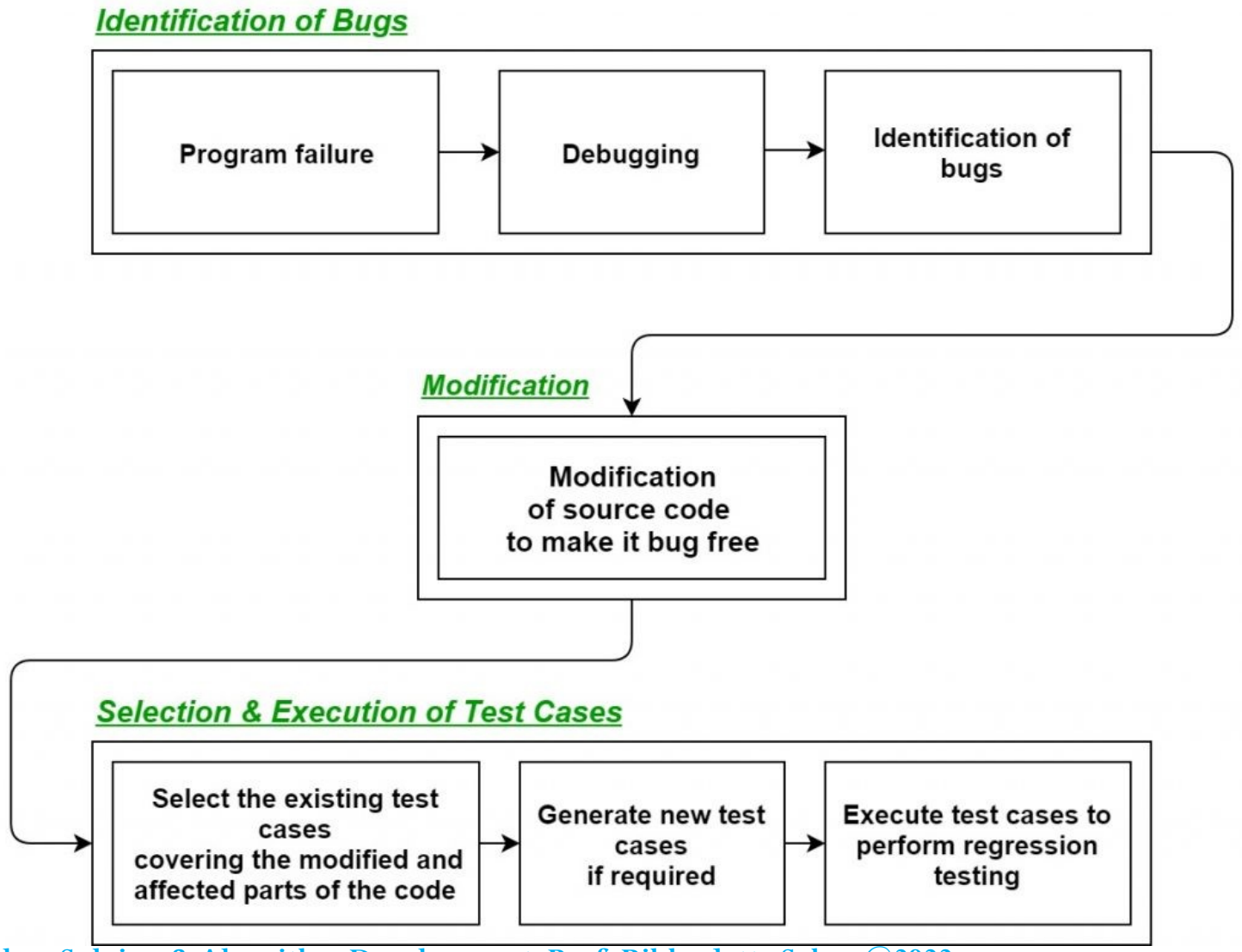
4. How to validate algorithms

- A **proof of correctness** requires that the solution be stated in two forms.
- One form is usually as a program which is annotated by a set of assertions about the input and output variables of the program. These assertions are often expressed in the **predicate calculus**.
- The second form is called a **specification** and this may also be expressed in the predicate calculus.
- A complete proof of program **correctness** requires that each statement of the programming language be precisely defined and that all basic operations be proved correct.

5. How to test algorithms

- Testing a program really consists of two phases : Debugging and Profiling.
- **Debugging** is the process of executing programs on sample data sets to determine if faulty results occur and, if so, to correct them !A proof of correctness is much more valuable than a thousand tests, (if that proof is correct), since it guarantees that the program will work correctly for all possible inputs.
- **Profiling** is the process of executing a correct program on data sets and measuring the time and space it takes to compute the results.
- This timing figures are useful in that they may confirm a previously done analysis and point out logical places to perform useful optimization.

Debugging and testing



Expressing Algorithm

- Reasoning about an algorithm is impossible without a careful description of the sequence of steps to be performed.
- The three most common forms of algorithmic notation are:
  I. **English,**
  II. **Pseudocode,**
  III. **A real programming language.**

Pseudo code

- Structured like a programming language, but ignores many syntactical details (like $ and ;)
- Complex operations can be written in natural language
- Still, we need to agree on some standard operations…

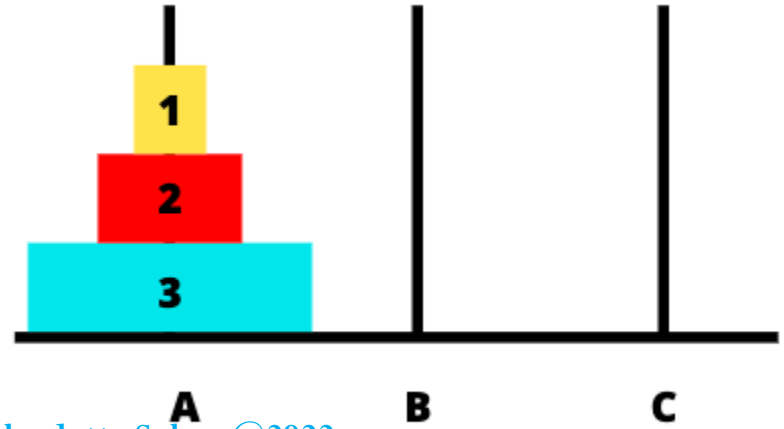
- We are following : Pseudo code as suggested in text "*Horowitz, Sahni @ Rajasekaran : Fundamentals of Algorithms*"
- Two good options for typesetting algorithms in LaTeX with packages, **algorithms** and **algorithm2e**
- *www.ctan.org/tex-archive/macros/**latex**/contrib/**algorithms**/*

Pseudo code for Selection Sort

```
1   Algorithm SelectionSort(a, n)
2   // Sort the array a[1 : n] into nondecreasing order.
3   {
4       for i := 1 to n do
5       {
6           j := i;
7           for k := i + 1 to n do
8               if (a[k] < a[j]) then j := k;
9           t := a[i]; a[i] := a[j]; a[j] := t;
10      }
11  }
```

Pseudo code Towers of Hanoi

```
1   Algorithm TowersOfHanoi(n, x, y, z)
2   // Move the top n disks from tower x to tower y.
3   {
4       if (n ≥ 1) then
5       {
6           TowersOfHanoi(n − 1, x, z, y);
7           write ("move top disk from tower", x,
8           "to top of tower", y);
9           TowersOfHanoi(n − 1, z, y, x);
10      }
11  }
```



Recursive permutation generator

```
1   Algorithm Perm(a, k, n)
2   {
3       if (k = n) then write (a[1 : n]); // Output permutation.
4       else // a[k : n] has more than one permutation.
5           // Generate these recursively.
6           for i := k to n do
7           {
8               t := a[k]; a[k] := a[i]; a[i] := t;
9               Perm(a, k + 1, n);
10              // All permutations of a[k + 1 : n]
11              t := a[k]; a[k] := a[i]; a[i] := t;
12          }
13  }
```

Latex algorithm

Algorithm 1 General Task Allocation Algorithm

Require: Task Matrix
Ensure: Utilization Matrix

1: $time, \tau \leftarrow 1$
2: Initialize Utilization Matrix, $U^* \leftarrow \phi$.
3: $R^* \leftarrow \phi$.
4: **while** $taskq \neq \phi$ **do**
5: jq = Get jobs from main queue($taskq$) where arrival time $\leq \tau$.
6: **while** $jq \neq \phi$ **do**
7: $j \leftarrow TaskChoosingPolicy()$
8: $i \leftarrow ResourceChoosingPolicy()$
9: **if** $i \neq Null$ **then**
10: Assign task t_j to R_i
11: $U_{(\tau,i)} \leftarrow U_{(\tau,i)} + utilization(t_j, i)$.
12: Remove task t_j from $taskq$ and jq.
13: **else**
14: Remove task t_j from jq.
15: **end if**
16: **end while**
17: $\tau \leftarrow \tau + 1$.
18: **end while**
19: **return** U.

Algorithm Paradigm

Algorithm Design Paradigms: General approaches to the construction of efficient solutions to problems

- **Interest of different paradigm**
  - They provide templates suited to solving a broad range of diverse problems.
  - They can be translated into common control and data structures provided by most high-level languages.
  - The temporal and spatial requirements of the algorithms which result can be precisely analyzed.

Brute force

- **Brute force** is a straightforward approach to solve a problem based on the problem's statement and definitions of the concepts involved. It is considered as one of the easiest approach to apply and is useful for solving small – size instances of a problem.
- Examples of brute force algorithms are:
  - Computing a^n ($a > 0$, n a nonnegative integer) by multiplying a*a*…*a
  - Computing $n!$
  - Selection sort
  - Bubble sort
  - Sequential search
  - Exhaustive search: Traveling Salesman Problem, Knapsack problem.

Algorithm Paradigm

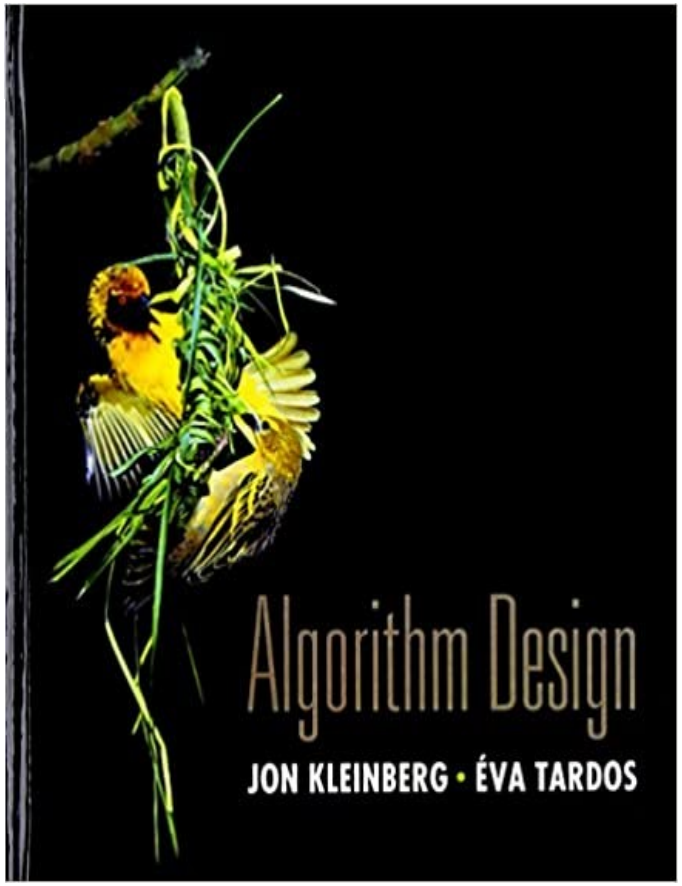
- Divide and Conquer
- Greedy
- Dynamic Programming
- Backtracking
- Branch and Bound

- Approximation algorithms
- Randomized algorithms
- Soft Computing Techniques

Algorithm Design: Suggested for problems: Chapter 3

9. There's a natural intuition that two nodes that are far apart in a communication network—separated by many hops—have a more tenuous connection than two nodes that are close together. There are a number of algorithmic results that are based to some extent on different ways of making this notion precise. Here's one that involves the susceptibility of paths to the deletion of nodes.

   Suppose that an n-node undirected graph $G = (V, E)$ contains two nodes s and t such that the distance between s and t is strictly greater than $n/2$. Show that there must exist some node v, not equal to either s or t, such that deleting v from G destroys all s-t paths. (In other words, the graph obtained from G by deleting v contains no path from s to t.) Give an algorithm with running time $O(m + n)$ to find such a node v.



Algorithm Design: Suggested for problems: Chapter 3

12. You're helping a group of ethnographers analyze some oral history data they've collected by interviewing members of a village to learn about the lives of people who've lived there over the past two hundred years.

From these interviews, they've learned about a set of n people (all of them now deceased), whom we'll denote $P_1, P_2, \ldots, P_n$. They've also collected facts about when these people lived relative to one another. Each fact has one of the following two forms:

- For some i and j, person P_i died before person P_j was born; or
- for some i and j, the life spans of P_i and P_j overlapped at least partially.

Naturally, they're not sure that all these facts are correct; memories are not so good, and a lot of this was passed down by word of mouth. So what they'd like you to determine is whether the data they've collected is at least internally consistent, in the sense that there could have existed a set of people for which all the facts they've learned simultaneously hold.

Give an efficient algorithm to do this: either it should produce proposed dates of birth and death for each of the n people so that all the facts hold true, or it should report (correctly) that no such dates can exist—that is, the facts collected by the ethnographers are not internally consistent.

Algorithm Design: Suggested for problems: Chapter 4

6. Your friend is working as a camp counselor, and he is in charge of organizing activities for a set of junior-high-school-age campers. One of his plans is the following mini-triathalon exercise: each contestant must swim 20 laps of a pool, then bike 10 miles, then run 3 miles. The plan is to send the contestants out in a staggered fashion, via the following rule: the contestants must use the pool one at a time. In other words, first one contestant swims the 20 laps, gets out, and starts biking. As soon as this first person is out of the pool, a second contestant begins swimming the 20 laps; as soon as he or she is out and starts biking, a third contestant begins swimming . . . and so on.)

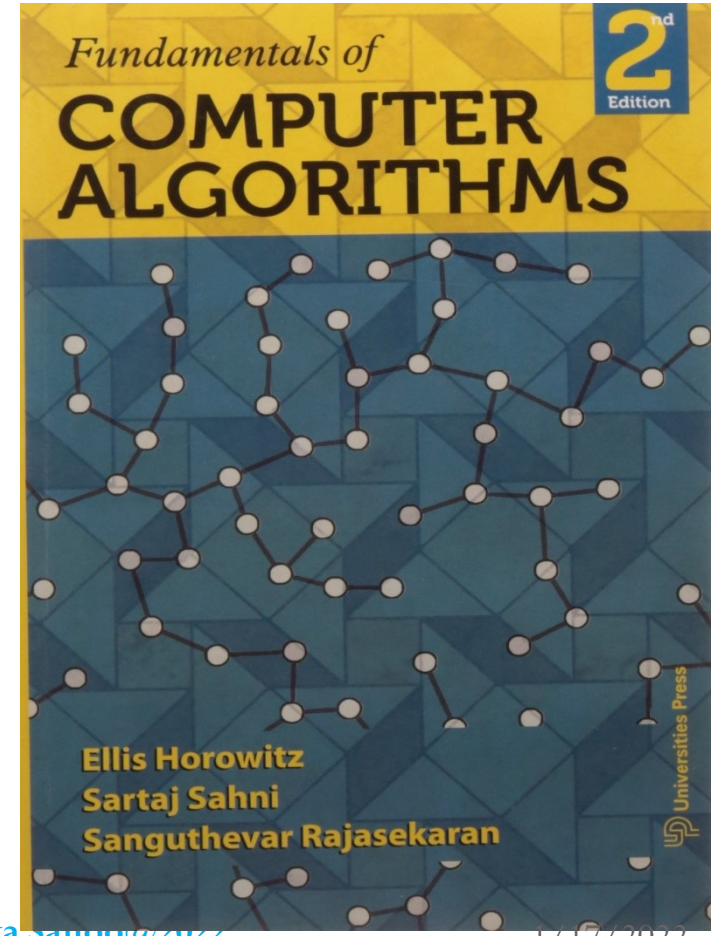
   Each contestant has a projected *swimming time* (the expected time it will take him or her to complete the 20 laps), a projected *biking time* (the expected time it will take him or her to complete the 10 miles of bicycling), and a projected *running time* (the time it will take him or her to complete the 3 miles of running). Your friend wants to decide on a *schedule* for the triathalon: an order in which to sequence the starts of the contestants. Let's say that the *completion time* of a schedule is the earliest time at which all contestants will be finished with all three legs of the triathalon, assuming they each spend exactly their projected swimming, biking, and running times on the three parts. (Again, note that participants can bike and run simultaneously, but at most one person can be in the pool at any time.) What's the best order for sending people out, if one wants the whole competition to be over as early as possible? More precisely, give an efficient algorithm that produces a schedule whose completion time is as small as possible.

Algorithmic paradigm

- An **algorithmic paradigm** or **algorithm design paradigm** is a generic model or framework which underlies the design of a **class of algorithms.**

Main algorithm design techniques:

- Brute-force or exhaustive search.
- Divide and Conquer.
- Greedy Algorithms.
- Dynamic Programming.
- Branch and Bound Algorithm.
- Backtracking.



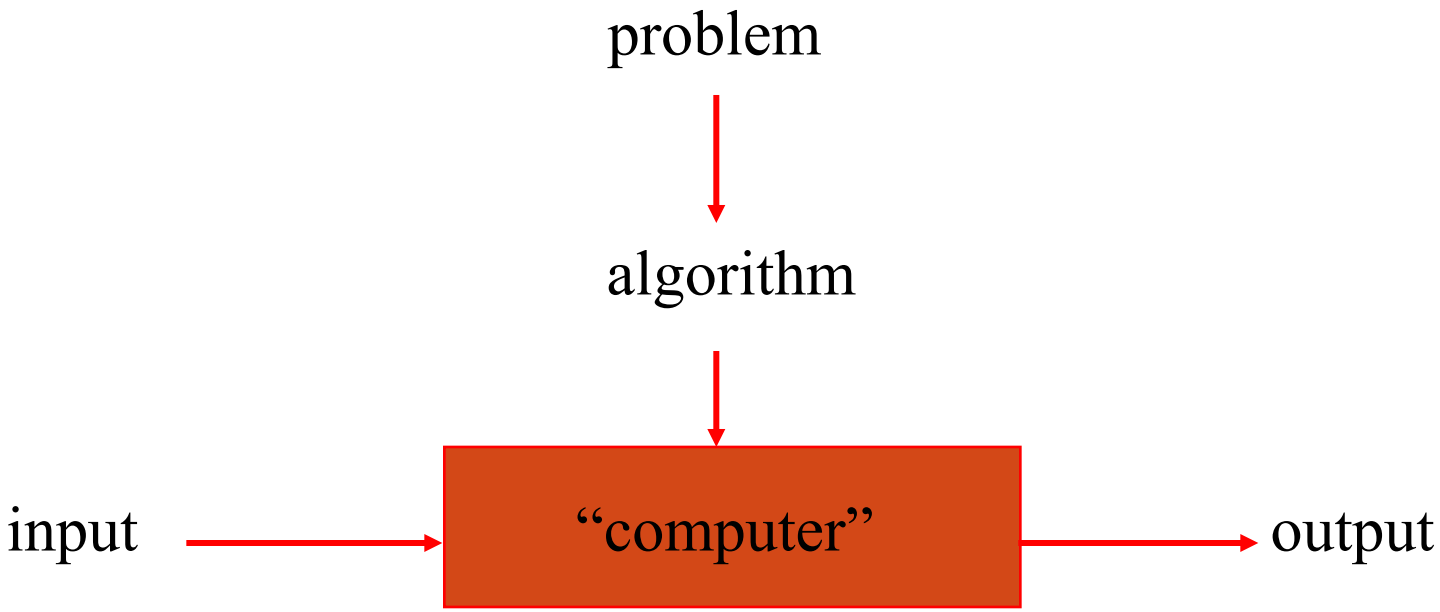
Analysis of algorithms

- How good is the algorithm?
  - Correctness
  - **Time efficiency**
  - **Space efficiency**
  - Simplicity
  - Generality
- Does there exist a better algorithm?
  - Lower bounds
  - Optimality

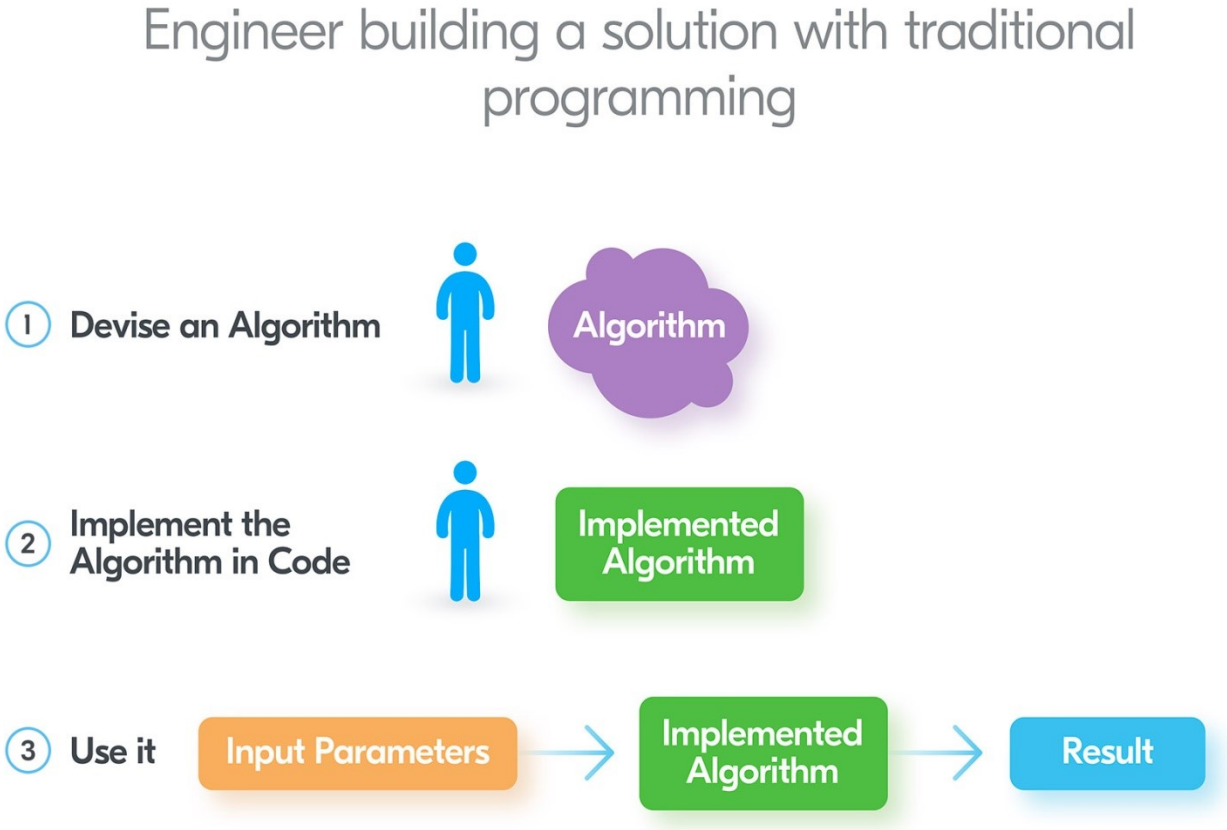
Problem

- An ***instance of a problem*** consists of all inputs needed to compute a solution to the problem.
- An algorithm is said to be ***correct*** if for every input instance, it halts with the correct output.
- A **correct algorithm** ***solves*** the given computational problem.
- An **incorrect algorithm** might not halt at all on some input instance, or it might halt with other than the desired answer.

Notion of algorithm: Algorithmic solution



Traditional Programming Approach



Problem Solving Techniques in Artificial Intelligence

- Problem-solving is commonly known as the method to reach the desired goal or finding a solution to a given situation.
- In computer science, problem-solving refers to artificial intelligence techniques, including various techniques such as forming efficient algorithms, heuristics, and performing root cause analysis to find desirable solutions.
- In **Artificial Intelligence**, the users can solve the problem by performing logical algorithms, utilizing polynomial and differential equations, and executing them using modeling paradigms.
- There can be various solutions to a single problem, which are achieved by different heuristic

Problem Solving in Artificial Intelligence

- The problem of AI is directly associated with the nature of humans and their activities. So we need a number of **finite st**eps to solve a problem which makes human easy works.
- **Goal Formulation:** This one is the first and simple step in problem-solving. It organizes finite steps to formulate a target/goals which require some action to achieve the goal. Today the formulation of the goal is based on AI agents.
- **Problem formulation:** It is one of the core steps of problem-solving which decides what action should be taken to achieve the formulated goal. In AI this core part is dependent upon software agent which consisted of the following components to formulate the associated problem.

Steps performed by Problem-solving agent

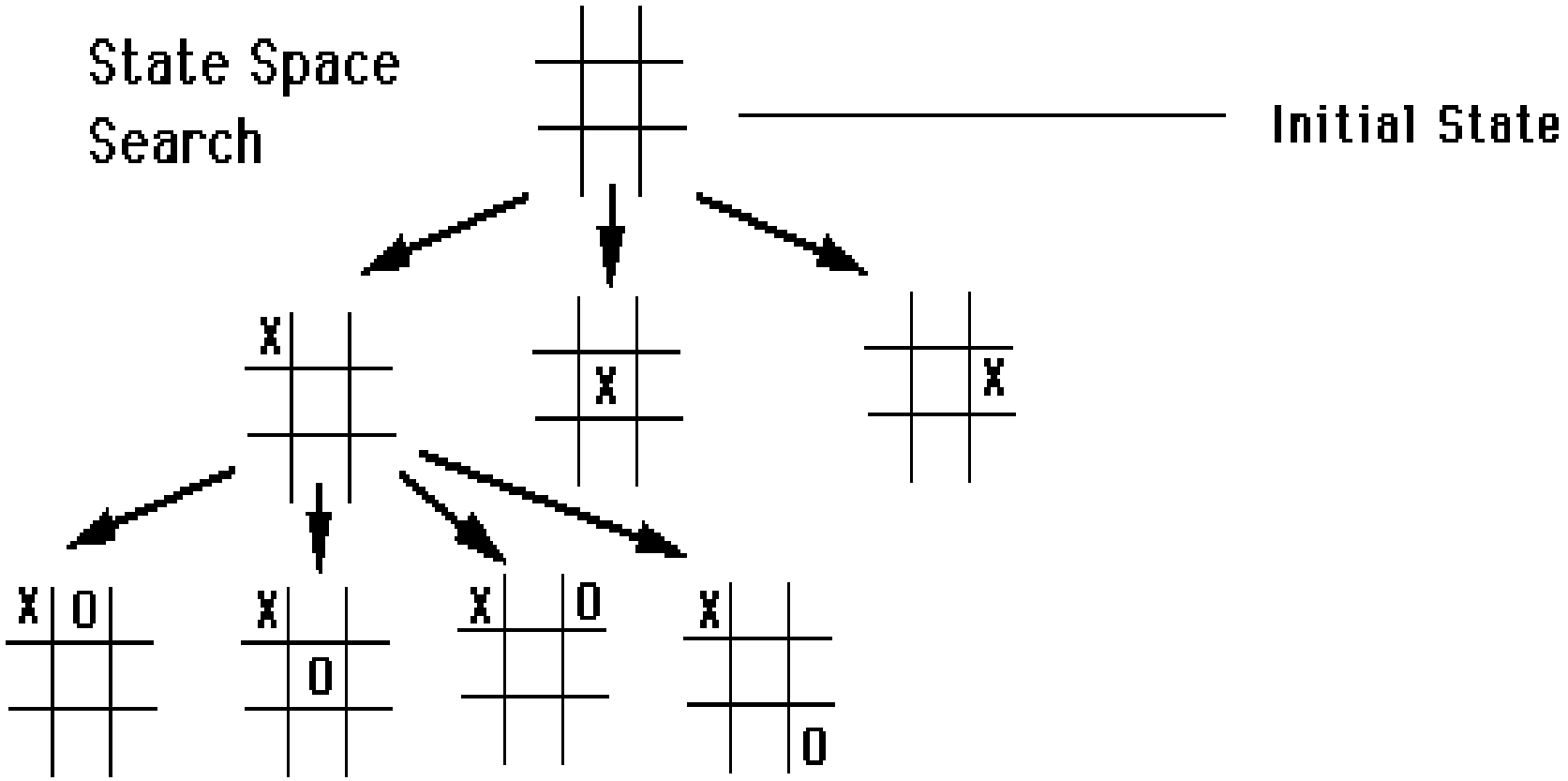
- **Goal Formulation:** It is the first and simplest step in problem-solving. It organizes the steps/sequence required to formulate one goal out of multiple goals as well as actions to achieve that goal. Goal formulation is based on the current situation and the agent's performance measure (discussed below).
- **Problem Formulation:** It is the most important step of problem-solving which decides what actions should be taken to achieve the formulated goal. There are following five components involved in problem formulation:
- **Initial State:** It is the starting state or initial step of the agent towards its goal.
- **Actions:** It is the description of the possible actions available to the agent.
- **Transition Model:** It describes what each action does.
- **Goal Test:** It determines if the given state is a goal state.
- **Path cost:** It assigns a numeric cost to each path that follows the goal. The problem-solving agent selects a cost function, which reflects its performance measure.

An optimal solution has the lowest path cost among all the solutions.

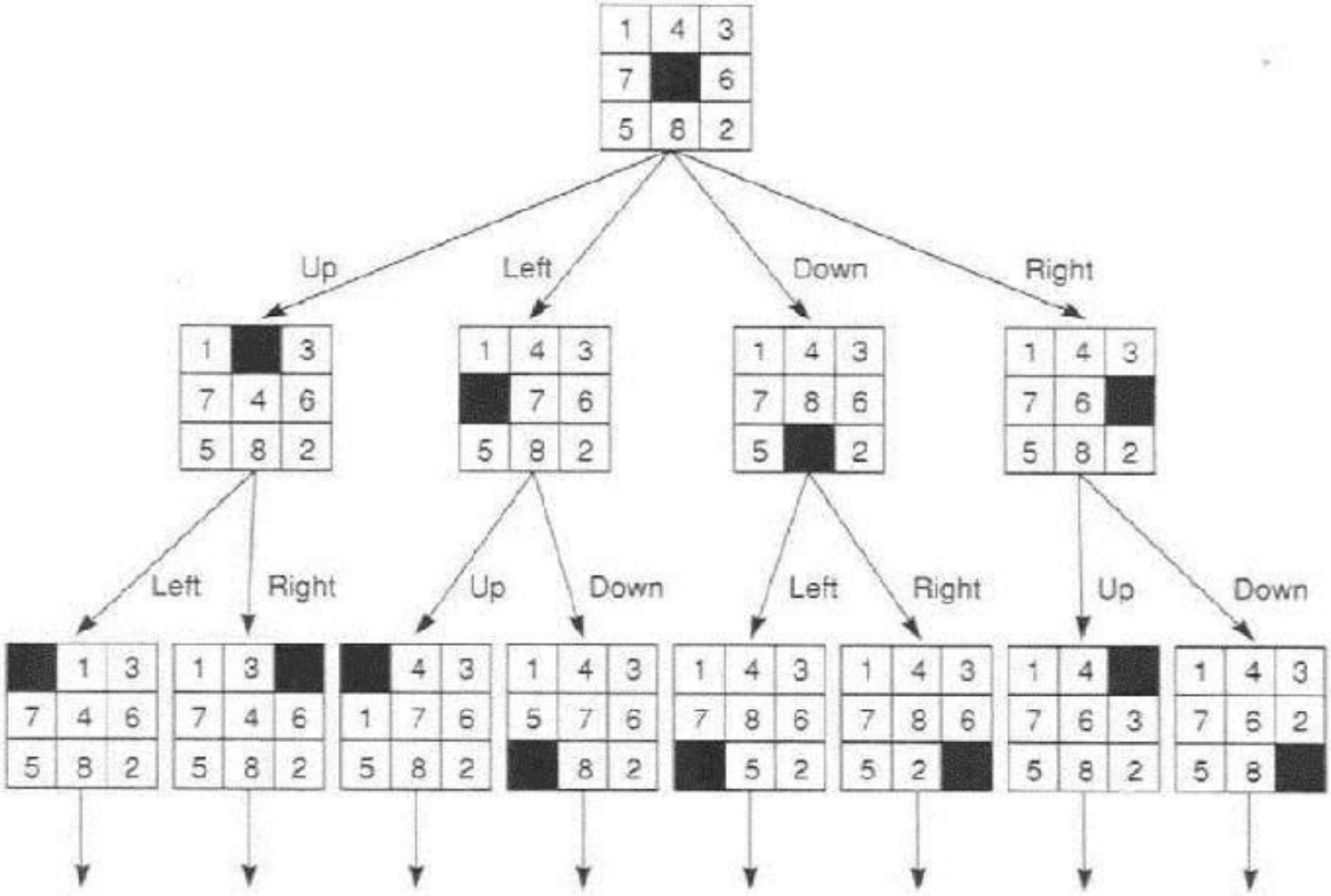
Problem-solving

- **Note: Initial state, actions**, and **transition model** together define the **state-space** of the problem implicitly.
- State-space of a problem is a set of all states which can be reached from the **initial state** followed by any sequence of actions.
- The state-space forms a directed map or graph where nodes are the states, links between the nodes are actions, and the path is a sequence of states connected by the sequence of actions.
- **Search:** It identifies all the best possible sequence of actions to reach the goal state from the current state. It takes a problem as an input and returns solution as its output.
- **Solution:** It finds the best algorithm out of various algorithms, which may be proven as the best optimal solution.
- **Execution:** It executes the best optimal solution from the searching algorithms to reach the goal state from the current state.

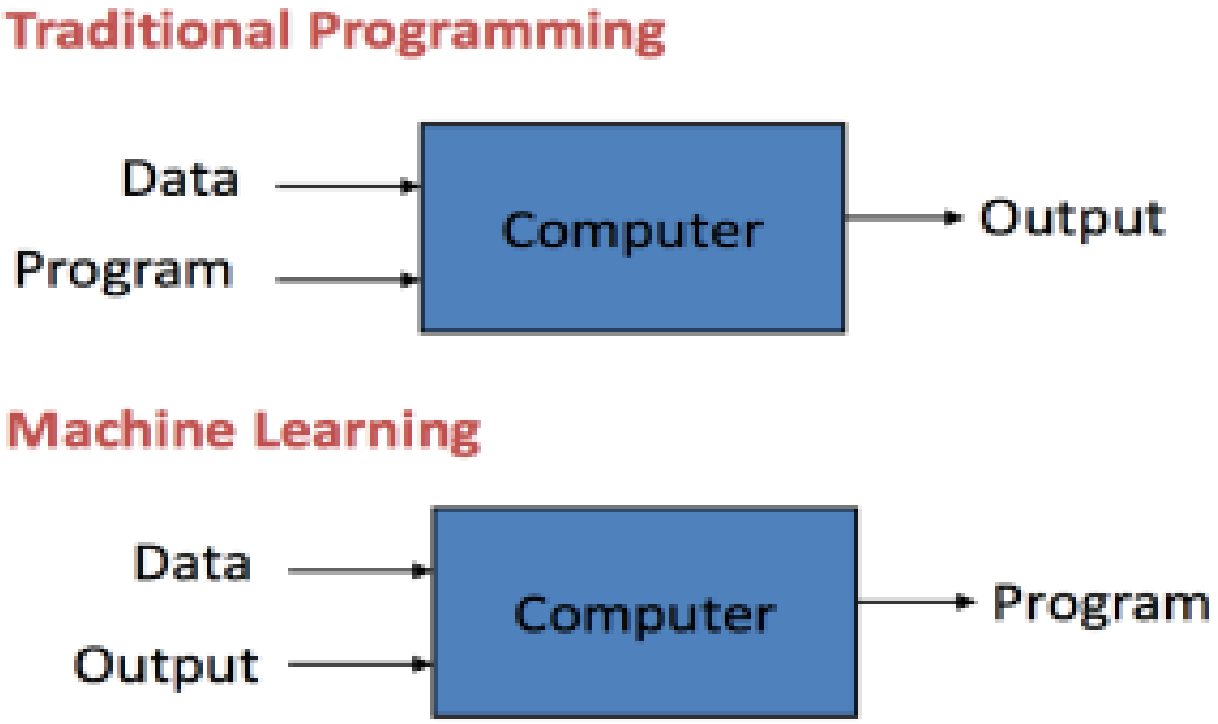
Problem soling by state space search



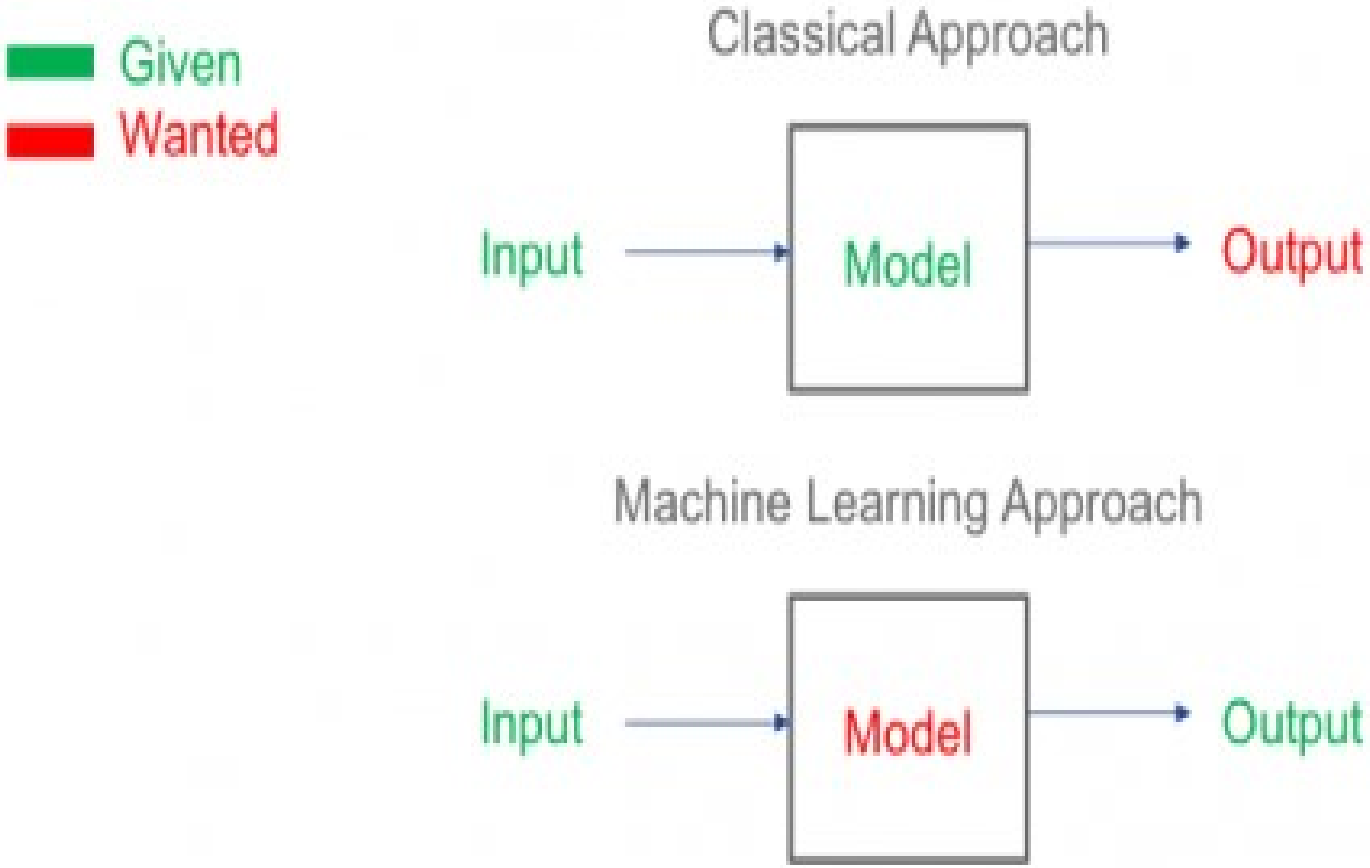
State space tree for 8 puzzle problem



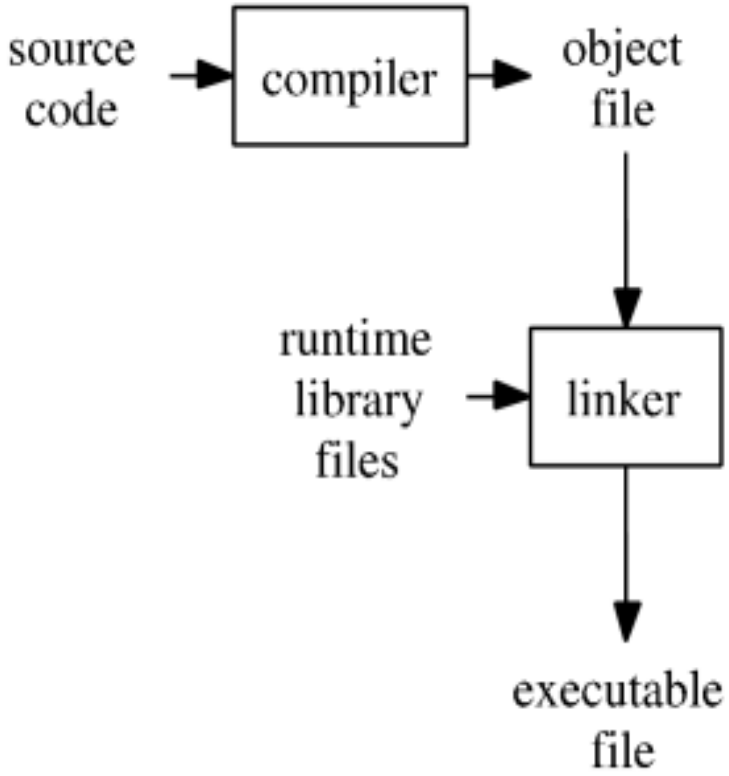
Traditional Programming vs Machine learning

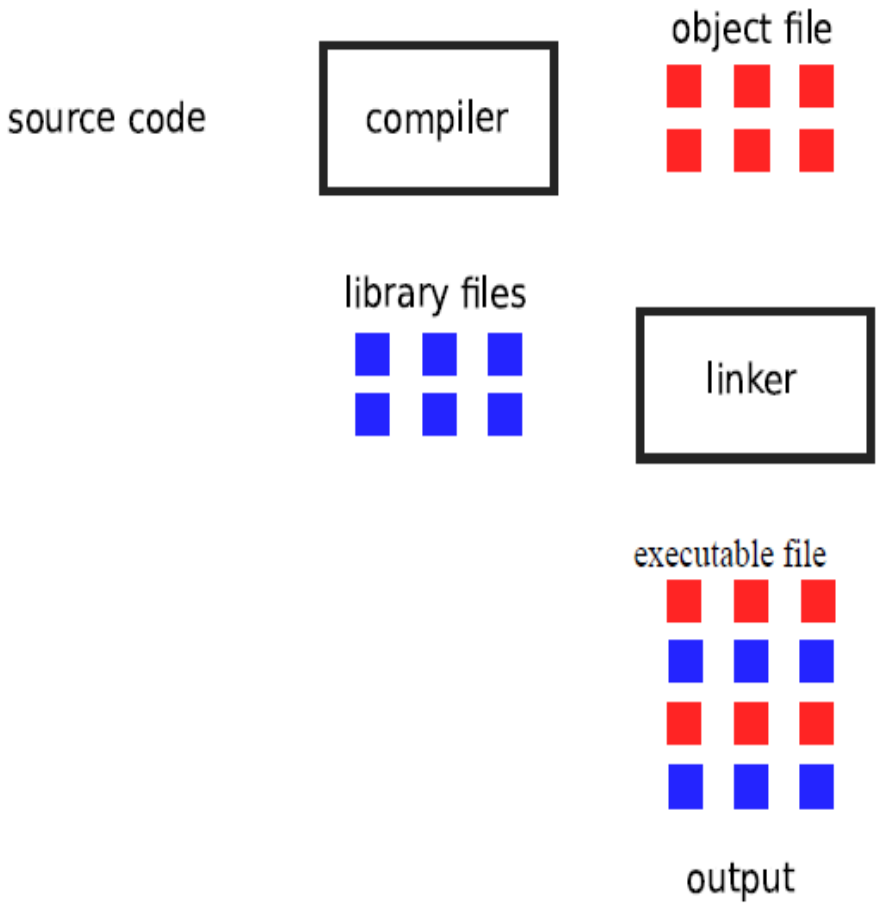


Machine Learning vs. Classical Approach



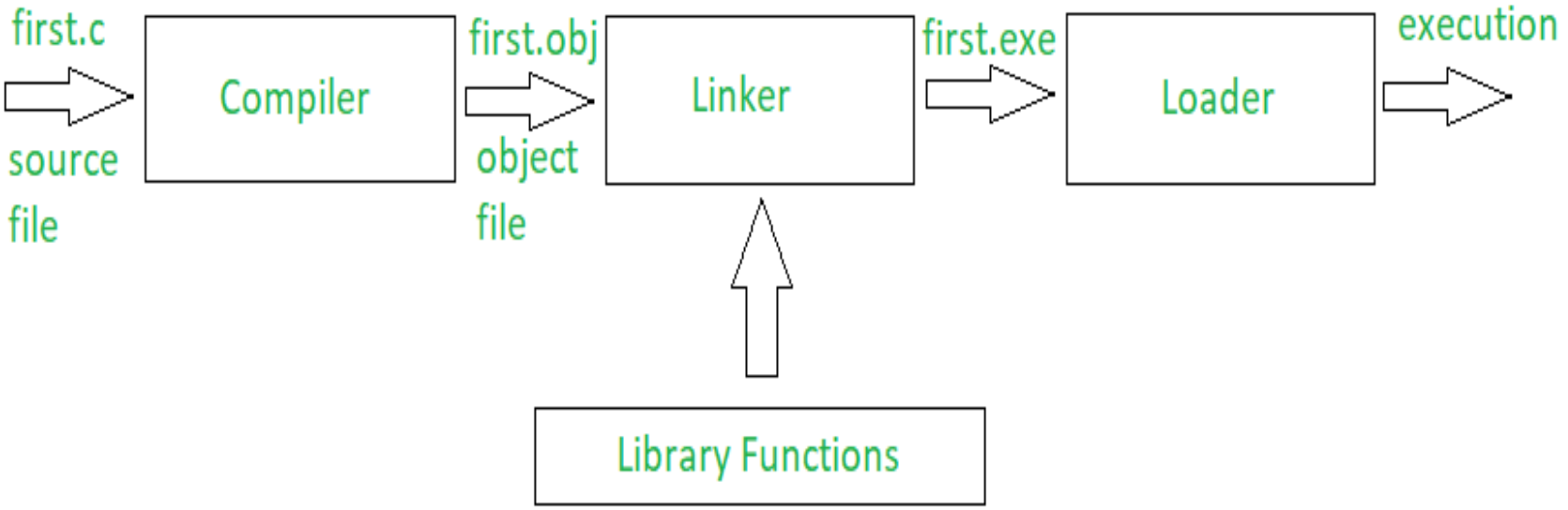
Complier


source code
compiler
object file
runtime library files
linker
executable file


source code
compiler
object file
library files
linker
executable file
output

The compilation process

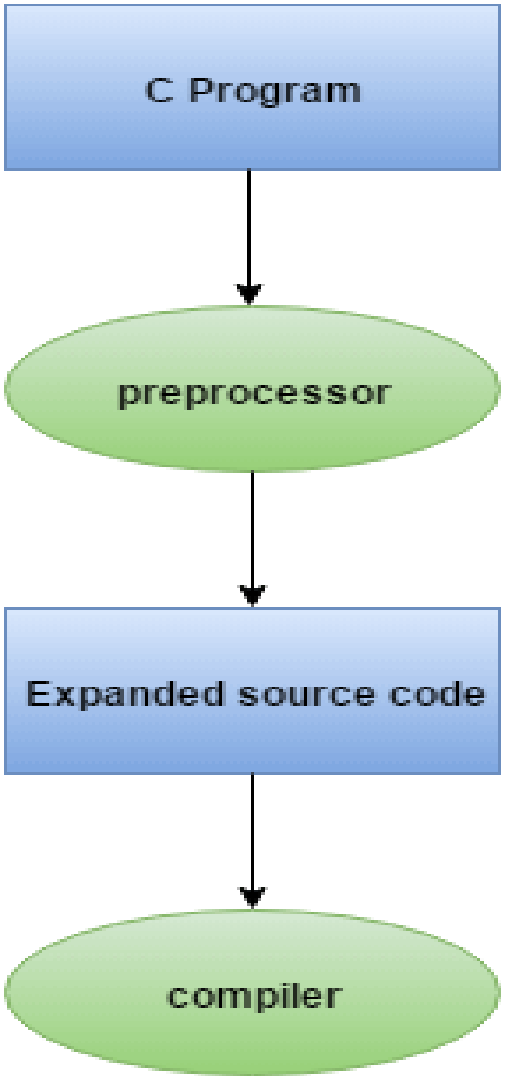
- The compilation process with the files created at each step of the compilation process



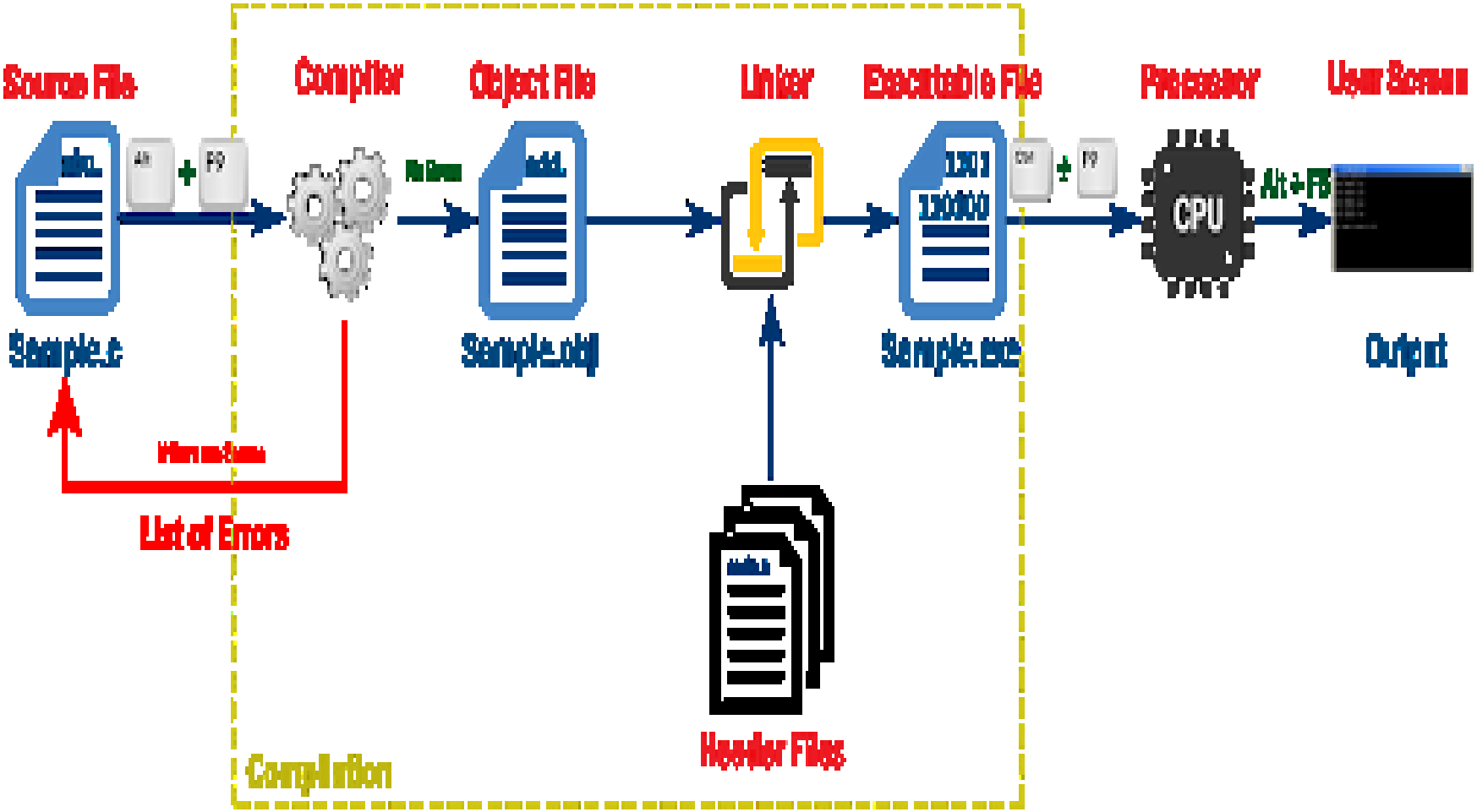
Preprocessing a C program

- All preprocessor directives starts with hash # symbol.
- A list of preprocessor directives are as following:

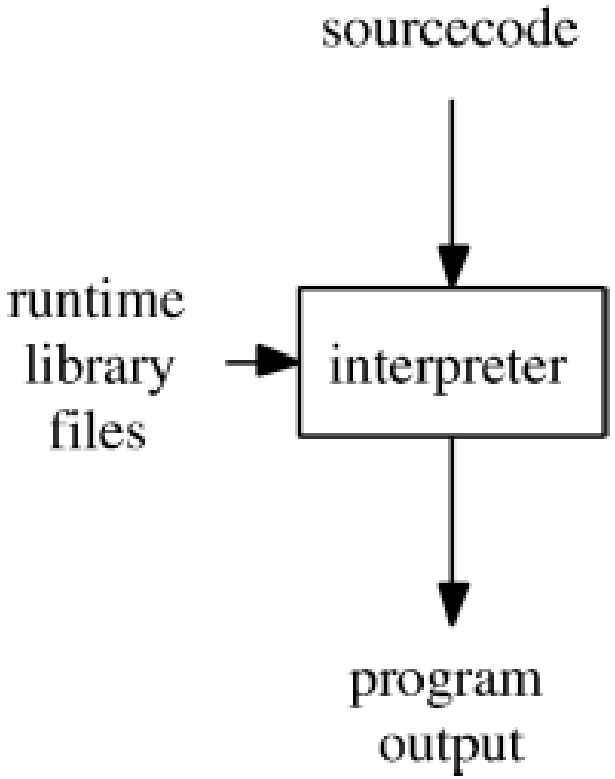
  #include
  #define
  #undef
  #ifdef
  #ifndef
  #if
  #else
  #elif
  #endif
  #error
  #pragma

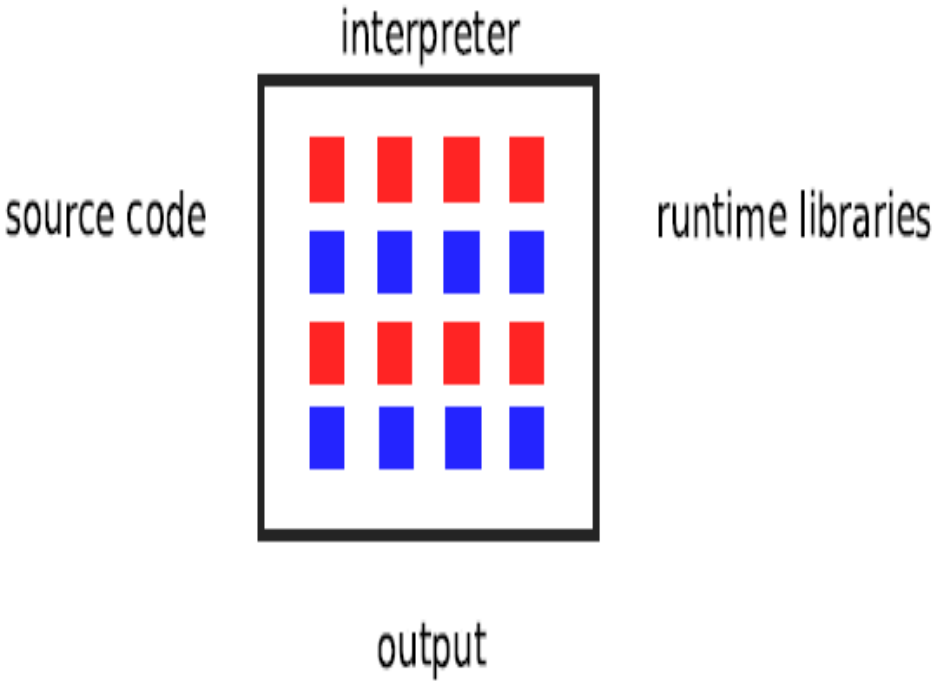


Execution Process of a C Program



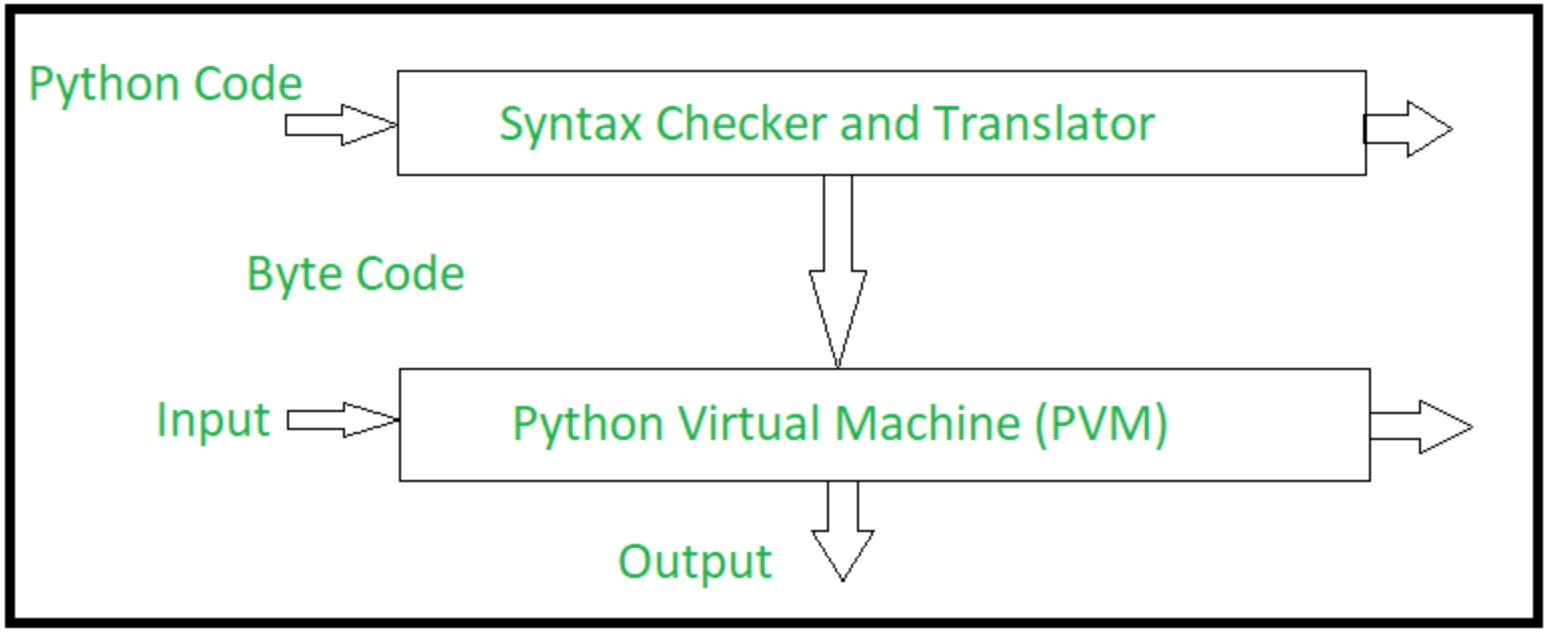
Interpreter



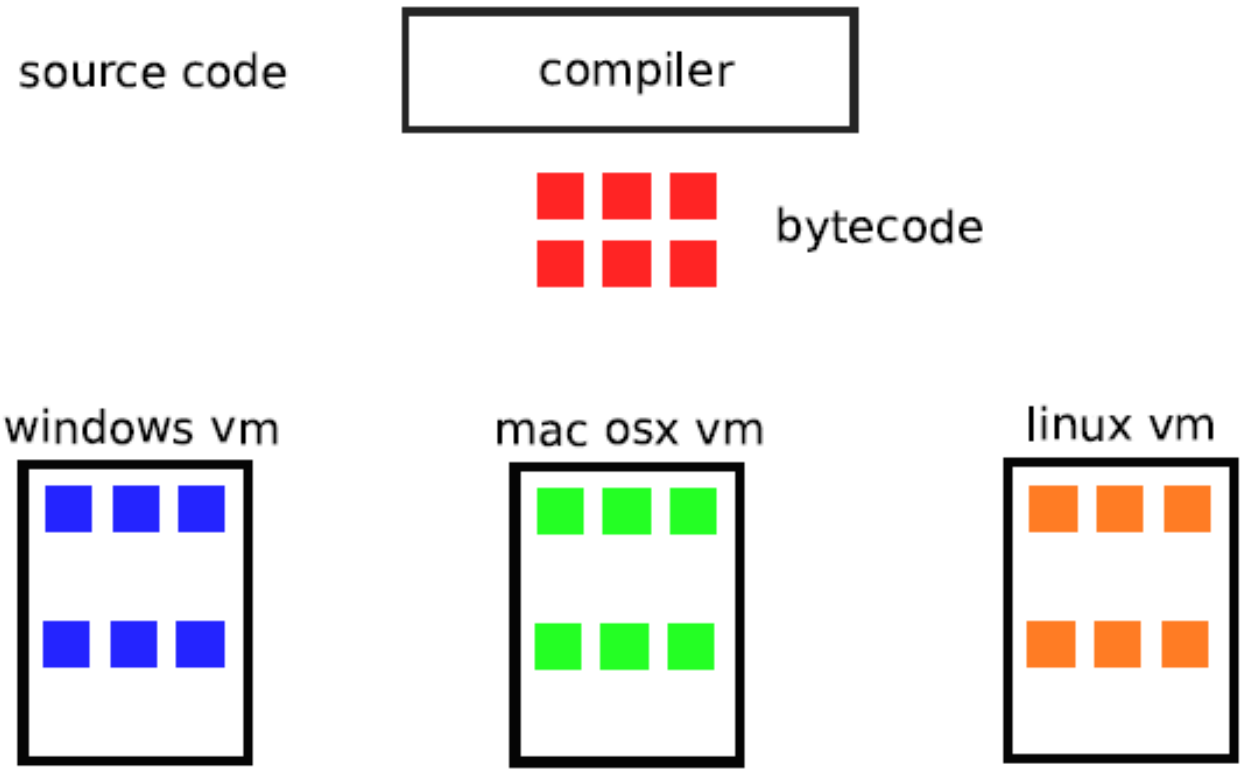


Execution of Python code on Virtual Machine

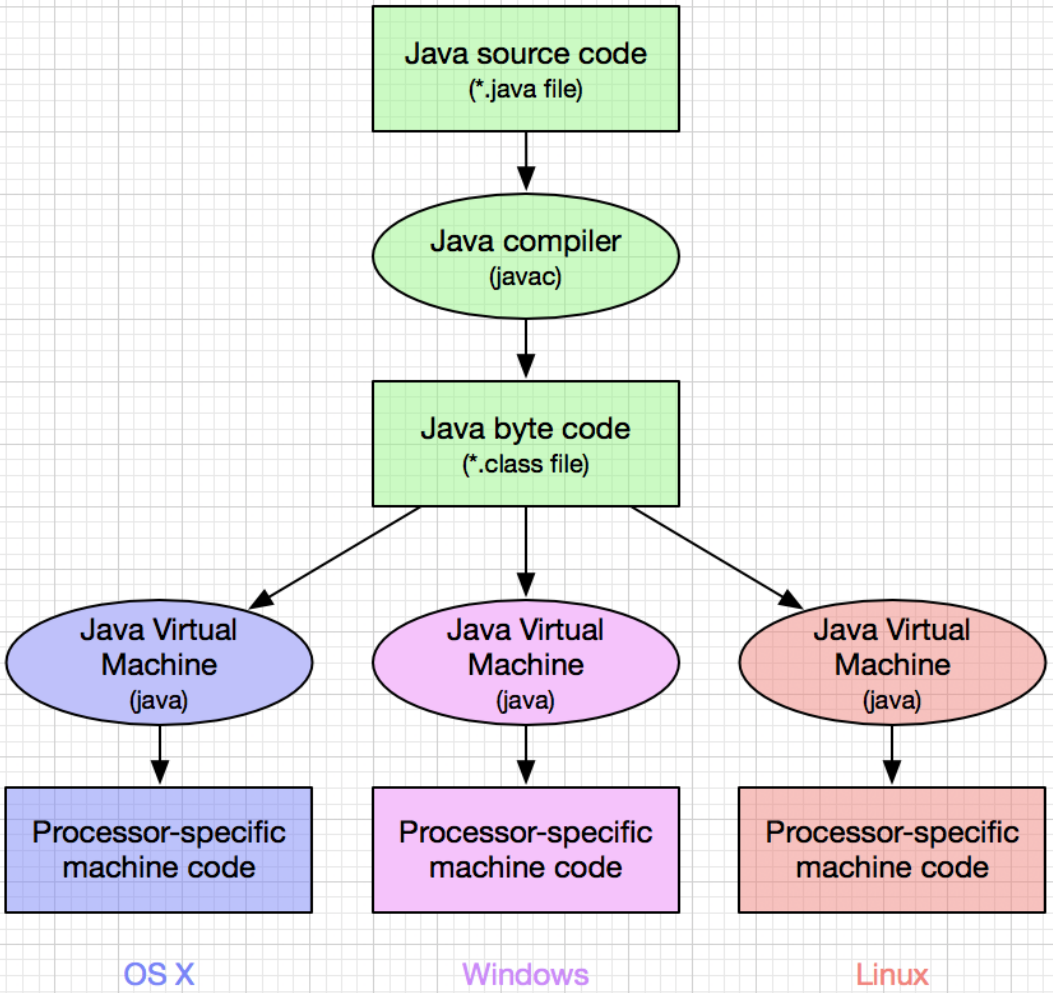
- **Python** is an object-oriented programming language .
- Python is called an interpreted language.



Compiler on virtual machine



Compiler on virtual machine



Humans read and write Java source code.

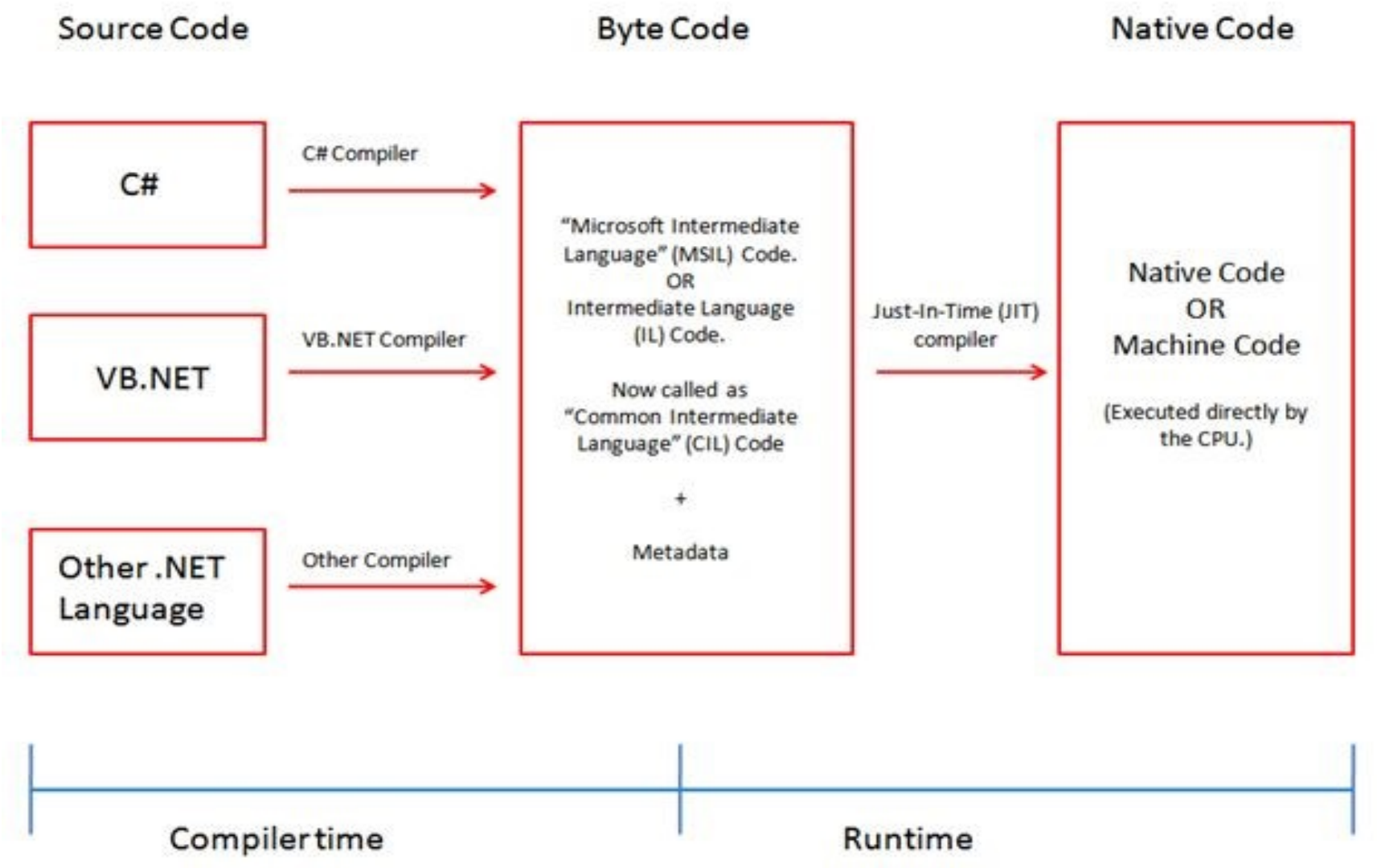
The java compiler converts that to byte code.

Byte code is the 'machine code' of the Java Virtual Machine.

To execute byte code, any Java Virtual Machine (JVM) interprets it and translates it to machine code specific to the computer the JVM is on.

Machine is what the actual processor ultimately executes.

Code execution process

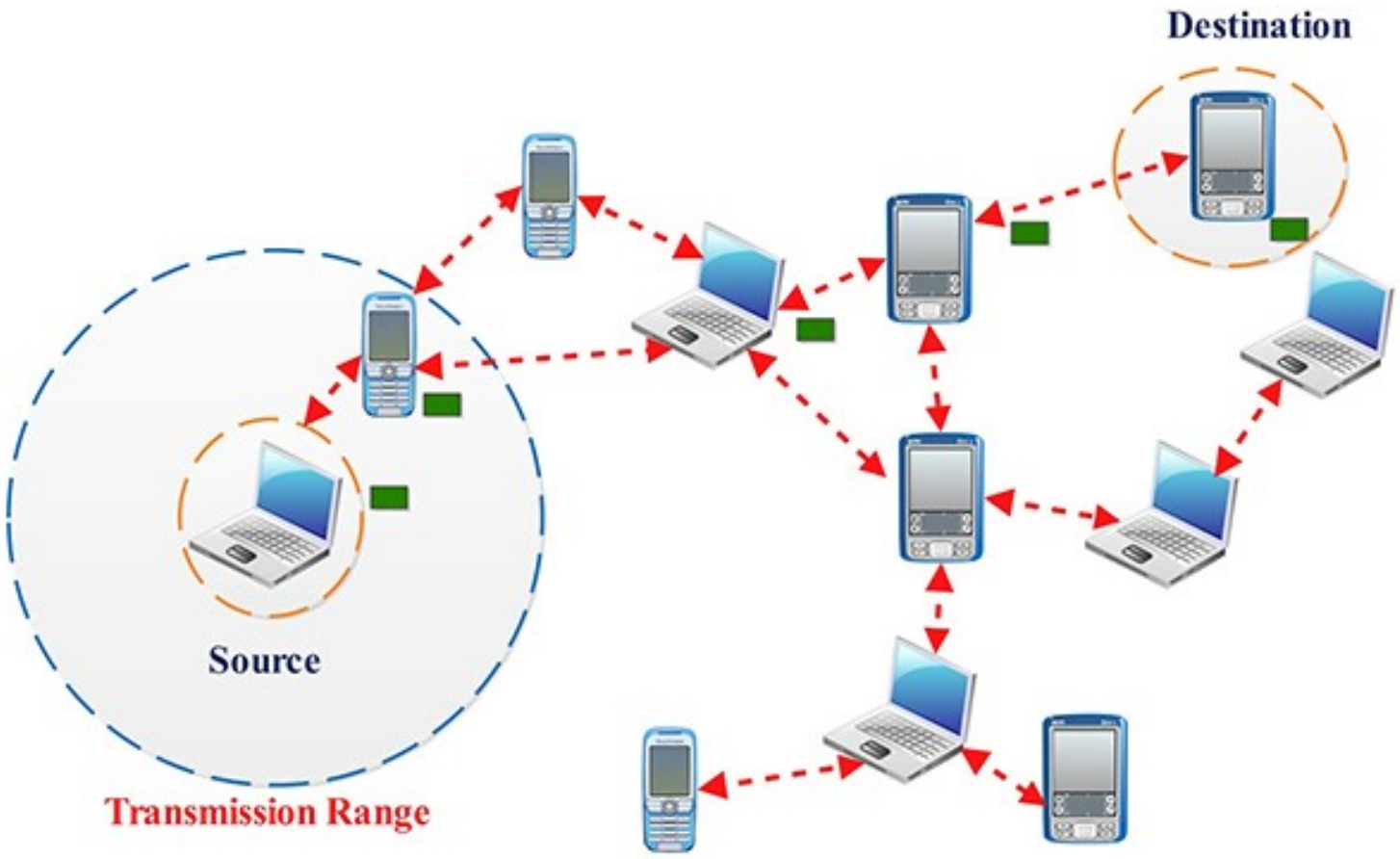


MODELING : Algorithm Design

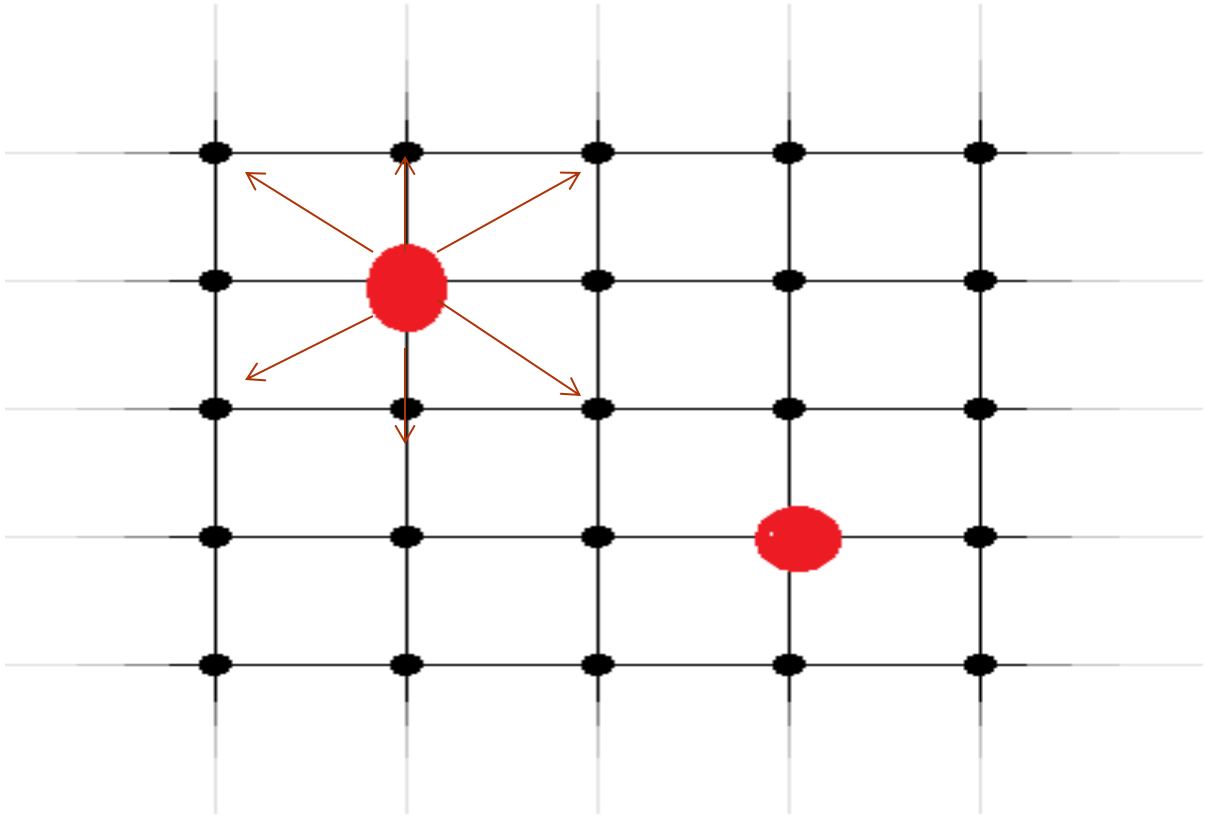
Modeling

- **Modeling** is the art of formulating your application in terms of precisely described, well-understood problems.
- Proper modeling is the key to applying **algorithmic design techniques** to any **real-world problem**.
- Real-world applications involve real-world objects.
- Most algorithms, however, are designed to work on rigorously defined abstract structures such as permutations, graphs, and sets.
- You must first describe your problem abstractly, in terms of fundamental structures and properties.

Mobile Ad Hoc Network(MANET)

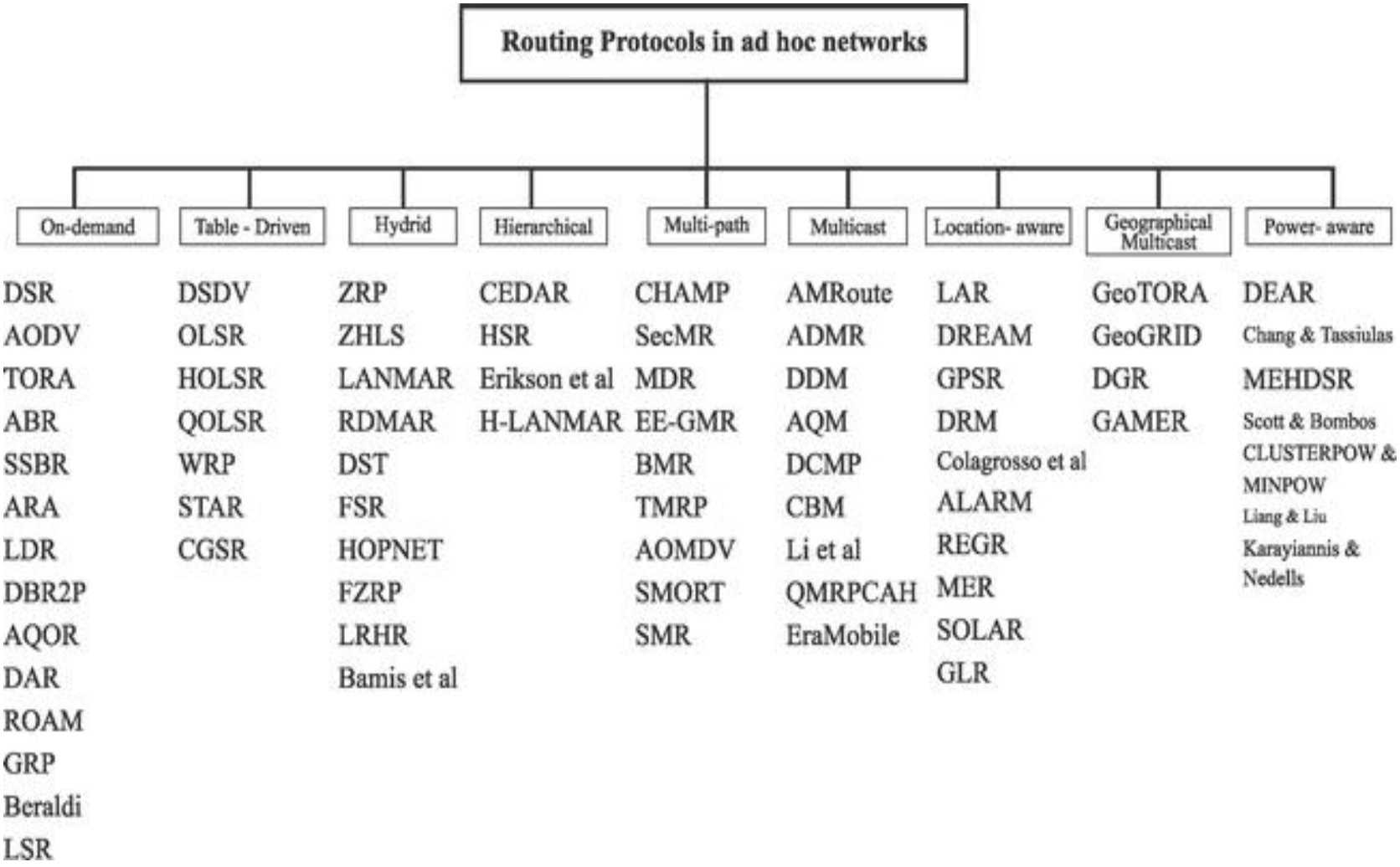


A model for MANET

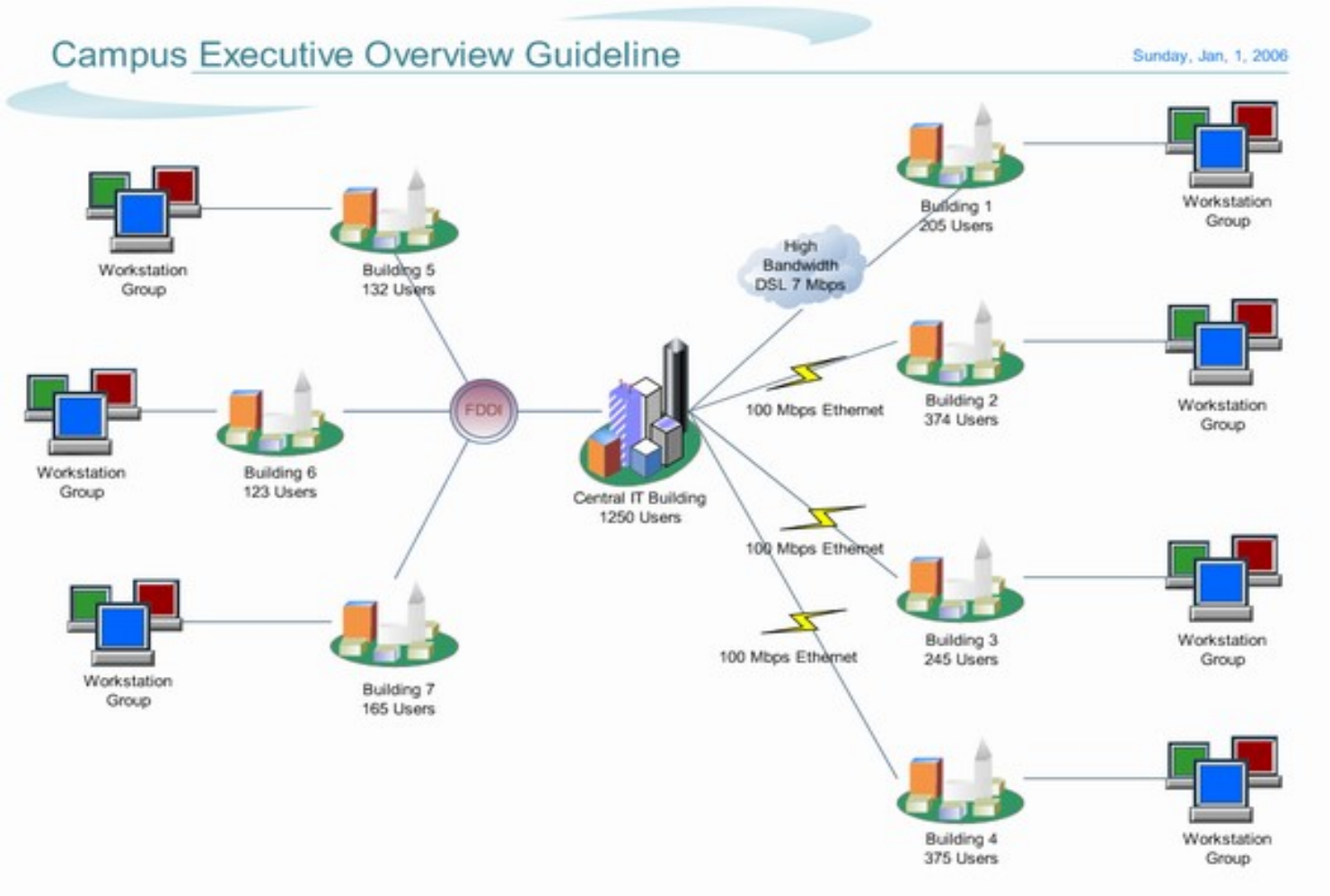


https://www.youtube.com/watch?v=nu3G3BL_ULU

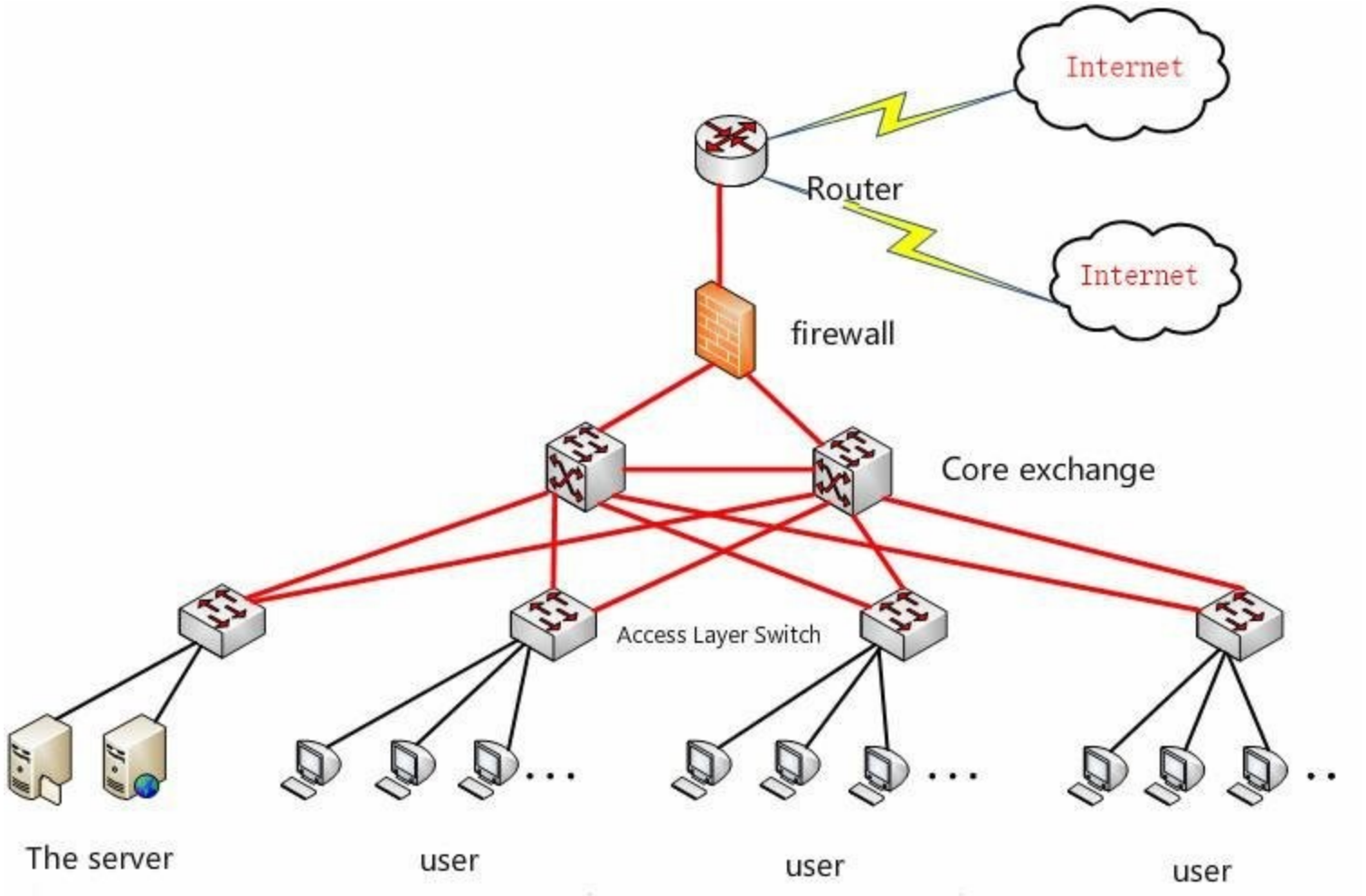
Mobile Ad Hoc Network(MANET) routing protocols



How to model a campus network ?



Model of a campus network topology



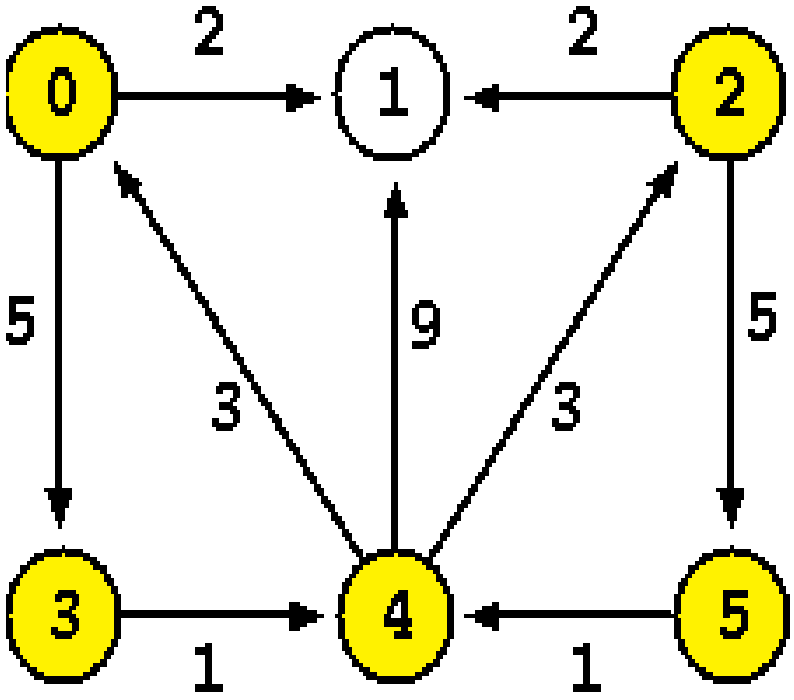
Rules for Algorithm design

- The secret to successful algorithm design, and problem solving in general, is to make sure *you ask the right questions.*
- Below, I give a possible series of questions for you to ask yourself as you try to solve difficult algorithm design problems

1. Do I really understand the problem?

- (a) What exactly does the input consist of?
- (b) What exactly are the desired results or output?
- (c) Can I construct some examples small enough to solve by hand? What happens when I solve them?
- (d) Are you trying to solve a numerical problem? A graph algorithm problem? A geometric problem? A string problem? A set problem? Might your problem be formulated in more than one way? Which formulation seems easiest?

Graph model for shortest path problem



	0	1	2	3	4	5
0	0	2	0	5	0	0
1	0	0	0	0	0	0
2	0	2	0	0	0	5
3	0	0	0	0	1	0
4	3	9	3	0	0	0
5	0	0	0	0	1	0

Rules for Algorithm design...

2. Can I find a simple algorithm for the problem?

(a) Can I find the solve my problem exactly by searching all subsets or arrangements and picking the best one?

I. If so, why am I sure that this algorithm always gives the correct answer?

II. How do I measure the quality of a solution once I construct it?

III. Does this simple, slow solution run in polynomial or exponential time?

IV. If I can't find a slow, *guaranteed* correct algorithm, am I sure that my problem is well defined enough to permit a solution?

Rules for Algorithm design...

(b) **Can I solve my problem by repeatedly trying some heuristic rule, like picking the biggest item first? The smallest item first? A random item first?**

I. If so, on what types of inputs does this heuristic rule work well? Do these correspond to the types of inputs that might arise in the application?

II. On what types of inputs does this heuristic rule work badly? If no such examples can be found, can I show that in fact it always works well?

III. How fast does my heuristic rule come up with an answer?

Rules for Algorithm design...

3. Are there special cases of this problem I know how to solve exactly?

(a) Can I solve it efficiently when I ignore some of the input parameters?

(b) What happens when I set some of the input parameters to trivial values, such as 0 or 1?

(c) Can I simplify the problem to create a problem I can solve efficiently? How simple do I have to make it?

(d) If I can solve a certain special case, why can't this be generalized to a wider class of inputs?

Rules for Algorithm design...

4. Which of the standard algorithm design paradigms seem most relevant to the problem?

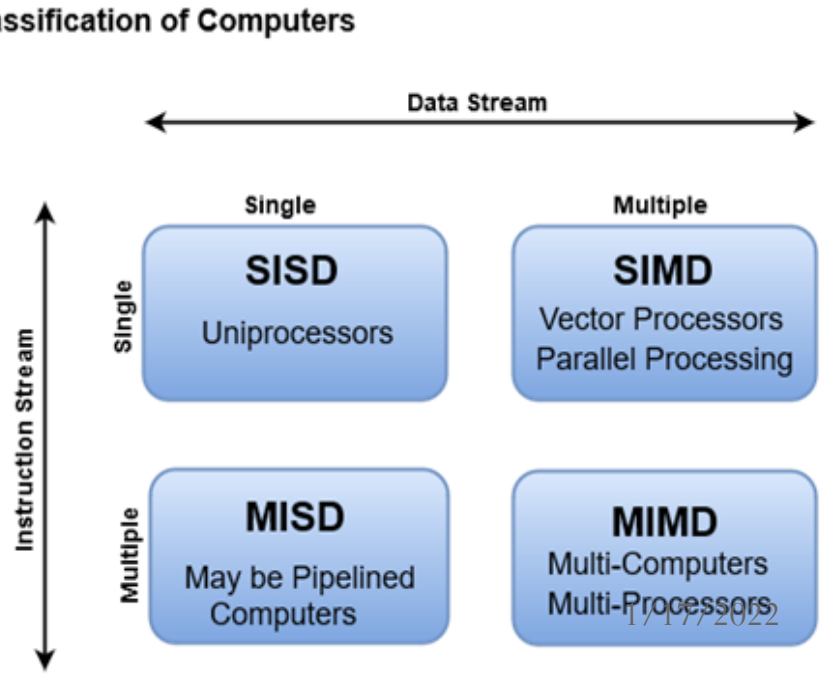
- (a) Is there a set of items which can be sorted by size or some key? Does this sorted order make it easier to find what might be the answer?
- (b) Is there a way to split the problem in two smaller problems, perhaps by doing a binary search or a partition of the elements into big and small, or left and right? If so, does this suggest a divide-and-conquer algorithm?
- (c) Are there certain operations being repeatedly done on the same data, such as searching it for some element, or finding the largest/smallest remaining element? If so, can I use a data structure of speed up these queries, like hash tables or a heap/priority queue?

Model of Computation

Model of Computation [Flynn's classification divides]

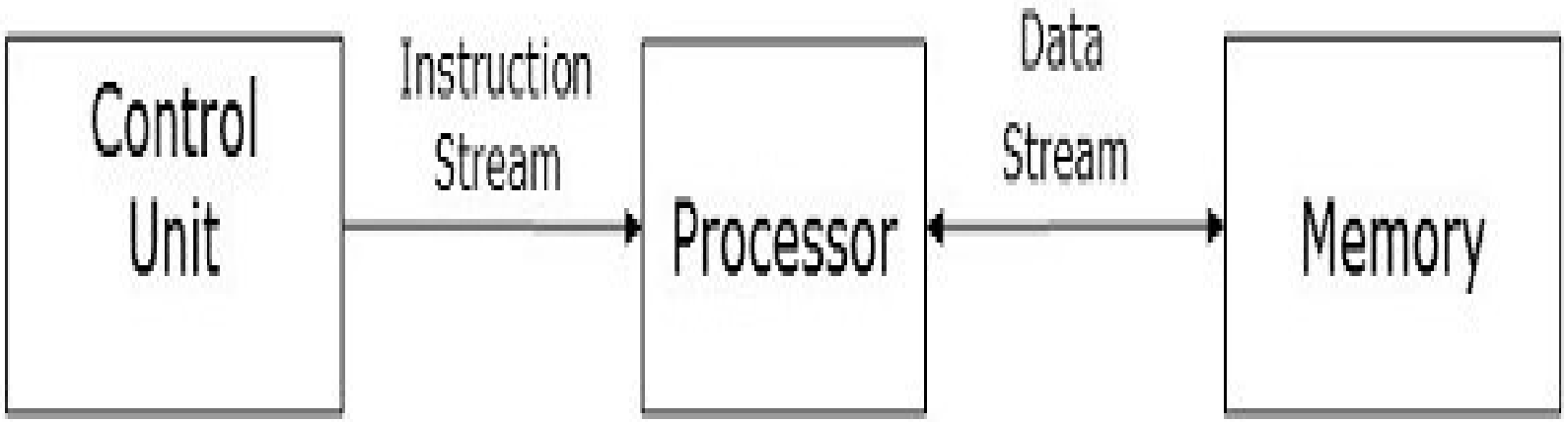
- Depending on the instruction stream and data stream, computers can be classified into four categories –

1. Single Instruction stream, Single Data stream (SISD) computers
2. Single Instruction stream, Multiple Data stream (SIMD) computers
3. Multiple Instruction stream, Single Data stream (MISD) computers
4. Multiple Instruction stream, Multiple Data stream (MIMD) computers



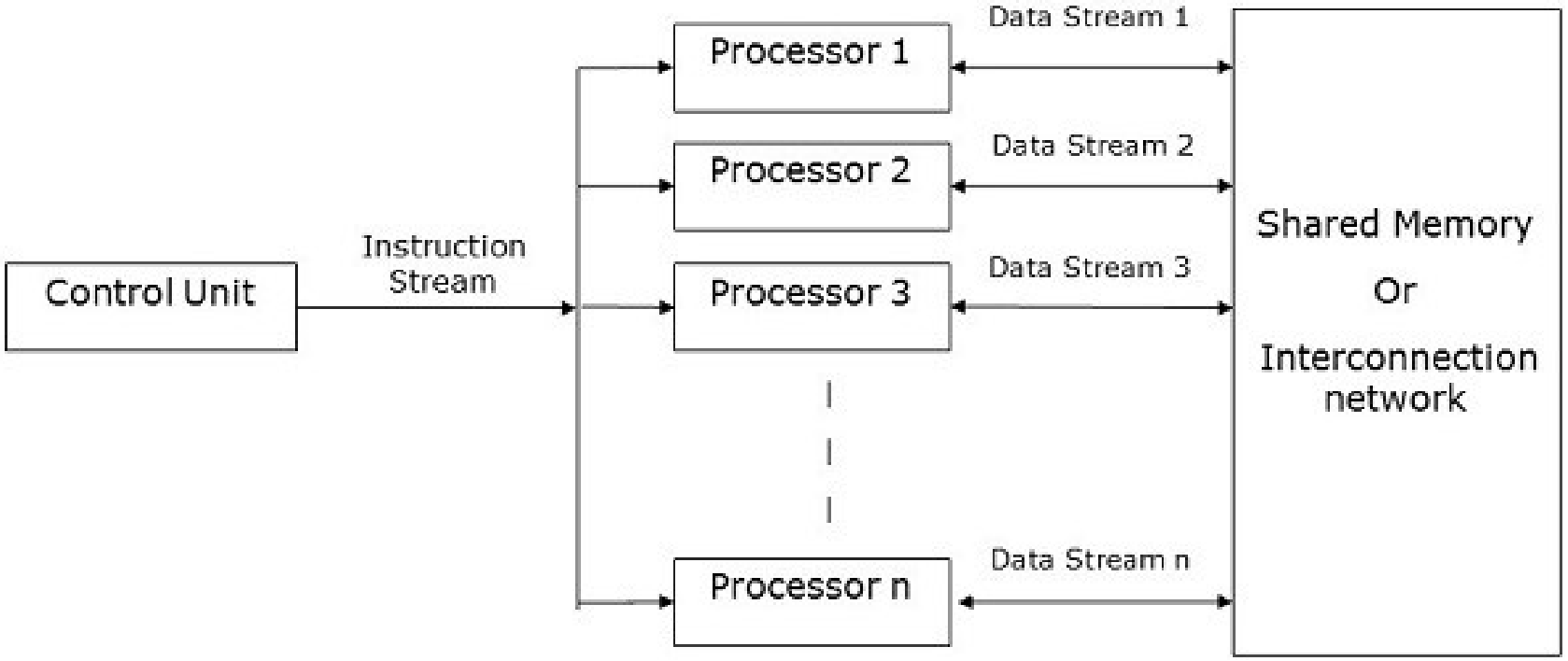
SISD Computer

- SISD computers contain **one control unit, one processing unit,** and **one memory unit**.

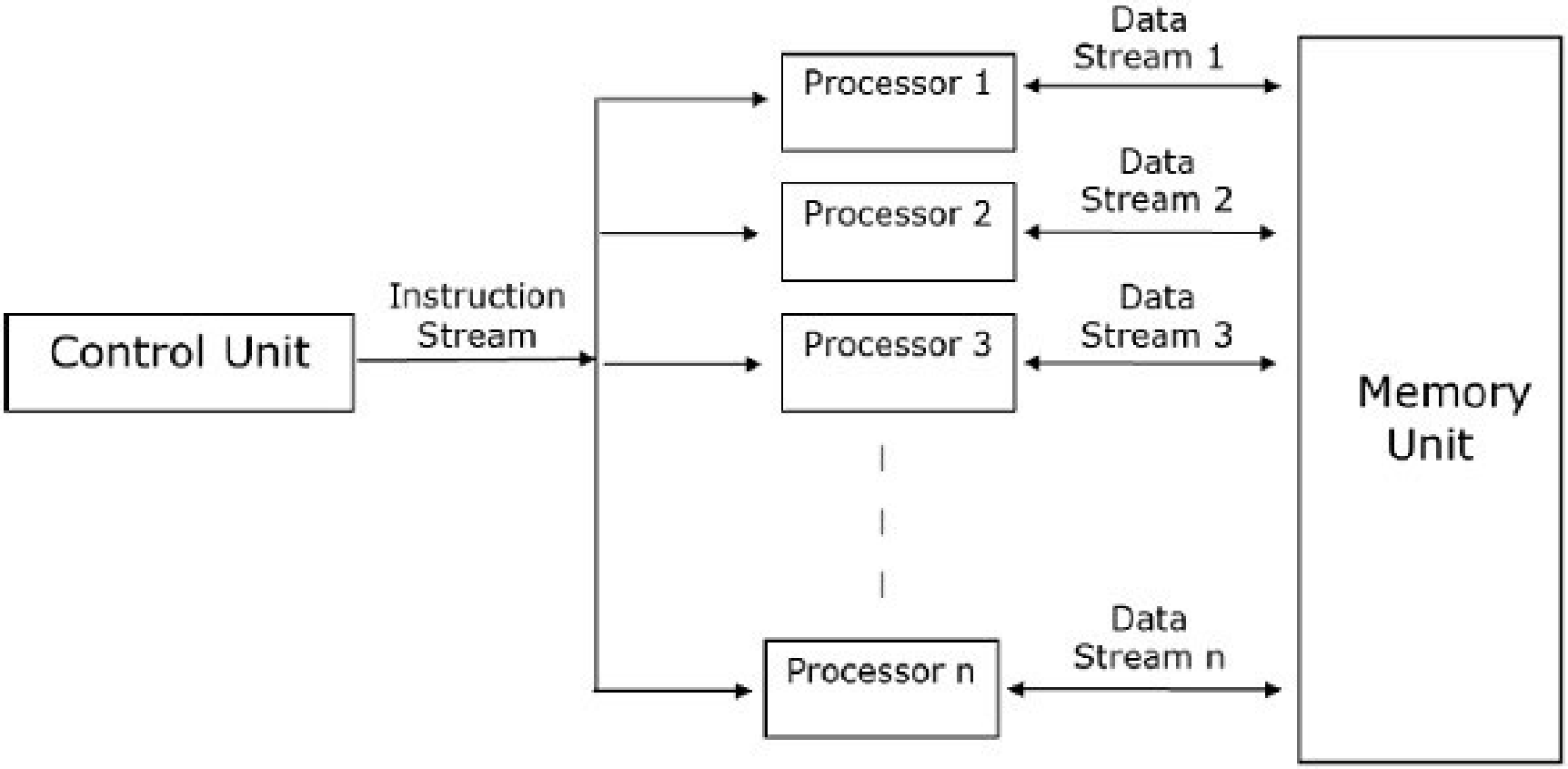


SIMD Computers

- SIMD computers contain **one control unit, multiple processing units,** and **shared memory or interconnection network.**

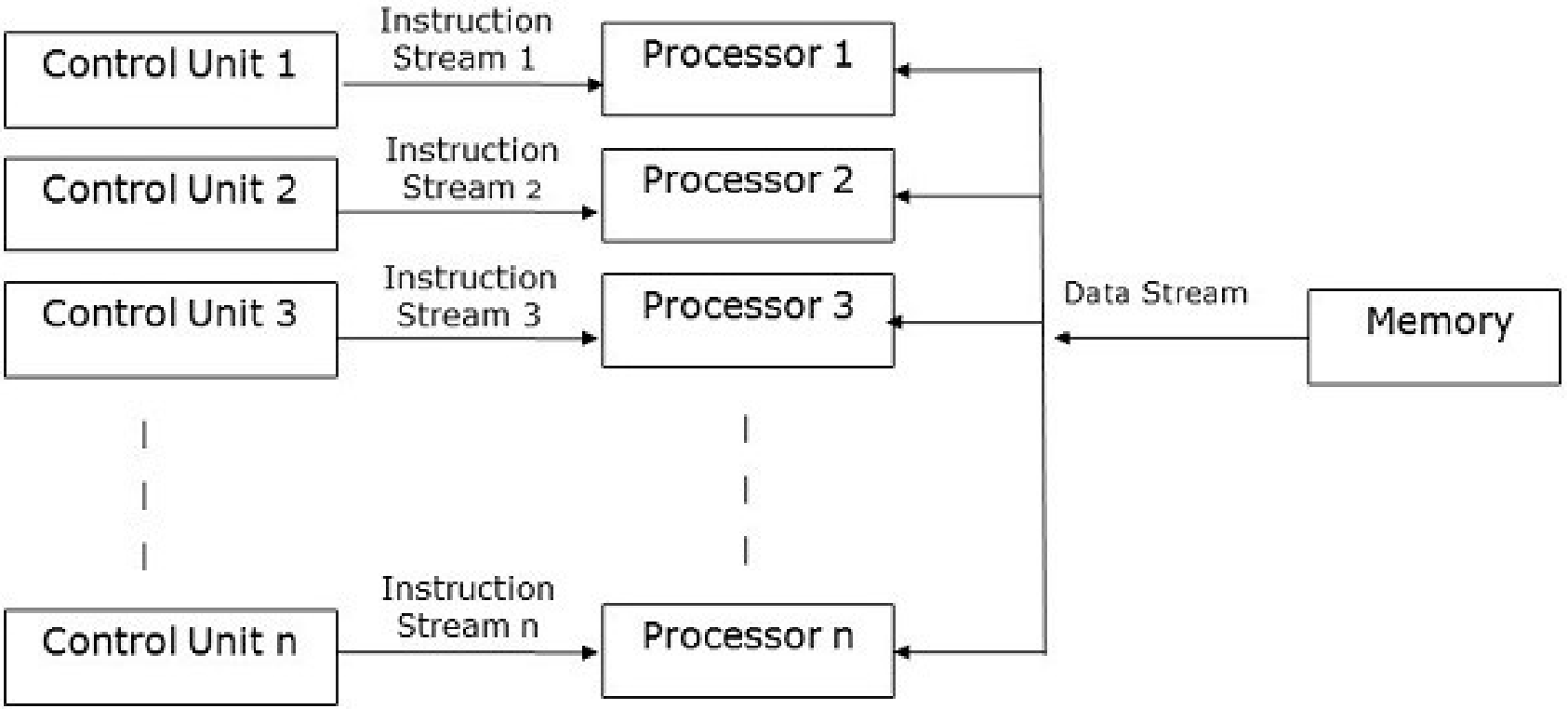


SIMD Computers



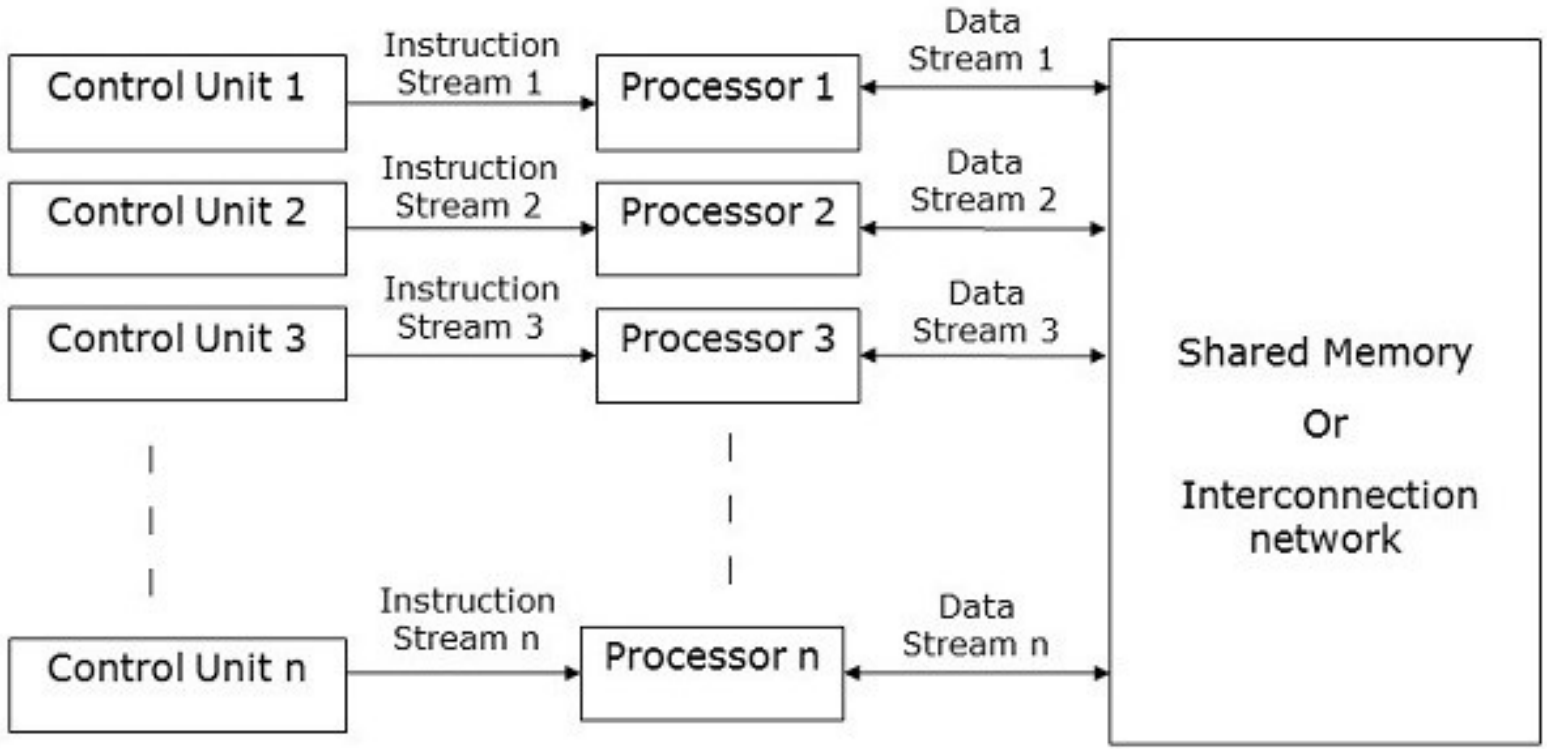
MISD Computers

- MISD computers contain **multiple control units, multiple processing units,** and **one common memory unit**.



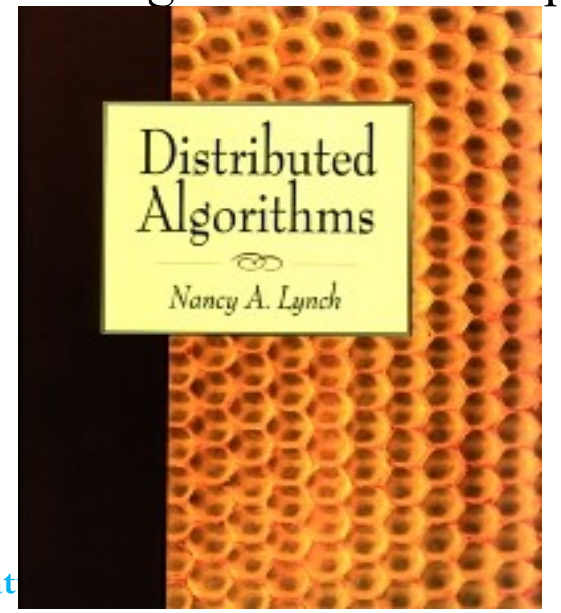
MIMD Computers

MIMD computers have **multiple control units, multiple processing units,** and a **shared memory** or **interconnection network**.

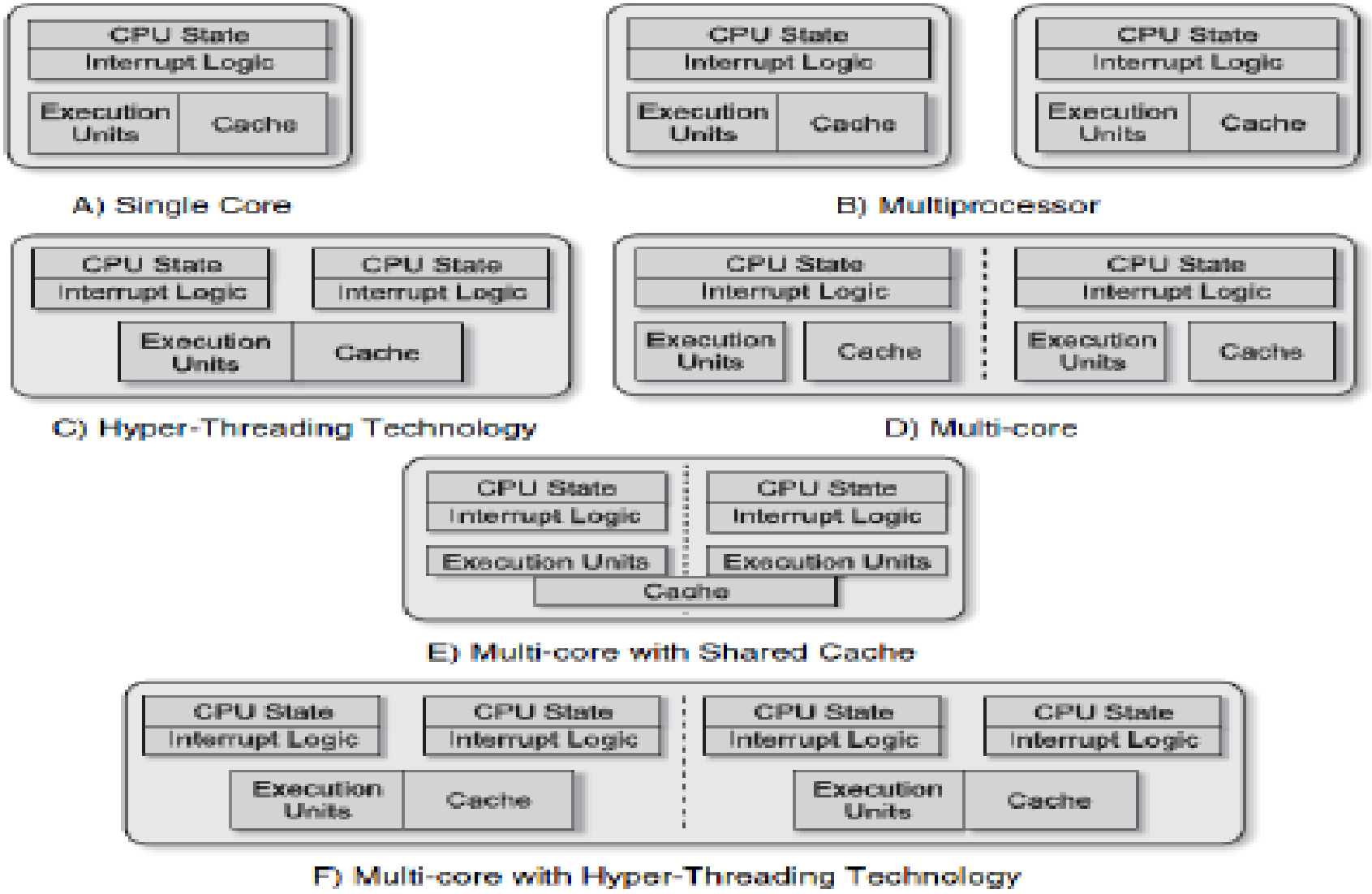


Distributed algorithm

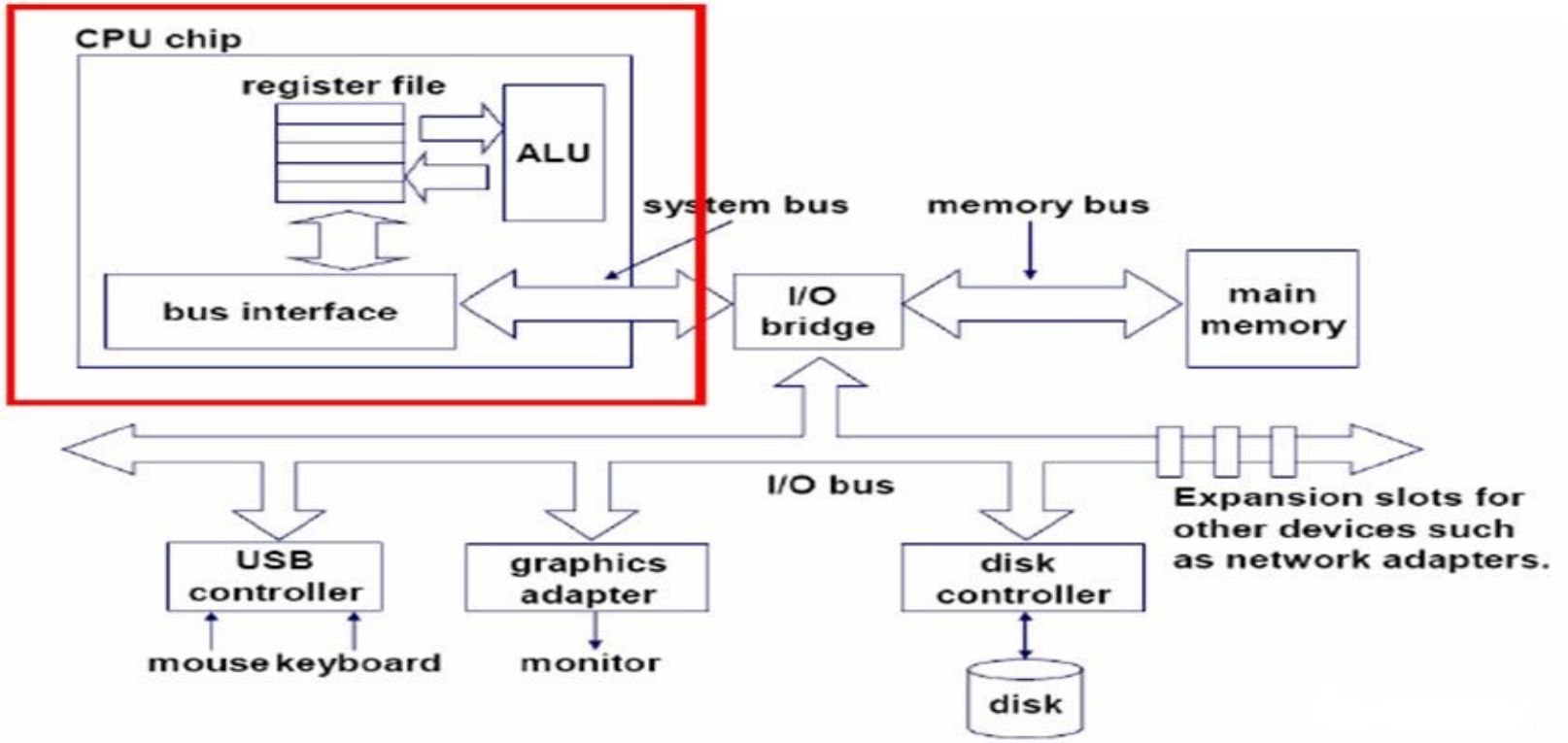
- A **distributed algorithm** is an algorithm, run on a distributed system.
- A **distributed algorithm** is an algorithm designed to run on computer hardware constructed from interconnected processors.
- Distributed algorithms are used in different application areas of distributed computing, such as telecommunications, scientific computing, distributed information processing, and real-time process control.



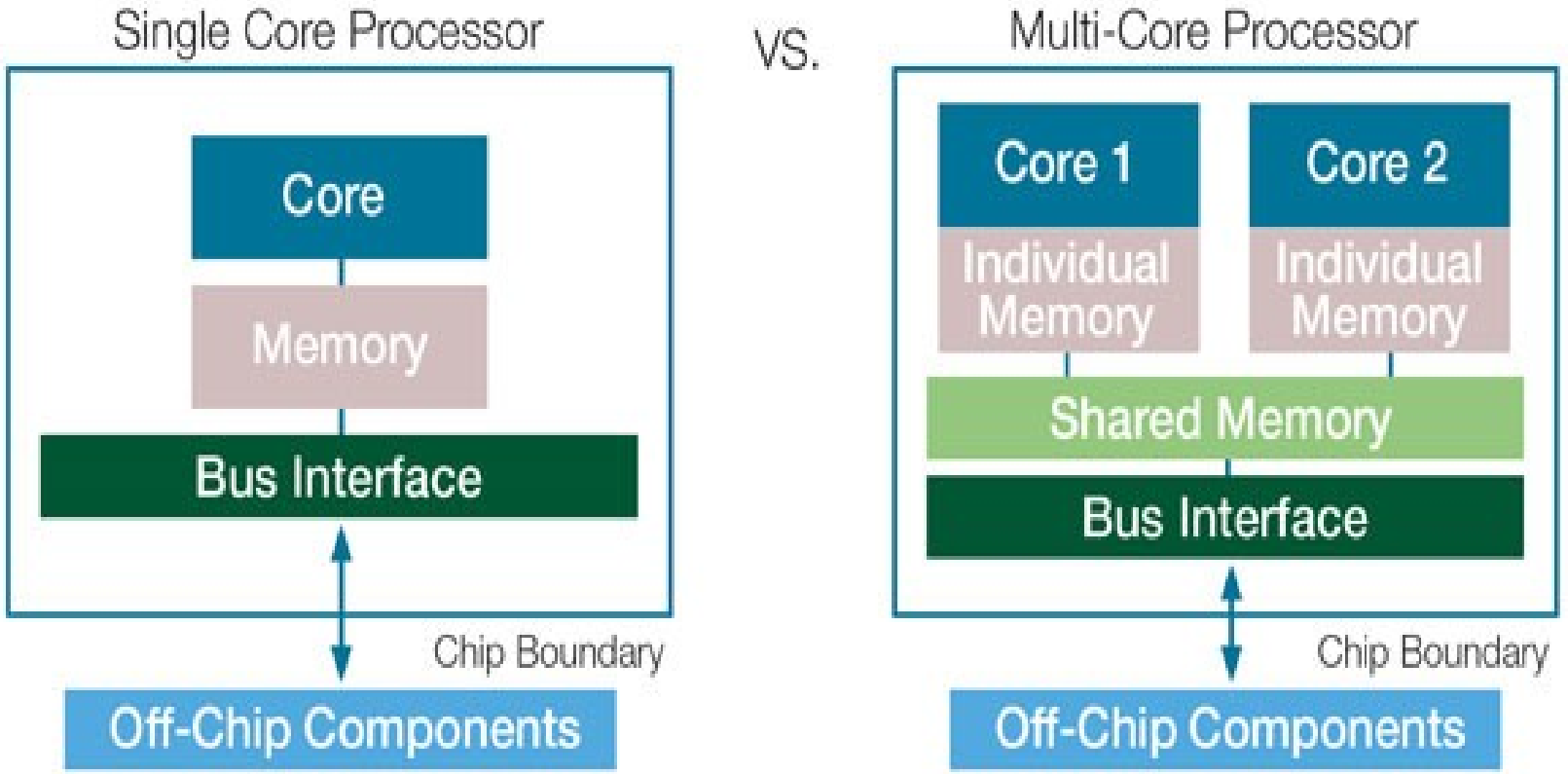
Multicore architecture
&
Programming


CPU State
Interrupt Logic
Execution Units
Cache
A) Single Core
B) Multiprocessor
C) Hyper-Threading Technology
D) Multi-core
E) Multi-core with Shared Cache
F) Multi-core with Hyper-Threading Technology

Single-core computer



Single Core Processor vs Multi Core Processor



Difference Between Multicore and Multiprocessor

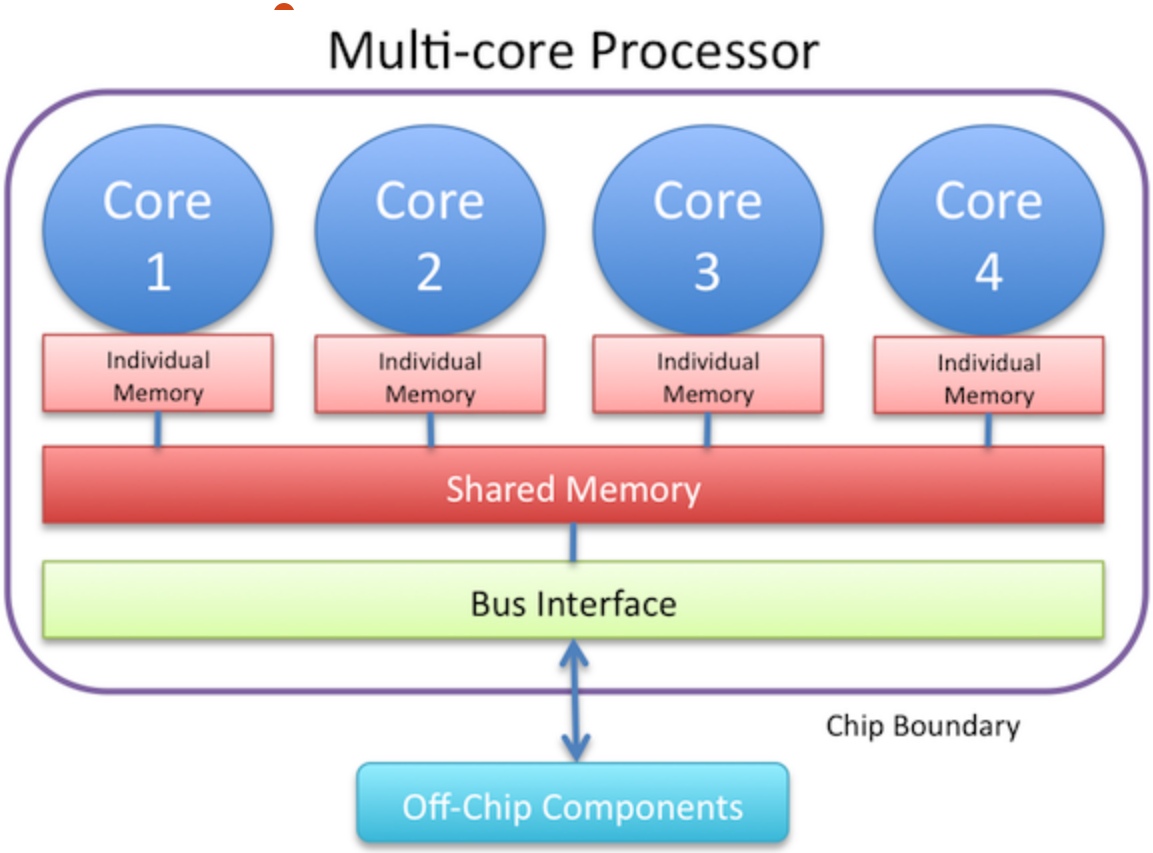
- The **main difference** between multicore and multiprocessor is that the **multicore refers to a single CPU with multiple execution units while the multiprocessor refers to a system that has two or more CPUs.**
- Multicores have multiple cores or processing units in a single CPU. A multiprocessor contains multiple CPUs. Both multicore and multiprocessors help to speed up the computing process. A multicore does not require complex configurations like a multiprocessor. On the other hand, a multiprocessor is more reliable and capable of executing multiple programs. In brief, a multicore has a single CPU whereas a multiprocessor has many CPUs.

Single Core Processor vs Multi Core Processor

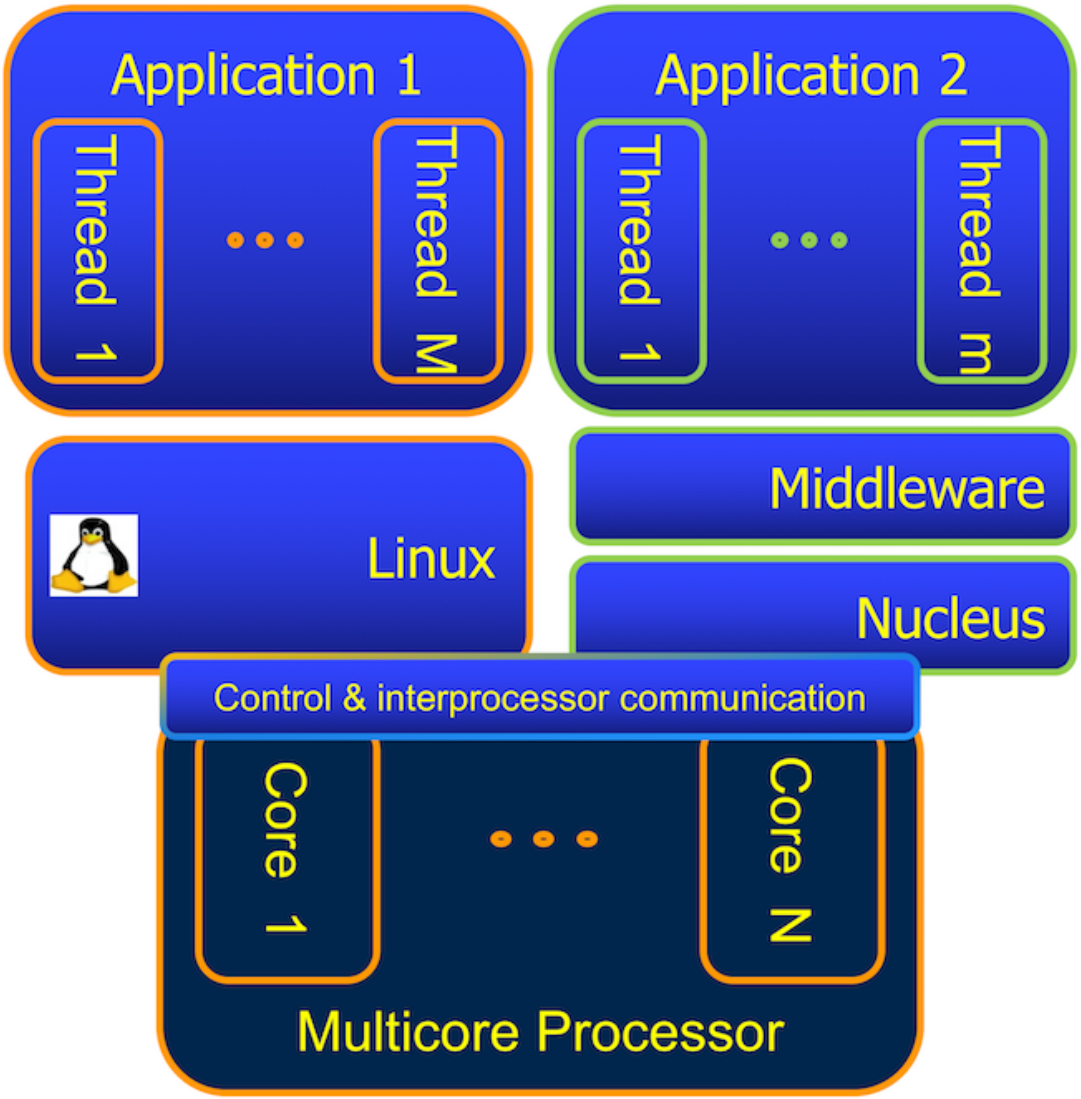
Parameter	Single-Core Processor	Multi-Core Processor
Number of cores on a die	Single	Multiple
Instruction Execution	Can execute Single instruction at a time	Can execute multiple instructions by using multiple cores
Gain	Speed up every program or software being executed	Speed up the programs which are designed for multi-core processors
Performance	Dependent on the clock frequency of the core	Dependent on the frequency, number of cores and program to be executed
Examples	Processor launched before 2005 like 80386,486, AMD 29000, AMD K6, Pentium I,II,III etc.	Processor launched after 2005 like Core-2-Duo,Athlon 64 X2, I3,I5 and I7 etc.

Mmulti-core architecture

- A **multi-core architecture** (or a chip multiprocessor) is a. general-purpose processor that consists of multiple. cores on the same die and can execute programs. simultaneously.



Multicore Processor



Multicore Architecture

- A multicore chip is a chip that has two or more processors. This processor configuration is referred to as CMP. CMPs currently range from dual core to octa - core.
- Hybrid multicore processors can contain different types of processors. The Cell broadband engine is a good example of a hybrid multicore.
- Multicore development can be approached from the bottom up or top down, depending on whether the developers in question are system programmers, kernel programmers, library developers, server developers, or application developers. Each group is faced with similar problems but looks at the cores from a different vantage point.

Multicore Architecture

- All developers that plan to write software that takes advantage of multiprocessor configurations should be familiar with the basic processor architecture of the target platform.
- The primary interface to the specific features of a multicore processor is the C/C++ compiler. To get the most from the target processor or family of target processors, the developer should be familiar with the options of the compiler, the assembler subcomponent of the compiler, and the linker.
- The secondary interface comprises the operating system calls and operating system synchronization and communication components.
- Parallel programming is the art and science of implementing an algorithm, a computer program, or a computer application using sets of instructions or tasks designed to be executed concurrently.
- Multicore application development and design is all about using parallel programming techniques and tools to develop software that can take advantage of CMP architectures.

Three embedded multicore architectures:

- Broadly speaking, there are

1. **Homogeneous multicore**: all the cores are identical.
2. **Heterogeneous multicore**: all the cores are different.
3. **Hybrid:** multiple identical cores, but some different.

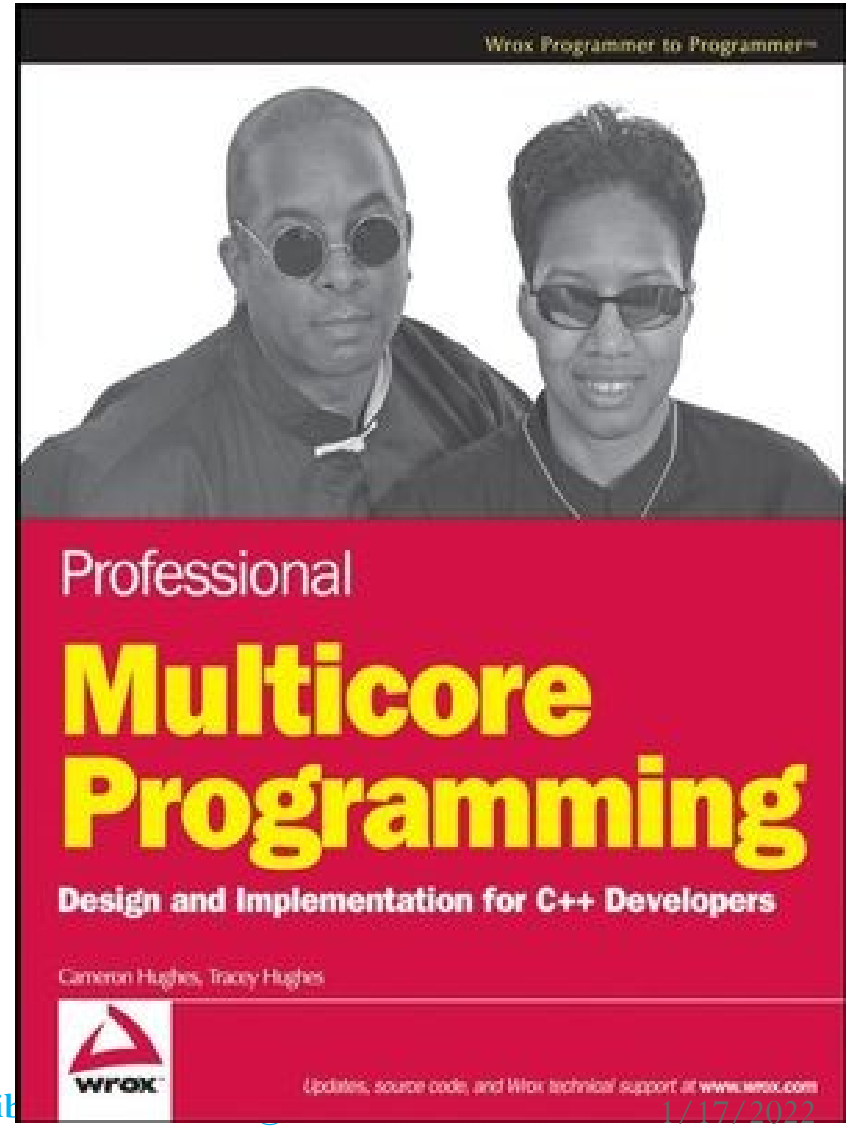
[This may be a selection of regular CPUs and specialized cores. It could also be pairs of CPUs, offering a low- and high-power option for each.]

MULTICORE VERSUS MULTIPROCESSOR

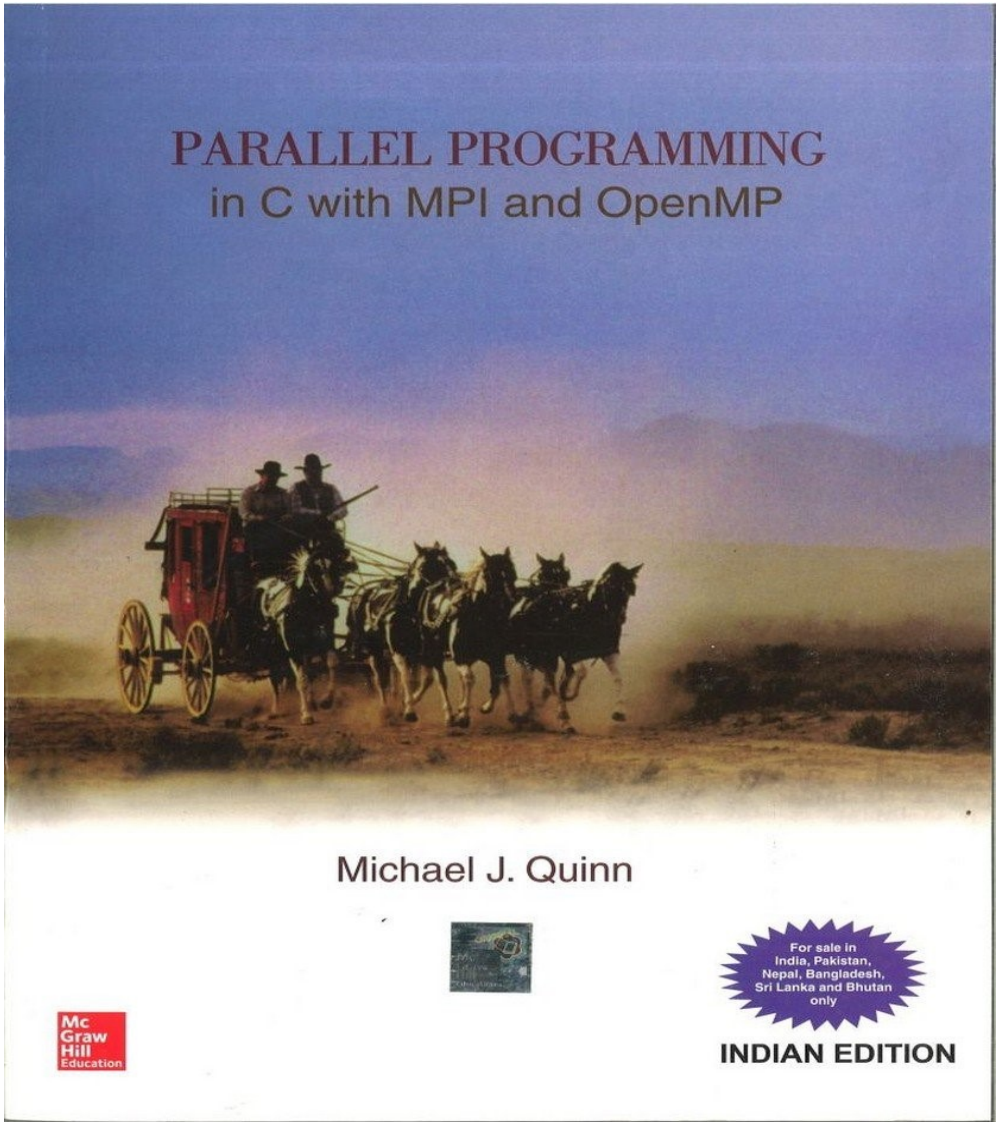
MULTICORE	MULTIPROCESSOR
A single CPU or processor with two or more independent processing units called cores that are capable of reading and executing program instructions	A system with two or more CPUs that allows simultaneous processing of programs
Executes a single program faster	Executes multiple programs faster
Not as reliable as a multiprocessor	More reliable since failure in one CPU will not affect the other
Have less traffic	Have more traffic

Visit www.PEDIAA.com

Multicore Programming reference book



Suggested reading for Multicore Architecture


PARALLEL PROGRAMMING
in C with MPI and OpenMP
Michael J. Quinn
Mc Graw Hill Education
For sale in India, Pakistan, Nepal, Bangladesh, Sri Lanka and Bhutan only
INDIAN EDITION

Algorithmic Thinking
(Computational thinking)

Dr. Bibhudatta Sahoo

Communication & Computing Group

Department of CSE, NIT Rourkela

Email: bdsahu@nitrkl.ac.in, 9937324437, 2462358

Why Study Algorithms?

- **Algorithmic thinking** skills support the development of general reasoning, problem-solving and communication skills by giving students the skills to fluently interpret and design structured procedures and rule systems.

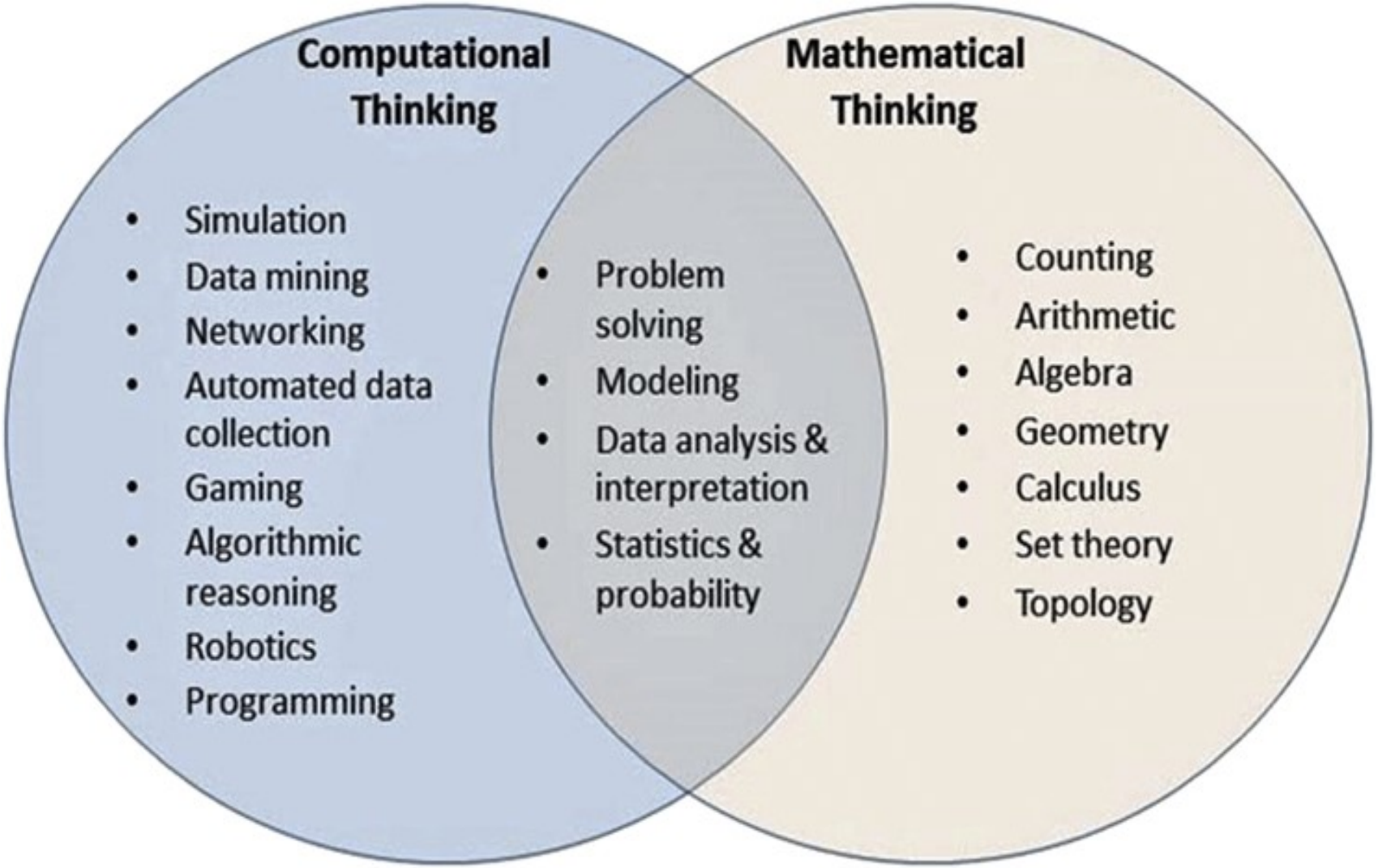
Algorithmic thinking?

- **Algorithmic thinking** skills support the development of general reasoning, problem-solving and communication skills by giving students the skills to fluently interpret and design structured procedures and rule systems.
- **Algorithmic Thinking** is to develop a step-by-step process to solve the problem so that the work is replicable by humans or computers.

Algorithmic thinking

- **Algorithmic thinking**, or **computational thinking**, refers to thinking about these processes for solving problems.
- **Algorithmic thinking** is a way of getting to a solution through the clear definition of the steps needed.
- **Algorithmic thinking** is the use of algorithms, or step-by-step sets of instructions, to complete a task.
- **Algorithmic thinking** has close ties to computer science and mathematics, as algorithms are the key to completing sequences of code or chunking big problems into smaller, more solvable parts.
- **Computational thinking** refers to the thought processes involved in expressing solutions as computational steps or algorithms that can be carried out by a computer. (Cuny, Snyder, & Wing, 2010; Aho, 2011; Lee, 2016).

Computational Thinking vs Mathematical Thinking



Algorithmic thinking

- Thinking computationally is not programming. It is not even thinking like a computer! Simply put, **programming** tells a computer what to do and how to do it.
- **Computational thinking** enables us to work out exactly what to tell the computer to do.

Algorithmic Thinking

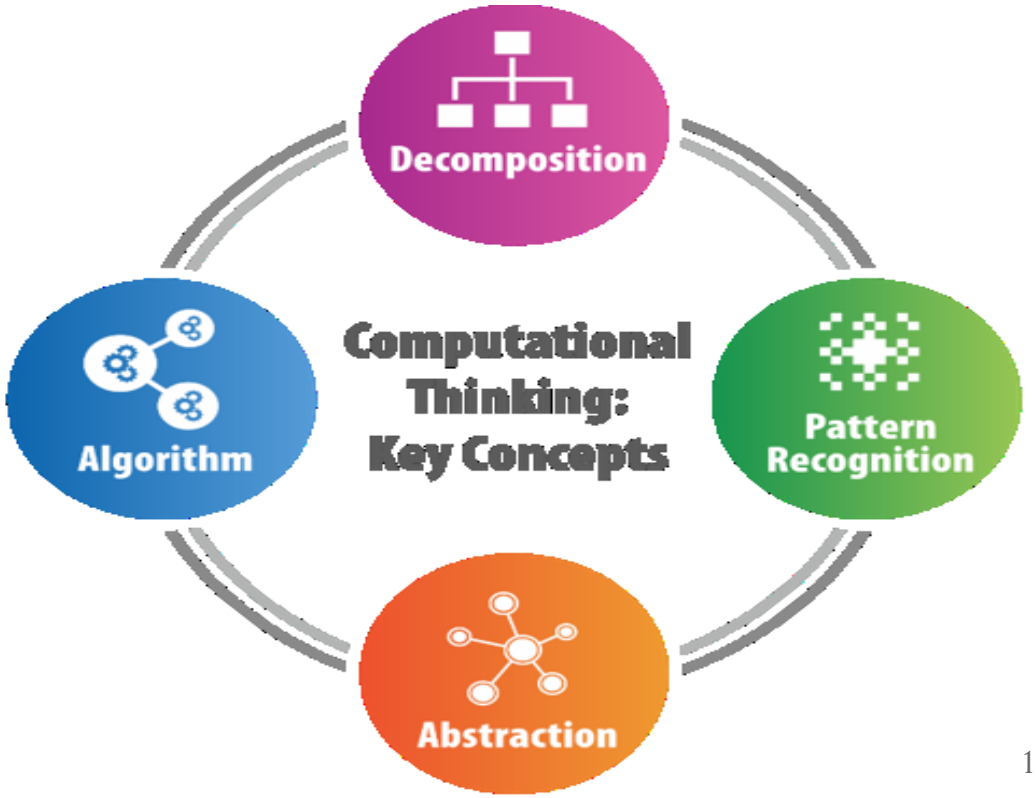
Algorithmic thinking is somehow a pool of abilities that are connected to constructing and understanding algorithms:

- - the ability to analyze given **problems**
- - the ability to specify a problem precisely
- - the ability to find the basic actions that are adequate to the given problem
- - the ability to construct a correct algorithm to a given problem using the basic actions
- - the ability to think about all possible special and normal cases of a problem
- - the ability to improve the efficiency of an algorithm

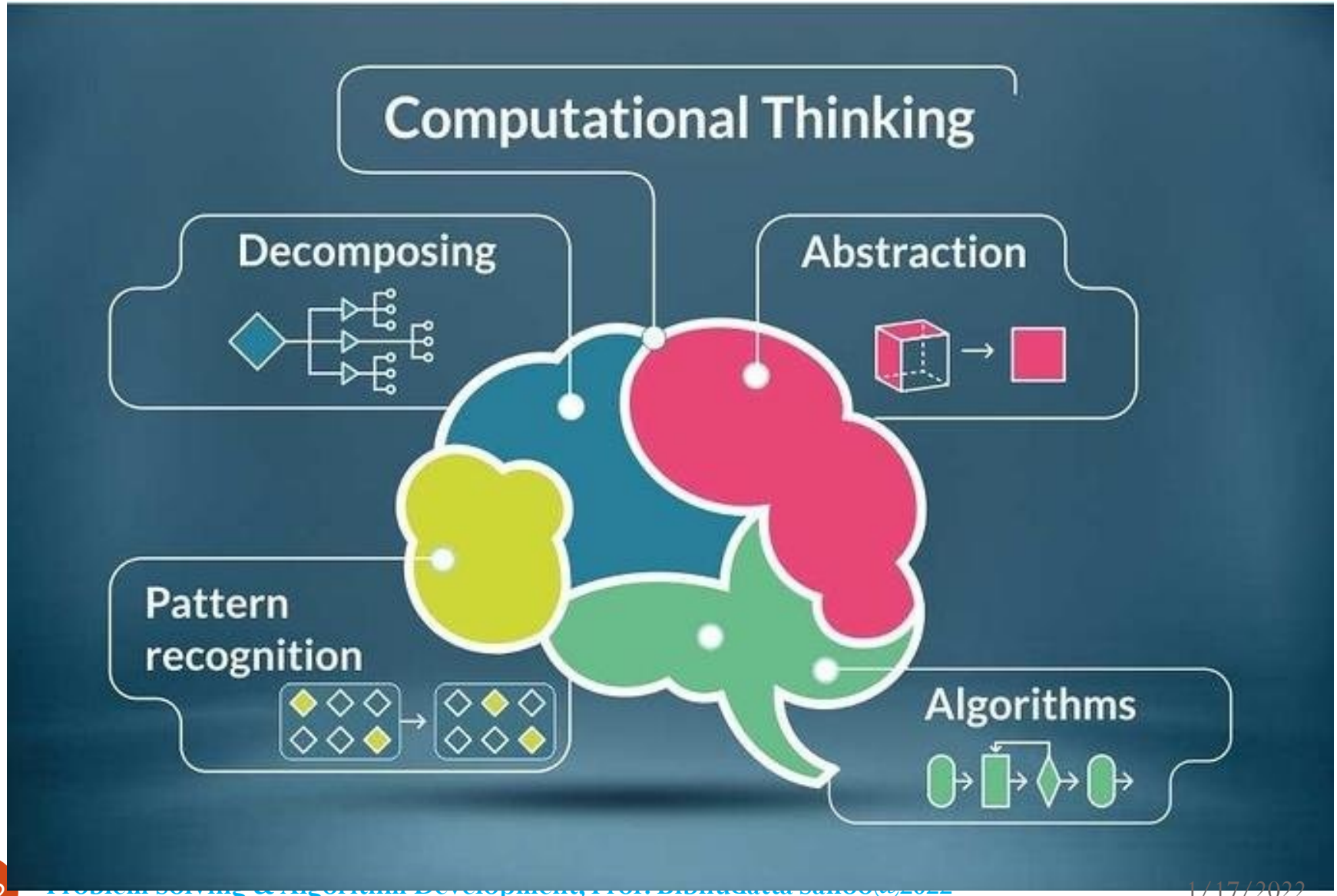
Algorithmic thinking has a strong creative aspect: the construction of new algorithms that solve given problems. If someone wants to do this he needs the ability of algorithmic thinking.

Computational Thinking

- **Computational Thinking** is an approach to problem solving with four key thinking processes; **decomposition**- taking ideas and problems apart, **pattern recognition**- looking for similarities or trends, **abstraction**- focusing on what's most important, and **algorithm design**- creating step-by-step instructions to solve a problem.



Computational thinking



Four key techniques of computational thinking:

- **Decomposition** - breaking down a complex problem or system into smaller, more manageable parts (e.g. where to go, how to complete the level)
- **Abstraction** - focusing on the important information only, ignoring irrelevant detail (e.g. weather, location of exit)
- **Pattern recognition** – looking for similarities among and within problems (e.g. knowledge of previous similar problems used)
- **Algorithms** - developing a step-by-step solution to the problem, or the rules to follow to solve the problem (e.g. to work out a step-by-step plan of action)

Computational thinking to solve a complex problem

STEP 1: SCOPE

- What would you like to do?
- What's the problem you have identified?
- Define problem statement

STEP 2: GENERATE

- Look for various solutions for your problem.
- Generate multiple scenarios – ideas

STEP 3: ANALYZE

- Go through each Idea/scenario and analyze

STEP 4: OPTIMIZE

- Fine tune the idea

Problem Solving and Algorithmic Thinking

Unit 1

- Problem Solving and Algorithmic Thinking Overview – problem definition, logical reasoning; Algorithm – definition, practical examples, properties, representation, algorithms vs programs.

Unit 2

- Algorithmic thinking – Constituents of algorithms – Sequence, Selection and Repetition, input-output; Computation – expressions, logic; algorithms vs programs, Problem Understanding and Analysis – problem definition, input-output, variables, name binding, data organization: lists, arrays etc. algorithms to programs.

Unit 3

- Problem solving with algorithms – Searching and Sorting, Evaluating algorithms, modularization, recursion. C for problem solving – Introduction, structure of C programs, data types, data input, output statements, control structures.

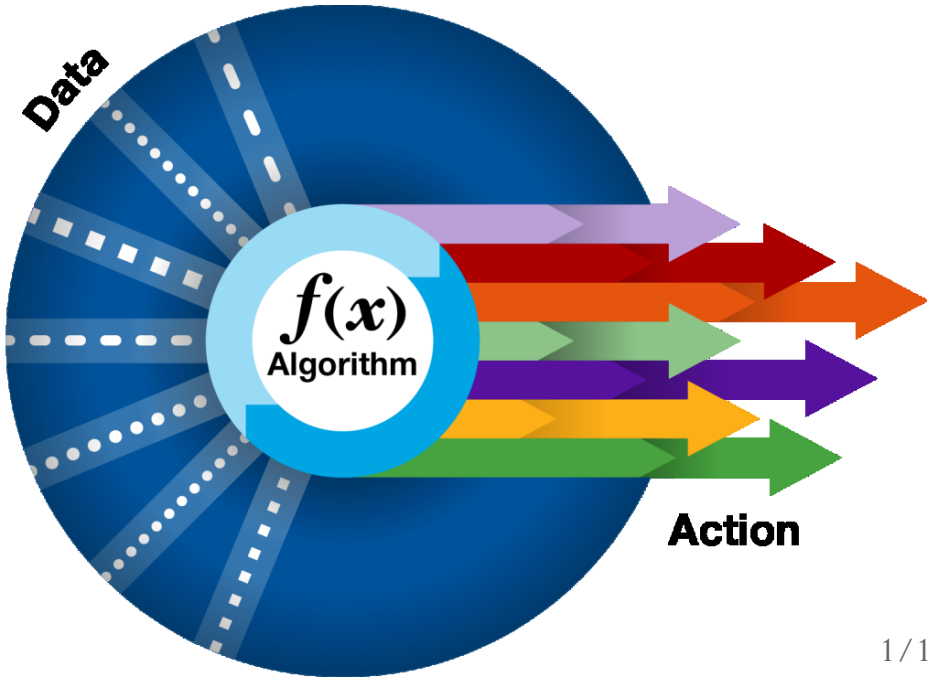
Reference: Problem Solving and Algorithmic Thinking

- Riley DD, Hunt KA. Computational Thinking for the Modern Problem Solver. CRC press; 2014 Mar 27.
- Ferragina P, Luccio Computational Thinking: First Algorithms, Then Code. Springer; 2018.
- Beecher Computational Thinking: A beginner's guide to Problem-solving and Programming.BCS Learning & Development Limited; 2017.
- Curzon P, McOwan The Power of Computational Thinking: Games, Magic and Puzzles to help you become a computational thinker. World Scientific Publishing Company; 2017.

Algorithmic business

Algorithmic business

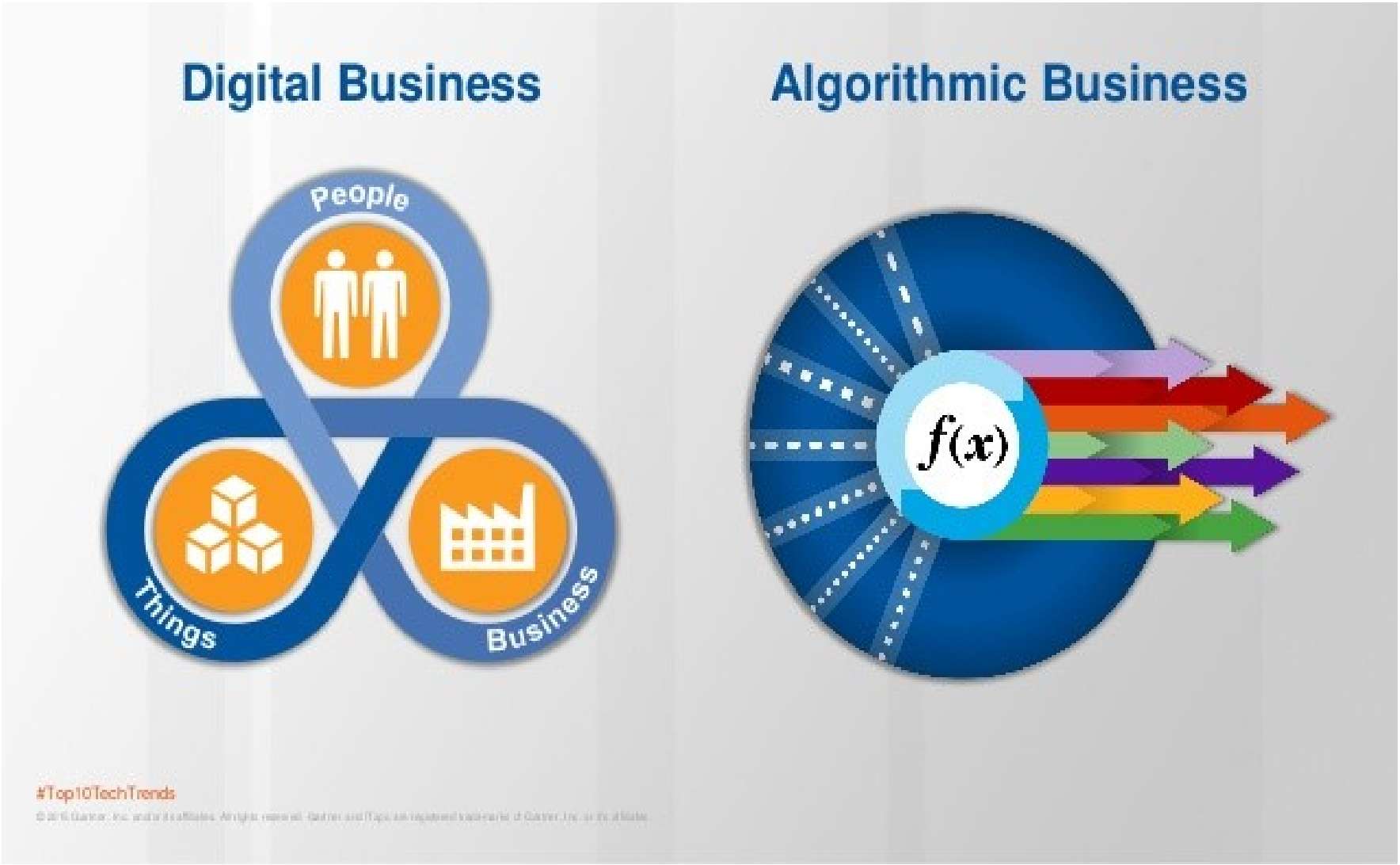
- **Algorithmic business** is the industrialized use of complex mathematical algorithms pivotal to driving improved business decisions or process automation for competitive differentiation.
- Algorithmic business provides the speed and scale to accelerate digital business to deliver even greater impact.



Algorithm application in business

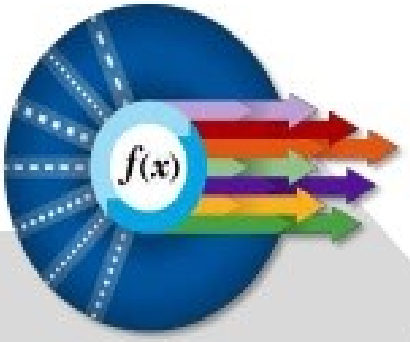
- Algorithms have actually been at the heart of some of the most successful corporate empires in the world.
- Google was born from an algorithm helping internet users find the information they need; while Coca-Cola boasts of its "secret recipe" – its own version of an algorithm.
- Meanwhile, the likes of the traders from Wall Street, Amazon and more have all relied on algorithms in the way they work.
- With organisations now generating more data than ever, collecting data is no longer a problem – instead, it's all about how that data can be used: and processing data is done best when it's achieved through algorithmic software.

Digital business vs Algorithmic business



What Is Changed?

The Internet of Things creates a digital version of physical world devices.



Digital business creates new business designs through merging the physical with the digital.

Algorithmic business applies the knowledge encapsulated in algorithms — business gets smarter and more powerful!

Gartner

Algorithmic business: An Introduction

- Algorithmic business is the industrialized use of complex mathematical algorithms pivotal to driving improved business decisions or process automation for competitive differentiation.
- Algorithmic business provides the speed and scale to accelerate digital business to deliver even greater impact.
- With sufficient data sources and enough data to ensure algorithms are capable of offering real-time and actionable results that benefit the enterprise.

Example: Netflix

- Netflix is a classic example of an algorithmic business model. In fact, the success of their streaming video service is owed almost entirely to algorithms.
- How does Netflix learn your preferences and make quality viewing recommendations? Algorithms.
- How does viewer behavior inform Netflix on the shows to make and the viewing demographics to cater their content toward? Once again, algorithms.
- Netflix as we know it would not exist but for sophisticated algorithms that allow Netflix executives to take meaningful action based on an algorithm's ability to provide clear insight into the big picture of big data.

Example: Shipping company

- Consider a shipping company such as UPS. Dropping off packages efficiently is a critical component of their business model, but there are a number of variables to consider.
- Truck maintenance, employee work hours, traffic and more all impact the ability to follow through on delivery times. A quality algorithm could sift through vast amounts of applicable data to determine the best route and delivery strategies for all the packages on each truck.
- Companies cannot afford to ignore this level of advanced efficiency. Before business algorithms transformed enterprise, companies were forced to rely on teams of analysts to find patterns that were used to create data trends to use for better decision-making. This advanced work may take weeks to complete and calculate, whereas business algorithms are equipped to perform those same calculations in mere seconds.

Millennium Problems ...

The Clay Mathematics Institute (CMI)

http://www.claymath.org/

Millennium Problems ...

Yang–Mills and Mass Gap

- Experiment and computer simulations suggest the existence of a "mass gap" in the solution to the quantum versions of the Yang-Mills equations. But no proof of this property is known.

Riemann Hypothesis

- The prime number theorem determines the average distribution of the primes. The Riemann hypothesis tells us about the deviation from the average. Formulated in Riemann's 1859 paper, it asserts that all the 'non-obvious' zeros of the zeta function are complex numbers with real part 1/2.

P vs NP Problem

- If it is easy to check that a solution to a problem is correct, is it also easy to solve the problem? This is the essence of the P vs NP question. Typical of the NP problems is that of the Hamiltonian Path Problem: given N cities to visit, how can one do this without visiting a city twice? If you give me a solution, I can easily check that it is correct. But I cannot so easily find a solution.

Millennium Problems

Navier–Stokes Equation

- This is the equation which governs the flow of fluids such as water and air. However, there is no proof for the most basic questions one can ask: do solutions exist, and are they unique? Why ask for a proof? Because a proof gives not only certitude, but also understanding.

Hodge Conjecture

- The answer to this conjecture determines how much of the topology of the solution set of a system of algebraic equations can be defined in terms of further algebraic equations. The Hodge conjecture is known in certain special cases, e.g., when the solution set has dimension less than four. But in dimension four it is unknown.

Millennium Problems ...

Poincaré Conjecture

- In 1904 the French mathematician Henri Poincaré asked if the three dimensional sphere is characterized as the unique simply connected three manifold. This question, the Poincaré conjecture, was a special case of Thurston's geometrization conjecture. Perelman's proof tells us that every three manifold is built from a set of standard pieces, each with one of eight well-understood geometries.

Birch and Swinnerton-Dyer Conjecture

- Supported by much experimental evidence, this conjecture relates the number of points on an elliptic curve mod p to the rank of the group of rational points. Elliptic curves, defined by cubic equations in two variables, are fundamental mathematical objects that arise in many areas: Wiles' proof of the Fermat Conjecture, factorization of numbers into primes, and cryptography, to name three.

References

- https://www.tutorialspoint.com/data_structures_algorithms/algorithms_basics.htm
- https://commons.wikimedia.org/wiki/File:Computer_Systems_-_Von_Neumann_Architecture_Large_poster_anchor_chart.svg

Thanks for Your Attention!

Sample problems

1. Write the most efficient algorithm you can think of for the following: Find the k-th item in an n-node doubly-linked list. What is the running time in terms of big-theta?
2. Write the most efficient algorithm you can think of for the following: Given a set of p points, find the pair closest to each other. Be sure and try a divide-and-conquer approach, as well as others. What is the running time in terms of big-theta?
3. Design and implement an algorithm that finds all the duplicates in a random sequence of integers.
4. Reconsider the three algorithms you just designed given the following change: the input has increased by 1,000,000,000 times.

Sample problems

5. You have two arrays of integers. Within each array, there are no duplicate values but there may be values in common with the other array. Assume the arrays represent two different sets. Define an O(n log n) algorithm that outputs a third array representing the union of the two sets. The value "n" in O(n log n) is the sum of the sizes of the two input arrays.
6. You are given an increasing sequence of numbers u_1, u_2, … u_m, and a decreasing series of numbers d_1, d_2, …d_n. You are given one more number C and asked to determine if C can be written as the sum of one u_i and one d_j. There is an obvious brute force method, just comparing all the sums to C, but there is a much more efficient solution. Define an algorithm that works in linear time.

Sample problems

7. Suppose you need to search a given ordered array of n integers for a specific value. The entries are sorted in non-decreasing order. Give a simple algorithm that has a worst-case complexity of $\Theta(n)$. We can do better by using the "binary search" strategy. Explain what it is and give an algorithm based on it. What is the worst-case complexity of this algorithm?
8. Write a program that fills an array of length **N** with random values. Then run the **maximum** code, incrementing a counter whenever a new maximum is found. Run this program for large values of **N** and try to infer a formula for that approximates how many times this happens. Run the program over and over again, with the same large **N**, then with double that size, quadruple that size, etc. to test your formula.

Sample problems

9 Peripatetic Shipping Lines, Inc., is a shipping company that owns n ships and provides service to n ports. Each of its ships has a *schedule* that says, for each day of the month, which of the ports it's currently visiting, or whether it's out at sea. (You can assume the "month" here has m days, for some $m > n$.) Each ship visits each port for exactly one day during the month. For safety reasons, PSL Inc. has the following strict requirement:

(†) *No two ships can be in the same port on the same day.*

The company wants to perform maintenance on all the ships this month, via the following scheme. They want to *truncate* each ship's schedule: for each ship S_i, there will be some day when it arrives in its scheduled port and simply remains there for the rest of the month (for maintenance). This means that S_i will not visit the remaining ports on its schedule (if any) that month, but this is okay. So the *truncation* of S_i's schedule will simply consist of its original schedule up to a certain specified day on which it is in a port P; the remainder of the truncated schedule simply has it remain in port P.

Now the company's question to you is the following: Given the schedule for each ship, find a truncation of each so that condition (†) continues to hold: no two ships are ever in the same port on the same day.

Show that such a set of truncations can always be found, and give an algorithm to find them.

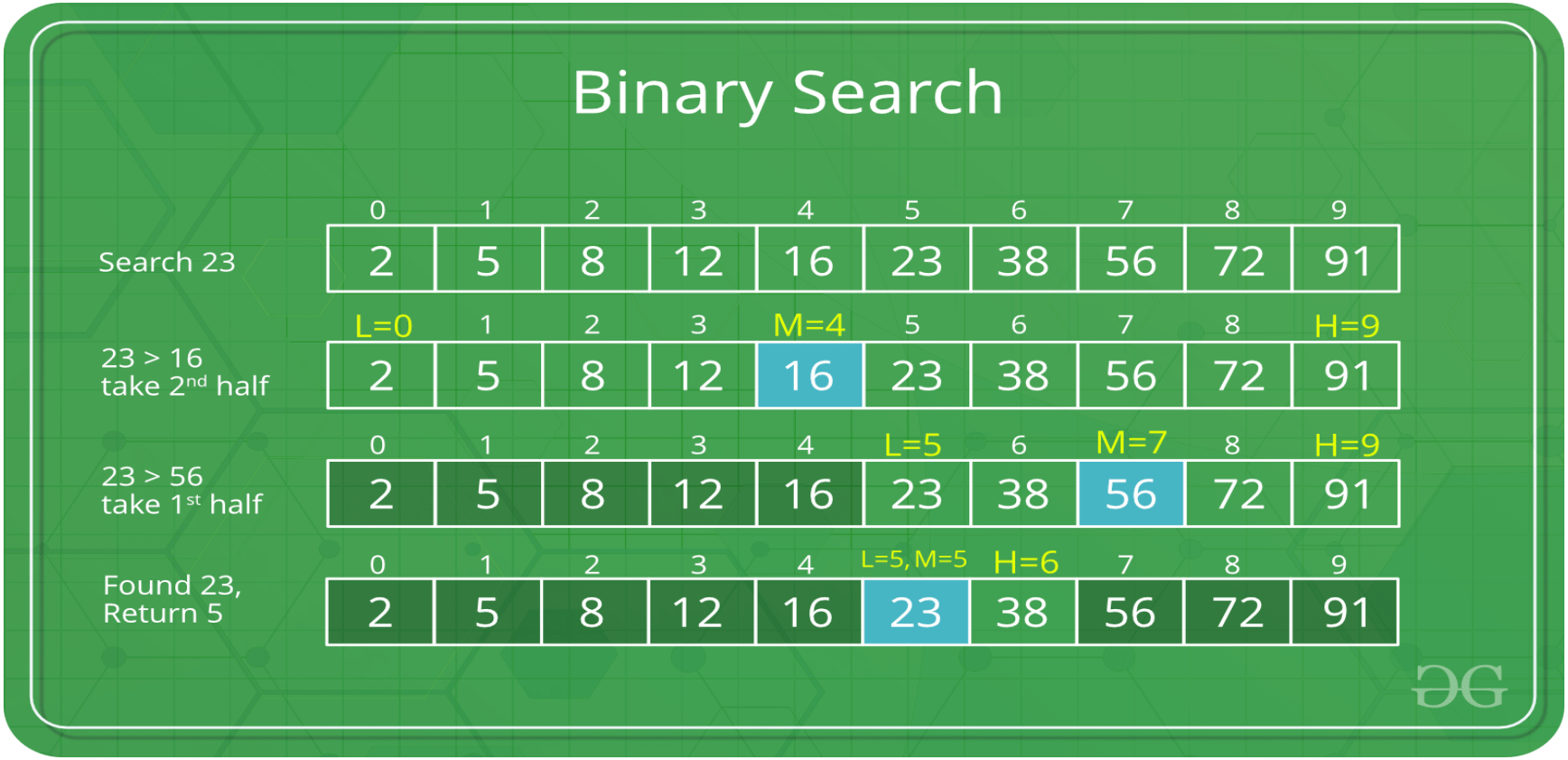
10 Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some d. Each node v of T is labeled with a real number x_v. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

You are given such a complete binary tree T, but the labeling is only specified in the following *implicit* way: for each node v, you can determine the value x_v by *probing* the node v. Show how to find a local minimum of T using only $O(\log n)$ *probes* to the nodes of T.

Review

Binary Search

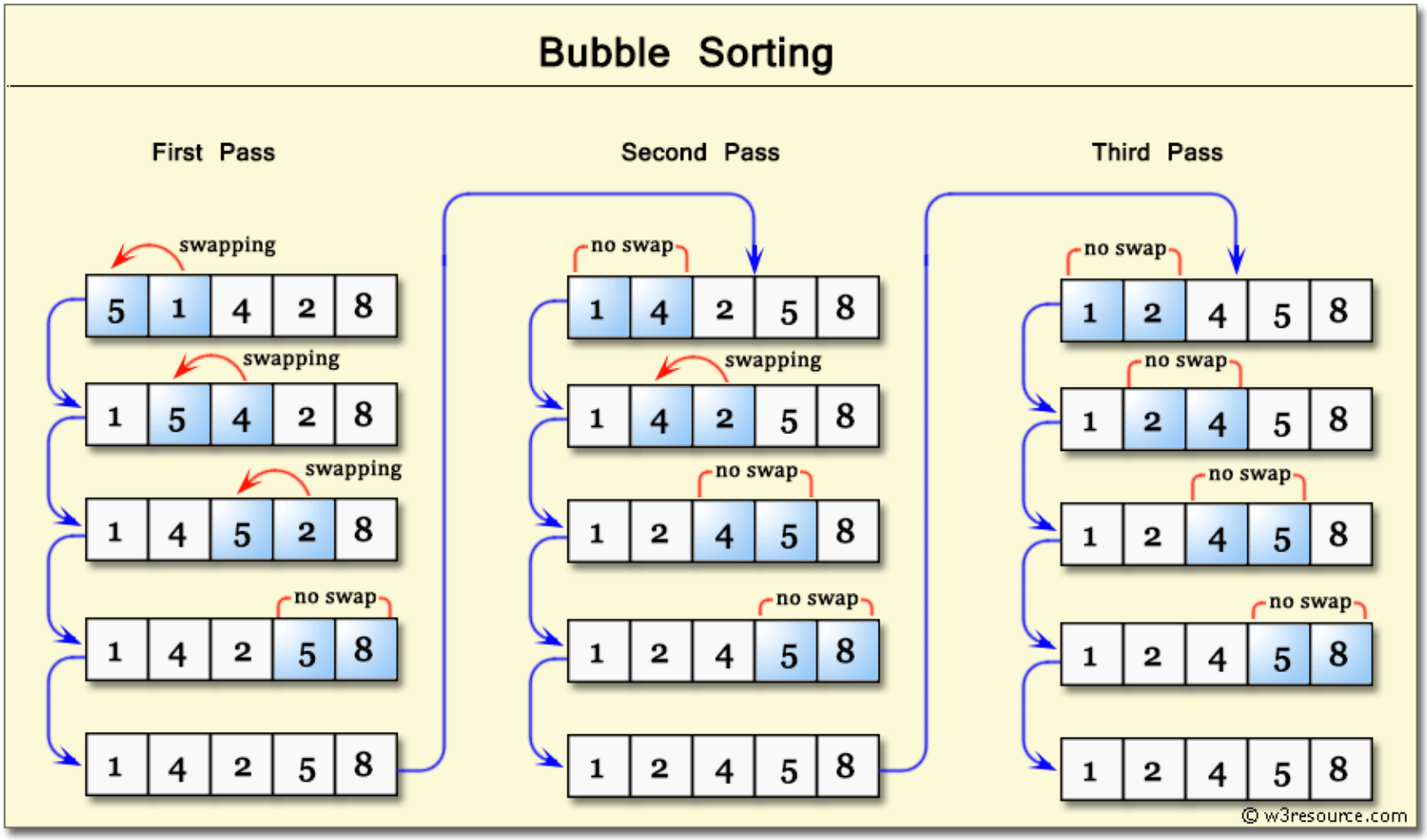
1. Data structure must be sorted in the **same** order as the one assumed by the binary search algorithm.
2. No duplicate element/item present in the array



Bubble Sort

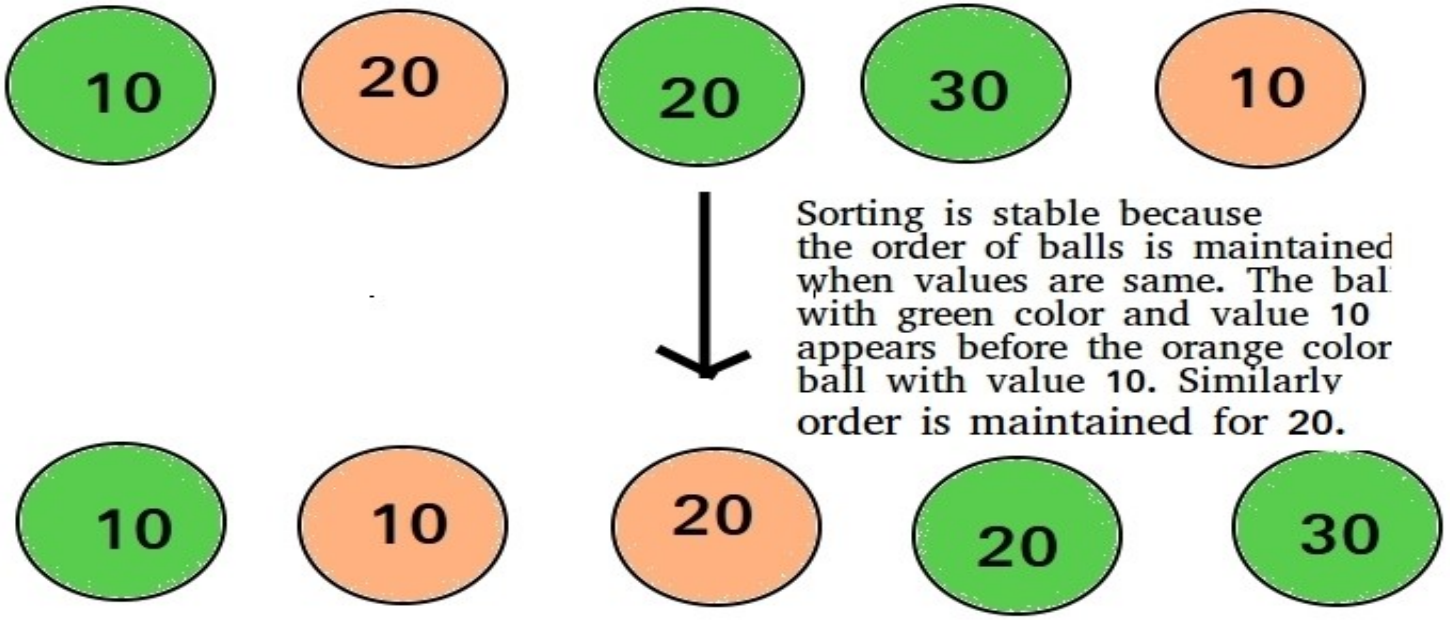
- Bubble Sort is a stable, in-place, internal sorting algorithm.

Bubble Sort


Bubble Sorting
First Pass
Second Pass
Third Pass
swapping
swapping
swapping
no swap
5 1 4 2 8
1 5 4 2 8
1 4 5 2 8
1 4 2 5 8
1 4 2 5 8
no swap
swapping
no swap
no swap
1 4 2 5 8
1 4 2 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
no swap
no swap
no swap
no swap
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
1 2 4 5 8
© w3resource.com

Stability in Sorting Algorithm

- A sorting algorithm is said to be stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.
- Stable algorithms **preserve the relative order of equal elements.**

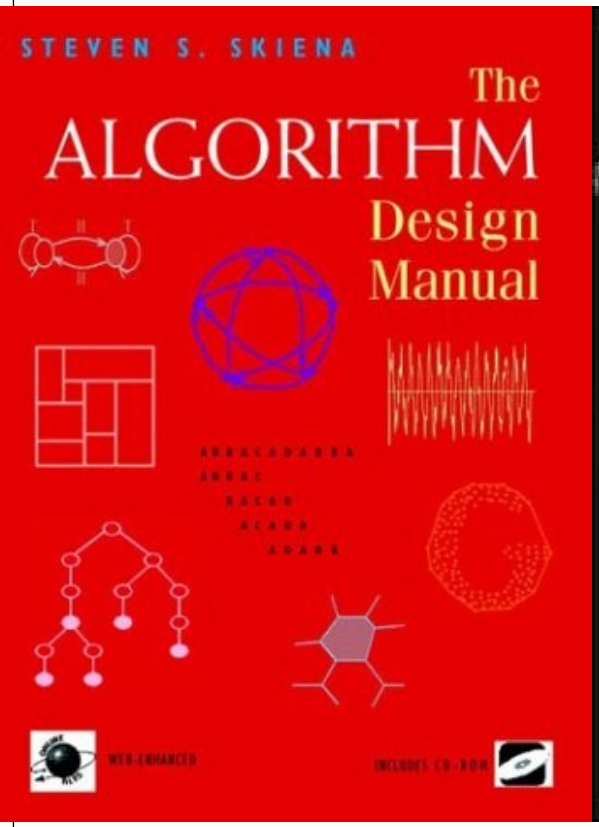


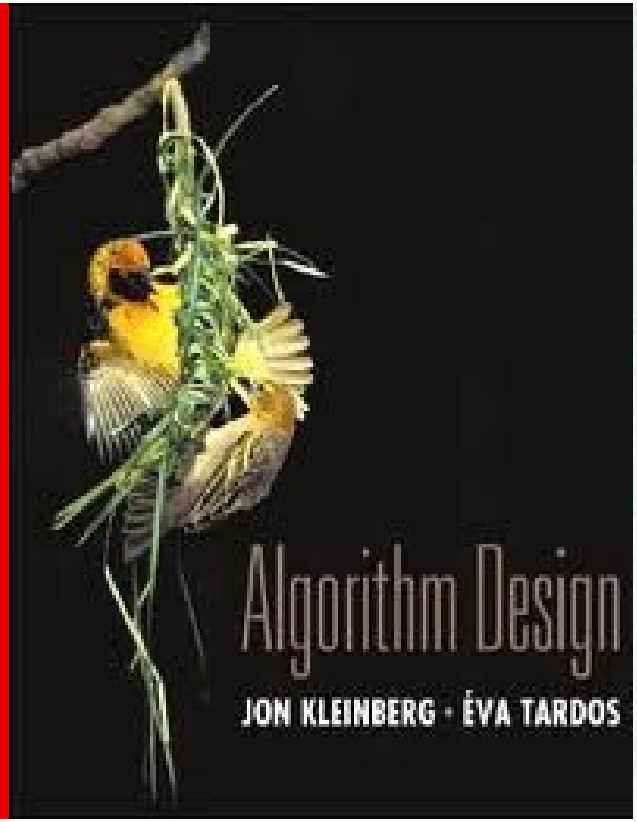
In place sorting algorithm

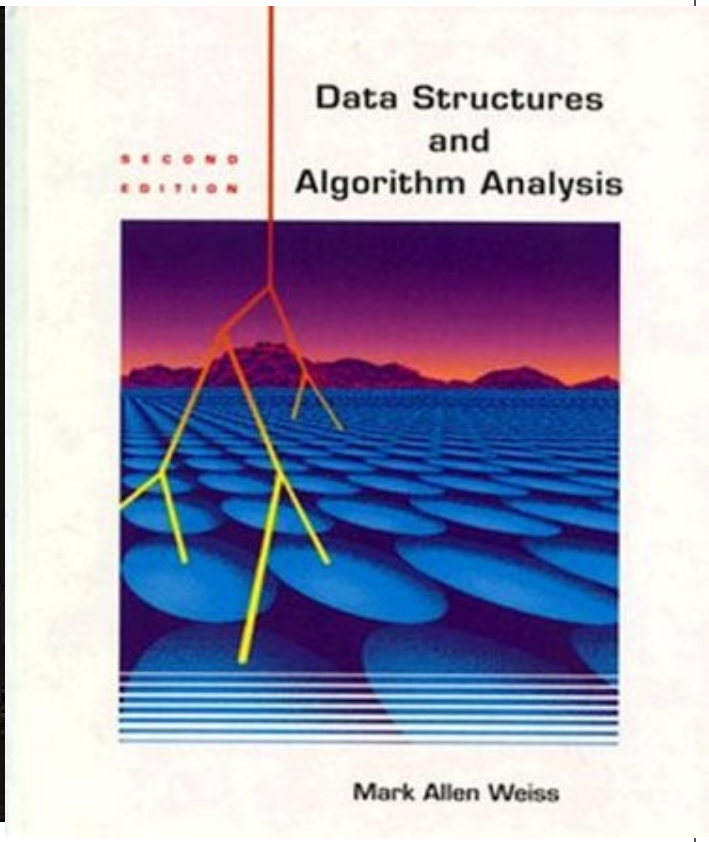
- Quick sort is an in-place algorithm as it does not use any extra memory. It shuffles the elements in the problem array itself.

Reference:

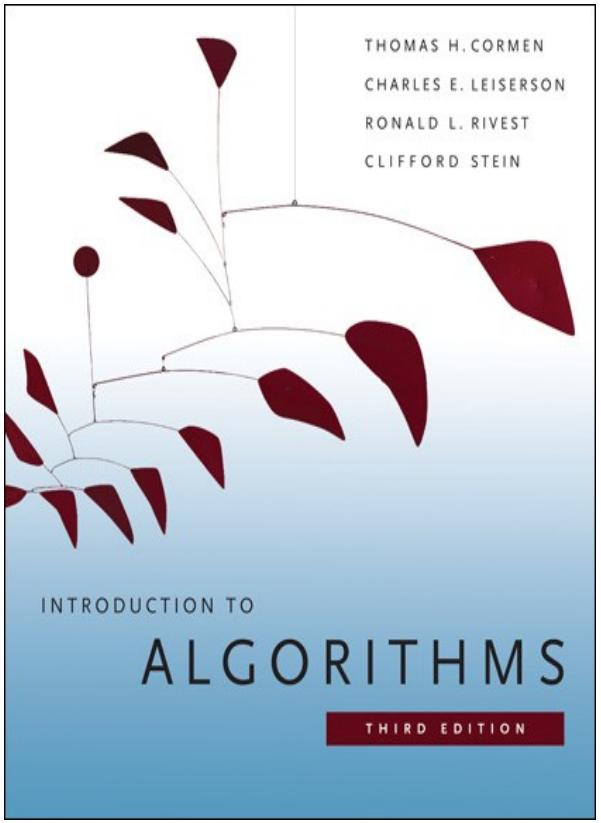
Steven S. Skiena, The Algorithm Design Manual, Springer-Verlag London Limited, 2008.

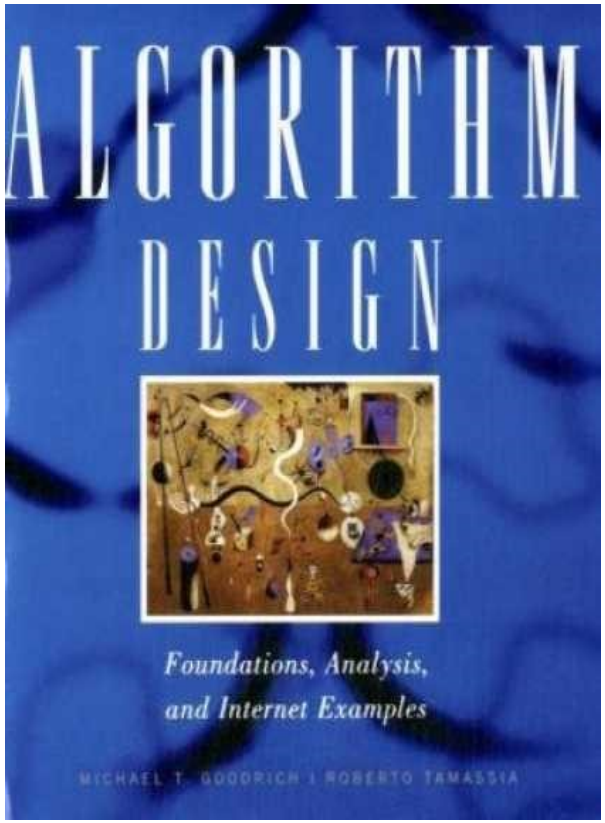


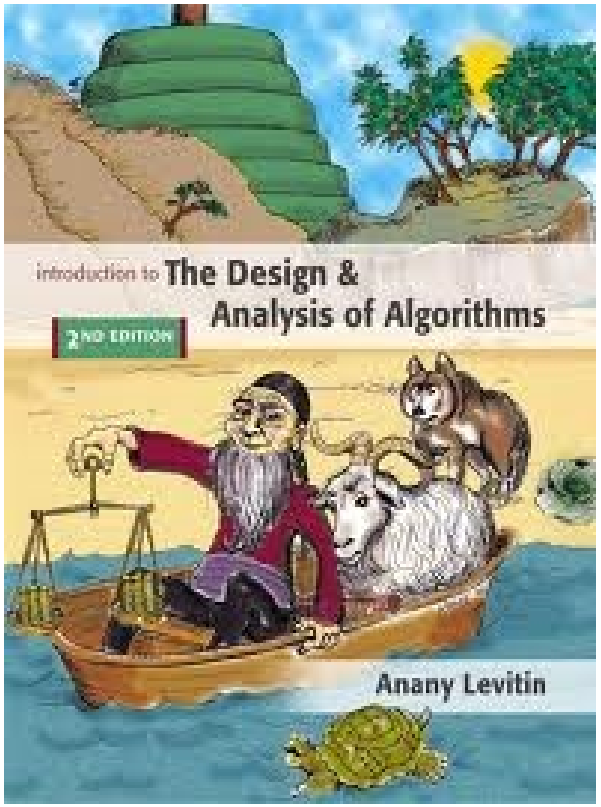




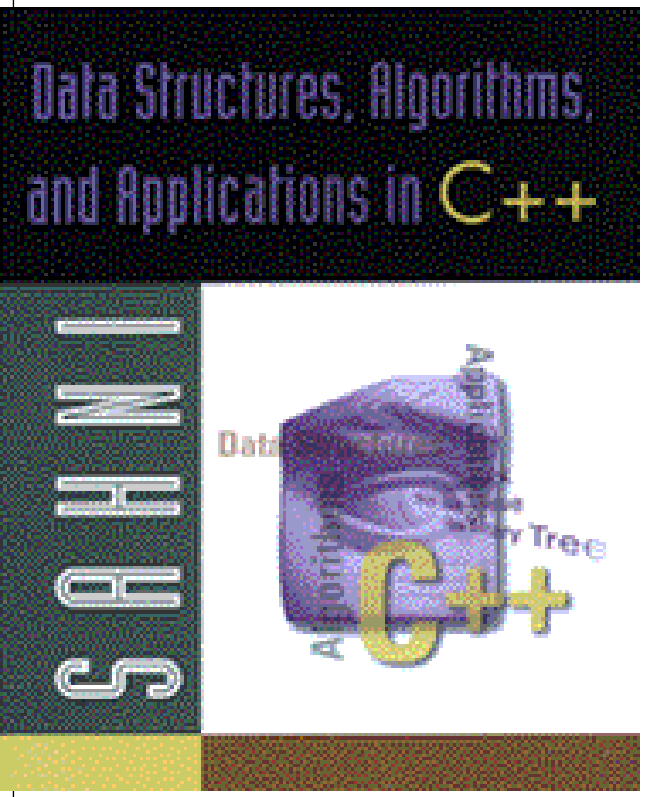
Reference:

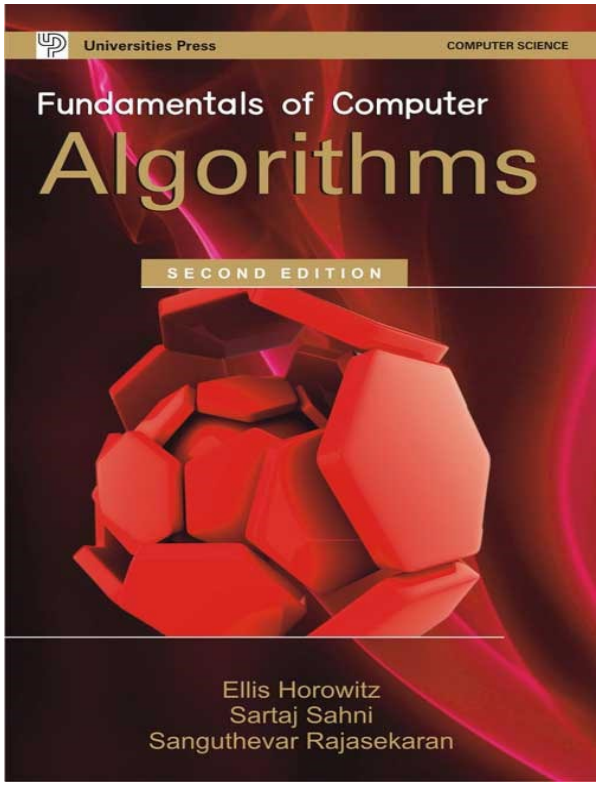

THOMAS H. CORMEN
CHARLES E. LEISERSON
RONALD L. RIVEST
CLIFFORD STEIN
INTRODUCTION TO
ALGORITHMS
THIRD EDITION

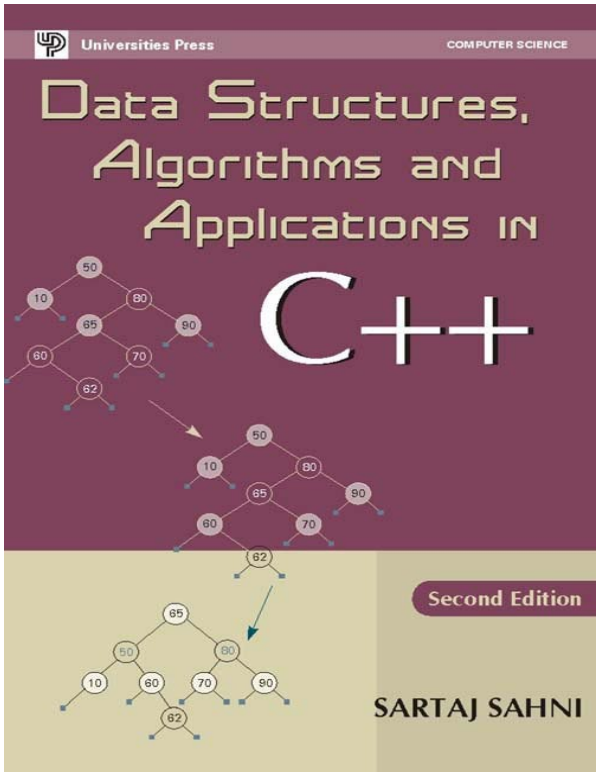

ALGORITHM
DESIGN
Foundations, Analysis,
and Internet Examples


Introduction to The Design &
Analysis of Algorithms
2ND EDITION
Anany Levitin

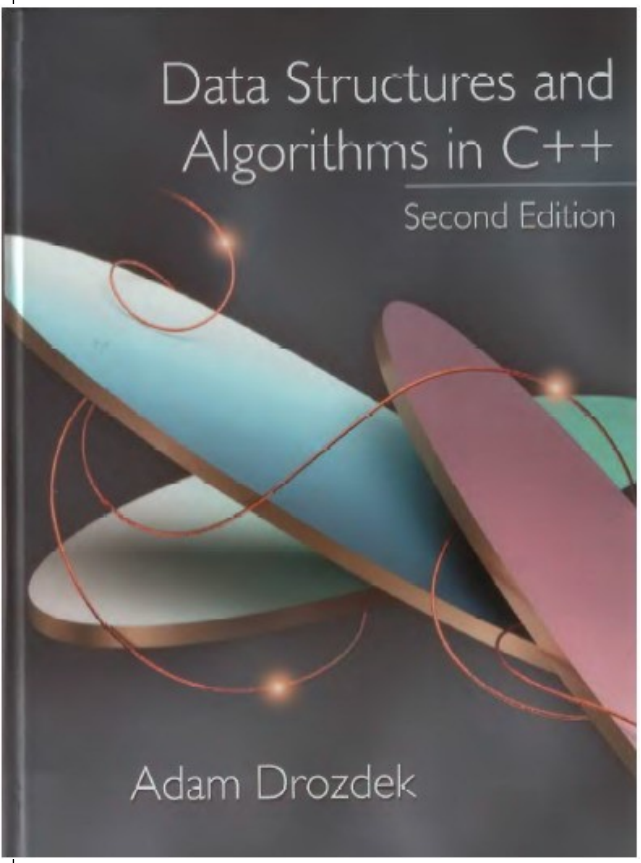
Reference:

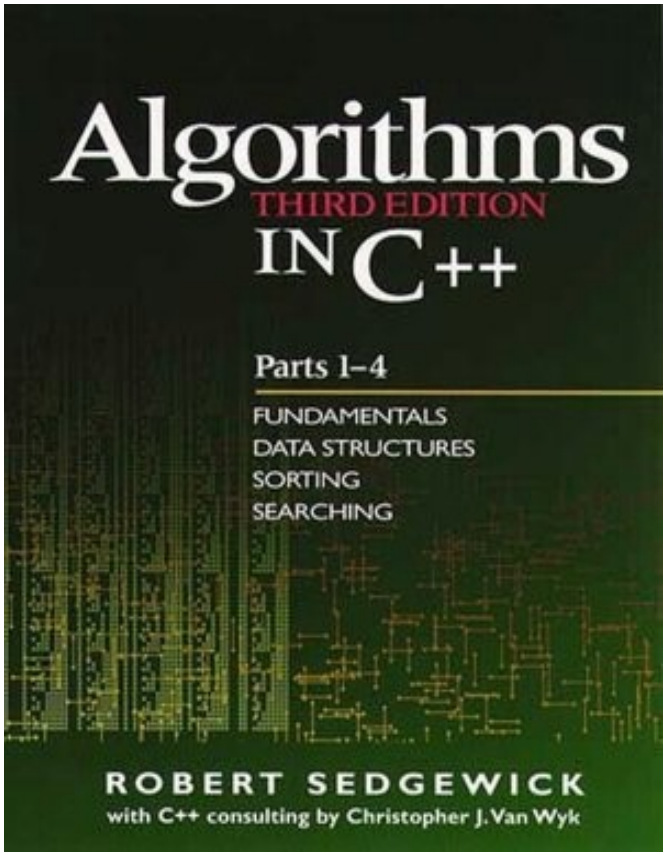

Data Structures, Algorithms, and Applications in C++
SAHNI

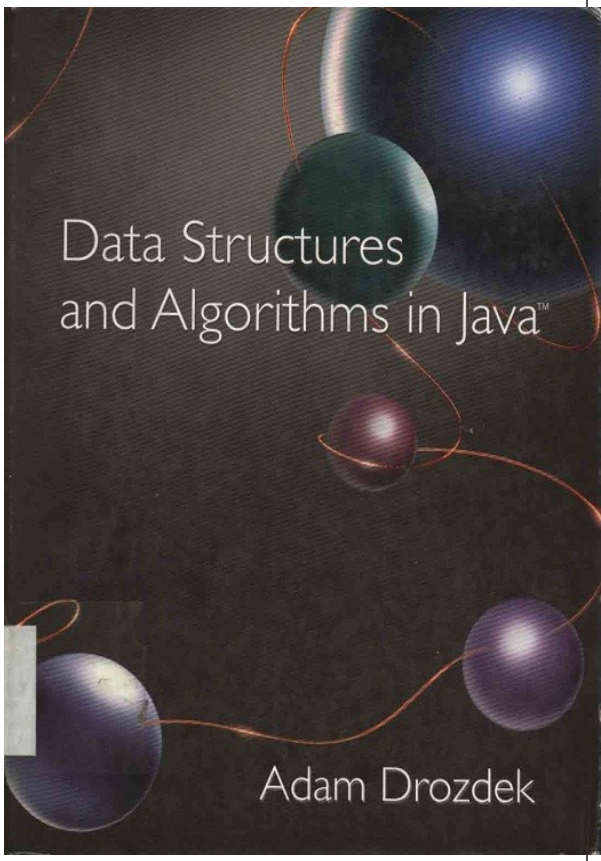

Universities Press
COMPUTER SCIENCE
Fundamentals of Computer
Algorithms
SECOND EDITION
Ellis Horowitz
Sartaj Sahni
Sanguthevar Rajasekaran


Universities Press
COMPUTER SCIENCE
Data Structures, Algorithms and Applications in C++
Second Edition
SARTAJ SAHNI

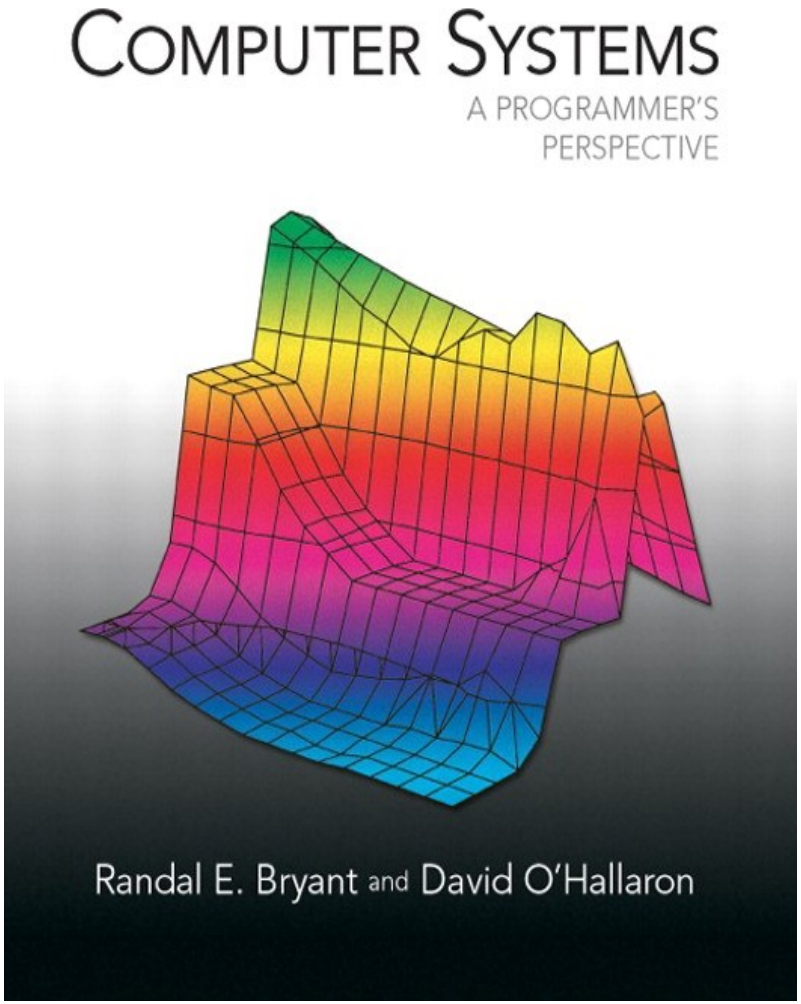
Reference:

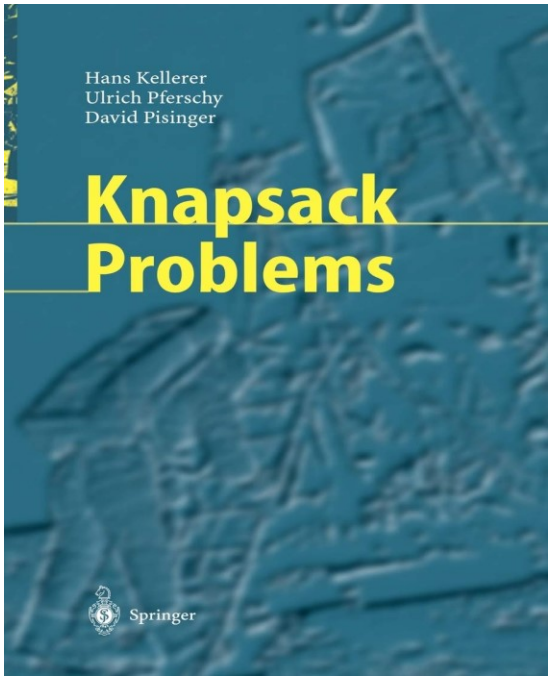

Data Structures and Algorithms in C++
Second Edition
Adam Drozdek

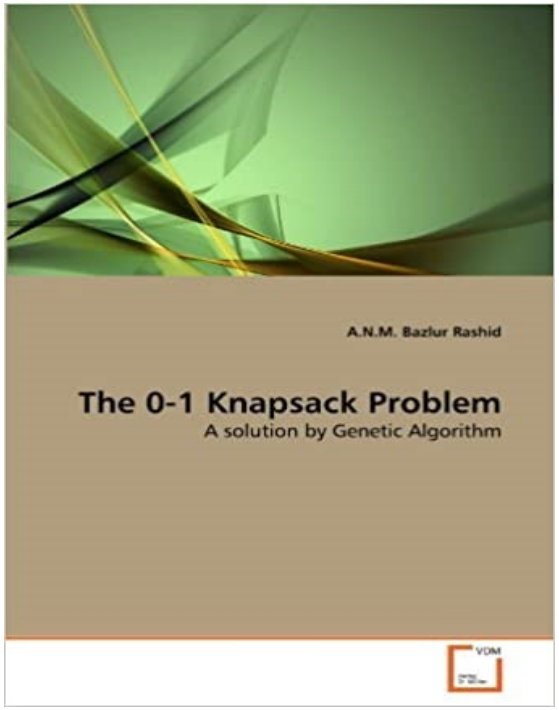

Algorithms
THIRD EDITION
IN C++
Parts 1–4
FUNDAMENTALS
DATA STRUCTURES
SORTING
SEARCHING
ROBERT SEDGEWICK
with C++ consulting by Christopher J. Van Wyk

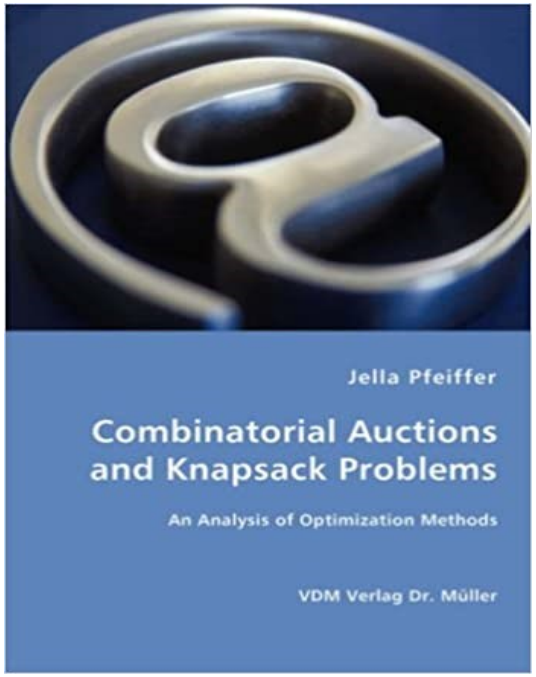

Data Structures and Algorithms in Java™
Adam Drozdek

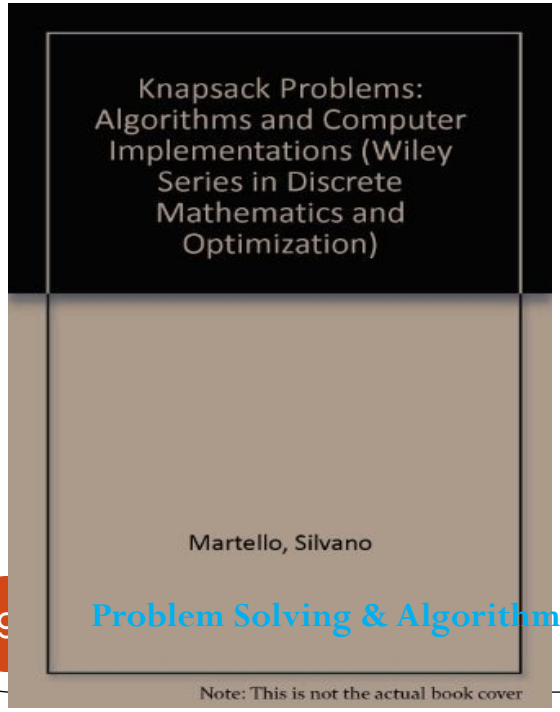
Reference

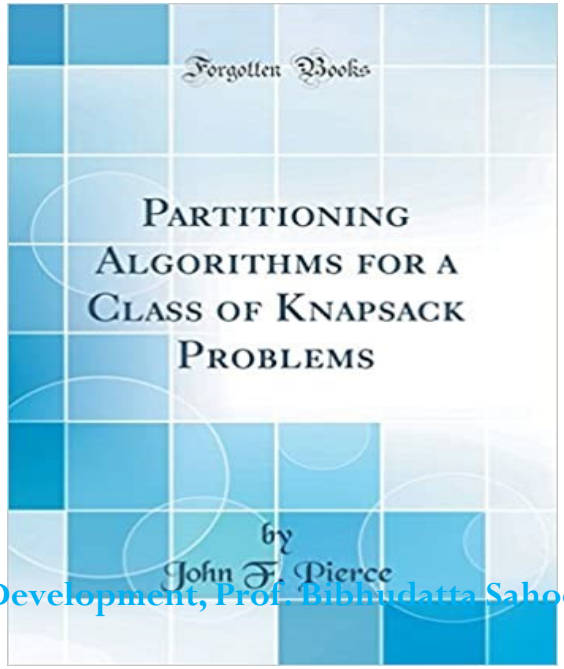

COMPUTER SYSTEMS
A PROGRAMMER'S
PERSPECTIVE
Randal E. Bryant and David O'Hallaron

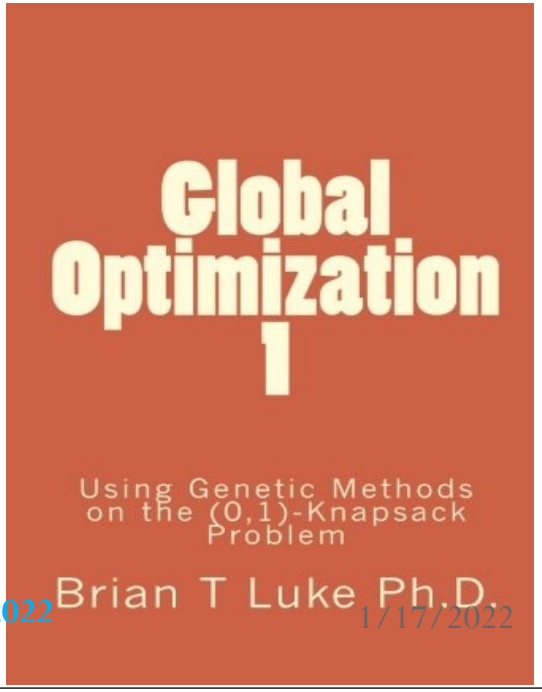

Hans Kellerer
Ulrich Pferschy
David Pisinger
Knapsack Problems
Springer


A.N.M. Bazlur Rashid
The 0-1 Knapsack Problem
A solution by Genetic Algorithm
VDM


Jella Pfeiffer
Combinatorial Auctions and Knapsack Problems
An Analysis of Optimization Methods
VDM Verlag Dr. Müller


Knapsack Problems: Algorithms and Computer Implementations (Wiley Series in Discrete Mathematics and Optimization)
Martello, Silvano
Note: This is not the actual book cover


Forgotten Books
Partitioning Algorithms for a Class of Knapsack Problems
by
John F. Pierce


Global Optimization 1
Using Genetic Methods on the (0,1)-Knapsack Problem
Brian T Luke Ph.D.