MODULE-2

Module-2

BRUTE FORCE APPROACHES (contd..): Exhaustive Search (Travelling Salesman probem and Knapsack Problem).

DECREASE-AND-CONQUER: Insertion Sort, Topological Sorting.

DIVIDE AND CONQUER: Merge Sort, Quick Sort, Binary Tree Traversals, Multiplication of Large Integers and Strassen's Matrix Multiplication.

Module-2

BRUTE FORCE APPROACHES (contd..): Exhaustive Search

Exhaustive search is simply a brute-force approach to combinatorial problems. It suggests generating each and every element of the problem domain, selecting those of them that satisfy all the constraints, and then finding a desired element. Note that although the idea of exhaustive search is quite straightforward, its implementation typically requires an algorithm for generating certain combinatorial objects.

Traveling Salesman Problem

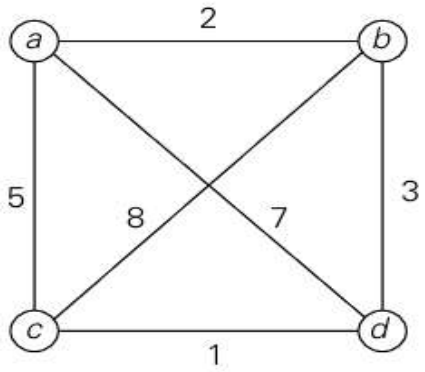
✓The problem asks to find the shortest tour through a given set of n cities that visits each city exactly once before returning to the city where it started.

✓The problem can be conveniently modeled by a weighted graph, with the graph's vertices representing the cities and the edge weights specifying the distances.

✓Then the problem can be stated as the problem of finding the shortest Hamiltonian circuit of the graph.

✓A Hamiltonian circuit is defined as a cycle that passes through all the vertices of the graph exactly once. It is named after the Irish mathematician Sir William Rowan Hamilton (1805–1865), who became interested in such cycles as an application of his algebraic discoveries.

Hamiltonian circuit can also be defined as a sequence of n+1 adjacent vertices $v_{i_0}, v_{i_1}, \ldots, v_{i_{n-1}}, v_{i_0}$, w here the first vertex of the sequence is the same as the last one and all the other n − 1 vertices are distinct.



Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	optimal
$a \rightarrow d \rightarrow b \rightarrow c \rightarrow a$	$l = 7 + 3 + 8 + 5 = 23$	
$a \rightarrow d \rightarrow c \rightarrow b \rightarrow a$	$l = 7 + 1 + 8 + 2 = 18$	

FIGURE 3.7 Solution to a small instance of the traveling salesman problem by exhaustive search.

Knapsack Problem

Here is another well-known problem in algorithmics. Given n items of known weights w1,w2,...,wn and values v1,v2,...,vn and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

Or a transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity.

The decrease-and-conquer

- Technique is based on exploiting the relationship between a solution to a given instance of a problem and a solution to its smaller instance.
- Once such a relationship is established, it can be exploited either top down or bottom up.
- The former leads naturally to a recursive implementation, an ultimate implementation may well be nonrecursive.
- The bottom-up variation is usually implemented iteratively, starting with a solution to the smallest instance of the problem; it is called sometimes the incremental approach. There are three major variations of decrease-and-conquer:

There are three major variations of decrease-and-conquer:

1. Decrease by a constant
2. Decrease by a constant factor
3. Variable size decrease

In the decrease-by-a-constant variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. Typically, this constant is equal to one (Figure 4.1), although other constant size reductions do happen occasionally.

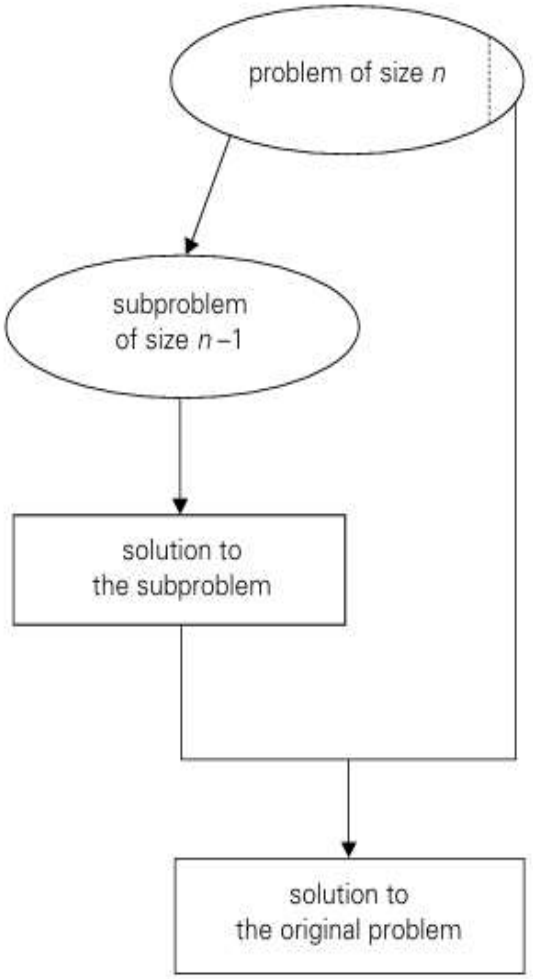


FIGURE 4.1 Decrease-(by one)-and-conquer technique.

The decrease-by-a-constant-factor technique suggests reducing a problem instance by the same constant factor on each iteration of the algorithm. In most applications, this constant factor is equal to two.

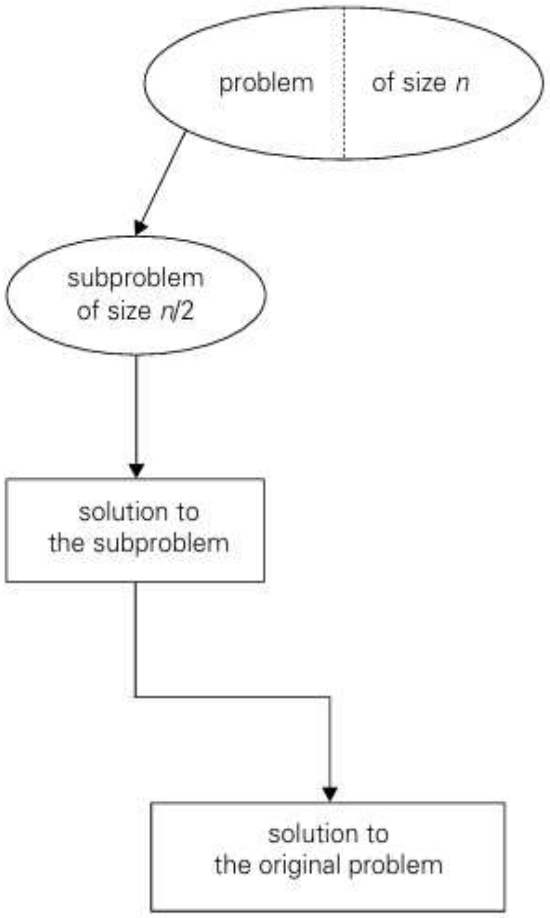


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

❖**variable-size-decrease** variety of decrease-and-conquer, the size-reduction pattern varies from one iteration of an algorithm to another.

❖Euclid's algorithm for computing the greatest common divisor provides a good ex ample of such a situation.

❖ Recall that this algorithm is based on the formula

gcd(m, n) =gcd(n, m mod n)

❖Though the value of the second argument is always smaller on the right-hand side than on the left-hand side, it decreases neither by a constant nor by a constant factor.

Insertion Sort

❖In this ,we consider an application of the decrease-by-one technique to sorting an array A[0..n − 1].

❖Following the technique's idea, we assume that the smaller problem of sorting the array A[0..n − 2] has already been solved to give us a sorted array of size n − 1: A[0]≤...≤A[n−2].

❖all we need is to find an appropriate position for A[n − 1] among the sorted elements and insert it there.

❖ This is usually done by scanning the sorted sub array from right to left until the first element smaller than or equal to A[n − 1] is encountered to insert A[n − 1] right after that element.

❖The resulting algorithm is called straight insertion sort or simply insertion sort.

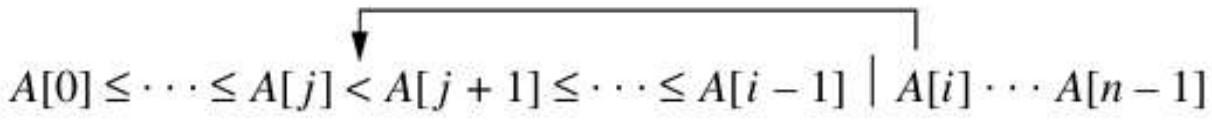
Though insertion sort is clearly based on a recursive idea, it is more efficient to implement this algorithm bottom up, i.e., iteratively. Algorithm starting with A[1] and ending with A[n−1], A[i] is inserted in its appropriate place among the first i elements of the array that have been already sorted

89 |	**45**	68	90	29	34	17
45	89 |	**68**	90	29	34	17
45	68	89 |	**90**	29	34	17
45	68	89	90 |	**29**	34	17
29	45	68	89	90 |	**34**	17
29	34	45	68	89	90 |	**17**
17	29	34	45	68	89	90

FIGURE 4.4 Example of sorting with insertion sort. A vertical bar separates the sorted part of the array from the remaining elements; the element being inserted is in bold.

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

//Sorts a given array by insertion sort
//Input: An array $A[0..n-1]$ of n orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 1$ **to** $n-1$ **do**
    $v \leftarrow A[i]$
    $j \leftarrow i-1$
    while $j \geq 0$ **and** $A[j] > v$ **do**
        $A[j+1] \leftarrow A[j]$
        $j \leftarrow j-1$
    $A[j+1] \leftarrow v$



$$C_{worst}(n) = \sum_{i=1}^{n-1} \sum_{j=0}^{i-1} 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \in \Theta(n^2).$$

$$C_{best}(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n).$$

$$C_{avg}(n) \approx \frac{n^2}{4} \in \Theta(n^2).$$

Topological Sorting

- **A directed graph,or digraph** for short, is a graph with directions specified for all its edges.
- The adjacency matrix and adjacency lists are still two principal means of representing a digraph.
- There are only two notable differences between undirected and directed graphs in representing them:

(1) the adjacency matrix of a directed graph does not have to be symmetric;

(2) an edge in a directed graph has just one (not two) corresponding nodes in the digraph's adjacency lists.

❖Depth-first search and breadth-first search are principal traversal algorithms for traversing digraphs as well, but the structure of corresponding forests can be more complex than for undirected graphs.

❖A directed cycle in a digraph is a sequence of three or more of its vertices that starts and ends with the same vertex and in which every vertex is connected to its immediate predecessor by an edge directed from the predecessor to the successor.

Divide-and-Conquer

Divide-and-conquer algorithms work according to the following general plan:

1. A problem is divided into several subproblems of the same type, ideally of about equal size.
2. The subproblems are solved (typically recursively, though sometimes a different algorithm is employed, especially when subproblems become small enough).
3. If necessary, the solutions to the subproblems are combined to get a solution to the original problem.

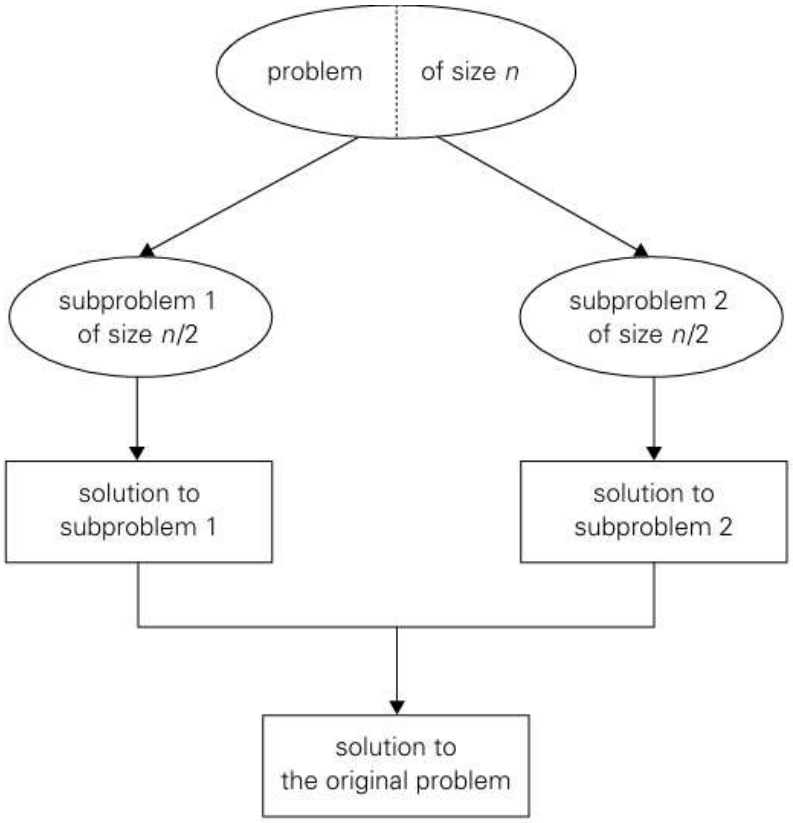


FIGURE 5.1 Divide-and-conquer technique (typical case).

Mergesort

Mergesort is a perfect example of a successful application of the divide-and conquer technique. It sorts a given array A[0..n − 1]by dividing it into two halves A[0.. n/2 −1] and A[n/2 ..n−1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

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

//Sorts array $A[0..n-1]$ by recursive mergesort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
if $n > 1$
    copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
    copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
    Mergesort$(B[0..\lfloor n/2 \rfloor - 1])$
    Mergesort$(C[0..\lceil n/2 \rceil - 1])$
    Merge(B, C, A) //see below

✓The merging of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged.

✓ The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from.

✓This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array

ALGORITHM *Merge*$(B[0..p-1], C[0..q-1], A[0..p+q-1])$

//Merges two sorted arrays into one sorted array
//Input: Arrays $B[0..p-1]$ and $C[0..q-1]$ both sorted
//Output: Sorted array $A[0..p+q-1]$ of the elements of B and C
$i \leftarrow 0;\ j \leftarrow 0;\ k \leftarrow 0$
while $i < p$ **and** $j < q$ **do**
    if $B[i] \leq C[j]$
        $A[k] \leftarrow B[i];\ i \leftarrow i+1$
    else $A[k] \leftarrow C[j];\ j \leftarrow j+1$
    $k \leftarrow k+1$
if $i = p$
    copy $C[j..q-1]$ to $A[k..p+q-1]$
else copy $B[i..p-1]$ to $A[k..p+q-1]$

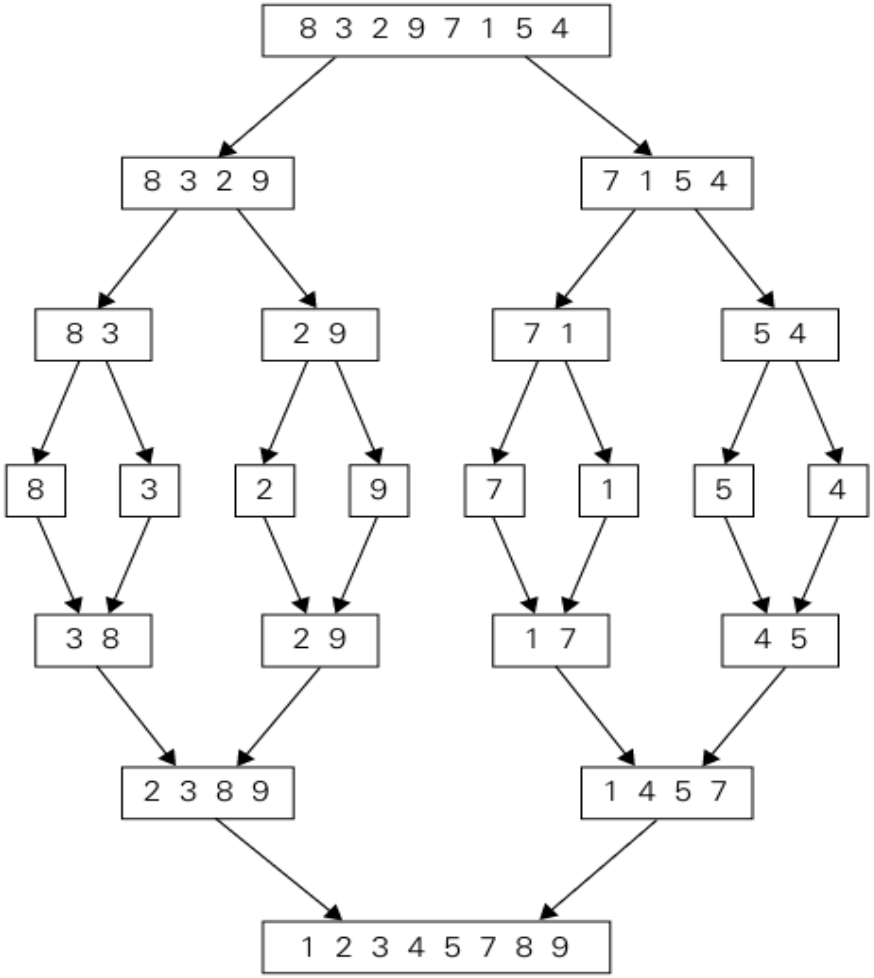


FIGURE 5.2 Example of mergesort operation.

Quicksort

•Quicksort is the other important sorting algorithm that is based on the divide-and conquer approach. Unlike mergesort, which divides its input elements according to their position in the array, quicksort divides them according to their value.

•A partition is an arrangement of the array's elements so that all the elements to the left of some element A[s] are less than or equal to A[s], and all the elements to the right of A[s]are greater than or equal to it

$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are} \le A[s]} \quad A[s] \quad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are} \ge A[s]}$$

✓Obviously, after a partition is achieved, A[s] will be in its final position in the sorted array, and we can continue sorting the two sub arrays to the left and to the right of A[s] independently.

✓Note the difference with mergesort: there, the division of the problem into two subproblems is immediate and the entire work happens in combining their solutions; here, the entire work happens in the division stage, with no work required to combine the solutions to the subproblem

ALGORITHM *Quicksort*$(A[l..r])$

//Sorts a subarray by quicksort
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r
//Output: Subarray $A[l..r]$ sorted in nondecreasing order
if $l < r$
  $s \leftarrow$ *Partition*$(A[l..r])$ //s is a split position
  Quicksort$(A[l..s-1])$
  Quicksort$(A[s+1..r])$

ALGORITHM *HoarePartition*$(A[l..r])$

//Partitions a subarray by Hoare's algorithm, using the first element
// as a pivot
//Input: Subarray of array $A[0..n-1]$, defined by its left and right
// indices l and r $(l < r)$
//Output: Partition of $A[l..r]$, with the split position returned as
// this function's value
$p \leftarrow A[l]$
$i \leftarrow l;\ j \leftarrow r+1$
repeat
  repeat $i \leftarrow i+1$ **until** $A[i] \geq p$
  repeat $j \leftarrow j-1$ **until** $A[j] \leq p$
  swap$(A[i], A[j])$
until $i \geq j$
swap$(A[i], A[j])$ //undo last swap when $i \geq j$
swap$(A[l], A[j])$
return j

Binary Tree Traversals and Related Properties

A binary tree T is defined as a finite set of nodes that is either empty or consists of a root and two disjoint binary trees TL and TR called, respectively , the left and right subtree of the root.

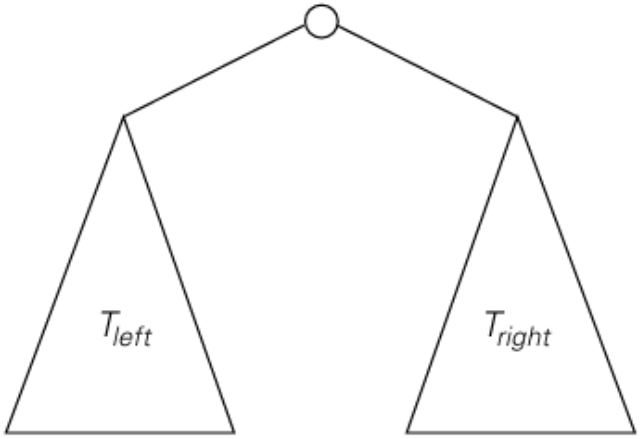


FIGURE 5.4 Standard representation of a binary tree.

Since the definition itself divides a binary tree into two smaller structures of the same type, the left subtree and the right subtree, many problems about binary trees can be solved by applying the divide-and-conquer technique.

Example

Let us consider a recursive algorithm for computing the height of a binary tree. Recall that the height is defined as the length of the longest path from the root to a leaf. Hence, it can be computed as the maximum of the heights of the root's left and right subtrees plus 1. (We have to add 1 to account for the extra level of the root.)

Also note that it is convenient to define the height of the empty tree as −1.

ALGORITHM *Height*(T)

//Computes recursively the height of a binary tree

//Input: A binary tree T

//Output: The height of T

if $T = \emptyset$ **return** -1

else return $\max\{Height(T_{left}), Height(T_{right})\} + 1$

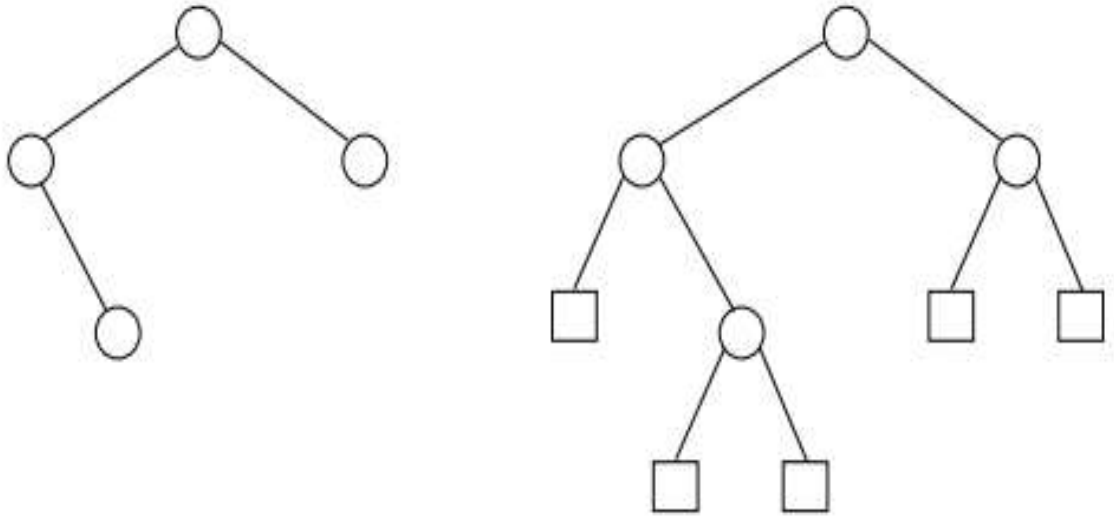


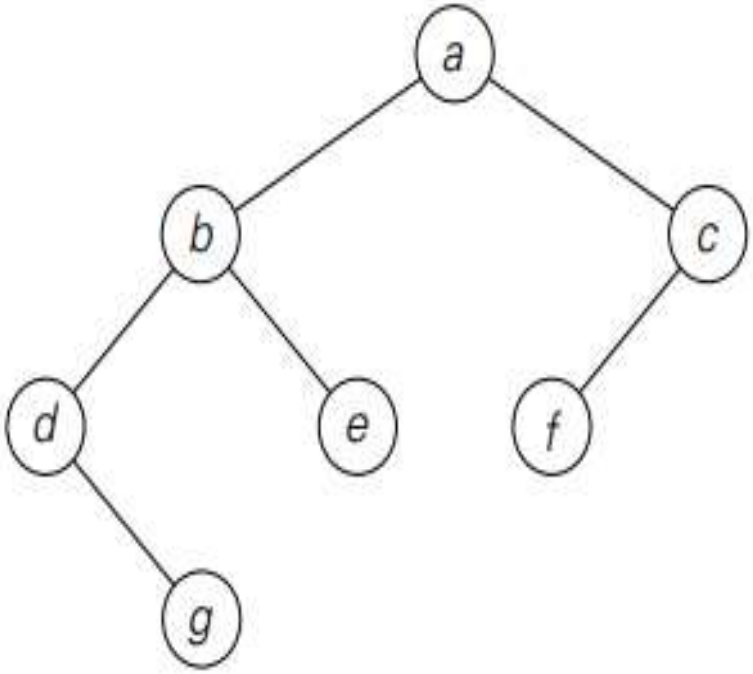
FIGURE 5.5 Binary tree (on the left) and its extension (on the right). Internal nodes are shown as circles; external nodes are shown as squares.

The most important divide-and-conquer algorithms for binary trees are the three classic traversals:

1. Preorder
2. Inorder
3. Postorder

All three traversals visit nodes of a binary tree recursively, i.e., by visiting the tree's root and its left and right subtrees. They differ only by the timing of the root's visit.

- ✓ In the preorder traversal, the root is visited before the left and right subtrees are visited .
- ✓ In the inorder traversal, the root is visited after visiting its left subtree but before visiting the right subtree.
- ✓ In the postorder traversal, the root is visited after visiting the left and right subtrees .



preorder: *a, b, d, g, e, c, f*
inorder: *d, g, b, e, a, f, c*
postorder: *g, d, e, b, f, c, a*

FIGURE 5.6 Binary tree and its traversals.

Multiplication of Large Integers and Strassen's Matrix Multiplication

Multiplication of Large Integers

- Some applications, notably modern cryptography, require manipulation of integers that are over 100 decimal digits long.
- Since such integers are too long to fit in a single word of a modern computer, they require special treatment.
- This practical need supports investigations of algorithms for efficient manipulation of large integers.

- In this section, we outline an interesting algorithm for multiplying such numbers.
- Obviously, if we use the conventional pen-and-pencil algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n2 digit multiplications.
- Though it might appear that it would be impossible to design an algorithm with fewer than n^2 digit multiplications, this turns out not to be the case. The miracle of divide-and-conquer comes to the rescue to accomplish this feat.

To demonstrate the basic idea of the algorithm, let us start with a case of two-digit integers, 71 and 56

Time Complexity = $O(n^{1.585})$

Strassen's Matrix Multiplication

✓Now that we have seen that the divide-and-conquer approach can reduce the number of one-digit multiplications in multiplying two integers, we should not be surprised that a similar feat can be accomplished for multiplying matrices.

✓Such an algorithm was published by V. Strassen in 1969. The principal insight of the algorithm lies in the discovery that we can find the product C of two 2 × 2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm.

This is accomplished by using the following formulas:

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$

$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix},$$

where

$$m_1 = (a_{00} + a_{11}) * (b_{00} + b_{11}),$$

$$m_2 = (a_{10} + a_{11}) * b_{00},$$

$$m_3 = a_{00} * (b_{01} - b_{11}),$$

$$m_4 = a_{11} * (b_{10} - b_{00}),$$

$$m_5 = (a_{00} + a_{01}) * b_{11},$$

$$m_6 = (a_{10} - a_{00}) * (b_{00} + b_{01}),$$

$$m_7 = (a_{01} - a_{11}) * (b_{10} + b_{11}).$$