S. J. P. N. TRUST'S
HIRASUGAR INSTITUTE OF TECHNOLOGY, NIDASOSHI
Accredited at 'A' Grade by NAAC
Programmes Accredited by NBA: CSE, ECE, EEE & ME.

Department of Computer Science & Engineering

Course: Design And Analysis of Algorithms (18CS42)

Module 1: Introduction, What is an Algorithm? Algorithm Specification Analysis Framework

Prof. A. A. Daptardar
Asst. Prof. , Dept. of Computer Science & Engg.,
Hirasugar Institute of Technology, Nidasoshi

Introduction

- The word “ALGORITHM” comes from the name of a Persian author, Abu Ja’far Mohammed ibn Musa al Khowarizmi (c. 825A.D.), who wrote a textbook on mathematics
- This word has taken on a special significance in computer science, where “ALGORITHM” has come to refer to a method that can be used by a computer for the solution of a problem

What is an Algorithm

- An "ALGORITHM" is a finite set of instructions that, if followed, accomplishes a particular task.
- In addition, all algorithm must satisfy the following criteria:
  1. **Input** – Zero or more quantities are externally supplied
  2. **Output** – At least one quantity is produced
  3. **Definiteness** – Each instruction is clear and unambiguous
  4. **Finiteness** – After tracing all instructions in ALG, the ALG terminates after a finite number of steps
  5. **Effectiveness** - Every instruction must be very basic so that it can be carried out, in principle, by a person using only pencil and paper. Also each operation must feasible

Algorithm Specification
Pseudocode Conventions

- ALG can be described using many ways
- Graphic representation called flowcharts are another possibility, but they work well only if the algorithm is small and simple
- In this text we present most of our algorithm using a Pseudocode that resembles C program
  1. Comments begins with // and continue until the end of line
  2. Blocks are indicated with matching braces { and }
  3. An identifier begins with a letter. The data types of variables are not explicitly declared

Algorithm Specification Pseudocode Conventions

4. Assignment of values to variables is done using the assignment statement - <variable> = <expression>;
5. Boolean values – true and false, logical operators – and, or and not, relational operators - <, >, == etc. are provided
6. Elements of multidimensional arrays are accessed using [and]. Array indices start at zero
7. for, while and repeat-until looping statements are provided-

```
while < condition> do
{
    statement 1;
        ….
    statement n;
}
```

```
for variable:= value1 to value2 step step do
 {
        statement 1;
            …….
        statement n;
 }
```

Pseudocode Conventions

A **repeat-until** statement is constructed as follows-

repeat
    <statement 1>
                ….
    <statement n>
until<condition>

8. A conditional statements has the following forms-
   if <condition> then <statement>

   if <condition> then <statement1> else <statement2>

   case
   {
        :<condition1>: <statement1>;
                ……
                ……
        :<condition n>: <statement n>;
        :else: <statement n + 1>

   }

Pseudocode Conventions

9. Input and Output are given using the instruction read and write
10. An algorithm consists of a heading and a body – Algorithm Name (<parameter list>)

Example:- ALG to find maximum element in Array

```
1  Algorithm Max(A, n)
2  // A is an array of size n.
3  {
4       Result := A[1];
5       for i := 2 to n do
6            if A[i] > Result then Result := A[i];
7       return Result;
8  }
```

Algorithm to find GCD of two numbers

ALGORITHM *Euclid(m, n)*

//Computes gcd(m, *n) by Euclid's algorithm*

!!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***

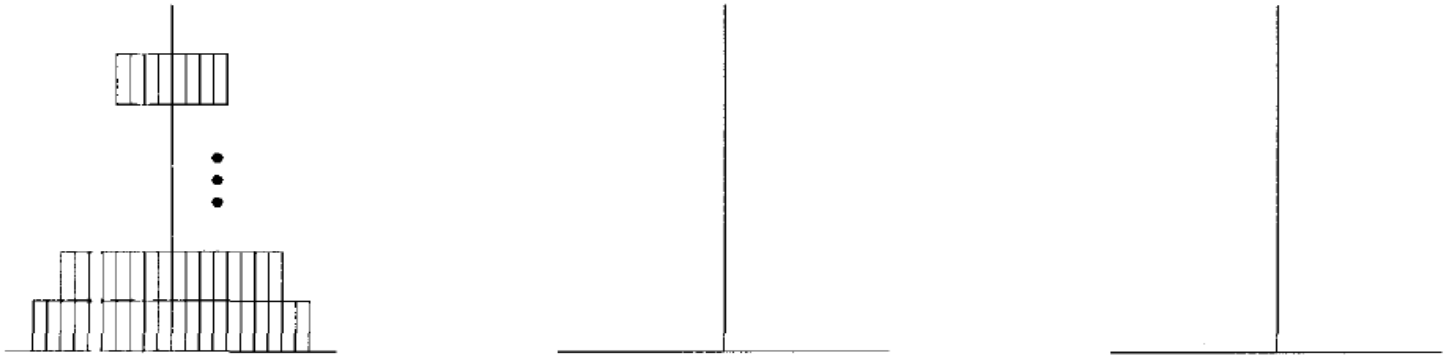
m ← *n*

n ← *r*

return *m*

Recursive Algorithms

- A recursive function is a function that is defined in terms of itself
- An ALG that calls itself is direct recursive
- ALG A is said to be indirect recursive if it calls another algorithm which in turns calls A



```
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  }
```

Analysis Framework

- The efficiency of an algorithm can be decided by measuring the performance of an algorithm.
- The performance of an algorithm can be measured by computing two factors:
  - Amount of time required by an algorithm to execute.
  - Amount of storage required by an algorithm.

Analysis Framework

- There are two kinds of efficiency – Time and Space
- Time efficiency indicates how fast an algorithm in question runs
- Space efficiency deals with the extra space the algorithm requires
- Measuring an Input's size – almost all ALGs run longer on larger inputs. For example, it takes longer to sort larger arrays, multiply larger matrices, and so on
- Units for Measuring Running Time – measuring an ALGs running time, simply can use a standard unit as seconds, milliseconds and so on. But, practically to identify the most important operation of the algorithm, called the basic operation, the operation contributing the most to the total running time, and compute the number of times the basic operation is executed. Hence, we can estimate, $T(n) \approx C_{op}C(n)$, where T(n) is running time of a program, C_{op} be the execution time of an ALGs basic operation on a particular computer, and C(n) be the number of times this basic operation needs to be executed for this algorithm.

Analysis Framework

- Worst-Case Efficiency –It is an efficiency when algorithm runs for a longest time. Example – $C_{worst}(n)=n$
- Best-Case Efficiency – It is an efficiency when the algorithm runs for short time. Example – $C_{best}(n)=1$
- Average-Case Efficiency – This type of efficiency gives information about the behaviour of an algorithm on specific or random input. Example – $C_{avg}(n)=p(n+1)/2 + n(1-p)$

Performance Analysis

- Major goal of this subject is to develop skills for making evaluative judgments about algorithm
- There are many criteria upon which we judge algorithm-
  1. Does it do what we want it to do?
  2. Does it work correctly according to the original specification of the task?
  3. Is there documentation that describes how to use it and how it works?
  4. Are procedures created in such a way that they perform logical sub-functions?
  5. Is the code readable?
- There are other important criteria for judging algorithms that have a more direct relationship to performance
  1. Space Complexity (Storage Requirement)
  2. Time Complexity (Computing Time)

Space Complexity (Storage Requirement)

- The space complexity of an algorithm is the amount of memory it needs to run to completion
- The space needed is the sum of a fixed part and variable part
  1. Fixed Part – is independent of characters of the inputs and outputs. It includes instruction space(space for code), space for simple variables and fixed size component variables, space for constants etc.
  2. Variable Part – consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, the space needed by referenced variables and recursion stack space
- The space requirement S(P) of any algorithm P may therefore be written as $S(P) = c + S_p(\text{instance characteristics})$, where c is constant

Time Complexity (Computing Time)

- The time complexity of an algorithm is the amount of computer time it needs to run to completion
- The time T(P) taken by a program P is the sum of the compile time and the run (or execution time)
- The compile time does not depend on the instance characteristics
- Hence, we concern with just the run time of a program
- This run time is denoted by t_p(instance characteristics)

Step Count for Array Element Addition

Statement	s/e	frequency	total steps
1 **Algorithm** Sum(a, n)	0	–	0
2 {	0	–	0
3 $s := 0.0$;	1	1	1
4 **for** $i := 1$ **to** n **do**	1	$n+1$	$n+1$
5 $s := s + a[i]$;	1	n	n
6 **return** s;	1	1	1
7 }	0	–	0
Total			$2n+3$

Step Count for Two Matrix Addition

Statement	s/e	frequency	total steps
1 **Algorithm** Add(a, b, c, m, n)	0	–	0
2 {	0	–	0
3 **for** $i := 1$ **to** m **do**	1	$m+1$	$m+1$
4 **for** $j := 1$ **to** n **do**	1	$m(n+1)$	$mn+m$
5 $c[i, j] := a[i, j] + b[i, j];$	1	mn	mn
6 }	0	–	0
Total			$2mn + 2m + 1$

Step Count for Rsum of Array Elements

Statement	s/e	frequency		total steps	
		$n=0$	$n>0$	$n=0$	$n>0$
1 **Algorithm** RSum(a,n)	0	–	–	0	0
2 {					
3 **if** $(n \leq 0)$ **then**	1	1	1	1	1
4 **return** 0.0;	1	1	0	1	0
5 **else return**					
6 RSum$(a,n-1)+a[n]$;	$1+x$	0	1	0	$1+x$
7 }	0	–	–	0	0
Total				2	$2+x$

$$x = t_{\mathsf{RSum}}(n-1)$$

When analyzing a recursive program for its step count, we often obtain a recursive formula for the step count, for example,

$$t_{\mathsf{RSum}}(n) = \begin{cases} 2 & \text{if } n = 0 \\ 2 + t_{\mathsf{RSum}}(n-1) & \text{if } n > 0 \end{cases}$$

$$\begin{aligned} t_{\mathsf{RSum}}(n) &= 2 + t_{\mathsf{RSum}}(n-1) \\ &= 2 + 2 + t_{\mathsf{RSum}}(n-2) \\ &= 2(2) + t_{\mathsf{RSum}}(n-2) \\ &\vdots \\ &= n(2) + t_{\mathsf{RSum}}(0) \\ &= 2n + 2, \qquad n \geq 0 \end{aligned}$$

So the step count for RSum (Algorithm 1.7) is $2n + 2$.

Module – 1

Asymptotic Notations

- Big-Oh notation (*O*)
- Big-Omega notation (*Ω*)
- Big-Theta notation (Θ)

Asymptotic Notations

- The efficiency analysis framework concentrates on the order of growth of an algorithm's basic operation count as the principal indicator of the algorithm's efficiency
- To compare and rank such orders of growth, computer scientists use three notations:- Big-Oh notation (O), Big-Omega notation (Ω) and Big-Theta notation (Θ)

Big-Oh notation (*O*)

- A function t(n) is said to be in O(g(n)), denoted t(n) ∈ O(g(n)), if t(n) is bounded above by some constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integer n_0 such that t(n) ≤ cg(n) for all n ≥ n_0

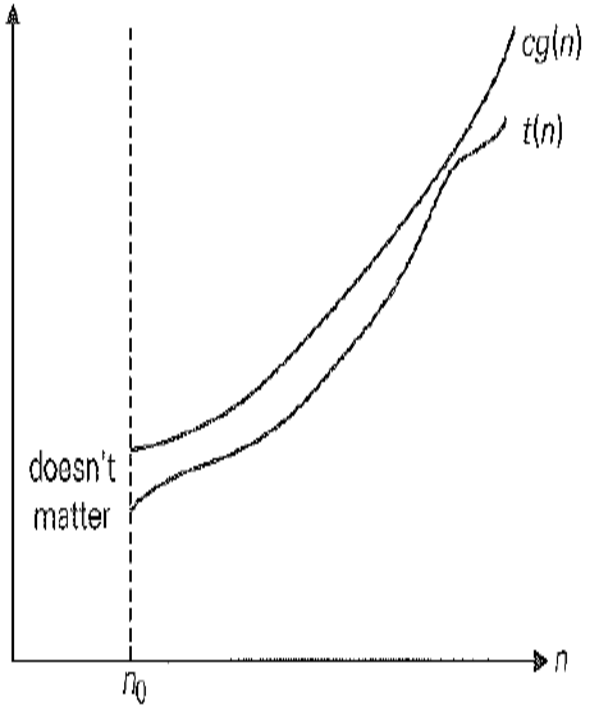


FIGURE 2.1 Big-oh notation: $t(n) \in O(g(n))$

Big-Oh notation (*O*) - Example

Prove the following example –

$$100n + 5 \in O(n^2)$$

Solution:-

$100n + 5 \leq 100n + n = n(100+1) = 101n$

Hence Constant, c = 101

Hence, $100n + 5 \in O(n^2)$ for all $n \geq 5$ and constant c=101

n0	100n + 5	≤	101n	Status
0	5	≤	0	False
1	105	≤	101	False
2	205	≤	202	False
3	305	≤	303	False
4	405	≤	404	False
5	505	≤	505	True

Prove the following example – $10n^2 + 4n + 2 \leq O(n^2)$

Solution - $= 10n^2 + n^2$

$= n^2(10 + 1)$

$= 11n^2$

Hence, constant, c = 11

Hence, $10n^2 + 4n + 2 \leq O(n^2)$ for all $n \geq 5$ and constant c=11

n_0	$10n^2 + 4n + 2$	≤	$11n^2$	Status
0	2	≤	0	False
1	16	≤	11	False
2	50	≤	44	False
3	104	≤	99	False
4	178	≤	176	False
5	272	≤	275	True

Big-Omega notation (Ω)

- A function t(n) is said to be in Ω(g(n)), denoted t(n) $\in$ Ω(g(n)), if t(n) is bounded below by some positive constant multiple of g(n) for all large n, i.e., if there exist some positive constant c and some nonnegative integers n_0 such that

  $t(n) \geq cg(n)$ for all $n \geq n_0$

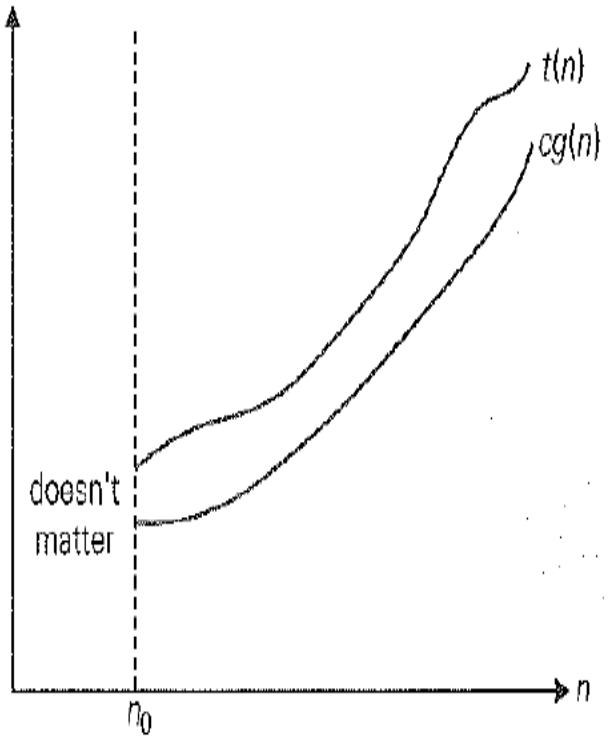


FIGURE 2.2 Big-omega notation: $t(n) \in \Omega(g(n))$

Big-Omega notation (Ω) - Example

- Prove the following examples-
  1. $n^3 \in \Omega(n^2)$ - where n^3 becomes greater than n^2 when we select constant, c=1 and n_0=0
  2. $3n + 2 \in \Omega(n)$ - where $3n + 2$ becomes greater than n when we select constant, c=3 and n_0=0

- Prove the following examples-
  3. $3n + 3 \in \Omega(1)$ - where $3n + 3$ becomes greater than 1 when we select constant, c=3 and n_0=0
  4. $6 * 2^n + n^2 \in \Omega(2^n)$ - where $6 * 2^n + n^2$ becomes greater than 2^n when we select constant, c=6 and n_0=0

Big-Theta notation (Θ)

- A function t(n) is said to be in Θ(g(n)), denoted t(n) ϵ Θ(g(n)), if t(n) is bounded both above and below by some positive constant multiples of g(n) for all large n, i.e., if there exist some positive constant c1 and c2 and some nonnegative integers n_0 such that

$c_2g(n) \leq t(n) \leq c_1g(n)$ for all n $\geq n_0$

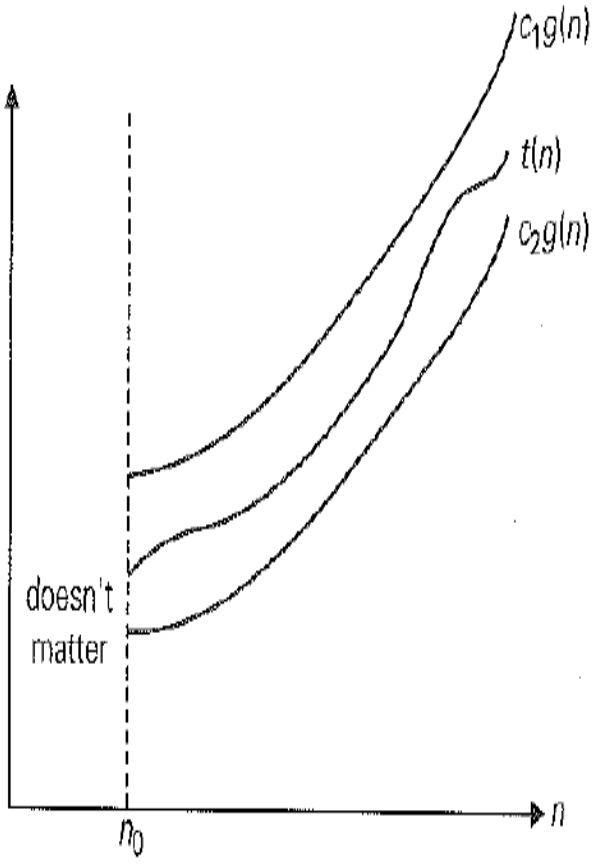


FIGURE 2.3 Big-theta notation: $t(n) \in \Theta(g(n))$

Big-Theta notation (Θ) - Example

- Prove the following example of Big-Theta

$$3n + 2 \in \Theta(n)$$

Solution:-

For $3n + 2 \geq n$, constant $c1 = 3$

For $3n + 2 \leq n$, $= 3n + n = n(3 + 1) = 4n$, constant $c2 = 4$

Hence $3n+2 \in \Theta(n)$ with constant $c1=3$, $c2=4$ and for all $n \geq 2$

n_0	$3n+2$	$\geq$	$3n$	Status
0	2	$\geq$	0	True
1	5	$\geq$	3	True
2	8	$\geq$	6	True

n_0	$3n+2$	$\leq$	$4n$	Status
0	2	$\leq$	0	False
1	5	$\leq$	4	False
2	8	$\leq$	8	True

THEOREM If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then

$$t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\}).$$

(The analogous assertions are true for the Ω and Θ notations as well.)

PROOF (As you will see, the proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, and b_2: if $a_1 \le b_1$ and $a_2 \le b_2$, then $a_1 + a_2 \le 2 \max\{b_1, b_2\}$.) Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \le c_1 g_1(n) \quad \text{for all } n \ge n_1.$$

Similarly, since $t_2(n) \in O(g_2(n))$,

$$t_2(n) \le c_2 g_2(n) \quad \text{for all } n \ge n_2.$$

Let us denote $c_3 = \max\{c_1, c_2\}$ and consider $n \ge \max\{n_1, n_2\}$ so that we can use both inequalities. Adding the two inequalities above yields the following:

$$\begin{aligned} t_1(n) + t_2(n) &\le c_1 g_1(n) + c_2 g_2(n) \\ &\le c_3 g_1(n) + c_3 g_2(n) = c_3[g_1(n) + g_2(n)] \\ &\le c_3 2 \max\{g_1(n), g_2(n)\}. \end{aligned}$$

Hence, $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$, with the constants c and n_0 required by the O definition being $2c_3 = 2 \max\{c_1, c_2\}$ and $\max\{n_1, n_2\}$, respectively. ■

TABLE 2.2 Basic asymptotic efficiency classes

Class	Name	Comments
1	*constant*	Short of best-case efficiencies, very few reasonable examples can be given since an algorithm's running time typically goes to infinity when its input size grows infinitely large.
$\log n$	*logarithmic*	Typically, a result of cutting a problem's size by a constant factor on each iteration of the algorithm (see Section 5.5). Note that a logarithmic algorithm cannot take into account all its input (or even a fixed fraction of it): any algorithm that does so will have at least linear running time.
n	*linear*	Algorithms that scan a list of size n (e.g., sequential search) belong to this class.
$n \log n$	"*n-log-n*"	Many divide-and-conquer algorithms (see Chapter 4), including mergesort and quicksort in the average case, fall into this category.
n^2	*quadratic*	Typically, characterizes efficiency of algorithms with two embedded loops (see the next section). Elementary sorting algorithms and certain operations on n-by-n matrices are standard examples.
n^3	*cubic*	Typically, characterizes efficiency of algorithms with three embedded loops (see the next section). Several nontrivial algorithms from linear algebra fall into this class.
2^n	*exponential*	Typical for algorithms that generate all subsets of an n-element set. Often, the term "exponential" is used in a broader sense to include this and larger orders of growth as well.
$n!$	*factorial*	Typical for algorithms that generate all permutations of an n-element set.

Module - 1

- Mathematical analysis of Non-Recursive Algorithms with Examples
- Mathematical analysis of Recursive Algorithms with Examples **(T1:2.2, 2.3, 2.4).**

Mathematical analysis of Non-Recursive Algorithms with Examples

- General plan for Analyzing Time Efficiency of Non-Recursive algorithms-
  1. Decide on a parameter indicating an input size
  2. Identify the algorithm's basic operation
  3. Check whether the number of times the basic operation is executed depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case and best-case efficiencies have to be investigated separately
  4. Set up a sum expressing the number of times the algorithm's basic operation is executed
  5. Using standard formulas and rules of sum manipulation, either find a closed-form formula for the count or, at the very least, establish its order of growth

Summation Rules & Formulas

- Summation Rules

$$\sum_{i=l}^{u} c a_i = c \sum_{i=l}^{u} a_i \qquad \textbf{(R1)}$$

$$\sum_{i=l}^{u} (a_i \pm b_i) = \sum_{i=l}^{u} a_i \pm \sum_{i=l}^{u} b_i \qquad \textbf{(R2)}$$

- Summation Formulae

$$\sum_{i=l}^{u} 1 = u - l + 1 \text{ where } l \leq u \text{ are some lower and upper integer limits} \qquad \textbf{(S1)}$$

$$\sum_{i=0}^{n} i = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2). \qquad \textbf{(S2)}$$

Maximum value in an array

ALGORITHM *MaxElement*$(A[0..n-1])$

//Determines the value of the largest element in a given array
//Input: An array $A[0..n-1]$ of real numbers
//Output: The value of the largest element in A
$maxval \leftarrow A[0]$
for $i \leftarrow 1$ **to** $n-1$ **do**
    if $A[i] > maxval$
        $maxval \leftarrow A[i]$
return *maxval*

Element Uniqueness Problem

ALGORITHM *UniqueElements*$(A[0..n-1])$

```
//Determines whether all the elements in a given array are distinct
//Input: An array A[0..n − 1]
//Output: Returns "true" if all the elements in A are distinct
//          and "false" otherwise
for i ← 0 to n − 2 do
    for j ← i + 1 to n − 1 do
        if A[i] = A[j] return false
return true
```

$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

$$= \sum_{i=0}^{n-2} (n-1) - \sum_{i=0}^{n-2} i = (n-1) \sum_{i=0}^{n-2} 1 - \frac{(n-2)(n-1)}{2}$$

$$= (n-1)^2 - \frac{(n-2)(n-1)}{2} = \frac{(n-1)n}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2).$$

Matrix Multiplication Example

ALGORITHM *MatrixMultiplication*$(A[0..n-1, 0..n-1], B[0..n-1, 0..n-1])$

//Multiplies two n-by-n matrices by the definition-based algorithm

//Input: Two n-by-n matrices A and B

//Output: Matrix $C = AB$

for $i \leftarrow 0$ **to** $n-1$ **do**

  for $j \leftarrow 0$ **to** $n-1$ **do**

    $C[i, j] \leftarrow 0.0$

    for $k \leftarrow 0$ **to** $n-1$ **do**

      $C[i, j] \leftarrow C[i, j] + A[i, k] * B[k, j]$

return C

and the total number of multiplications $M(n)$ is expressed by the following triple sum:

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1.$$

Now we can compute this sum by using formula $(S1)$ and rule $(R1)$ (see above). Starting with the innermost sum $\sum_{k=0}^{n-1} 1$, which is equal to n (why?), we get

$$M(n) = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{k=0}^{n-1} 1 = \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} n = \sum_{i=0}^{n-1} n^2 = n^3.$$

Mathematical analysis of Recursive Algorithms with Examples

- General plan for Analyzing Time Efficiency of Recursive algorithms-
  1. Decide on a parameter/parameter's indicating an input's size
  2. Identify the algorithm's basic operation
  3. Check whether the number of times the basic operation is executed can vary on different inputs of the same size; if it can, the worst-case, average-case and best-case efficiencies must be investigated separately
  4. Set up a recurrence relation, with an appropriate initial condition for number of times the basic operation is executed.
  5. Solve the recurrence or at least ascertain the order of growth of its solution

Tower of Hanoi Example

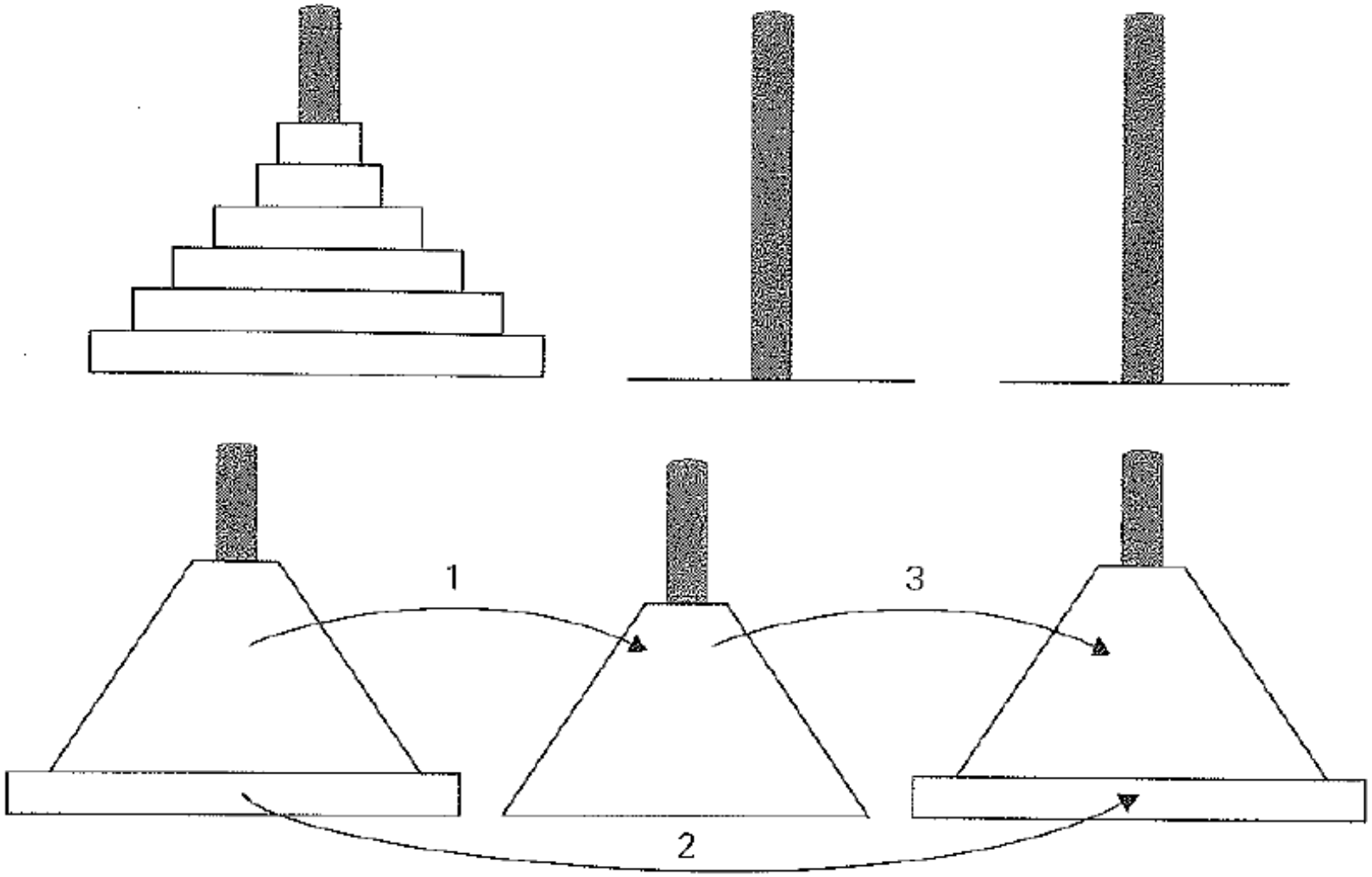


FIGURE 2.4 Recursive solution to the Tower of Hanoi puzzle

```
Algorithm TowerofHanoi(n, A, B, C)
{
if (n = 1)
{
write(“move disk from peg A to peg C”)
return;
}
else
{
TowerofHanoi(n-1, A, C, B);
TowerofHanoi(n-1, B, C, A);
}
```

Mathematical Analysis

Let us apply the general plan to the Tower of Hanoi problem. The number of disks n is the obvious choice for the input's size indicator, and so is moving one disk as the algorithm's basic operation. Clearly, the number of moves $M(n)$ depends on n only, and we get the following recurrence equation for it:

$$M(n) = M(n-1) + 1 + M(n-1) \quad \text{for } n > 1.$$

With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for the number of moves $M(n)$:

$$\begin{aligned} M(n) &= 2M(n-1) + 1 \quad \text{for } n > 1, \\ M(1) &= 1. \end{aligned} \tag{2.3}$$

We solve this recurrence by the same method of backward substitutions:

$$\begin{aligned} M(n) &= 2M(n-1) + 1 && \text{sub. } M(n-1) = 2M(n-2) + 1 \\ &= 2[2M(n-2) + 1] + 1 = 2^2 M(n-2) + 2 + 1 && \text{sub. } M(n-2) = 2M(n-3) + 1 \\ &= 2^2[2M(n-3) + 1] + 2 + 1 = 2^3 M(n-3) + 2^2 + 2 + 1. \end{aligned}$$

The pattern of the first three sums on the left suggests that the next one will be $2^4 M(n-4) + 2^3 + 2^2 + 2 + 1$ and, generally, after i substitutions, we get

$$M(n) = 2^i M(n-i) + 2^{i-1} + 2^{i-2} + \cdots + 2 + 1 = 2^i M(n-i) + 2^i - 1.$$

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$, we get the following formula for the solution to recurrence (2.3):

$$\begin{aligned} M(n) &= 2^{n-1} M(n-(n-1)) + 2^{n-1} - 1 \\ &= 2^{n-1} M(1) + 2^{n-1} - 1 = 2^{n-1} + 2^{n-1} - 1 = 2^n - 1. \end{aligned}$$

Module - 1

Important Problem Types

1. Sorting
2. Searching
3. String processing
4. Graph Problems
5. Combinatorial Problems

Important Problem Types

- Sorting- The sorting problem asks us to rearrange the items of a given list in ascending or descending order. As a practical matter, we need to sort list of numbers, characters from alphabets, character string and most important records about students, employees, libraries about holding books etc.. The special piece of information called "key" is used to sort list items. For example, student name, number or grade point in student records. Many types of sorting algorithms are – quick sort, merge sort, bubble sort, selection sort etc.
- Searching- The searching problem deals with finding a given value, called a search key, in a given set. There are two types of searching algorithms – sequential searching and binary searching

Important Problem Types

- String Processing- a string is a sequence of characters from an alphabets. Text strings comprise letters, numbers and special characters. Bit strings comprise of zeros and once. String processing ALGs are important for computer languages and compiling issues. Several string processing algorithms are available like string matching, string comparison, string concatenation, finding string length etc.
- Graph Problems- A graph can be thought of as a collection of points called vertices, some of which are connected by line segments called edges. Graphs can be used for modeling a wide variety of real-life applications, including transportation and communication network, project scheduling and games. Examples are Traveling Salesman Problem, Graph Coloring etc.

Important Problem Types

- Combinatorial Problems- The Travelling Salesman Problem and Graph Coloring Problem are examples of 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 and has some desired property (e.g., maximize a value or minimize a cost)
- Geometric Problems- Geometric algorithms deals with geometric objects such as points, lines and polygons. Ancient Greeks developed solution for variety of geometric problems, including problems of constructing simple geometric shapes- triangles, circles and so on
- Numerical Problems- Numerical problems, are problems that involve mathematical objects of continuous nature: solving equations and systems of equations, computing definite integrals, evaluating functions and so on

Module - 1

Fundamental Data Structures

1. Stacks
2. Queues
3. Graphs
4. Trees
5. Sets and Dictionaries

Fundamental Data Structures

- A data structure can be defined as a particular scheme of organizing related data items
- Linear data structures are array and linked lists
- Array - A (one dimensional) array is a sequence of n items of the same data type that are stored contiguously in computer memory and made accessible by specifying a value of the array's index as shown in below figure-

Fundamental Data Structures

- 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.
- Two types of linked lists are singly linked lists and doubly linked lists. There working principles are shown in below diagram-

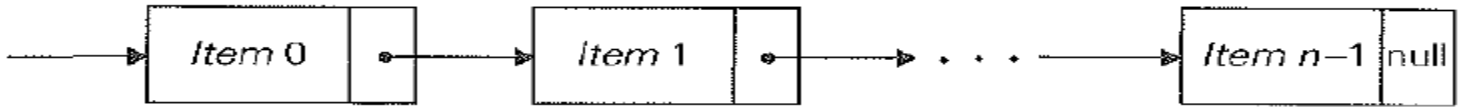


FIGURE 1.4 Singly linked list of n elements

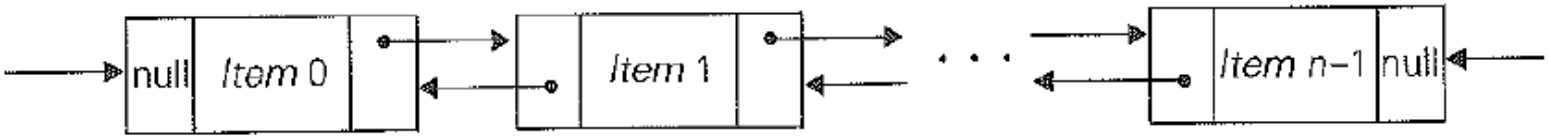


FIGURE 1.5 Doubly linked list of n elements

Fundamental Data Structures

- Stack- A Stack is a list in which insertions and deletions can be done only at the end. This end is called the top because a stack is usually visualized not horizontally but vertically. It operates in the “last-in-first-out” (LIFO) fashion.
- Queue- is a list from which elements are deleted from one end of the structure, called the front (dequeue), and new elements are added to the other end, called the rear (enqueue). It operates in the “first-in-first-out” (FIFO) fashion.

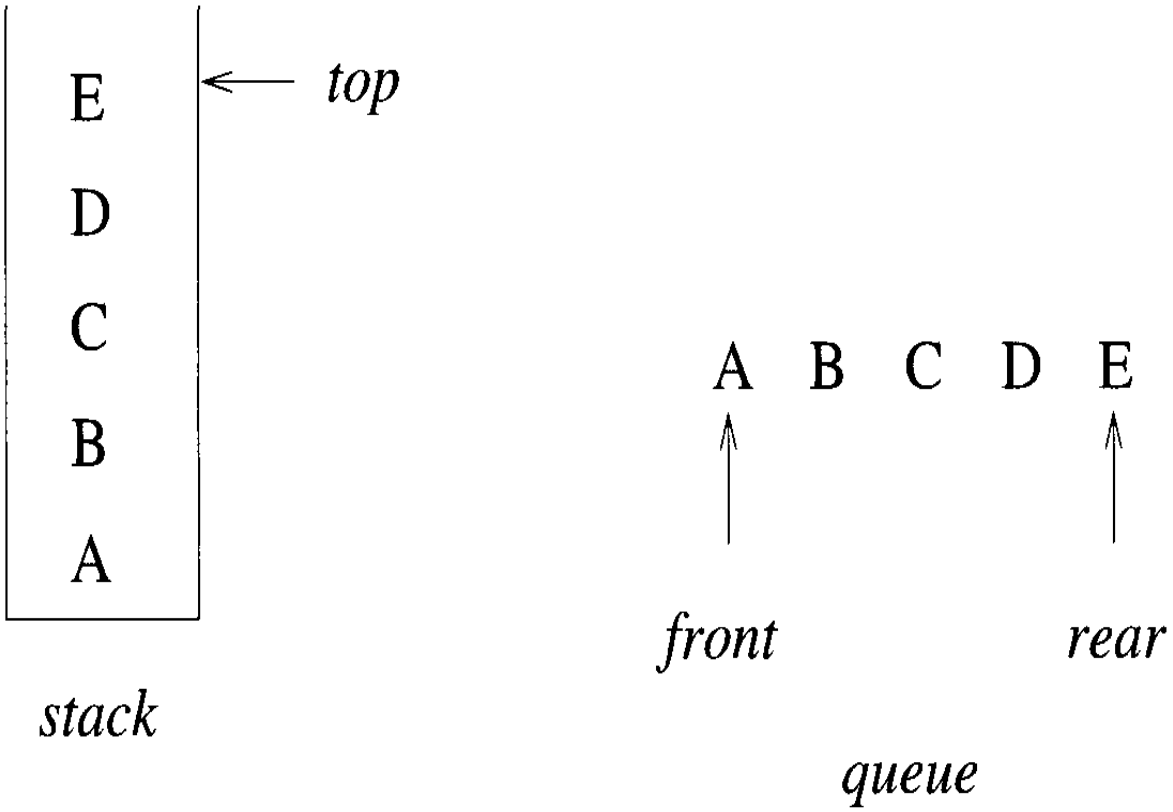
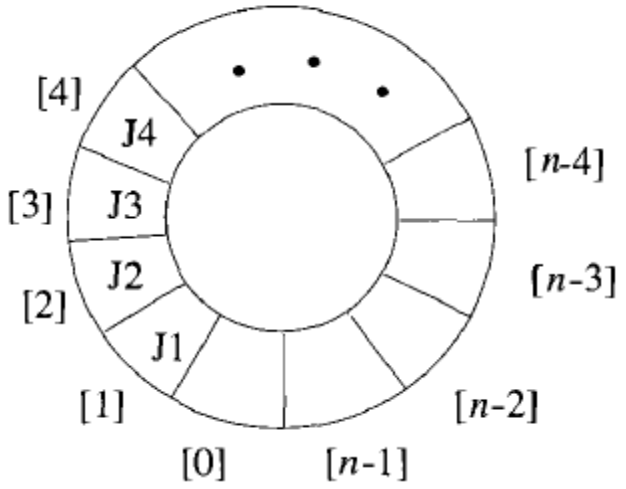


Figure 2.1 Example of a stack and a queue

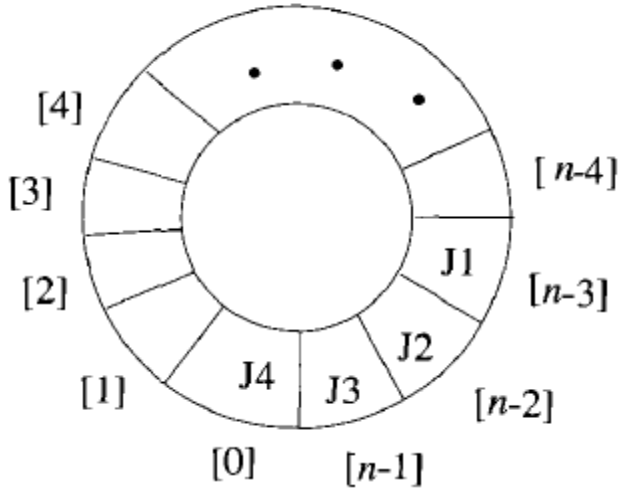
```
1   Algorithm Add(item)
2   // Push an element onto the stack. Return true if successful;
3   // else return false. item is used as an input.
4   {
5       if (top ≥ n − 1) then
6       {
7           write ("Stack is full!"); return false;
8       }
9       else
10      {
11          top := top + 1; stack[top] := item; return true;
12      }
13  }
```

```
1   Algorithm Delete(item)
2   // Pop the top element from the stack. Return true if successful
3   // else return false. item is used as an output.
4   {
5       if (top < 0) then
6       {
7           write ("Stack is empty!"); return false;
8       }
9       else
10      {
11          item := stack[top]; top := top - 1; return true;
12      }
13  }
```

Circular Queue



front = 0; rear = 4



front = n-4; rear = 0

```
1   Algorithm DeleteQ(item)
2   // Removes and returns the front element of the queue q[0 : n − 1].
3   {
4       if (front = rear) then
5       {
6           write ("Queue is empty!");
7           return false;
8       }
9       else
10      {
11          front := (front + 1) mod n; // Advance front clockwise.
12          item := q[front]; // Set item to front of queue.
13          return true;
14      }
15  }
```

(b) Deletion of an element

```
1   Algorithm AddQ(item)
2   // Insert item in the circular queue stored in q[0 : n − 1].
3   // rear points to the last item, and front is one
4   // position counterclockwise from the first item in q.
5   {
6       rear := (rear + 1) mod n; // Advance rear clockwise.
7       if (front = rear) then
8       {
9           write ("Queue is full!");
10          if (front = 0) then rear := n − 1;
11          else rear := rear − 1;
12          // Move rear one position counterclockwise.
13          return false;
14      }
15      else
16      {
17          q[rear] := item; // Insert new item.
18          return true;
19      }
20  }
```

(a) Addition of an element

Fundamental Data Structures

- Graphs- A graph G=<V,E> is defined by a pair of two sets: a finite set V of items called vertices and a set E of pairs of these items called edges

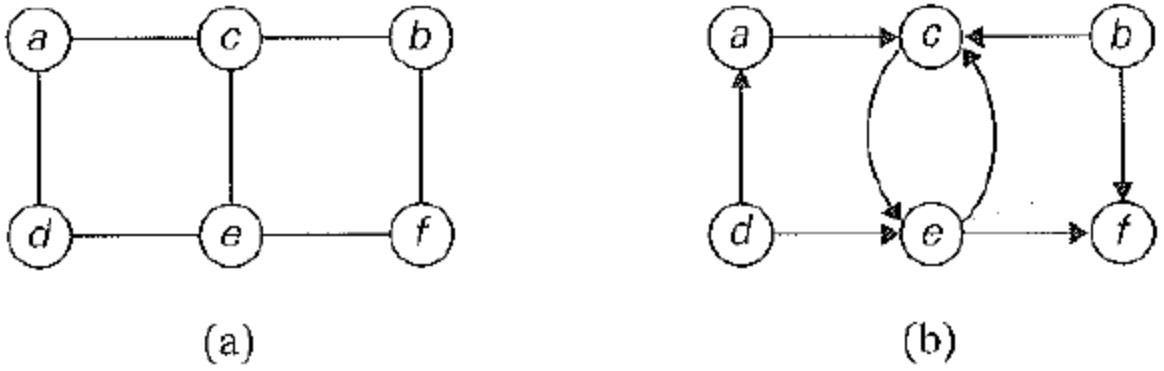
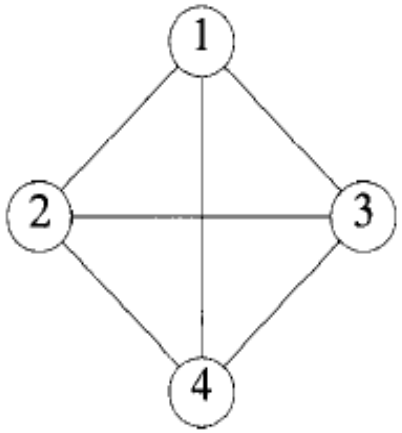
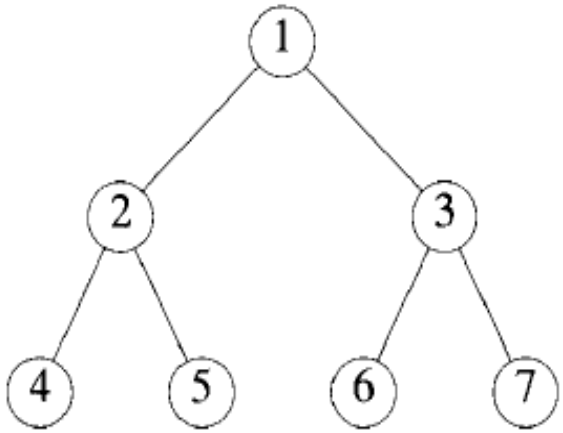


FIGURE 1.6 (a) Undirected graph. (b) Digraph.



(a) G_1



(b) G_2

(c) G_3

Figure 2.25 Three sample graphs

The set representations of these graphs are

$$V(G_1) = \{1,2,3,4\} \qquad E(G_1) = \{(1,2),(1,3),(1,4),(2,3),(2,4),(3,4)\}$$
$$V(G_2) = \{1,2,3,4,5,6,7\} \qquad E(G_2) = \{(1,2),(1,3),(2,4),(2,5),(3,6),(3,7)\}$$
$$V(G_3) = \{1,2,3\} \qquad E(G_3) = \{\langle 1,2\rangle,\langle 2,1\rangle,\langle 2,3\rangle\}$$

Fundamental Data Structures

- Graph Representation- A graphs for computer algorithms can be represented in two principal ways – the adjacency matrix and adjacency lists

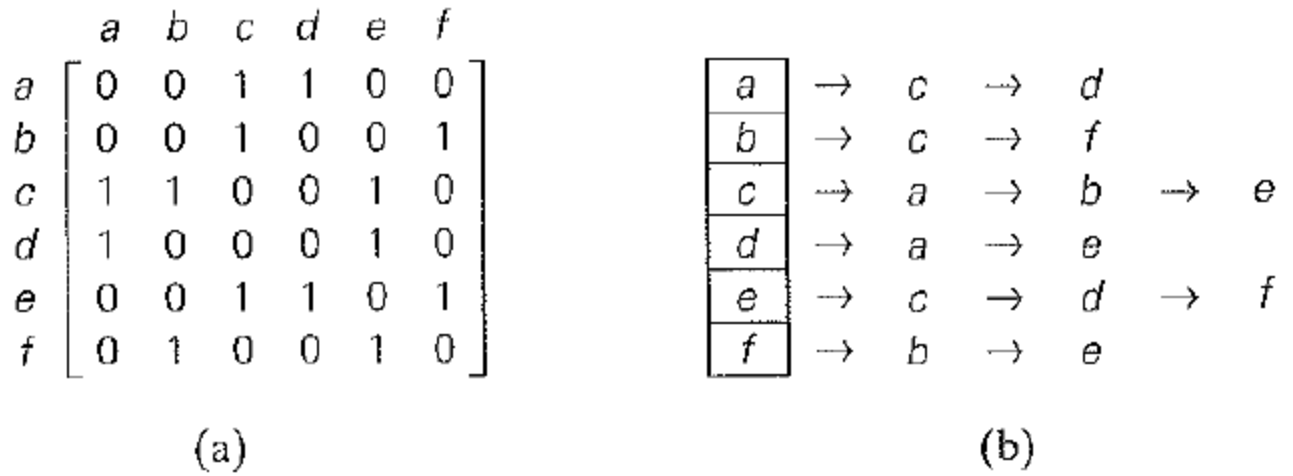


FIGURE 1.7 (a) Adjacency matrix and (b) adjacency lists of the graph in Figure 1.6a

Fundamental Data Structures

- Weighted Graph - A weighted graph is a graph with numbers assigned to its edges. These numbers are called weights or costs.

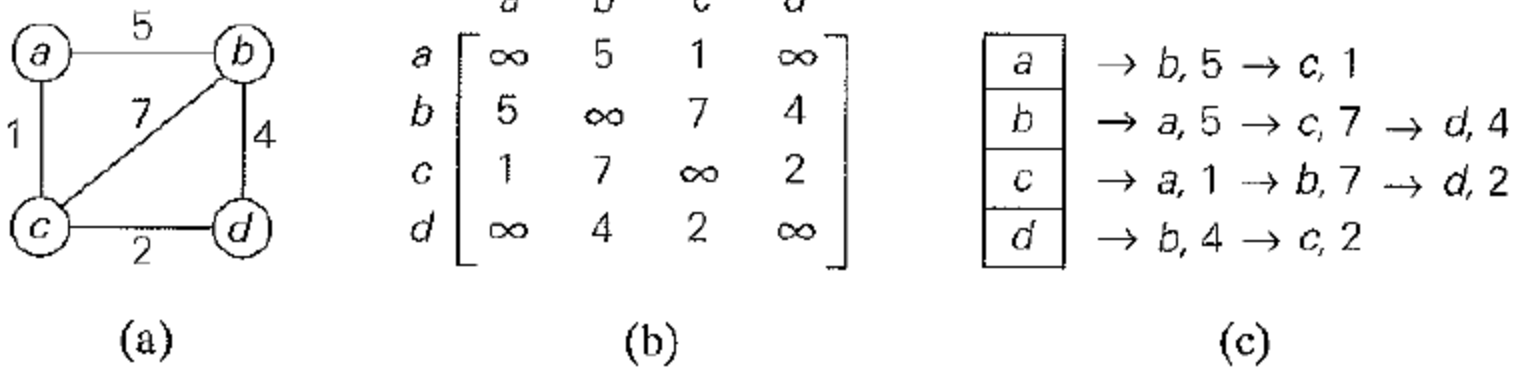


FIGURE 1.8 (a) Weighted graph. (b) Its weight matrix. (c) Its adjacency lists.

Fundamental Data Structures

- Trees – A tree is a connected acyclic graph. A graph that has no cycles but is not necessarily connected is called a forest.

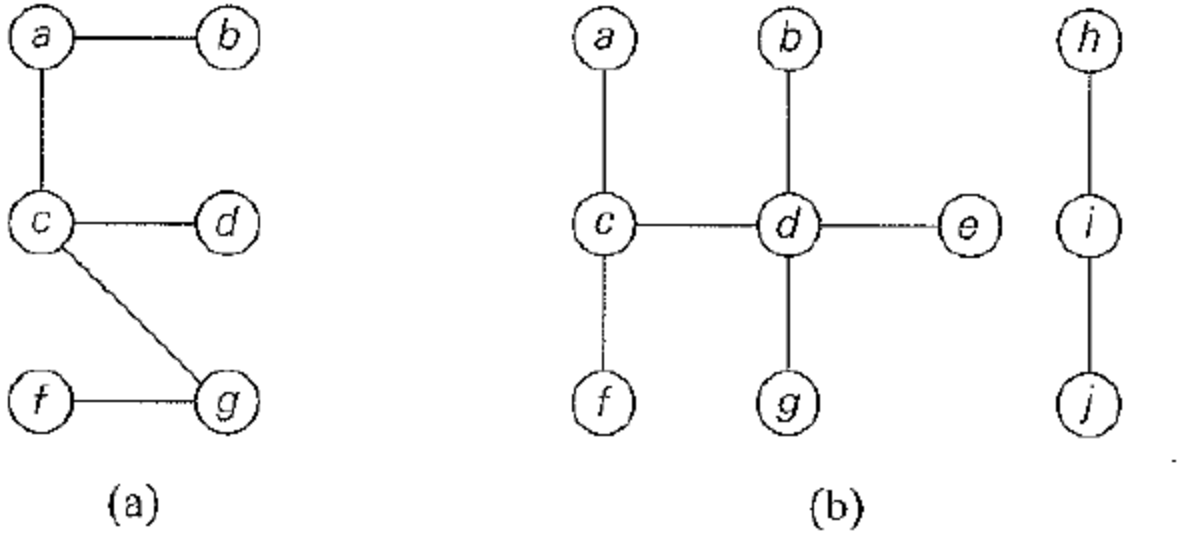


FIGURE 1.10 (a) Tree. (b) Forest.

Fundamental Data Structures

- Free Tree & Rooted Trees

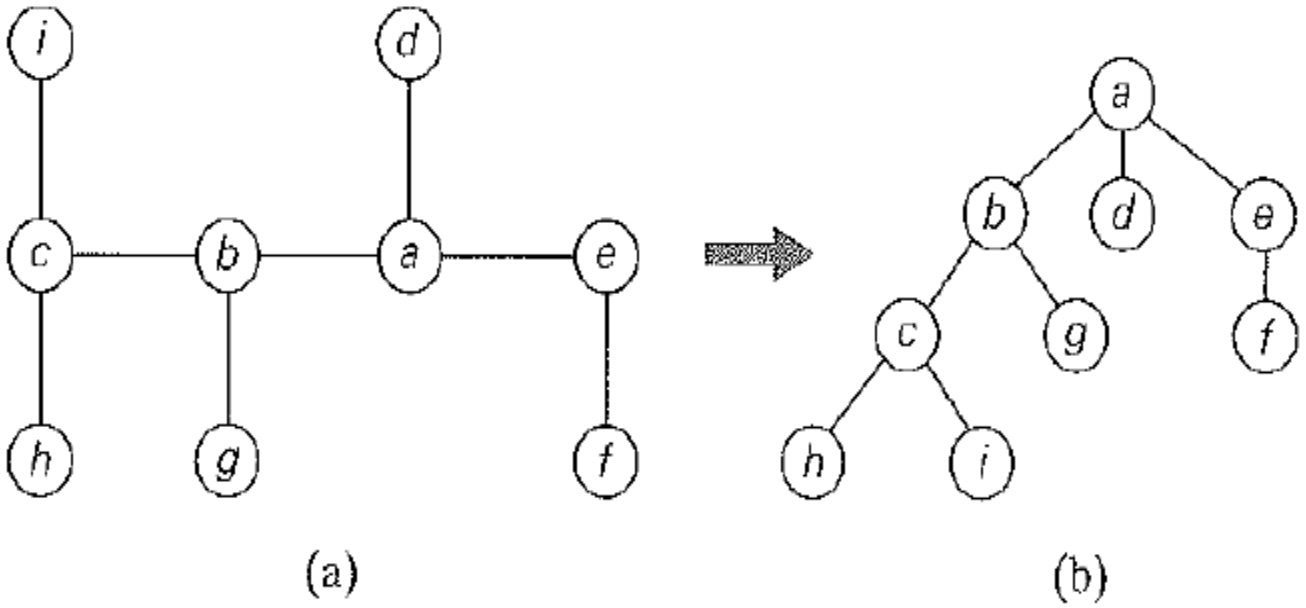
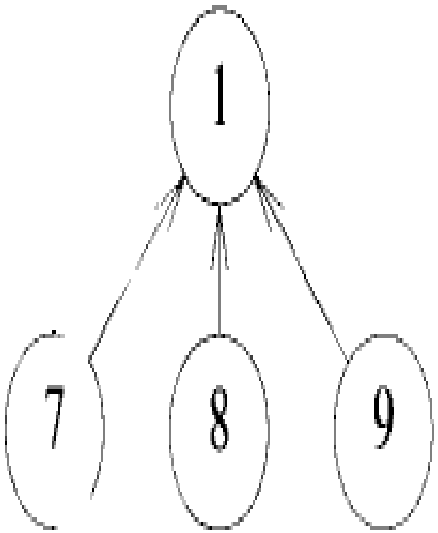


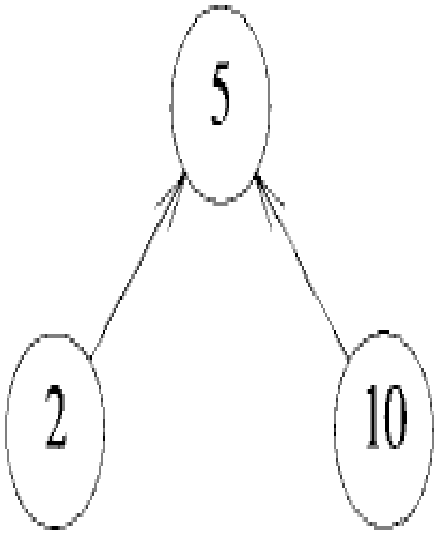
FIGURE 1.11 (a) Free tree. (b) Its transformation into a rooted tree.

Fundamental Data Structures

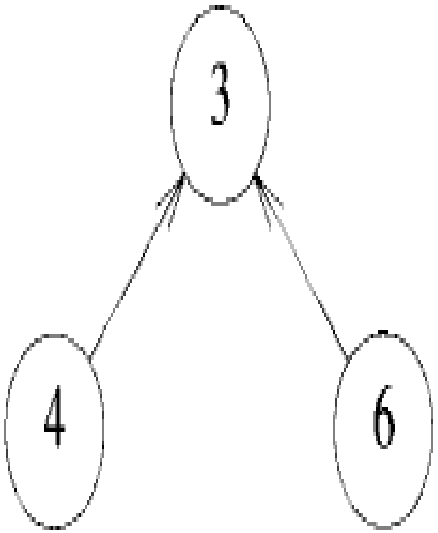
- Sets and Dictionaries- The notion of a set plays a central role in mathematics.
- A set - can be described as an unordered collection of distinct items called elements of the set
- The dictionary - a data structure that implements three operations that is searching for a given item, adding a new item and deleting an item on given set or multiset is called dictionary



S_1



S_2



S_3