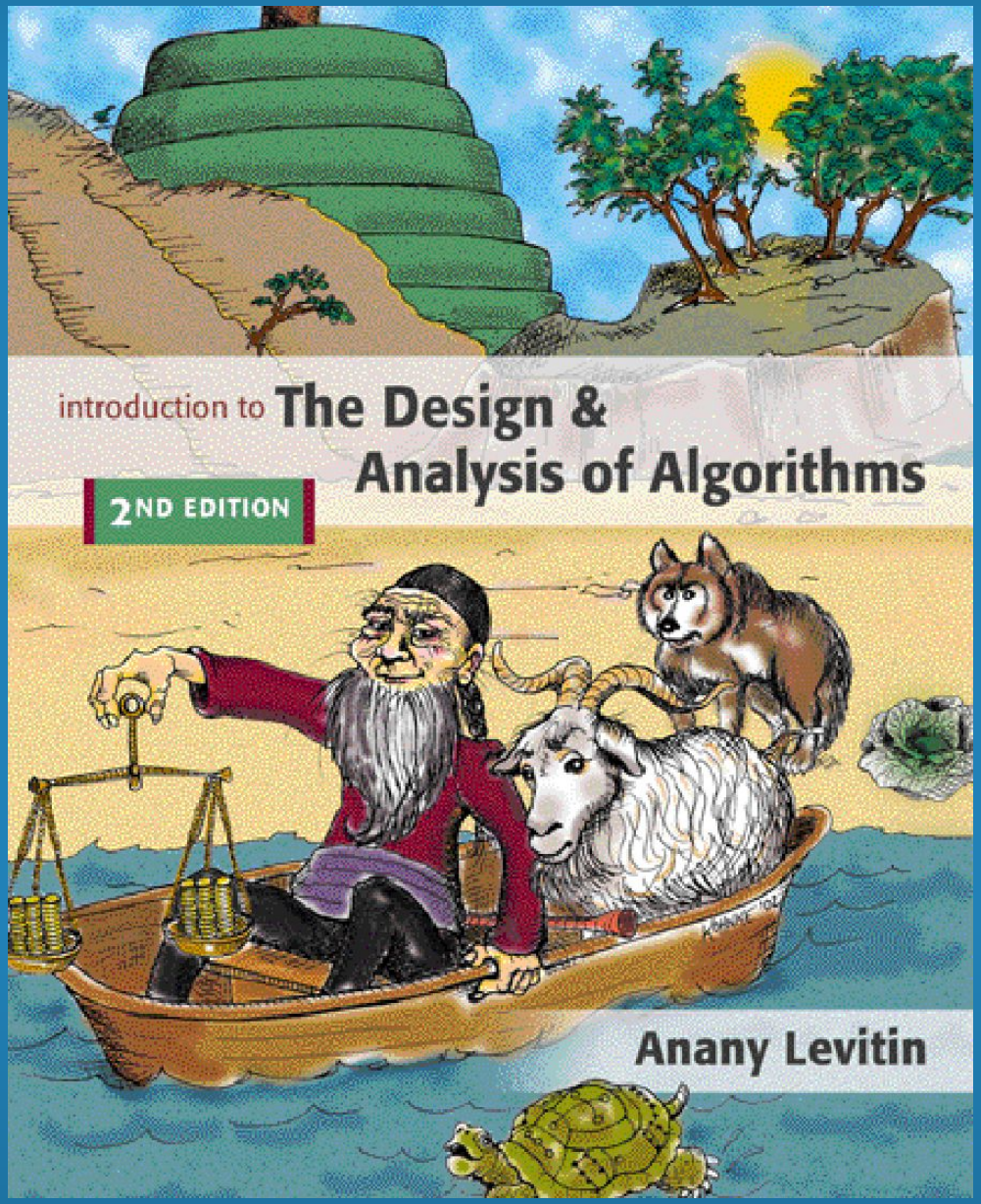
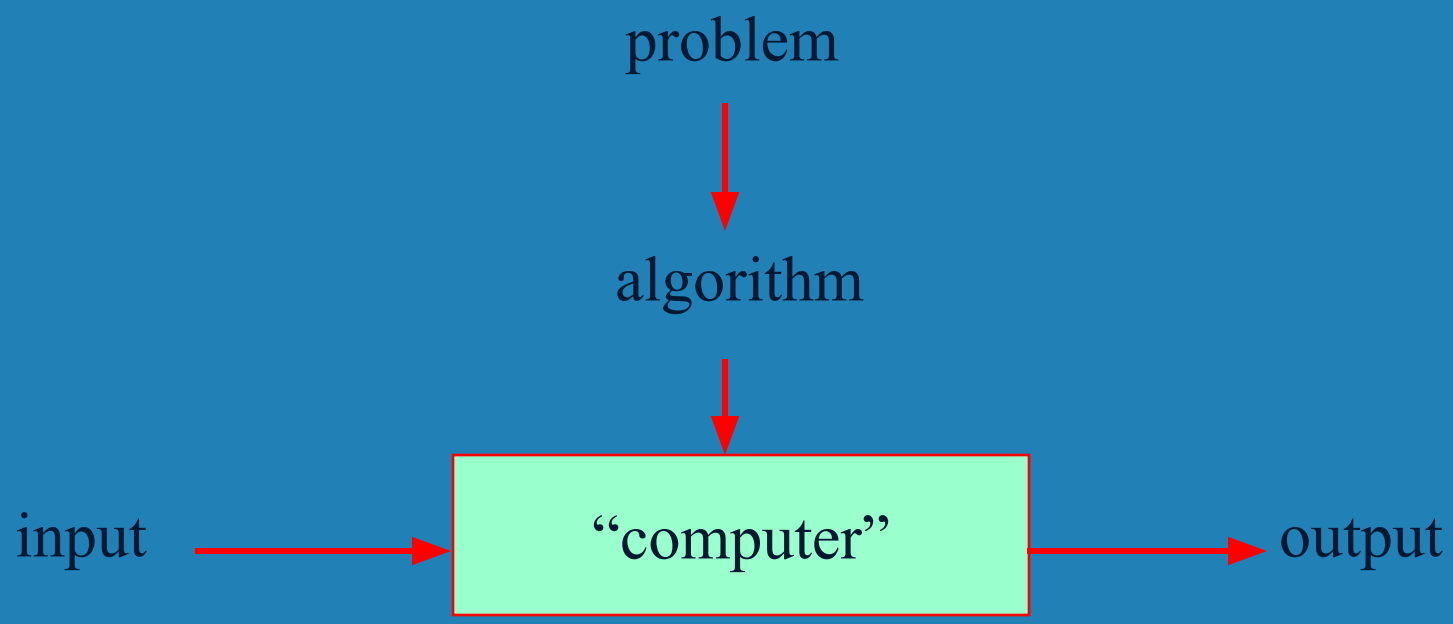
Chapter 1

Introduction



What is an algorithm?

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.



Algorithm

- **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.**

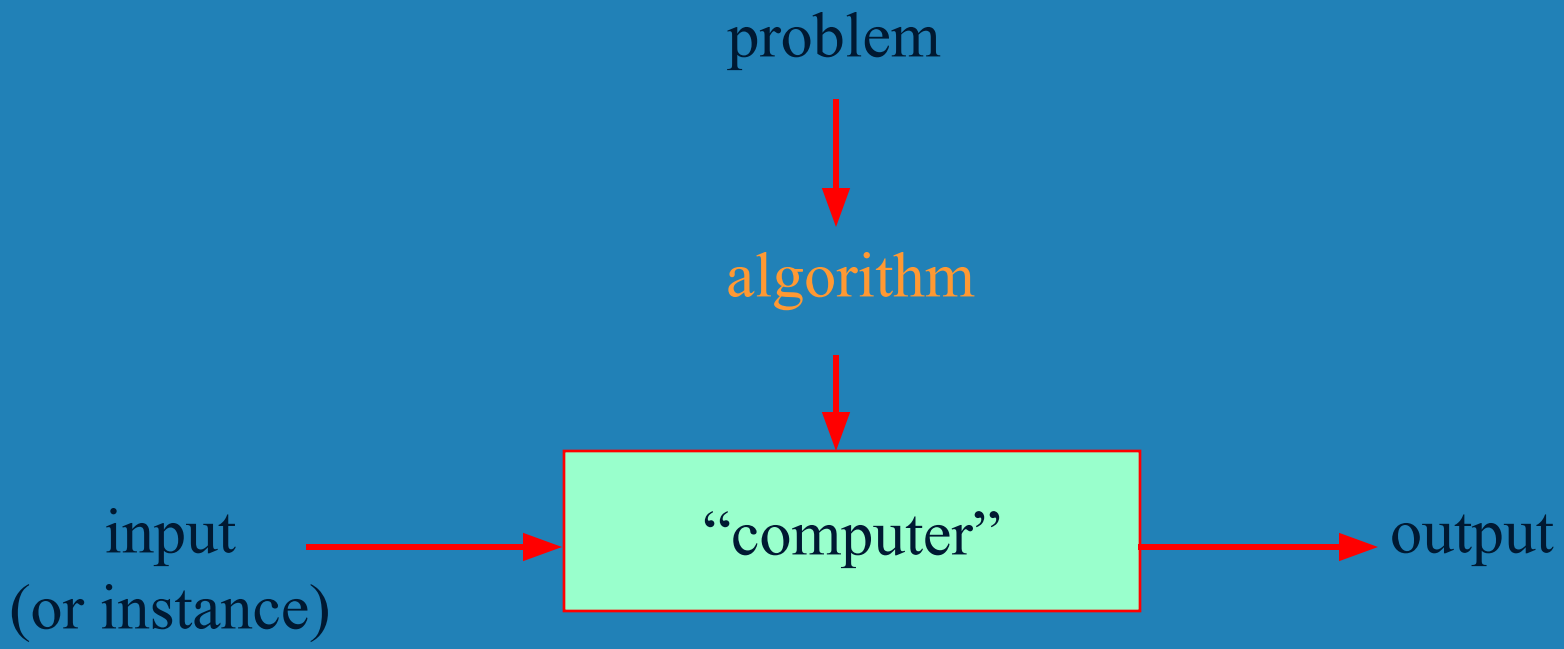
  - Can be represented various forms
  - Unambiguity/clearness
  - Effectiveness
  - Finiteness/termination
  - Correctness

Historical Perspective

- **Euclid's algorithm for finding the greatest common divisor**

- **Muhammad ibn Musa al-Khwarizmi – 9th century mathematician**
  www.lib.virginia.edu/science/parshall/khwariz.html

Notion of algorithm and problem



algorithmic solution
(different from a conventional solution)

Selection Sort

- **Input: array a[1], …, a[n]**
- **Output: array a sorted in non-decreasing order**
- **Algorithm:**

for i=1 to n
swap a[i] with smallest of a[i], …, a[n]

- Is this unambiguous? Effective?
- See also pseudocode, section 3.1

Selection Sort

Algorithm ***SelectionSort(A[0..n-1])***

//The algorithm sorts a given array by selection sort

//Input: An array A[0..n-1] of orderable elements

//Output: Array A[0..n-1] sorted in ascending order

```
for i ← 0 to n – 2 do
        min ← i
        for j ← i + 1 to n – 1 do
                if A[j] < A[min]
                        min ← j
        swap A[i] and A[min]
```

Example of computational problem: sorting

- **Statement of problem:**
  - ***Input:*** **A sequence of *n* numbers $\langle a_1, a_2, \ldots, a_n \rangle$**
  - ***Output:*** **A reordering of the input sequence $\langle a'_1, a'_2, \ldots, a'_n \rangle$ so that $a'_i \leq a'_j$ whenever $i < j$**
- **Instance: The sequence <5, 3, 2, 8, 3>**
- **Algorithms:**
  - **Selection sort**
  - **Insertion sort**
  - **Merge sort**
  - **(many others)**

Basic Issues Related to Algorithms

- How to design algorithms
- How to express algorithms
- Proving correctness
- Efficiency (or complexity) analysis
  - Theoretical analysis
  - Empirical analysis
- Optimality

Algorithm design strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Greedy approach
- Dynamic programming
- Backtracking and branch-and-bound
- Space and time tradeoffs

Analysis of Algorithms

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

What is an algorithm?

- Recipe, process, method, technique, procedure, routine,… with the following requirements:

1. Finiteness
   - terminates after a finite number of steps
2. Definiteness
   - rigorously and unambiguously specified
3. Clearly specified input
   - valid inputs are clearly specified
4. Clearly specified/expected output
   - can be proved to produce the correct output given a valid input
5. Effectiveness
   - steps are sufficiently simple and basic

Why study algorithms?

- **Theoretical importance**
  - **the core of computer science**
- **Practical importance**
  - **A practitioner's toolkit of known algorithms**
  - **Framework for designing and analyzing algorithms for new problems**

Example: Google's PageRank Technology

Euclid's Algorithm

Problem: Find gcd(m,n), the greatest common divisor of two nonnegative, not both zero integers m and n

Examples: gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

Euclid's algorithm is based on repeated application of equality

$$\gcd(m,n) = \gcd(n, m \bmod n)$$

until the second number becomes 0, which makes the problem trivial.

Example: gcd(60,24) = gcd(24,12) = gcd(12,0) = 12

Two descriptions of Euclid's algorithm

Step 1 If $n = 0$, return m and stop; otherwise go to Step 2

Step 2 Divide m by n and assign the value of the remainder to r

Step 3 Assign the value of n to m and the value of r to n. Go to Step 1.

while $n \neq 0$ **do**

$r \leftarrow m \bmod n$

$m \leftarrow n$

$n \leftarrow r$

return m

Other methods for computing gcd(*m*,*n*)

Consecutive integer checking algorithm

Step 1 **Assign the value of min{*m*,*n*} to *t***

Step 2 **Divide *m* by *t*. If the remainder is 0, go to Step 3; otherwise, go to Step 4**

Step 3 **Divide *n* by *t*. If the remainder is 0, return *t* and stop; otherwise, go to Step 4**

Step 4 **Decrease *t* by 1 and go to Step 2**

Is this slower than Euclid's algorithm?
How much slower?

O(n), if n <= m , vs O(log n)

Other methods for gcd(*m*,*n*) [cont.]

Middle-school procedure

Step 1 Find the prime factorization of *m*

Step 2 Find the prime factorization of *n*

Step 3 Find all the common prime factors

Step 4 Compute the product of all the common prime factors and return it as gcd(*m*,*n*)

Is this an algorithm?

How efficient is it?

Time complexity: O(sqrt(n))

Sieve of Eratosthenes

Input: Integer $n \geq 2$

Output: List of primes less than or equal to n

for $p \leftarrow 2$ **to** n **do** $A[p] \leftarrow p$

for $p \leftarrow 2$ **to** n **do**

  if $A[p] \neq 0$ //p hasn't been previously eliminated from the list

    $j \leftarrow p * p$

    while $j \leq n$ **do**

      $A[j] \leftarrow 0$ //mark element as eliminated

      $j \leftarrow j + p$

Example: 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

Time complexity: O(n)

Two main issues related to algorithms

- How to design algorithms
- How to analyze algorithm efficiency

Analysis of algorithms

- How good is the algorithm?
  - time efficiency
  - space efficiency
  - correctness ignored in this course
- Does there exist a better algorithm?
  - lower bounds
  - optimality

Some Well-known Computational Problems

- Sorting
- Searching
- Shortest paths in a graph
- Minimum spanning tree
- Primality testing
- Traveling salesman problem
- Knapsack problem
- Chess
- Towers of Hanoi
- Program termination

Some of these problems don't have efficient algorithms, or algorithms at all!

Algorithm Design and analysis process

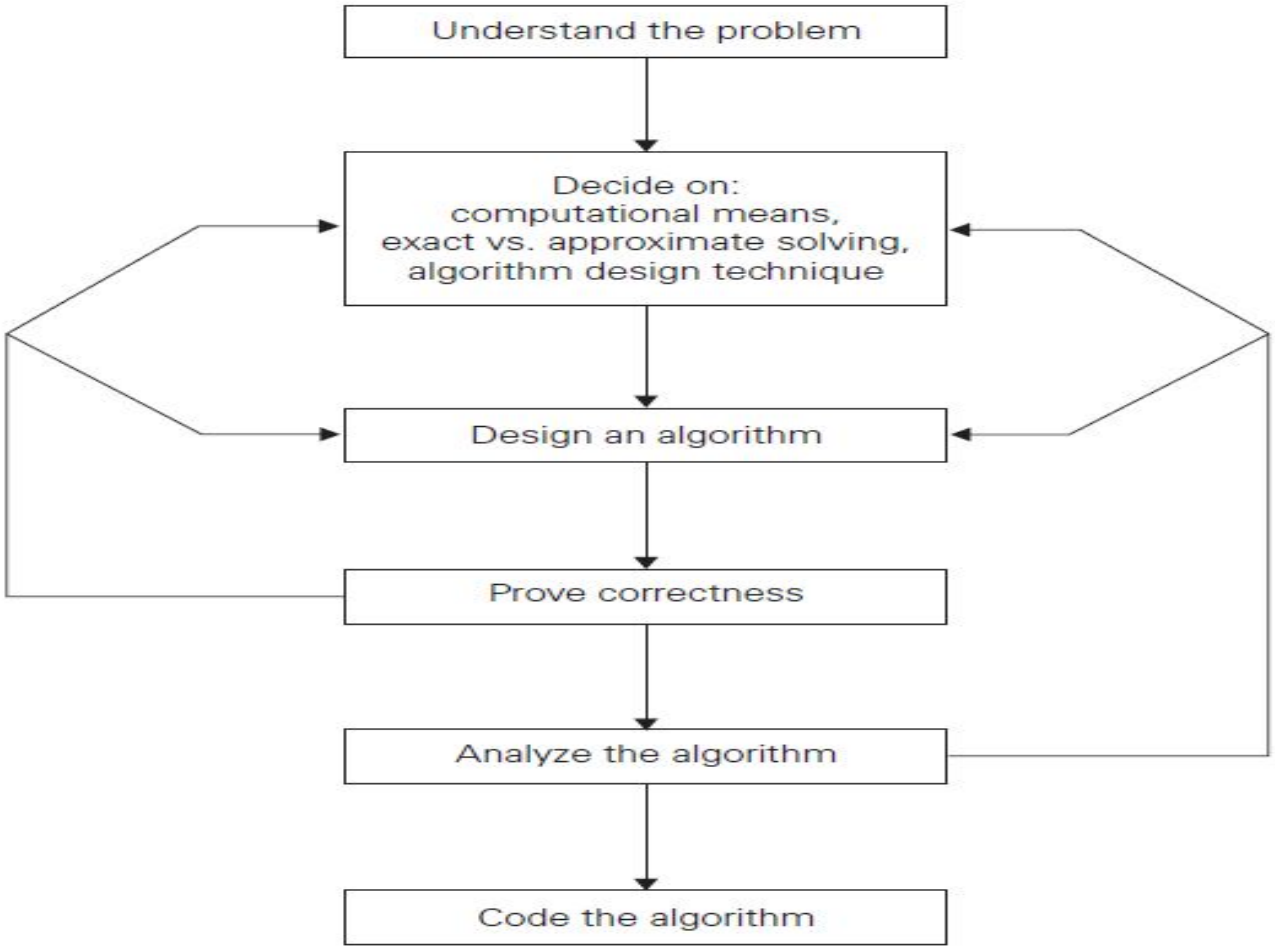


FIGURE 1.2 Algorithm design and analysis process.

Algorithm Design and analysis process

(i) Understanding the Problem

- This is the first step in designing of algorithm.
- Read the problem's description carefully to understand the problem statement completely.
- Ask questions for clarifying the doubts about the problem.
- Identify the problem types and use existing algorithm to find solution.
- Input (instance) to the problem and range of the input get fixed.

Algorithm Design and analysis process

(ii) Decision making The Decision making is done on the following:

(a) Ascertaining the Capabilities of the Computational Device In random-access machine (RAM), instructions are executed one after another (The central assumption is that one operation at a time).

- Accordingly, algorithms designed to be executed on such machines are called sequential algorithms.
- In some newer computers, operations are executed concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called parallel algorithms.
- Choice of computational devices like Processor and memory is mainly based on space and time efficiency

Algorithm Design and analysis process

(b) Choosing between Exact and Approximate Problem Solving

- **The next principal decision is to choose between solving the problem exactly or solving it approximately.**
- **An algorithm used to solve the problem exactly and produce correct result is called an exact algorithm.**
- **If the problem is so complex and not able to get exact solution, then we have to choose an algorithm called an approximation algorithm.**
- **E.g., extracting square roots, solving nonlinear equations, and evaluating definite integrals.**

Algorithm Design and analysis process

(c) Algorithm Design Techniques

- An algorithm design technique (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.
- Algorithms + Data Structures = Programs
- Though Algorithms and Data Structures are independent, but they are combined together to develop program. Hence the choice of proper data structure is required before designing the algorithm.
- Implementation of algorithm is possible only with the help of Algorithms and Data Structures
- Algorithmic strategy / technique / paradigm are a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and so on.

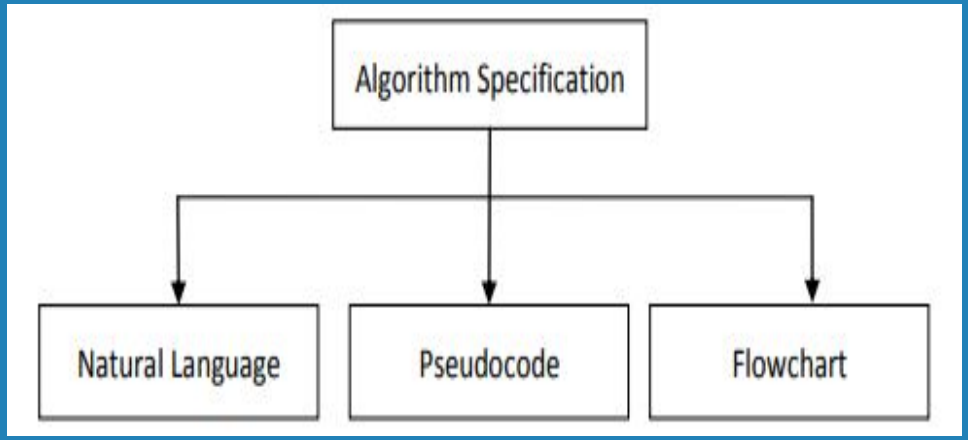
Algorithm Design and analysis process

- (iii) Methods of Specifying an Algorithm
- There are three ways to specify an algorithm. They are:

  a. Natural language

  b. Pseudocode

  c. Flowchart



- Pseudocode and flowchart are the two options that are most widely used nowadays for specifying algorithms.

a. Natural Language

- It is very simple and easy to specify an algorithm using natural language. But many times

Algorithm Design and analysis process

- specification of algorithm by using natural language is not clear and thereby we get brief specification.
- Example: An algorithm to perform addition of two numbers.

  Step 1: Read the first number, say a.

  Step 2: Read the first number, say b.

  Step 3: Add the above two numbers and store the result in c.

  Step 4: Display the result from c.
- Such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of Pseudocode.

Algorithm Design and analysis process

b. Pseudocode

- **Pseudocode is a mixture of a natural language and programming language constructs.**
- **Pseudocode is usually more precise than natural language.**
- **For Assignment operation left arrow "←", for comments two slashes "//",if condition, for, while loops are used.**

ALGORITHM Sum(a,b)

//Problem Description: This algorithm performs addition of two numbers

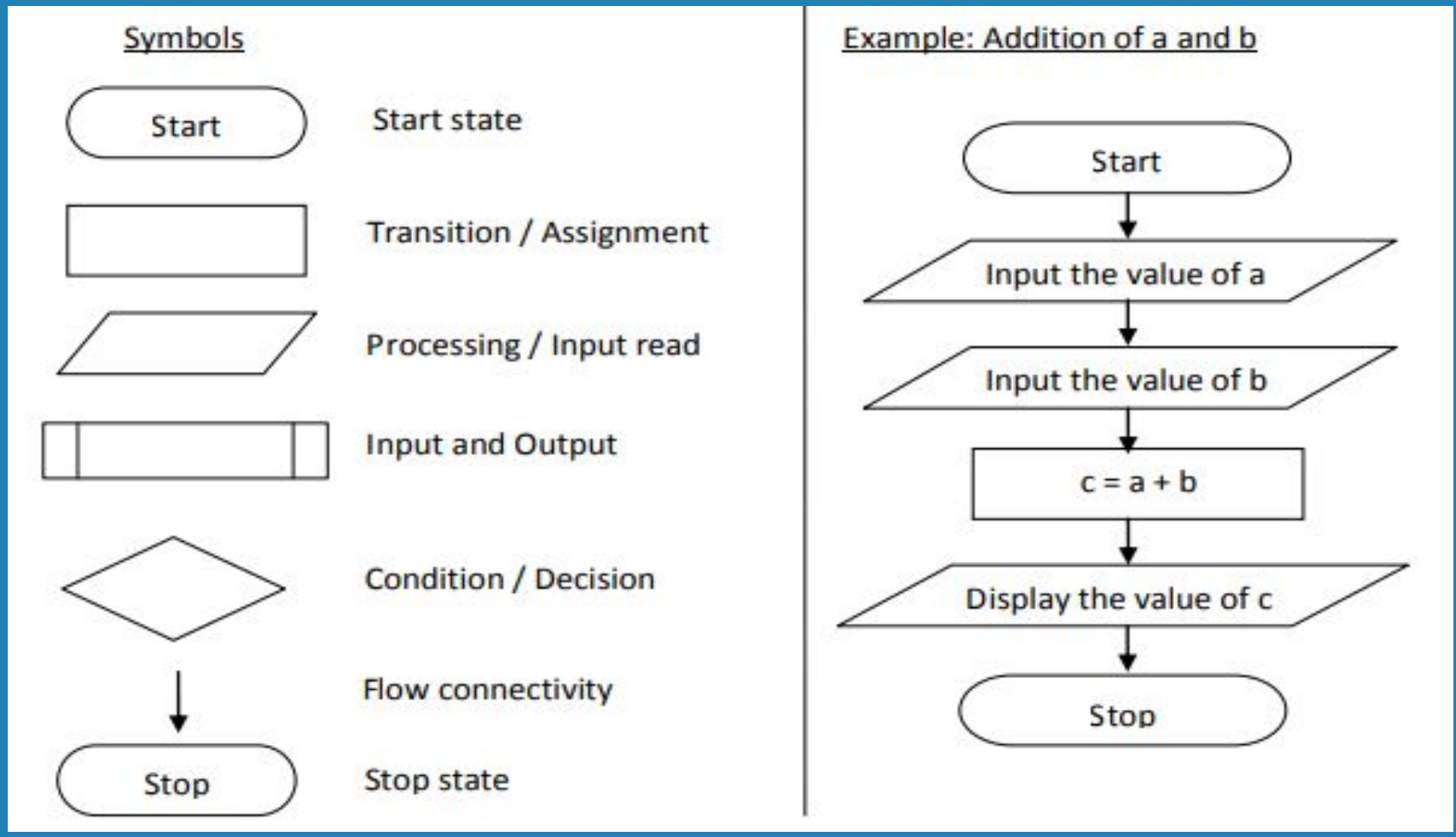
- **//Input: Two integers a and b**
- **//Output: Addition of two integers**
- **c←a+b**
- **return c**
- **This specification is more useful for implementation of any language.**

Algorithm Design and analysis process

c. Flowchart

- In the earlier days of computing, the dominant method for specifying algorithms was a flowchart, this representation technique has proved to be inconvenient.
- Flowchart is a graphical representation of an algorithm. It is a a method of expressing an algorithm by a collection of connected geometric shapes containing descriptions of the algorithm's steps.

Algorithm Design and analysis process



Algorithm Design and analysis process

(iv) Proving an Algorithm's Correctness

- Once an algorithm has been specified then its correctness must be proved.
- An algorithm must yields a required result for every legitimate input in a finite amount of time.
- For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd(m, n) = gcd(n, m mod n).
- A common technique for proving correctness is to use mathematical induction because an algorithm's iterations provide a natural sequence of steps needed for such proofs.
- The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms.
- The error produced by the algorithm should not exceed a predefined limit.

Algorithm Design and analysis process

(v) Analyzing an Algorithm

- For an algorithm the most important is efficiency. In fact, there are two kinds of algorithm efficiency. They are:
  - Time efficiency, indicating how fast the algorithm runs, and
  - Space efficiency, indicating how much extra memory it uses.
  - The efficiency of an algorithm is determined by measuring both time efficiency and space efficiency.
  - So factors to analyze an algorithm are:
    - Time efficiency of an algorithm
    - Space efficiency of an algorithm
    - Simplicity of an algorithm
    - Generality of an algorithm

Algorithm Design and analysis process

- (vi) Coding an Algorithm
- The coding / implementation of an algorithm is done by a suitable programming language like C, C++, JAVA.
- The transition from an algorithm to a program can be done either incorrectly or very inefficiently. Implementing an algorithm correctly is necessary. The Algorithm power should not reduced by inefficient implementation.
- Standard tricks like computing a loop's invariant (an expression that does not change its value) outside the loop, collecting common subexpressions, replacing expensive operations by cheap ones, selection of programming language and so on should be known to the programmer.
- Typically, such improvements can speed up a program only by a constant factor, whereas a better algorithm can make a difference in running time by orders of magnitude. But once an algorithm is selected, a 10–50% speedup may be worth an effort.
- It is very essential to write an optimized code (efficient code) to reduce the burden of compiler.

Algorithm design techniques/strategies

- Brute force
- Divide and conquer
- Decrease and conquer
- Transform and conquer
- Space and time tradeoffs
- Greedy approach
- Dynamic programming
- Iterative improvement
- Backtracking
- Branch and bound

Important problem types

- sorting
- searching
- string processing
- graph problems
- combinatorial problems
- geometric problems
- numerical problems

Sorting (I)

- Rearrange the items of a given list in ascending order.
  - Input: A sequence of n numbers $\langle a_1, a_2, \ldots, a_n \rangle$
  - Output: A reordering $\langle a'_1, a'_2, \ldots, a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq \ldots \leq a'_n$.
- Why sorting?
  - Help searching
  - Algorithms often use sorting as a key subroutine.
- Sorting key
  - A specially chosen piece of information used to guide sorting. E.g., sort student records by names.

Sorting (II)

- Examples of sorting algorithms
  - Selection sort
  - Bubble sort
  - Insertion sort
  - Merge sort
  - Heap sort …
- Evaluate sorting algorithm complexity: the number of key comparisons.
- Two properties
  - Stability: A sorting algorithm is called stable if it preserves the relative order of any two equal elements in its input.
  - In place : A sorting algorithm is in place if it does not require extra memory, except, possibly for a few memory units.

Selection Sort

Algorithm ***SelectionSort(A[0..n-1])***

//The algorithm sorts a given array by selection sort

//Input: An array A[0..n-1] of orderable elements

//Output: Array A[0..n-1] sorted in ascending order

```
for i ← 0 to n – 2 do
    min ← i
    for j ← i + 1 to n – 1 do
        if A[j] < A[min]
            min ← j
    swap A[i] and A[min]
```

Searching

- **Find a given value, called a search key, in a given set.**
- **Examples of searching algorithms**
  - **Sequential search**
  - **Binary search …**

Input: sorted array a_i < … < a_j and key x;
m □(i+j)/2;
while i < j and x != a_m do
    if x < a_m then j □ m-1
                else i □ m+1;
 if x = a_m then output a_m;

Time: O(log n)

String Processing

- **A string is a sequence of characters from an alphabet.**
- **Text strings: letters, numbers, and special characters.**
- **String matching: searching for a given word/pattern in a text.**

Examples:

(i) searching for a word or phrase on WWW or in a Word document

(ii) searching for a short read in the reference genomic sequence

Graph Problems

- Informal definition
  - A graph is a collection of points called vertices, some of which are connected by line segments called edges.
- Modeling real-life problems
  - Modeling WWW
  - Communication networks
  - Project scheduling …
- Examples of graph algorithms
  - Graph traversal algorithms
  - Shortest-path algorithms
  - Topological sorting

Combinatorial problems

- These are problems that ask, explicitly or implicitly, to find a combinatorial object such as a permutation, a combination, or a subset that satisfies certain constraints.
- A desired combinatorial object may also be required to have some additional property such as a maximum value or a minimum cost.
- In practical, the combinatorial problems are the most difficult problems in computing.
- The traveling salesman problem and the graph coloring problem are examples of combinatorial problems.

Geometric problems

- Geometric algorithms deal with geometric objects such as points, lines, and polygons.
- Geometric algorithms are used in computer graphics, robotics, and tomography.
- The closest-pair problem and the convex-hull problem are comes under this category.

Numerical problems

- Numerical problems are problems that involve mathematical equations, systems of equations, computing definite integrals, evaluating functions, and so on.
- The majority of such mathematical problems can be solved only approximately.