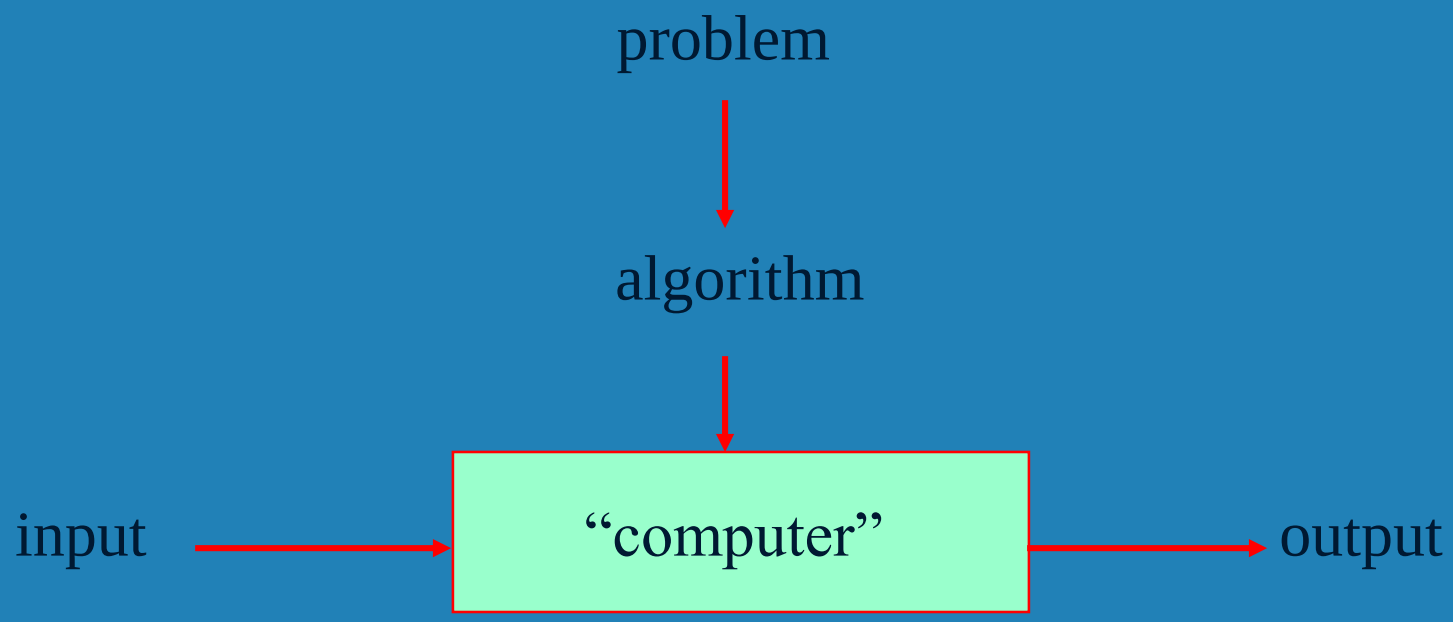
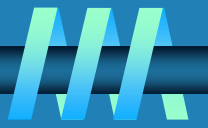
What is an algorithm?

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.



What is an algorithm?



Unambiguous : Well defined

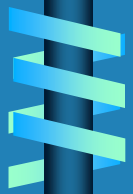
Clear, precise, having or exhibiting a single clearly defined meaning.

Instructions: procedure

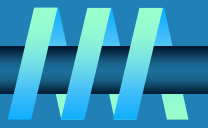
Series of actions conducted in certain manner

Another definition:

A process or set of rules to be followed in calculations or other problem-solving operations, especially by a computer.

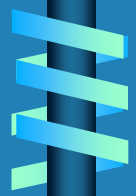


Reasons to Study Algorithm

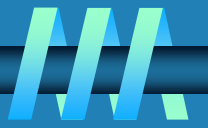


Theoretical:

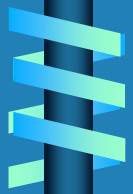
- **Core of computer science**
- **Relevant to most of science business and technology**
- **Computer programs would not exists without algorithm**
- **Useful in developing analytical skills**
- **Design techniques can be interpreted as problem solving strategies.**



Requirements of an algorithm.



1. ***Finiteness:*** **terminates after a finite number of steps**
2. ***Definiteness:*** **logically and unambiguously specified**
3. ***Input:*** **Valid inputs are clearly specified**
4. ***Output:*** **Can be proved to produce the correct output given a valid input**
5. ***Effectiveness:*** **steps are sufficiently simple and basic**



Important points for algorithms:

- The non-ambiguity requirements for each step of an algorithm cannot be compromised .
- The range of inputs for which an algorithm Works has to be specified carefully.
- The same algorithm can be represented in several different ways.
- There may exist several algorithms for solving the same problem.
- Algorithms for the same problem can be based on very different ideas and can solve the problem with dramatically different speeds.

Algorithm Design & Analysis Process

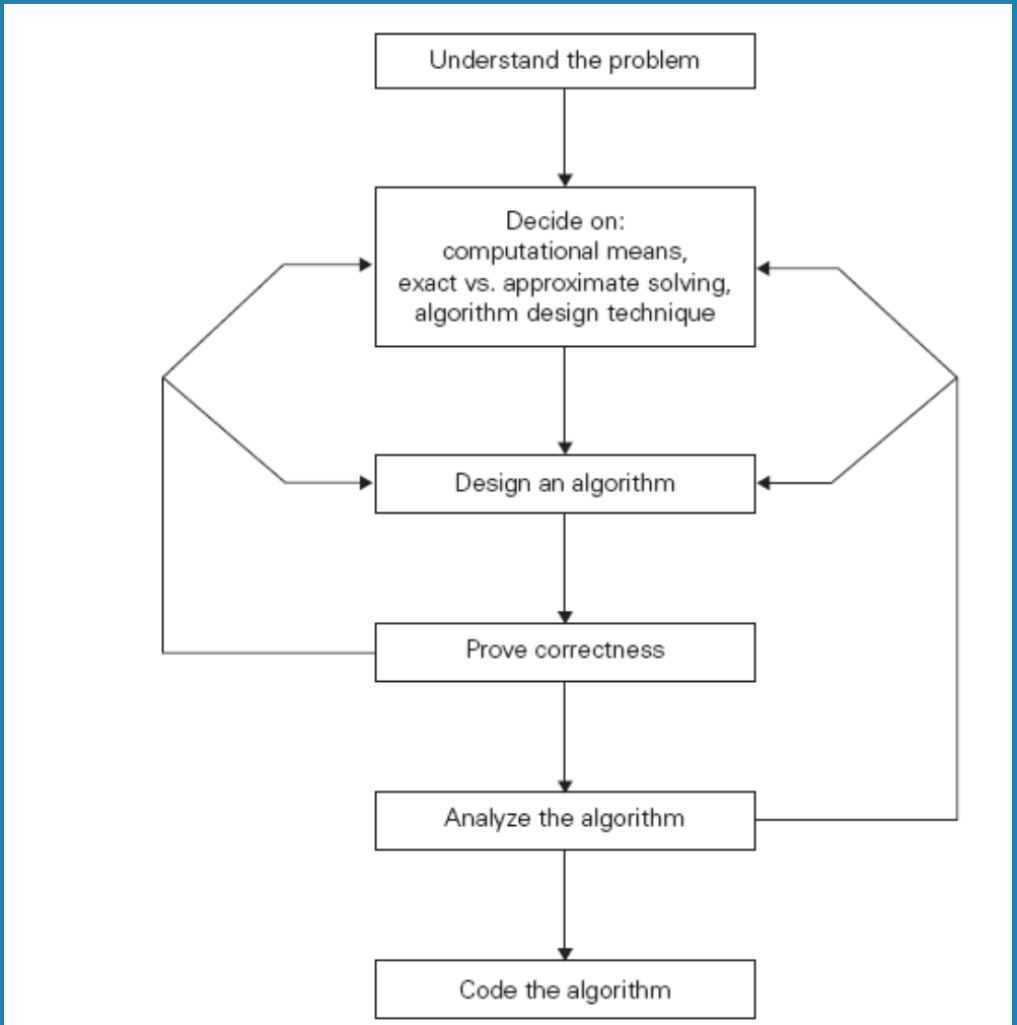


FIGURE 1.2 Algorithm design and analysis process.

Algorithms (Sequential/ Parallel)

Cornerstones:

- The von Neumann machine—a computer architecture outlined by the prominent Hungarian-American mathematician **John von Neumann** (1903– 1957), in collaboration with **A. Burks and H. Goldstone**, in 1946. The essence of this architecture is captured by the so-called ***random-access machine*** (***RAM***).

Algorithms designed to be executed on such machines are called ***sequential algorithms***.

- The central assumption of the RAM model does not hold for some newer computers that can execute operations concurrently, i.e., in parallel. Algorithms that take advantage of this capability are called ***parallel algorithms***.

Studying the classic techniques for design and analysis of algorithms under the RAM model remains the cornerstone of algorithmics for the foreseeable future.

Exact and Approximate Problem Solving

The next principal decision is to choose between solving the problem exactly or solving it approximately. In the former case, an algorithm is called an ***exact algorithm***; in the latter case, an algorithm is called an ***approximation algorithm***.

Why would one opt for an approximation algorithm?

First, there are important problems that simply cannot be solved exactly for most of their instances; examples include extracting square roots, solving nonlinear equations, and evaluating definite integrals.

Second, available algorithms for solving a problem exactly can be unacceptably slow because of the problem's intrinsic complexity. This happens, in particular, for many problems involving a very large number of choices.

Third, an approximation algorithm can be a part of a more sophisticated algorithm that solves a problem exactly.

Proving an Algorithm's Correctness

Correctness: Prove that the algorithm 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

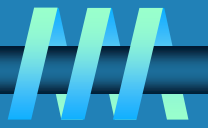
$\gcd(m, n) = \gcd(n, m \bmod n)$

The simple observation that the second integer gets smaller on every iteration of the algorithm, and the fact that the algorithm stops when the second integer becomes 0.

For some algorithms, a proof of correctness is quite easy; for others, it can be quite complex.

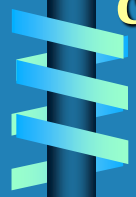
In **order to show that an algorithm is incorrect, you need just one instance of its input for which the algorithm fails.**

Analyzing an Algorithm



We usually want our algorithms to possess several qualities. After correctness, by far the most important is ***efficiency***. In fact, there are two kinds of algorithm efficiency: ***time efficiency***, indicating how fast the algorithm runs, and ***space efficiency***, indicating how much extra memory it uses.

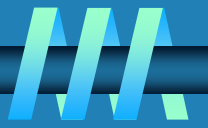
Another desirable characteristic of an algorithm is ***simplicity***. Unlike efficiency, which can be precisely defined and investigated with mathematical rigor, simplicity, like beauty, is to a considerable degree in the eye of the beholder. For example, most people would agree that **Euclid's algorithm is simpler than the middle-school procedure for computing gcd*(m, n)*, but it is not clear whether Euclid's algorithm is simpler than the consecutive integer checking algorithm**.



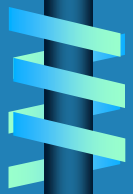
Algorithm design techniques/strategies

- 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

Analysis of algorithms efficiency



- How good is the algorithm?
  - time efficiency
  - space efficiency
- Does there exist a better algorithm?
  - lower bounds
  - optimality

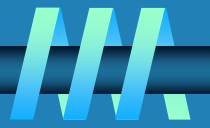


Important problem types

- **Sorting :** Rearrange the items of a given list in increasing order. A sorting algorithm is called stable if it preserves the relative order of any two equal elements in it input. In other words, if an input list contains two equal elements in position i and j where i < j, then in the sorted list they have to be in the position i' and j'. in-place if it does not require extra memory, except, possibly, for a few memory units.

- **Stable Sorting Algorithms:**
  - Insertion Sort
  - Merge Sort
  - Bubble Sort
  - Tim Sort
  - Counting Sort
- **Unstable Sorting Algorithms:**
  - Heap Sort
  - Selection sort
  - Shell sort
  - Quick Sort

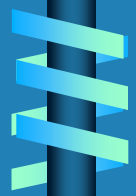
Important problem types



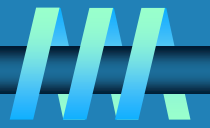
- **Searching :** Find a given value in a given set

- **string processing:** Deal with non-numerical data, strings that is a sequence of characters from an alphabet. Example: searching a given word in a text (*string matching*)

- **graph problems:** Graphs are used for modeling a wide variety of applications including transportations, communication, social and economic networks, project scheduling, and games. The most well known examples are the travelling salesman problem (finding shortest path tour through n cities that visits every city exactly once) and graph-coloring problem (assign the smallest numbers of colors to the vertices of a graph so that no two adjacent vertices are the same color).



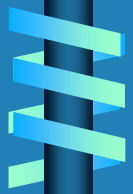
Important problem types



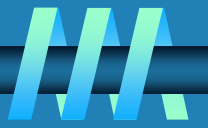
- **combinatorial problems:** These are problems that ask, explicitly or implicitly to find combinatorial objects such as a permutation, a combination or a subset that satisfies certain constraints.

- **geometric problems:** Deal with geometric objects such as points, lines, and polygons.

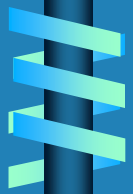
- **numerical problems:** Involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions and so on.



Fundamental data structures



- list
  - array
  - linked list
  - string
- stack
- queue
- priority queue
- graph
- tree
- set and dictionary



Fundamental data structures

- **Array: A one dimensional array is a sequence of n items of the same data type (e.g., integers or characters) that are stored contiguously in the computer memory and made accessible by specifying a value of the array index**

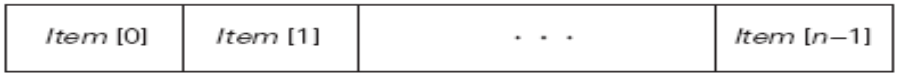


FIGURE 1.3 Array of n elements.

- **Linked List: A Linked List is a sequence of zero or more elements called nodes, each containing two kinds of information: some data and one or more links called pointers to other nodes of the linked list. A special pointer null is used to indicate the absence of a node successor. In a singly linked list, each node except the last one contains a single pointer to the next element**

FIGURE 1.4 Singly linked list of n elements.

Fundamental data structures

- **Doubly Linked List: every node except the first and last pointers contains pointers to both its successor and its predecessor.**

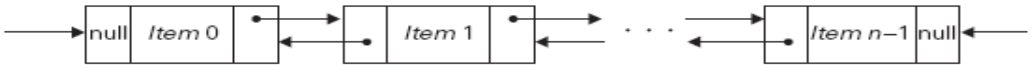


FIGURE 1.5 Doubly linked list of n elements.

The array and linked list are two principle choices in representing a more abstract data structure called linear list or simply a list. A list is a finite sequence of data items, a collection of data items arranged in a certain linear order. Basic operations on this data structure are

searching for,

inserting,

deleting

An element.

- **Stacks and Queues**

Fundamental data structures

- **Graphs: a graph is informally thought of as a collection of points in the plane called "vertices" or "nodes," some of them connected by line segments called "edges" or "arcs." Formally, a graphG = V,E is defined by a pair of two sets: a finite nonempty set V of items called vertices and a set E of pairs of these items called edges. If these pairs of vertices are unordered, i.e., a pair of vertices (u, v) is the same as the pair (v, u), we say that the vertices u and v are adjacent to each other and that they are connected by the undirected edge (u, v). We call the vertices u and v endpoints of the edge (u, v) and say that u and v are incident to this edge; we also say that the edge (u, v) is incident to its endpoints u and v. A graph G is called undirected if every edge in it is undirected. Graphs used for representing relations. In a directed graph every edge has a direction. Directed graphs are also called as digraphs**

FIGURE 1.6 (a) Undirected graph. (b) Digraph.

Fundamental data structures

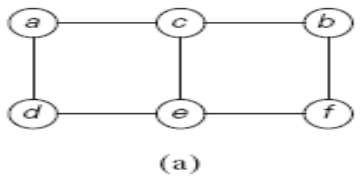
If a pair of vertices (u, v) is not the same as the pair (v, u), we say that the edge (u, v) is directed from the vertex u, called the edge's tail, to the vertex v, called the edge's head. We also say that the edge (u, v) leaves u and enters v. A graph whose every edge is directed is called directed. Directed graphs are also called digraphs.

It is normally convenient to label vertices of a graph or a digraph with letters, integer numbers, or, if an application calls for it, character strings (Figure 1.6). The graph depicted in Figure 1.6a has six vertices and seven undirected edges:

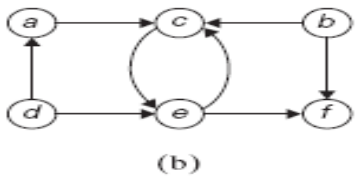
V = {a, b, c, d, e, f }, E = {(a, c), (a, d), (b, c), (b, f), (c, e), (d, e), (e, f)}.

The digraph depicted in Figure 1.6b has six vertices and eight directed edges:

V = {a, b, c, d, e, f }, E = {(a, c), (b, c), (b, f), (c, e), (d, a), (d, e), (e, c), (e, f)}.



(a)



(b)

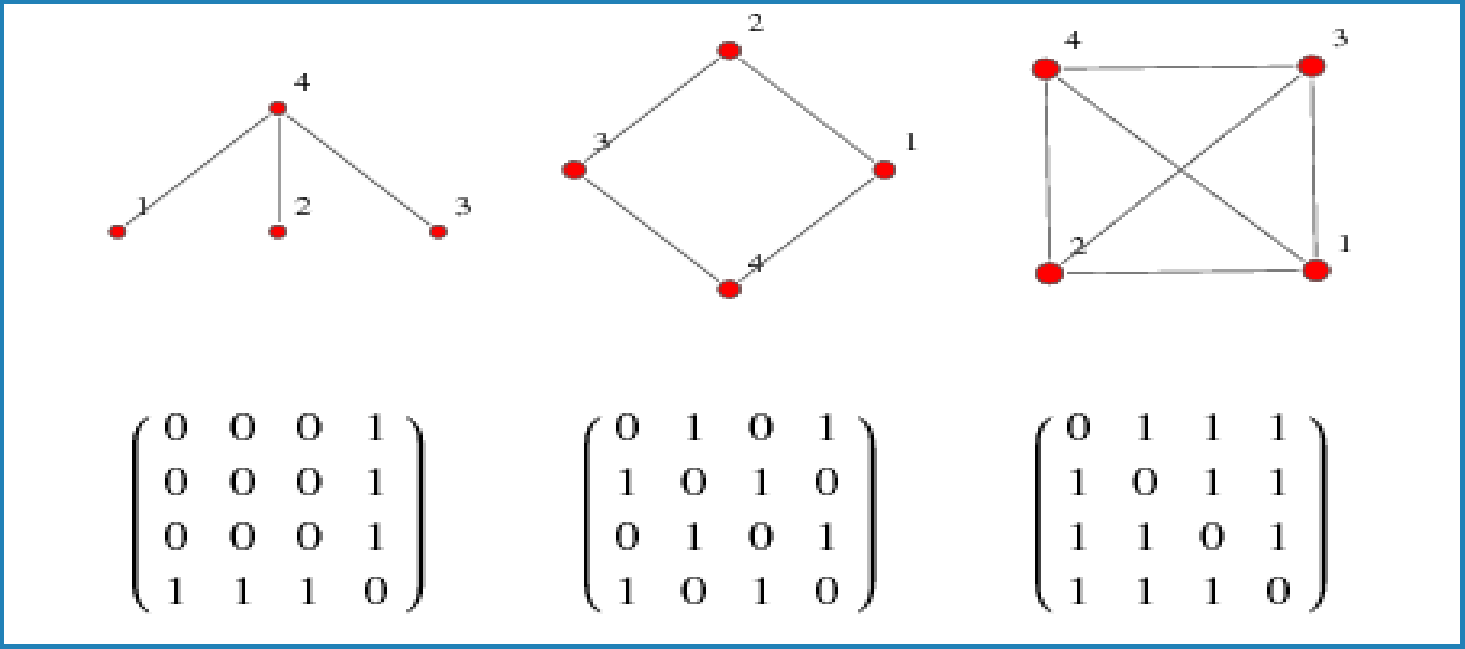
FIGURE 1.6 (a) Undirected graph. (b) Digraph.

Fundamental data structures

- **Adjacency Matrix**

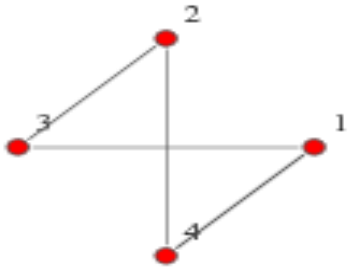
The adjacency matrix, sometimes also called the connection matrix, of a simple labeled graph is a matrix with rows and columns labeled by graph vertices, with a 1 or 0 in position (vi, vj) according to whether vi, vj are adjacent or not. For a simple graph with no self-loops, the adjacency matrix must have 0s on the diagonal. For an undirected graph, the adjacency matrix is symmetric.

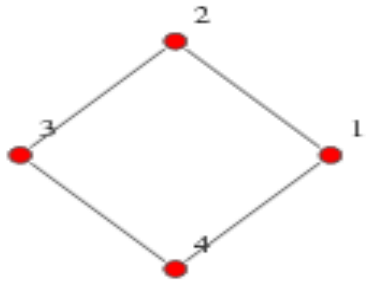
CLAW GRAPH **CYCLE GRAPH** **COMPLETE GRAPH**

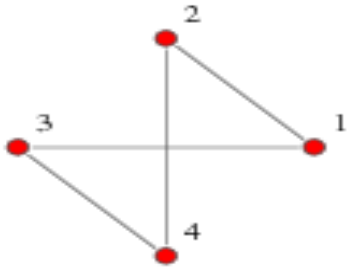


$$\begin{pmatrix} 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 1 \\ 1 & 0 & 1 & 1 \\ 1 & 1 & 0 & 1 \\ 1 & 1 & 1 & 0 \end{pmatrix}$$

Fundamental data structures

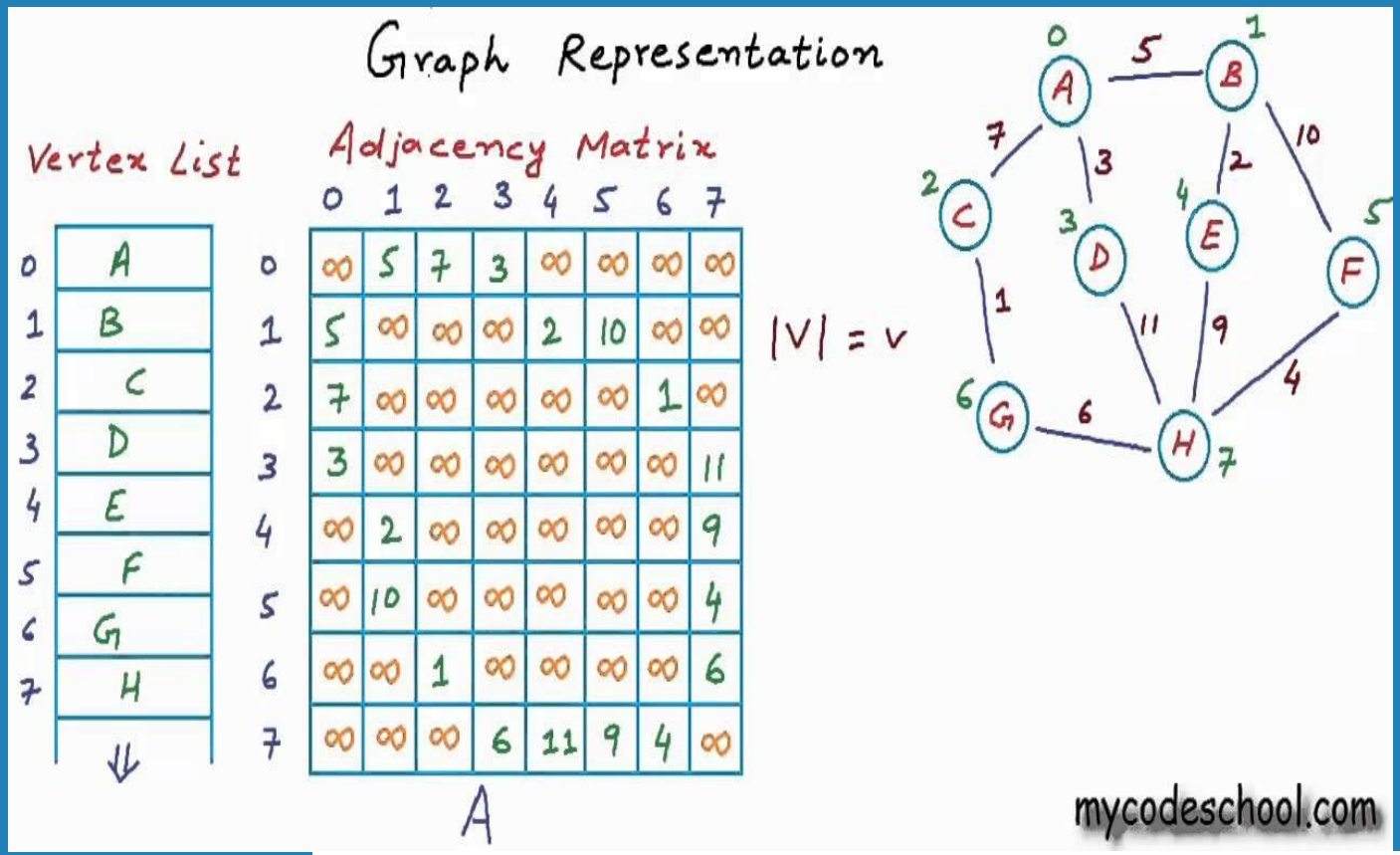






$$\begin{pmatrix} 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & 1 & 0 \end{pmatrix} \quad \begin{pmatrix} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$$

Fundamental data structures



Adjacency lists specifying this graph:

A => b,5 => c,7 =>d,3

B => a,5 =>e,2 =>f,10 …….

Fundamental data structures

- **Trees: A tree (free tree) is a connected acyclic graph. If a graph is acyclic but not connected, it is called as forest and each of its connected components is a tree. An important property of trees is the fact that for every vertices in a tree, there always exactly one simple path form on of these vertices to the other.**

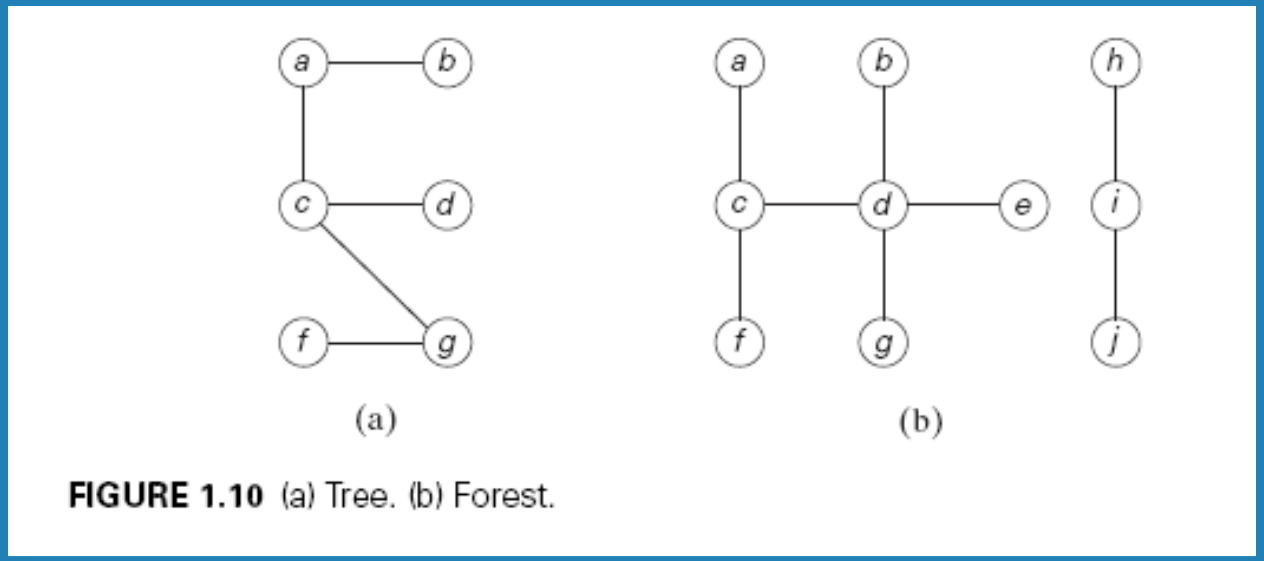


FIGURE 1.10 (a) Tree. (b) Forest.

Fundamental data structures

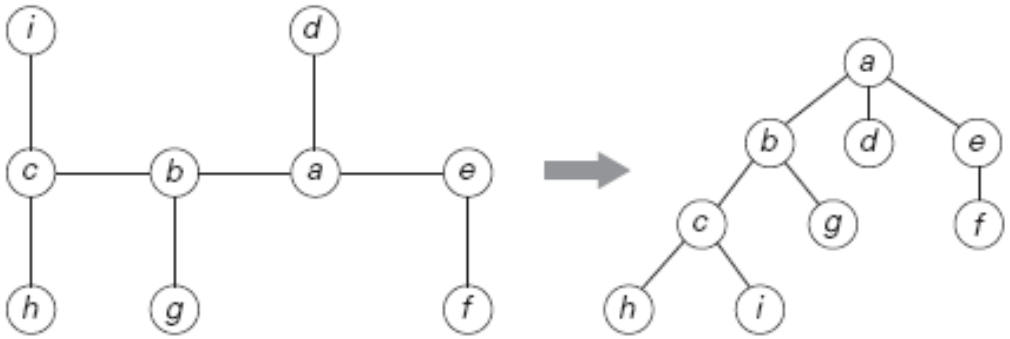


FIGURE 1.11 (a) Free tree. (b) Its transformation into a rooted tree.

Rooted Trees Another very important property of trees is the fact that for every two vertices in a tree, there always exists exactly one simple path from one of these vertices to the other. This property makes it possible to select an arbitrary vertex in a free tree and consider it as the root of the so-called rooted tree. A rooted tree is usually depicted by placing its root on the top (level 0 of the tree), the vertices adjacent to the root below it (level 1), the vertices two edges apart from the root still below (level 2), and so on. Figure 1.11 presents such a transformation from a free tree to a rooted tree.

Fundamental data structures

Ordered Trees An ordered tree is a rooted tree in which all the children of each vertex are ordered. It is convenient to assume that in a tree's diagram, all the children are ordered left to right. A binary tree can be defined as an ordered tree in which every vertex has no more than two children and each child is designated as either a left child or a right child of its parent; a binary tree may also be empty. An example of a binary tree is given in Figure 1.12a. The binary tree with its root at the left (right) child of a vertex in a binary tree is called the left (right) subtree of that vertex. Since left and right subtrees are binary trees as well, a binary tree can also be defined recursively. This makes it possible to solve many problems involving binary trees by recursive algorithms.

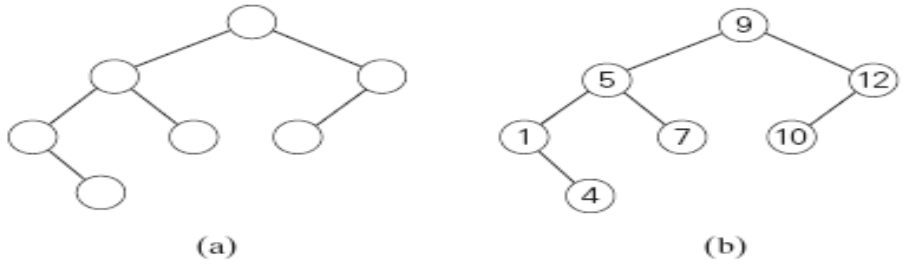


FIGURE 1.12 (a) Binary tree. (b) Binary search tree.

Fundamental data structures

A binary tree is usually implemented for computing purposes by a collection of nodes corresponding to vertices of the tree. Each node contains some information associated with the vertex (its name or some value assigned to it) and two pointers to the nodes representing the left child and right child of the vertex, respectively. Figure 1.13 illustrates such an implementation for the binary search tree in Figure 1.12b.

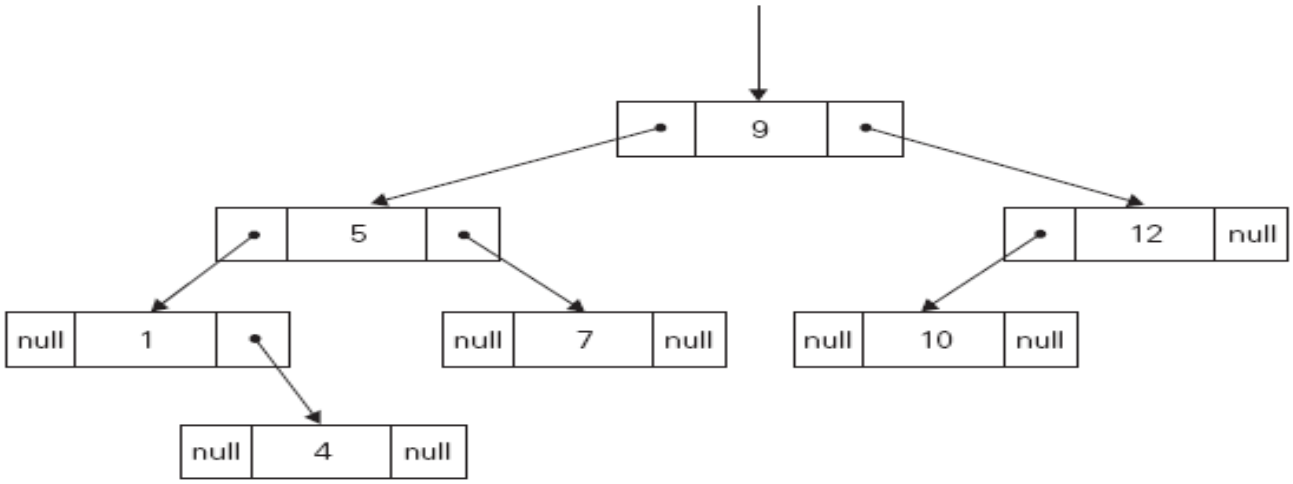
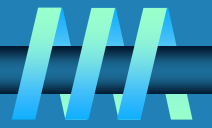


FIGURE 1.13 Standard implementation of the binary search tree in Figure 1.12b.

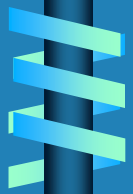
Fundamental data structures



- **Sets and Dictionaries**

The notion of a set plays a central role in mathematics. A set can be described as an unordered collection (possibly empty) of distinct items called elements of the set. A specific set is defined either by an explicit listing of its elements (e.g., S = {2,3, 5, 7}) or by specifying a property that all the set's elements and only they must satisfy (e.g., S = {n: n is a prime number smaller than 10}). The most important set operations are: checking membership of a given item in a given set; finding the union of two sets, which comprises all the elements in either or both of them; and finding the intersection of two sets, which comprises all the common elements in the sets.

In computing, the operations we need to perform for a set or a multiset most often are searching for a given item, adding a new item, and deleting an item from the collection. A data structure that implements these three operations is called the dictionary.

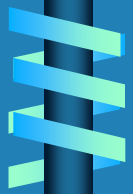


Example:

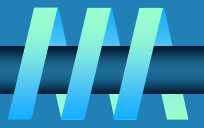
Different ways to calculate Greatest Common Divisor

- Euclid's Algorithm
- Consequtive Integer Checking
- Middle-school Procedure: Compute the product of all the common prime factors.

All the above algorithms have different algorithm design techniques. Their speeds dramatically different from each other.



Euclid's Algorithm



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

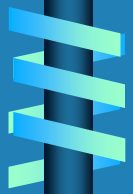
Examples: gcd(60,24) = 12, gcd(60,0) = 60, gcd(0,0) = ?

Euclid's algorithm is based on repeated application of equality

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

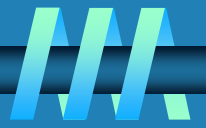
until the second number becomes 0, which makes the problem trivial.

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

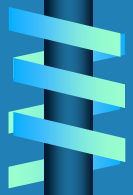


Descriptions of Euclid's algorithm for computing *GCD*

Stuructural Description

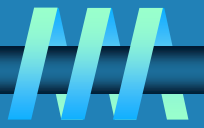


How can we write down structural description of the Euclid's algorithm? (gcd(m,n))



Descriptions of Euclid's algorithm for computing *GCD*

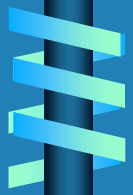
Stuructural Description



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

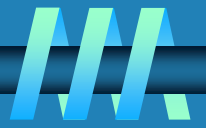
Step 2 Divide m by n and assign the value for the remainder to r

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

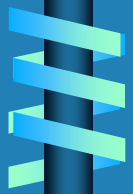


Descriptions of Euclid's algorithm for computing *GCD)*

Pseudocode

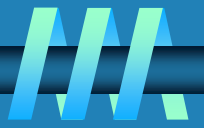


How can we write the pseudocode for Euclid's Algorithm?



Descriptions of Euclid's algorithm for computing *GCD*

Pseudocode



Ex: Below is a function that takes two positive integer and returns GCD which is found by Euclid's algorithm..

ALGORITHM Euclid(m,n)

//Input: Two nonnegative, not-both-zero integers m and n

//Output: Greatest common divisor of m and n

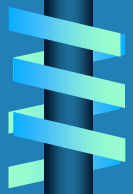
while n ≠ 0 do

r ← m mod n

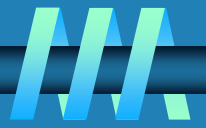
m← n

n ← r

return m



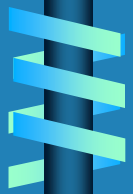
Consecutive integer checking algorithm for computing GCD



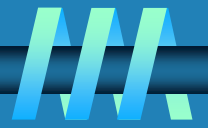
Lets write a Structural Description that :

- takes two positive integers and

- returns GCD which is found by Consecutive integer checking algorithm



Consecutive integer checking algorithm for computing GCD



Consecutive integer checking algorithm for computing gcd(m, n)

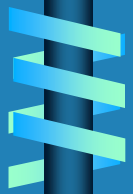
Step 1 Assign the value of min{m, n} to t.

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

Step 3 Divide n by t. If the remainder of this division is 0, return the value of t as the answer and stop; otherwise, proceed to Step 4.

Step 4 Decrease the value of t by 1. Go to Step 2.

Note that unlike Euclid's algorithm, this algorithm, in the form presented, does not work correctly when one of its input numbers is zero. This example illustrates why it is so important to specify the set of an algorithm's inputs explicitly and carefully.



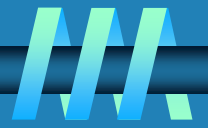
Consecutive integer checking algorithm for computing GCD

Consecutive integer checking algorithm

Pseudocode

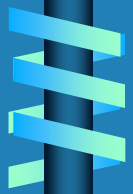
```
ALGORITHM CIC(m,n)
//Input: Two nonnegative, not-both-zero integers m and n
//Output: Greatest common divisor of m and n
t ← min(m,n)
while t ≠ 0 do
    r ← m mod t
    if  r = 0
        then r ← n mod t
        if  r = 0
            then return t
   t ← t-1
return t
```

Example on GCD

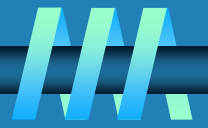


a. Find gcd(31415, 14142) by applying Euclid's algorithm.

b. Estimate how many times faster it will be to find gcd(31415, 14142) by Euclid's algorithm compared with the algorithm based on checking consecutive integers from min{m, n} down to gcd(m, n).



Example on GCD

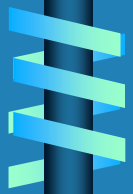


a. Find gcd(31415, 14142) by applying Euclid's algorithm.

b. Estimate how many times faster it will be to find gcd(31415, 14142) by Euclid's algorithm compared with the algorithm based on checking consecutive integers from min$\{m, n\}$ down to gcd(m, n).

a. $\gcd(31415, 14142) = \gcd(14142, 3131) = \gcd(3131, 1618) =$
$\gcd(1618, 1513) = \gcd(1513, 105) = \gcd(105, 43) =$
$\gcd(43, 19) = \gcd(19, 5) = \gcd(5, 4) = \gcd(4, 1) = \gcd(1, 0) = 1.$

b. To answer the question, we need to compare the number of divisions the algorithms make on the input given. The number of divisions made by Euclid's algorithm is 11 (see part a). The number of divisions made by the consecutive integer checking algorithm on each of its 14142 iterations is either 1 and 2; hence the total number of multiplications is between $1 \cdot 14142$ and $2 \cdot 14142$. Therefore, Euclid's algorithm will be between $1 \cdot 14142/11 \approx 1300$ and $2 \cdot 14142/11 \approx 2600$ times faster.



Middle-school procedure for gcd(*m*,*n*)

Middle-school procedure for computing gcd(m, n)

Step 1 Find the prime factors of m.

Step 2 Find the prime factors of n.

Step 3 Identify all the common factors in the two prime expansions found in Step 1 and Step 2. (If p is a common factor occurring p_m and p_n times in m and n, respectively, it should be repeated $\min\{p_m, p_n\}$ times.)

Step 4 Compute the product of all the common factors and return it as the greatest common divisor of the numbers given.

Thus, for the numbers 60 and 24, we get

$$60 = 2 \cdot 2 \cdot 3 \cdot 5$$
$$24 = 2 \cdot 2 \cdot 2 \cdot 3$$
$$\gcd(60, 24) = 2 \cdot 2 \cdot 3 = 12.$$

Is this an algorithm? in the form presented, as a legitimate algorithm. Why? Because the prime factorization steps are not defined unambiguously: they require a list of prime numbers. Unless this issue is resolved, we cannot say, write a program implementing this procedure. Incidentally, Step 3 is also not defined clearly enough.

A simple algorithm for generating consecutive primes not exceeding any given integer $n > 1$. (Sieve of Eratosthenes)

Example: Find the list of primes not exceeding $n = 25$

2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25
2	3		5		7		9		11		13		15		17		19		21		23		25
2	**3**		5		7				11		13				17		19				23		25
2	3		**5**		7				11		13				17		19				23		

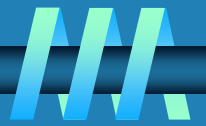
Sieve of Eratosthenes (Finding Prime Numbers)

ALGORITHM *Sieve*(n)

//Implements the sieve of Eratosthenes
//Input: A positive integer $n > 1$
//Output: Array L of all prime numbers less than or equal to n

```
for p ← 2 to n do A[p] ← p
for p ← 2 to ⌊√n⌋ do     //see note before pseudocode
    if A[p] ≠ 0          //p hasn't been eliminated on previous passes
        j ← p * p
        while j ≤ n do
            A[j] ← 0     //mark element as eliminated
            j ← j + p
//copy the remaining elements of A to array L of the primes
i ← 0
for p ← 2 to n do
    if A[p] ≠ 0
        L[i] ← A[p]
        i ← i + 1
return L
```

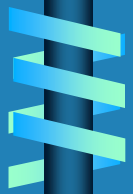
Example :
Pseudocode for finding Binary Representation



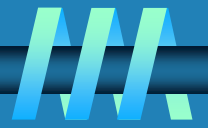
Describe the standard algorithm for finding the binary representation of a positive decimal integer

a. in English.

b. in a pseudocode.



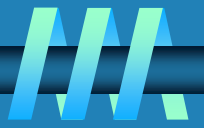
Example – Pseudocode



a. Divide the given number n by 2: the remainder r_n (0 or 1) will be the next (from right to left) digit of the binary representation in question. Replace n by the quotient of the last division and repeat this operation until n becomes 0.

b. **Algorithm** *Binary*(n)
//The algorithm implements the standard method for finding
//the binary expansion of a positive decimal integer
//Input: A positive decimal integer n
//Output: The list $b_k\ b_{k-1}...b_1\ b_0$ of n's binary digits
$k \leftarrow 0$
while $n \neq 0$
    $b_k \leftarrow n \bmod 2$
    $n \leftarrow \lfloor n/2 \rfloor$
    $k \leftarrow k + 1$

Example: Comparison Counting



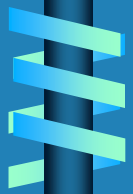
1. Consider the algorithm for the sorting problem that sorts an array by counting, for each of its elements, the number of smaller elements and then uses this information to put the element in its appropriate position in the sorted array:

Algorithm *ComparisonCountingSort*$(A[0..n-1], S[0..n-1])$
//Sorts an array by comparison counting
//Input: Array $A[0..n-1]$ of orderable values
//Output: Array $S[0..n-1]$ of A's elements sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-1$ **do**
    $Count[i] \leftarrow 0$
for $i \leftarrow 0$ **to** $n-2$ **do**
    for $j \leftarrow i+1$ **to** $n-1$ **do**
        if $A[i] < A[j]$
            $Count[j] \leftarrow Count[j] + 1$
        else $Count[i] \leftarrow Count[i] + 1$
for $i \leftarrow 0$ **to** $n-1$ **do**
    $S[Count[i]] \leftarrow A[i]$

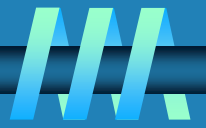
a. Apply this algorithm to sorting the list 60, 35, 81, 98, 14, 47.

b. Is this algorithm stable?

c. Is it in place?



Example: Comparison Counting



1. a. Sorting 60, 35, 81, 98, 14, 47 by comparison counting will work as follows:

Array $A[0..5]$		60	35	81	98	14	47
Initially	$Count[]$	0	0	0	0	0	0
After pass $i = 0$	$Count[]$	3	0	1	1	0	0
After pass $i = 1$	$Count[]$		1	2	2	0	1
After pass $i = 2$	$Count[]$			4	3	0	1
After pass $i = 3$	$Count[]$				5	0	1
After pass $i = 4$	$Count[]$					0	2
Final state	$Count[]$	3	1	4	5	0	2

Array $S[0..5]$	14	35	47	60	81	98

b. The algorithm is not stable. Consider, as a counterexample, the result of its application to $1'$, $1''$.

c. The algorithm is not in place because it uses two extra arrays of size n: $Count$ and S.