MCA-20106 DESIGN AND ANALYSIS OF ALGORITHMS

Instruction: 4Periods/week **Time: 3Hours** **Credits: 4**
Internal: 25Marks **External: 75Marks** **Total: 100Marks**

UNIT I

Introduction: Fundamentals of algorithmic problem solving, important problem types.

Fundamentals of analysis of algorithms and efficiency: Analysis framework, Asymptotic Notations and Basic Efficiency classes, Mathematical Analysis of Non-recursive Algorithms, Mathematical Analysis of recursive Algorithms, Empirical Analysis of Algorithms, Algorithm Visualization.

Brute Force: Selection Sort and Bubble sort, Sequential Search and Exhaustive Search.

UNIT II

Divide-and-Conquer: Merge Sort, Quick sort, Binary Search, Binary Tree Traversals and Related Properties.

Decrease-and-Conquer: Insertion Sort, Depth-First Search and Breadth-First Search-Topological Sorting, Decrease-by-a-Constant-Factor Algorithms.

Transform-and-Conquer: Balanced Search Trees, Heaps and Heap sort, Problem Reduction.

UNIT III

Dynamic Programming: Warshall's and Floyd's Algorithm, Optimal Binary Search Trees, The 0/1 Knapsack Problem and Memory Functions.

Greedy Technique: Prim's Algorithm, Kruskal's Algorithm, Dijkstra's Algorithm

UNIT IV

Limitations of Algorithm Power: Decision Trees, P, NP and NP- complete problems.

Coping with the Limitations of Algorithms Power: Backtracking-n-queens problem, Hamiltonian circuit problem, Subset-sum problem. Branch-and-Bound- The Knapsack Problem, Travelling salesperson problem, Approximation Algorithms for NP-hard Problems.

Text Book:

1. Introduction to Design & Analysis of Algorithms by Anany Levitin, Pearson Education, New Delhi, 2003

Reference Books:

1. Introduction to Algorithms by Thomas H. Corman, Charles E. Leiserson, Ron ald R. Rivest& Clifford Stein, Prentice Hall of India, NewDelhi.
2. The Design and Analysis of computer Algorithms, Aho, Hopcroft & Ullman, Pearson Education, New Delhi, 2003
3. Fundamentals of algorithmics, Gilles Brassard & Paul Bratley, Prentice Hall of India, NewDelhi

UNIT-I

INTRODUCTION

Algorithm

Definition: An algorithm is a set of steps of operations to solve a problem performing calculation, data processing, and automated reasoning tasks. An algorithm is an efficient method that can be expressed within finite amount of time and space.

Algorithm Design

The important aspects of algorithm design include creating an efficient algorithm to solve a problem in an efficient way using minimum time and space.

To solve a problem, different approaches can be followed. Some of them can be efficient with respect to time consumption, whereas other approaches may be memory efficient.

Characteristics of Algorithms

The main characteristics of algorithms are as follows −

- Effectiveness: Algorithms must have a unique name
- Input and Output: Algorithms should have explicitly defined set of inputs and outputs
- Definiteness: Algorithms are well-ordered with unambiguous operations
- Finiteness: Algorithms halt in a finite amount of time. Algorithms should not run for infinity, i.e., an algorithm must end at some point

Pseudo-code for Expressing Algorithms

Algorithm is basically sequence of instructions written in simple English language. Based on algorithm there are 2 more representations used by programmer, these are flow chart and pseudo-code.

Flow chart: is a graphical representation of an algorithm.

Pseudo-code: is a representation of an algorithm in which instruction sequence can be given with the help of programming constructs.

Pseudo-code gives a high-level description of an algorithm without the ambiguity associated with plain text but also without the need to know the syntax of a particular programming language.

The running time can be estimated in a more general manner by using Pseudo-code to represent the algorithm as a set of fundamental operations which can then be counted.

Steps for writing an algorithm:

1. An algorithm is a procedure. It has two parts; the first part is **head** and the second part is
   body.

2. The Head section consists of keyword **Algorithm** and Name of the algorithm with parameter list.
   E.g: Algorithm name1(p1, p2,…,p3)
   The head section also has the following:
   //Problem Description:

   //Input:

   //Output:

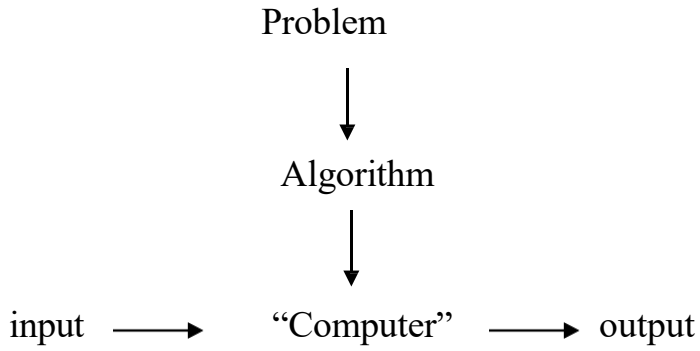
3. In the body of an algorithm various programming constructs like **if, for, while** and somestatements like assignments are used.
4. The compound statements may be enclosed with **{** and **}** brackets. **if**, **for**, **while** can beclosed by **endif**, **endfor**, **endwhile** respectively. Proper indention is must for block.
5. Comments are written using // at the beginning.
6. The **identifier** should begin by a letter and not by digit. It contains alpha numeric lettersafter first letter. No need to mention data types.
7. The left arrow **"←"** used as assignment operator. E.g. v←10.
8. **Boolean** operators (TRUE, FALSE), **Logical** operators (AND, OR, NOT) and **Relational** operators (<,<=, >, >=,=, ≠, <>) are also used.
9. Input and Output can be done using **read** and **write**.
10. **Array[]**, **if then else condition**, **branch** and **loop** can be also used in algorithm.

Introduction

Notation of an Algorithm

Define Algorithm:

An Algorithm is a sequence of unambiguous instructions for solving a problem that is for obtaining a required output for any legitimate (valid) input in a finite amount of time.



Example of an algorithm finding GCD of two numbers m and n

Euclid's algorithm for computing gcd(m,n)

Step 1: If n=0, return the value of '**m**' as the answer and stop; otherwise, Proceed to Step 2.

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

Step 3: Assign the value of **n** to **m** and the value of **r** to **n**. Goto Step 1.

Pseudo-code of this algorithm

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

Tracing:
Euclid(24,18) → Euclid(18,6) → Euclid(6,0)
Hence gcd(24,18) is 6.

Fundamentals of Algorithmic Problem Solving

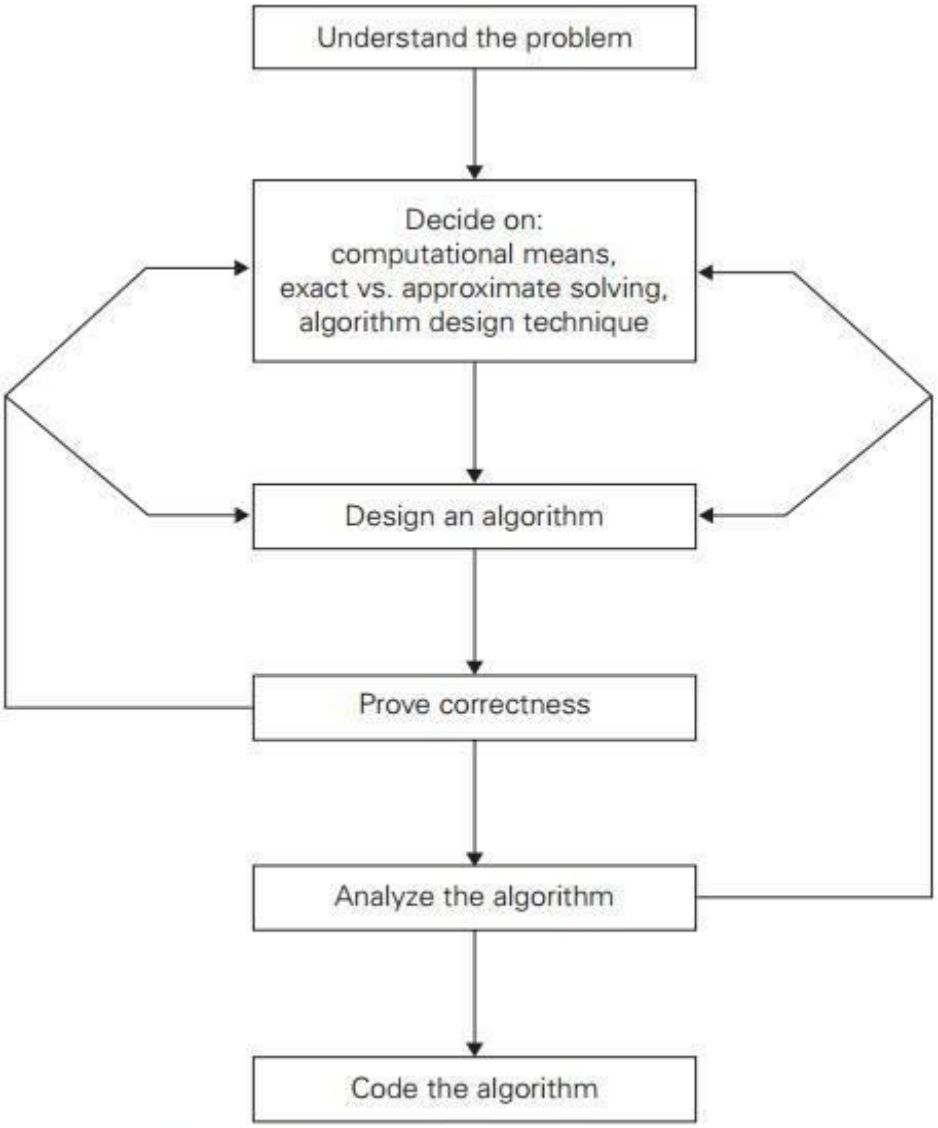


FIGURE 1.2 Algorithm design and analysis process.

(i) Understanding the Problem

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

(ii) Decision making

The Decision making is done on the following:

(a) Ascertaining the Capabilities of the Computational Device

- In ***random-access machine*** (***RAM***), instructions are executed one after another (Thecentral assumption is that one operation at a time). Accordingly, algorithms designed to be executed on such machines are called ***sequential algorithms***.
- In some newer computers, operations are executed **concurrently**, i.e., in parallel. Algorithms that take advantage of this capability are called ***parallel algorithms***.
- Choice of computational devices like Processor and memory is mainly based on **space and time efficiency.**

(b) Choosing between Exact and Approximate Problem Solving

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

(c) Algorithm Design Techniques

- An ***algorithm design technique*** (or "strategy" or "paradigm") is a general approach to solving problems algorithmically that is applicable to a variety of problems from different areas of computing.

Algorithms*+ Data Structures = Programs*

- Though Algorithms and Data Structures are independent, but they are combined together to develop program. Hence the choice of proper data structure is required before designing the algorithm.
- **Implementation** of algorithm is possible only with the help of Algorithms and Data Structures.
- **Algorithmic strategy / technique / paradigm** are a general approach by which many problems can be solved algorithmically. E.g., Brute Force, Divide and Conquer, Dynamic Programming, Greedy Technique and so on.

(iii) Methods of Specifying an Algorithm

There are three ways to specify an algorithm. They are**:**

a. **Natural language**
b. **Pseudocode**
c. **Flowchart**

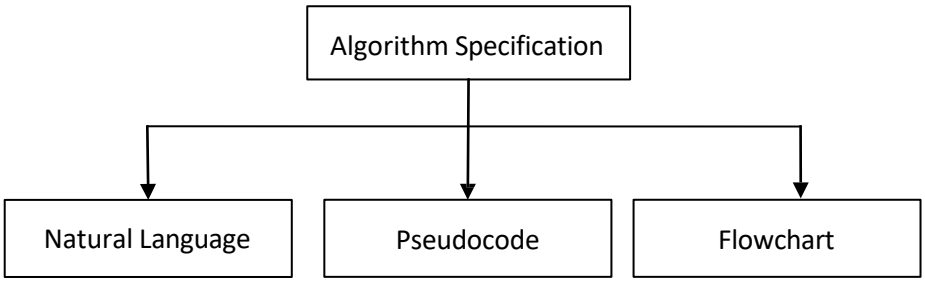


FIGURE 1.3 Algorithm Specifications

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

a. Natural Language

It is very simple and easy to specify an algorithm using natural language. But many times specification of algorithm by using natural language is not clear and thereby we get brief specification.

Step 1: Read the first number, say a.
Step 2: Read the first number, say b.
Step 3: Add the above two numbers and store the result in c.
Step 4: Display the result from c.

Example: An algorithm to perform addition of two numbers.

Such a specification creates difficulty while actually implementing it. Hence many programmers prefer to have specification of algorithm by means of Pseudocode.

b. Pseudocode

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

ALGORITHM *Sum(a,b)*

//Problem Description: This algorithm performs addition of two numbers
//Input: Two integers a and b
//Output: Addition of two integers
c←a+b

This specification is more useful for implementation of any language.

c. Flowchart

In the earlier days of computing, the dominant method for specifying algorithms was a ***flowchart***, this representation technique has proved to be inconvenient.

Flowchart is a graphical representation of an algorithm. It is a a method of expressing an algorithmby a collection of connected geometric shapes containing descriptions of the algorithm's steps.

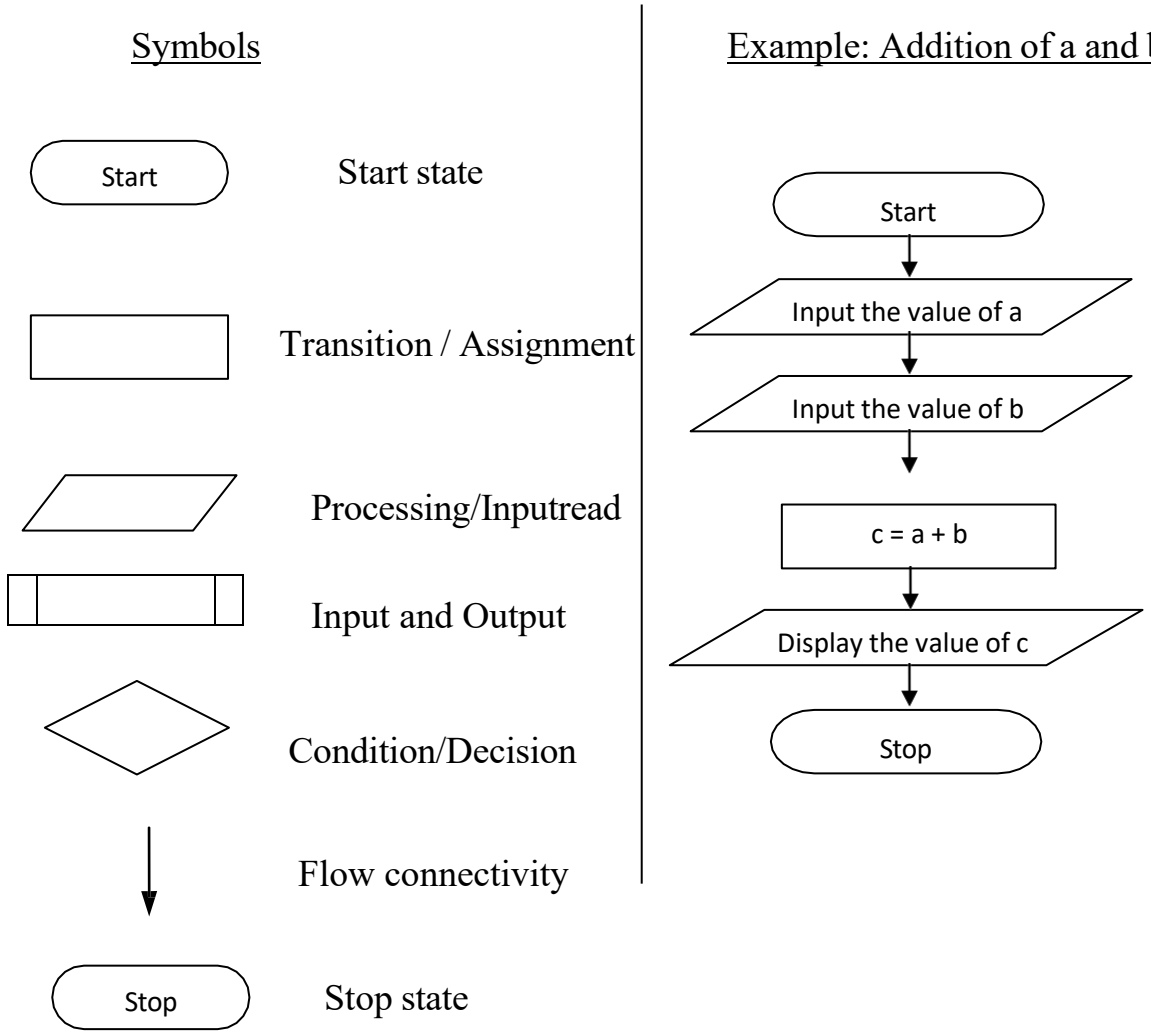


FIGURE 1.4 Flowchart symbols and Example for two integer addition.

(iv) Proving an Algorithm's Correctness

- Once an algorithm has been specified then its ***correctness*** must be proved.
- An algorithm must yields a required **result** for every legitimate input in a finite amount oftime.
- For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd*(m, n)* = gcd*(n, m* mod *n)*.

- A common technique for proving correctness is to use **mathematical induction** because an algorithm's iterations provide a natural sequence of steps needed for such proofs.

- The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The **error** produced by the algorithm should not exceed a predefinedlimit.

(v) Analyzing an Algorithm

For an algorithm the most important is ***efficiency***. In fact, there are two kinds of algorithmefficiency. They are:

- ***Time efficiency***, indicating how fast the algorithm runs, and
- ***Space efficiency***, indicating how much extra memory it uses.
- The efficiency of an algorithm is determined by measuring both time efficiency and spaceefficiency.
- So factors to analyze an algorithm are:
    - Time efficiency of an algorithm
    - Space efficiency of an algorithm
    - Simplicity of an algorithm
    - Generality of an algorithm

(vi) Coding an Algorithm

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

Important Problem Types

We are going to introduce the most important problem types:

- Sorting
- Searching
- String processing
- Graph problems
- Combinatorial problems
- Geometric problems
- Numerical problems

(i) Sorting

- The ***sorting problem*** is to rearrange the items of a given list in nondecreasing (ascending)order.
- Sorting can be done on numbers, characters, strings or records.
- To sort student records in alphabetical order of names or by student number or by studentgrade-point average. Such a specially chosen piece of information is called a ***key***.
- An algorithm is said to be **in-place** if it does not require extra memory, E.g., Quick sort.
- A sorting algorithm is called **stable** if it preserves the relative order of any two equalelements in its input.

(ii) Searching

- The ***searching problem*** deals with finding a given value, called a ***search key***, in a given set.
- There are plenty of searching algorithms to choose from.
- E.g., Ordinary Linear search and fast binary search.

(iii) String processing

- A ***string*** is a sequence of characters from an alphabet.
- Strings comprise letters, numbers, and special characters; bit strings, which comprise zeros and ones; and gene sequences, which can be modeled by strings of characters from the four- character alphabet {A, C, G, T}. It is very useful in bioinformatics.
- Searching for a given word in a text is called string matching.

(iv) Graph problems

- A ***graph*** is a collection of points called vertices, some of which are connected by linesegments called edges.
- Some of the graph problems are graph traversal, shortest path algorithm, topological sort, traveling salesman problem and the graph-coloring problem and so on.

(v) Combinatorial problems

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

(vi) Geometric problems

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

(vii) Numerical problems

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

FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

The efficiency of an algorithm can be in terms of time and space. The algorithm efficiencycan be analyzed by the following ways.

a. Analysis Framework.
b. Asymptotic Notations and its properties.
c. Mathematical analysis for Recursive algorithms.
d. Mathematical analysis for Non-recursive algorithms.

Analysis Framework

There are two kinds of efficiencies to analyze the efficiency of any algorithm. They are:

- ***Time efficiency***, indicating how fast the algorithm runs, and
- ***Space efficiency***, indicating how much extra memory it uses.

The algorithm analysis framework consists of the following:

- Measuring an Input's Size
- Units for Measuring Running Time
- Orders of Growth
- Worst-Case, Best-Case, and Average-Case Efficiencies

(i) Measuring an Input's Size

With the obvious observation of all the algorithms, some algorithms runs longer if the input size is large. And some algorithms takes small amount of time for smaller inputs.
Example for larger inputs – Sorting of arrays, multiplication of larger matrices.
Example for smaller inputs – GCD, Swapping etc.

(ii) Units for Measuring Running Time

We simply use some standard unit of time measurement such as a second, or millisecond, and so on. To measure the running time of the program implementing the algorithm is always not possible because of following drawbacks:

- Dependence on the speed of a particular computer.
- Dependence on the quality of a program implementing the algorithm.
- The compiler used in generating the machine code.
- The difficulty of clocking the actual running time of the program.

One possible approach is to ***count the number of times each of the algorithm's operations are executed***. This approach is excessively difficult.

The most important operation (+, -, *, /) of the algorithm, called the ***basic operation***. Computing the number of times the basic operation is executed is easy. The total running time is determined by basic operations count.

(iii) Orders of Growth

- A difference in running times on small inputs is not what really distinguishes efficient algorithms from inefficient ones.
- For example, the greatest common divisor of two small numbers, it is not immediately clear how much more efficient Euclid's algorithm is compared to the other algorithms, the difference in algorithm efficiencies becomes clear for larger numbers only.
- For large values of *n*, it is the function's order of growth that counts just like the Table 1.1, which contains values of a few functions particularly important for analysis of algorithms.

TABLE 1.1 Values (approximate) of several functions important for analysis of algorithms

n	$\sqrt{n}$	$\log_2 n$	n	$n \log_2 n$	n^2	n^3	2^n	$n!$
1	**1**	**0**	**1**	**0**	**1**	**1**	**2**	**1**
2	**1.4**	**1**	**2**	**2**	**4**	**4**	**4**	**2**
4	**2**	**2**	**4**	**8**	**16**	**64**	**16**	**24**
8	**2.8**	**3**	**8**	$\mathbf{2.4 \cdot 10^1}$	**64**	$\mathbf{5.1 \cdot 10^2}$	$\mathbf{2.6 \cdot 10^2}$	$\mathbf{4.0 \cdot 10^4}$
10	**3.2**	**3.3**	**10**	$\mathbf{3.3 \cdot 10^1}$	$\mathbf{10^2}$	$\mathbf{10^3}$	$\mathbf{10^3}$	$\mathbf{3.6 \cdot 10^6}$
16	**4**	**4**	**16**	$\mathbf{6.4 \cdot 10^1}$	$\mathbf{2.6 \cdot 10^2}$	$\mathbf{4.1 \cdot 10^3}$	$\mathbf{6.5 \cdot 10^4}$	$\mathbf{2.1 \cdot 10^{13}}$
$\mathbf{10^2}$	**10**	**6.6**	$\mathbf{10^2}$	$\mathbf{6.6 \cdot 10^2}$	$\mathbf{10^4}$	$\mathbf{10^6}$	$\mathbf{1.3 \cdot 10^{30}}$	$\mathbf{9.3 \cdot 10^{157}}$
10^3	31	10	10^3	$1.0 \cdot 10^4$	10^6	10^9	Very big computation	
10^4	10^2	13	10^4	$1.3 \cdot 10^5$	10^8	10^{12}		
10^5	$\mathbf{3.2 \cdot 10^2}$	17	10^5	$1.7 \cdot 10^6$	10^{10}	10^{15}		
10^6	10^3	20	10^6	$2.0 \cdot 10^7$	10^{12}	10^{18}		

(iv) Worst-Case, Best-Case, andAverage Case Efficiencies

We establish that it si reasonable to measure the algorithm's effiency as a function of parameter indicating the size of the algorithm's input. But for many algorithms the runtime depends not only on the input size but also the specific type of input.

For example, sequential search. In this case we are going to search each and every element in the list until a search element is found.

Consider Sequential Search algorithm some search key K

ALGORITHM SequentialSearch(A[0..n - 1], K)

// Searches for a given value in a given array by sequential search

// Input: An array A[0..n - 1] and a search key K

// Output: The index of the first element in A that matches K or -1 if there are // no matching elements

i ←0

while i < n **and** A[i] ≠ K **do**

i ←i +1

if i < n **return** i

else return -1

Clearly, the running time of this algorithm can be quite different for the same list size *n*.

In the worst case, there is no matching of elements or the first matching element can foundat last on the list. In the best case, there is matching of elements at first on the list.

Worst-case efficiency

The worst-case efficiency is found in two cases. They are

1. When there is no matching element.
2. The first matching element happens to be the last element in the list.

The large no. of key comparisons among all possible inputs of size 'n'.

$C_{worst}(n) = n$.

The time taken for worst cas is highest than remaining cases.

Best case efficiency

The ***best-case efficiency*** of an algorithm is its efficiency for the best case input of size *n*. The algorithm runs the fastest among all possible inputs of that size n.

In sequential search, If we search a first element in list of size *n. (i.e.* first element equal toa search key), then the running time is $C_{best}(n) = 1$.

Neither the worst-case efficiency nor the best-case efficiency keeps the necessary information on the 'random' or 'typical' input.

Average case efficiency

The Average case efficiency lies between best case and worst case.

To analyze the algorithm's average case efficiency, we must make some

assumptions about possible inputs of size *n*.

The standard assumptions are that

- The probability of a successful search is equal to p $(0 \le p \le 1)$ and
- The probability of the first match occurring in the *i*th position of the

$$C_{avg}(n) = [1 \cdot \frac{p}{n} + 2 \cdot \frac{p}{n} + \cdots + i \cdot \frac{p}{n} + \cdots + n \cdot \frac{p}{n}] + n \cdot (1-p)$$
$$= \frac{p}{n}[1 + 2 + \cdots + i + \cdots + n] + n(1-p)$$
$$= \frac{p}{n}\frac{n(n+1)}{2} + n(1-p) = \frac{p(n+1)}{2} + n(1-p).$$

list is the samefor every *i*.

ASYMPTOTIC NOTATIONS

Asymptotic notation is a notation, which is used to take meaningful statement about theefficiency of a program.

The efficiency analysis framework concentrates on the order of growth of an algorithm'sbasic operation count as the principal indicator of the algorithm's efficiency.

To compare and rank such orders of growth, computer scientists use three notations, they are:

- O - Big oh notation
- Ω - Big omega notation
- Θ - Big theta notation

Let *t(n)* and *g(n)* can be any nonnegative functions defined on the set of natural numbers.

The algorithm's running time *t(n)* usually indicated by its basic operation count *C(n),* and *g(n),*
some simple function to compare with the count.

Example:

$n \in O(n^2)$,	$100n + 5 \in O(n^2)$,	$\frac{1}{2}n(n-1) \in O(n^2)$.
$n^3 \notin O(n^2)$,	$0.00001n^3 \notin O(n^2)$,	$n^4 + n + 1 \notin O(n^2)$.
$n^3 \in \Omega(n^2)$,	$\frac{1}{2}n(n-1) \in \Omega(n^2)$,	but $100n + 5 \notin \Omega(n^2)$.

(i) O - Big oh notation

Definition: A function t*(n)* is said to be in *O(g(n))*, denoted $t\ (n) \in O(g(n))$, 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

$$t\ (n) \leq\ cg(n) \text{ for } all\ n\ \geq\ n_0.$$

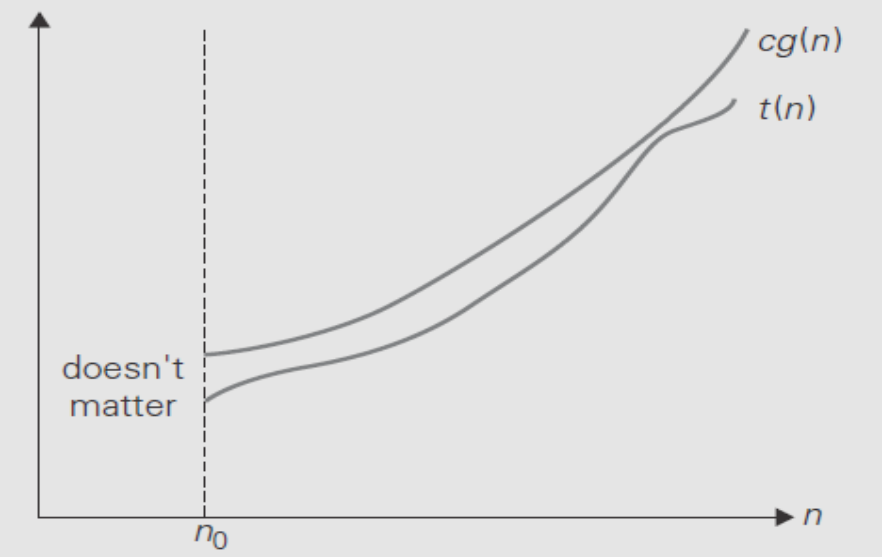


FIGURE 1.5 Big-oh notation: $t\ (n) \in\ O(g(n))$.

Example 1: Prove the assertions $100n + 5 \in O(n^2)$.

Proof: $100n + 5 \leq 100n + n$ (for all $n \geq 5$)

$= 101n$

$\leq 101n^2$

Since, the definition gives us a lot of freedom in choosing specific values for constants **c** and $\mathbf{n_0}$. We have c=101 and n_0=5

Example 2: Prove the assertions $100n + 5 \in O(n)$.

Proof: $100n + 5 \leq 100n + 5n$ (for all $n \geq 1$)

$= 105n$

i.e., $100n + 5 \leq 105n$

i.e., $t(n) \leq cg(n)$

$\pm 100n + 5 \in O(n)$ with c=105 and n_0=1

(ii) Ω - Big omega notation

Definition: A function *t(n)* is said to be in *Ω(g(n)),* denoted *t(n)* ∈ *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 integer n_0 such that

$$t(n) \geq cg(n) \text{ for all } n \geq n_0.$$

Where *t(n)* and *g(n)* are nonnegative functions defined on the set of natural numbers.

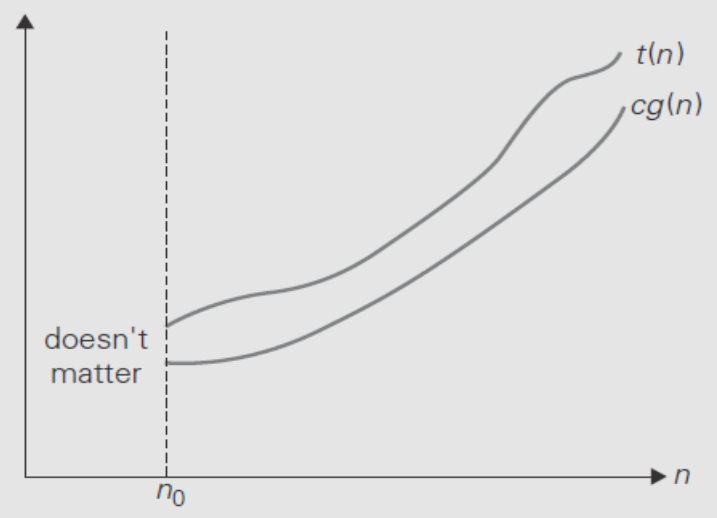


FIGURE 1.6 Big-omega notation: *t (n)* ∈ **Ω** *(g(n))*.

Example: Prove the assertions $n^3+5n \in \Omega(n^2)$.

Proof: $n^3+5n \geq n^2$ (for all $n \geq 0$)
i.e., by definition $t(n) \geq cg(n)$, where $c=1$ and $n_0=0$

(iii) Θ - Big theta notation

Definition: A function t(n) is said to be in *Θ(g(n)),* denoted t(n) ∈ 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 constants c_1 and c_2 and some nonnegative integer n_0 such that

$$c_2\, g(n) \leq t(n) \leq c_1\, g(n) \text{ for all } n \geq n_0.$$

Where *t(n)* and *g(n)* are nonnegative functions defined on the set of natural numbers.

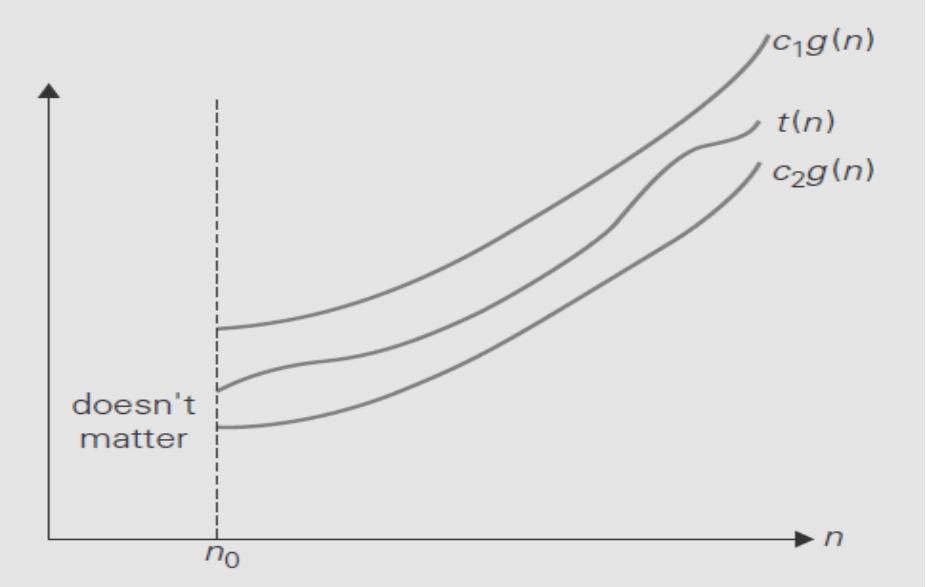


FIGURE 1.7 Big-theta notation: $t(n) \in \Theta(g(n))$.

Example :

Mathematical Analysis of Non-recursive Algorithms

General Plan for Analyzing the Time Efficiency of Non-recursive Algorithms

1. Decide on a *parameter* (or parameters) indicating an input's size.
2. Identify the algorithm's *basic operation.*
3. Check whether the *number of times the basic operation is executed* depends only on the sizeof an input. If it also depends on some additional property, the worst-case, average-case, and, if necessary, 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 least, establish its *order of growth*.

EXAMPLE 1: Consider the problem of finding the value of **the largest element in a list of nnumbers**. Assume that the list is implemented as an array for simplicity.

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 ←A[0]
for i ←1 to n − 1 do
        if A[i]>maxval
                maxval←A[i]
return maxval
```

Algorithm analysis

The measure of an input's size here is the number of elements in the array, i.e., n. There are two operations in the for loop's body:

- The comparison A[i]> maxval and
- The assignment maxval←A[i].

Let C(n) denotes the number of times this comparison is executed. The algorithm makesone comparison on each execution of the loop, which is repeated for each value of the loop's variable i within the bounds 1 and n-1. Therefore, the sum for C(n) is calculated as follows:

This is an easy sum to compute because it is nothing other than 1 repeated n-1 times. Thus,

Basic rules of Sum manipulations:

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

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

Summation formulas:

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

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

EXAMPLE 2: Consider the **element uniqueness problem**: check whether all the Elements in agiven array of n elements are distinct.

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

Algorithm Analysis

- The natural measure of the input's size here is again n (the number of elements in the array).

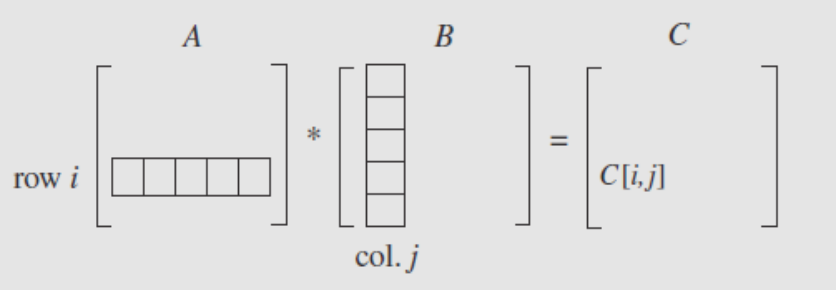
- Since the innermost loop contains a single operation (the comparison of two elements), we should consider it as the algorithm's basic operation.
- The number of element comparisons depends not only on n but also on whether there are equal elements in the array and, if there are, which array positions they occupy. We will limit our investigation to the worst case only**.**
- One comparison is made for each repetition of the innermost loop, i.e., for each value of the loop variable j between its limits i + 1 and n − 1; this is repeated for each value of the outer loop, i.e., for each value of the loop variable i between its limits 0 and n − 2.

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

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

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

Example 3: Consider matrix multiplication. Given two n × n matrices A and B, find the timeefficiency of the definition-based algorithm for computing their product C = AB. By definition C an n × n matrix whose elements are computed as the scalar (dot) products of the rows of matrix Aand the columns of matrix B:



where C[i, j]= A[i, 0]B[0, j]+ . . . + A[i, k]B[k, j]+ . . . + A[i, n − 1]B[n − 1, j] for every pair ofindices 0 ≤ i, j ≤ n − 1.

ALGORITHM MatrixMultiplication(A[0..n − 1, 0..n − 1], B[0..n − 1, 0..n − 1])

//Multiplies two square matrices of order n by the definition-based algorithm

//Input: Two n × n matrices A and B

//Output: Matrix C = AB

for i ←0 **to** n − 1 **do**

for j ←0 **to** n − 1 **do**

C[i, j]←0.0

for k←0 to n − 1 do

C[i, j]←C[i, j]+ A[i, k] ∗ B[k, j]

return C

Algorithm Analysis:

Here the basic operations are multiplication and addition. One multiplication executed on each repetition in the inner most loop. Which is denoted by 'k' and it is from 0 to n-1. Therefore,

$$\sum_{k=0}^{n-1} 1$$

The total number of multiplications expressed as

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

Now, we can compute this sum by using formula

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

The time efficiency of matrix based on the algorithm

Where Cm is only matrix multiplication content. But, then there is also addition basic operation.

Therefore the total time efficiency of matrix is

Mathematical Analysis of Recursive Algorithms

General Plan for Analyzing the Time Efficiency of Recursive Algorithms

1. Decide on a *parameter* (or parameters) 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 the number of times the basic operation is executed.
5. Solve the recurrence or, at least, ascertain the *order of growth* of its solution.

EXAMPLE 1: Compute the factorial function F(n) = n! for an arbitrary nonnegative integer n.

Since $n! = 1 * \ldots * (n-1) * n = (n-1)! * n$, for $n \geq 1$ and 0!= 1 by definition, we can compute

$F(n) = F(n-1) * n$ with the following recursive algorithm.

ALGORITHM *F(n)*

//Computes *n*! recursively
//Input: A nonnegative integer *n*
//Output: The value of *n*!
if *n* = 0 **return** 1

else return *F(n* − 1*) * n*

Algorithm analysis

For simplicity, we consider *n* itself as an indicator of this algorithm's input size. i.e. 1.

The basic operation of the algorithm is multiplication, whose number of executions we denote *M(n)*. Since the function *F(n)* is computed according to the formula $F(n) = F(n-1) \cdot n$ for $n > 0$.

The number of multiplications *M(n)* needed to compute it must satisfy the equality

Recurrence relations

The last equation defines the sequence M(n) that we need to find. This equation defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely n − 1. Such equations are called recurrence relations or recurrences. There are several techniques for solving the recurrence relations.

Here, we are going to use the back ward substitutions. The way it applies for solving a particular equation is as follows:

$M(n) = M(n-1) + 1$ substitute $M(n-1) = M(n-2) + 1$

$= [M(n-2) + 1] + 1$

$= M(n-2) + 2$ substitute $M(n-2) = M(n-3) + 1$

$= [M(n-3) + 1] + 2$

$= M(n-3) + 3$

…

$= M(n-i) + i$

…

$= M(n-n) + n$

$= n.$

Therefore $M(n)=n$.

EXAMPLE 2: consider educational workhorse of recursive algorithms: the ***Tower of Hanoi***puzzle. We have n disks of different sizes that can slide onto any of three pegs. Consider A (source), B (auxiliary), and C (Destination). Initially, all the disks are on the first peg in order of size, the largest on the bottom and the smallest on top. The goal is to move all the disks to the third peg, using the second one as an auxiliary.

ALGORITHM TOH(n, A, C, B)

//Move disks from source to destination recursively

//Input: *n disks and* 3 pegs A, B, and C

//Output: Disks moved to destination as in the source order.

if n=1

Move disk from A to C

else

Move top n-1 disks from A to B using C
TOH(n - 1, A, B, C)
Move top n-1 disks from B to C using A
TOH(n - 1, B, C, A)

Algorithm analysis

The number of moves *M(n)* depends on *n* only, and we get the following recurrenceequation for it: $M(n) = M(n-1) + 1 + M(n-1)$ for $n > 1$. With the obvious initial condition $M(1) = 1$, we have the following recurrence relation for thenumber of moves *M(n)*:

$$M(n) = 2M(n-1) + 1 \text{ for } n > 1,$$
$$M(1) = 1.$$

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

$M(n) = 2M(n-1) + 1$ sub. $M(n-1) = 2M(n-2) + 1$
$= 2[2M(n-2) + 1] + 1$
$= 2^2M(n-2) + 2 + 1$ sub. $M(n-2) = 2M(n-3) + 1$
$= 2^2[2M(n-3) + 1] + 2 + 1$
$= 2^3M(n-3) + 2^2 + 2 + 1$ sub. $M(n-3) = 2M(n-4) + 1$
$= 2^4M(n-4) + 2^3 + 2^2 + 2 + 1$
…

$= 2^iM(n-i) + 2^{i-1} + 2i^{-2} + \ldots + 2 + 1 = 2^iM(n-i) + 2^i - 1.$

…

Since the initial condition is specified for $n = 1$, which is achieved for $i = n - 1$,

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

Thus, we have an exponential time algorithm.

Empirical Analysis of Algorithms

Empirical analysis is an evidence-based approach to the study and interpretation of information. The empirical approach relies on real-world data, metrics and results rather than theories and concepts. Empirical analysis never gives an absolute answer, however, only a most likely answer based on probability.

The principal alternative to the mathematical analysis of an algorithm's efficiency is its empirical analysis. This approach implies steps spelled out in the following plan.

General Plan for the Empirical Analysis of Algorithm Time Efficiency

1. Understand the experiment's purpose.
2. Decide on the efficiency metric ***M*** to be measured and the measurement unit (an operation count vs. a time unit).
3. Decide on characteristics of the input sample (its range, size, and so on).
4. Prepare a program implementing the algorithm (or algorithms) for the exper-imentation.
5. Generate a sample of inputs.
6. Run the algorithm (or algorithms) on the sample's inputs and record the data observed.
7. Analyze the data obtained.

Let us discuss these steps one at a time. There are several different goals one can pursue in analyzing algorithms empirically. They include checking the accuracy of a theoretical assertion about the algorithm's efficiency, comparing the efficiency of several algorithms for solving the same problem or different imple-mentations of the same algorithm, developing a hypothesis about the algorithm's efficiency class, and ascertaining the efficiency of the program implementing the algorithm on a particular machine.

In particular, the goal of the experiment should influence, if not dictate, how the algorithm's efficiency is to be measured. The first alternative is to insert a counter (or counters) into a program implementing the algorithm to count the number of times the algorithm's basic operation is executed. This is usually a straightforward operation.

The second alternative is to time the program implementing the algorithm in question. The easiest way to do this is to use a system's command, such as the time command in UNIX.

The empirical data obtained as the result of an experiment need to be recorded and then presented for an analysis. Data can be presented numerically in a table or graphically in a ***scatterplot***, i.e., by points in a Cartesian coordinate system. It is a good idea to use both these options whenever it is feasible because both methods have their unique strengths and weaknesses.

The principal advantage of tabulated data lies in the opportunity to manipulate it easily. For example, one can compute the ratios ***M(n)/g(n)*** where ***g(n)*** is a candidate to represent the efficiency class of the algorithm in question. If the algorithm is indeed in ***(g(n)),*** most likely these ratios will converge to some positive constant as ***n*** gets large.

One of the possible applications of the empirical analysis is to predict the algorithm's performance on an instance not included in the experiment sample. For example, if you observe that the ratios ***M(n)/g(n)*** are close to some constant ***c*** for the sample instances, it could be sensible to approximate ***M(n)*** by the product ***cg(n)*** for other instances, too.

We can implement one of several known algorithms for gener-ating (pseudo)random numbers. The most widely used and thoroughly studied of such algorithms is the ***linear congruential method***.

ALGORITHM *Random****(n, m,*** *seed****, a, b)***

//Generates a sequence of ***n*** pseudorandom numbers according to the linear congruential method

//Input: A positive integer ***n*** and positive integer parameters ***m,*** *seed****, a, b***

//Output: A sequence $r_1, \ldots, r_n$ of ***n*** pseudorandom integers uniformly distributed among integer values between 0 and ***m*** − 1

//Note: Pseudorandom numbers between 0 and 1 can be obtained by treating the integers generated as digits after the decimal point

$r_0 \leftarrow$ *seed*

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

$r_i \leftarrow (a * r_{i-1} + b) \bmod m$

Here is a partial list of recommen-dations based on the results of a sophisticated mathematical analysis : *seed* may be chosen arbitrarily and is often set to the current date and time; ***m*** should be large and may be conveniently taken as 2^w, where ***w*** is the computer's word size; ***a*** should be selected as an integer between $0.01m$ and $0.99m$ with no particular pattern in its digits but such that $a \bmod 8 = 5$; and the value of ***b*** can be chosen as 1.

Algorithm Visualization

In addition to the mathematical and empirical analyses of algorithms, there is yet a third way to study algorithms. It is called ***algorithm visualization*** and can be defined as the use of images to convey some useful information about algorithms. That information can be a visual illustration of an algorithm's operation, of its performance on different kinds of inputs, or of its execution speed versus that of other algorithms for the same problem. To accomplish this goal, an algorithm visualization uses graphic elements—points, line segments, two- or three-dimensional bars, and so on—to represent some "interesting events" in the algorithm's operation.

There are two principal variations of algorithm visualization: Static algorithm visualization Dynamic algorithm visualization, also called algorithm animation.

Static algorithm visualization shows an algorithm's progress through a series of still images. Algorithm animation, on the other hand, shows a continuous, movie-like presentation of an algorithm's operations. Animation is an arguably more sophisticated option, which, of course, is much more difficult to implement.

Early efforts in the area of algorithm visualization go back to the 1970s. The watershed event happened in 1981 with the appearance of a 30-minute color sound film titled *Sorting Out Sorting*. This algorithm visualization classic was produced at the University of Toronto by Ronald Baecker with the assistance of D.

There are two principal applications of algorithm visualization: **research** and **education**. Potential benefits for researchers are based on expectations that algorithm visualization may help uncover some unknown features of algorithms. For example, one researcher used a visualization of the recursive Tower of Hanoi algorithm in which odd- and even-numbered disks were colored in two different colors. He noticed that two disks of the same color never came in direct contact during the algorithm's execution. This observation helped him in developing a better non-recursive version of the classic algorithm.

The application of algorithm visualization to education seeks to help students learning algorithms. The available evidence of its effectiveness is decisively mixed. Although some experiments did register positive learning outcomes, others failed to do so. The increasing body of evidence indicates that creating sophisticated software systems is not going to be enough. In fact, it appears that the level of student involvement with visualization might be more important than specific features of visualization software.

FIGURE 2.8 Initial and final screens of a typical visualization of a sorting algorithm using the bar representation.

Selection Sort

We start selection sort by scanning the entire given list to find its smallest element and exchange it with the first element, putting the smallest element in its final position in the sorted list. Then we scan the list, starting with the second element, to find the smallest among the last $n-1$ elements and exchange it with the second element, putting the second smallest element in its final position. Generally, on the ith pass through the list, which we number from 0 to $n-2$, the algorithm searches for the smallest item among the last $n-i$ elements and swaps it with A_i:

$$\underbrace{A_0 \le A_1 \le \cdots \le A_{i-1}}_{\text{in their final positions}} \;\big|\; \underbrace{A_i, \ldots, A_{min}, \ldots, A_{n-1}}_{\text{the last } n-i \text{ elements}}$$

After $n-1$ passes, the list is sorted.

Here is pseudocode of this algorithm, which, for simplicity, assumes that the list is implemented as an array:

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

```
//Sorts a given array by selection sort
//Input: An array A[0..n − 1] of orderable elements
//Output: Array A[0..n − 1] sorted in nondecreasing order
for i ← 0 to n − 2 do
    min ← i
    for j ← i + 1 to n − 1 do
        if A[j] < A[min]  min ← j
    swap A[i] and A[min]
```

As an example, the action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated in Figure 3.1.

The analysis of selection sort is straightforward. The input size is given by the number of elements n; the basic operation is the key comparison $A[j] < A[min]$. The number of times it is executed depends only on the array size and is given by the following sum:

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

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

FIGURE 3.1 Example of sorting with selection sort. Each line corresponds to one iteration of the algorithm, i.e., a pass through the list's tail to the right of the vertical bar; an element in bold indicates the smallest element found. Elements to the left of the vertical bar are in their final positions and are not considered in this and subsequent iterations.

Since we have already encountered the last sum in analyzing the algorithm of Example 2 in Section 2.3, you should be able to compute it now on your own. Whether you compute this sum by distributing the summation symbol or by immediately getting the sum of decreasing integers, the answer, of course, must be the same:

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

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs. Note, however, that the number of key swaps is only $\Theta(n)$, or, more precisely, $n-1$ (one for each repetition of the i loop). This property distinguishes selection sort positively from many other sorting algorithms.

Bubble Sort

Another brute-force application to the sorting problem is to compare adjacent elements of the list and exchange them if they are out of order. By doing it repeatedly, we end up "bubbling up" the largest element to the last position on the list. The next pass bubbles up the second largest element, and so on, until after $n-1$ passes the list is sorted. Pass i $(0 \le i \le n-2)$ of bubble sort can be represented by the following diagram:

$$A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \le \cdots \le A_{n-1}$$

in their final positions

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

//Sorts a given array by bubble sort
//Input: An array $A[0..n-1]$ of orderable elements
//Output: Array $A[0..n-1]$ sorted in nondecreasing order
for $i \leftarrow 0$ **to** $n-2$ **do**
  for $j \leftarrow 0$ **to** $n-2-i$ **do**
    if $A[j+1] < A[j]$ swap $A[j]$ and $A[j+1]$

The action of the algorithm on the list 89, 45, 68, 90, 29, 34, 17 is illustrated as an example in Figure 3.2.

The number of key comparisons for the bubble-sort version given above is the same for all arrays of size n; it is obtained by a sum that is almost identical to the sum for selection sort:

89 ?↔ 45 68 90 29 34 17
45 89 ?↔ 68 90 29 34 17
45 68 89 ?↔ 90 ?↔ 29 34 17
45 68 89 29 90 ?↔ 34 17
45 68 89 29 34 90 ?↔ 17
45 68 89 29 34 17 | 90

45 ?↔ 68 ?↔ 89 ?↔ 29 34 17 | 90
45 68 29 89 ?↔ 34 17 | 90
45 68 29 34 89 ?↔ 17 | 90
45 68 29 34 17 | 89 90

etc.

FIGURE 3.2 First two passes of bubble sort on the list 89, 45, 68, 90, 29, 34, 17. A new line is shown after a swap of two elements is done. The elements to the right of the vertical bar are in their final positions and are not considered in subsequent iterations of the algorithm.

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

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

The number of key swaps, however, depends on the input. In the worst case of decreasing arrays, it is the same as the number of key comparisons:

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

Sequential Search

We have already encountered a brute-force algorithm for the general searching problem: it is called sequential search (see Section 2.1). To repeat, the algorithm simply compares successive elements of a given list with a given search key until either a match is encountered (successful search) or the list is exhausted without finding a match (unsuccessful search). A simple extra trick is often employed in implementing sequential search: if we append the search key to the end of the list, the search for the key will have to be successful, and therefore we can eliminate the end of list check altogether. Here is pseudocode of this enhanced version.

ALGORITHM *SequentialSearch2*$(A[0..n], K)$

```
//Implements sequential search with a search key as a sentinel
//Input: An array A of n elements and a search key K
//Output: The index of the first element in A[0..n − 1] whose value is
//        equal to K or −1 if no such element is found
A[n] ← K
i ← 0
while A[i] ≠ K do
        i ← i + 1
if i < n return i
else return −1
```

Another straightforward improvement can be incorporated in sequential search if a given list is known to be sorted: searching in such a list can be stopped as soon as an element greater than or equal to the search key is encountered.

Best case efficiency

The ***best-case efficiency*** of an algorithm is its efficiency for the best case input of size *n*. The algorithm runs the fastest among all possible inputs of that size n.

In sequential search, If we search a first element in list of size *n. (i.e.* first element equal toa search key), then the running time is $C_{best}(n) = 1$.

Worst-case efficiency

The worst-case efficiency is found in two cases. They are

1. When there is no matching element.
2. The first matching element happens to be the last element in the list.

The large no. of key comparisons among all possible inputs of size 'n'.

$C_{worst}(n) = n$.

The Time Complexity of Sequential Search is O(n).

The time taken for worst cas is highest than remaining cases.

Exhaustive Search

Definition: 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.

A brute force solution to a problem involving search for an element with a special property, usually among combinatorial objects such as a permutations, combinations, or subsets of a set.

Method

• Construct a way of listing all potential solutions to the problem in a systematic manner
• all solutions are eventually listed
• no solution is repeated
• Evaluate solutions one by one, perhaps disqualifying infeasible ones and keeping track of the best one found so far
• When search ends, announce the winner

We illustrate exhaustive search by applying it to three important problems: the traveling salesman problem, the knapsack problem, and the assignment problem.

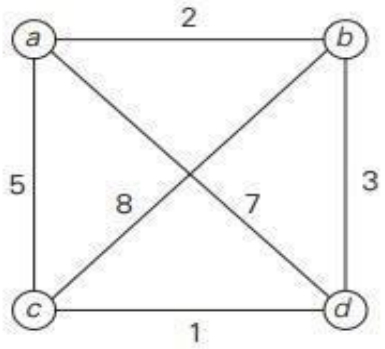
Traveling Salesman Problem

The ***traveling salesman problem (TSP)*** has been intriguing researchers for the last 150 years by its seemingly simple formulation, important applications, and interesting connections to other combinatorial problems.

In layman's terms, 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).

Example:



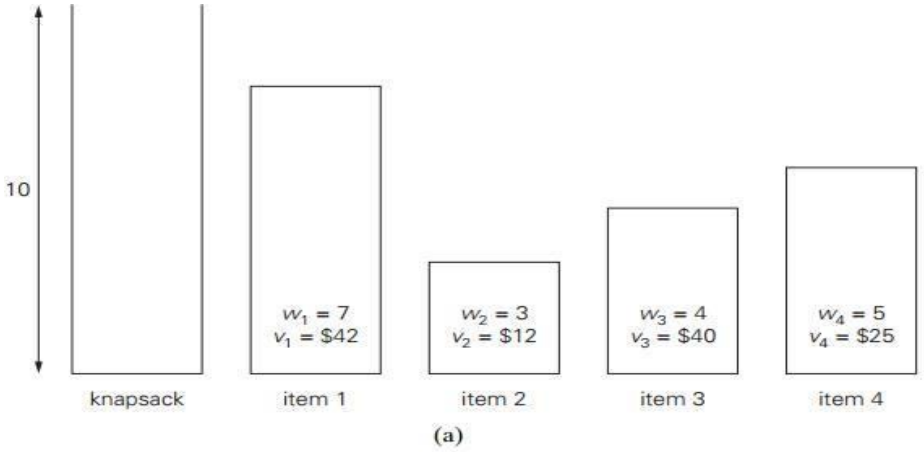
Tour	Length	
$a \rightarrow b \rightarrow c \rightarrow d \rightarrow a$	$l = 2 + 8 + 1 + 7 = 18$	
$a \rightarrow b \rightarrow d \rightarrow c \rightarrow a$	$l = 2 + 3 + 1 + 5 = 11$	optimal
$a \rightarrow c \rightarrow b \rightarrow d \rightarrow a$	$l = 5 + 8 + 3 + 7 = 23$	
$a \rightarrow c \rightarrow d \rightarrow b \rightarrow a$	$l = 5 + 1 + 3 + 2 = 11$	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 $w_1, w_2, \ldots, w_n$ and values $v_1, v_2, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

The exhaustive-search approach to this problem leads to generating all the subsets of the set of n items given, computing the total weight of each subset in order to identify feasible subsets (i.e., the ones with the total weight not exceeding the knapsack capacity), and finding a subset of the largest value among them. As an example, the solution to the instance of Figure 3.8a is given in Figure 3.8b. Since the number of subsets of an n-element set is 2^n, the exhaustive search leads to a (2^n) algorithm, no matter how efficiently individual subsets are generated.



(a)

Subset	Total weight	Total value
∅	0	$ 0
{1}	7	$42
{2}	3	$12
{3}	4	$40
{4}	5	$25
{1, 2}	10	$54
{1, 3}	11	not feasible
{1, 4}	12	not feasible
{2, 3}	7	$52
{2, 4}	8	$37
{3, 4}	**9**	**$65**
{1, 2, 3}	14	not feasible
{1, 2, 4}	15	not feasible
{1, 3, 4}	16	not feasible
{2, 3, 4}	12	not feasible
{1, 2, 3, 4}	19	not feasible

(b)

FIGURE 3.8 (a) Instance of the knapsack problem. (b) Its solution by exhaustive search. The information about the optimal selection is in bold.

Assignment Problem

In our third example of a problem that can be solved by exhaustive search, there are n people who need to be assigned to execute n jobs, one person per job. (That is, each person is assigned to exactly one job and each job is assigned to exactly one person.) The cost that would accrue if the ith person is assigned to the j th job is a known quantity $C[i, j]$ for each pair $i, j = 1, 2, \ldots, n$. The problem is to find an assignment with the minimum total cost.

A small instance of this problem follows, with the table entries representing the assignment costs $C[i, j]$:

	Job 1	Job 2	Job 3	Job 4
Person 1	9	2	7	8
Person 2	6	4	3	7
Person 3	5	8	1	8
Person 4	7	6	9	4

We can describe feasible solutions to the assignment problem as n-tuples $j_1, \ldots, j_n$ in which the ith component, $i = 1, \ldots, n$, indicates the column of the element selected in the ith row (i.e., the job number assigned to the ith person). For example, for the cost matrix above, 2, 3, 4, 1 indicates the assignment of Person 1 to Job 2, Person 2 to Job 3, Person 3 to Job 4, and Person 4 to Job 1.

The requirements of the assignment problem imply that there is a one-to-one correspondence between feasible assignments and permutations of the first n integers. Therefore, the exhaustive-search approach to the assignment problem would require generating all the permutations of integers $1, 2, \ldots, n$, computing the total cost of each assignment by summing up the corresponding elements of the cost matrix, and finally selecting the one with the smallest sum. A few first iterations of applying this algorithm to the instance given above are shown in Figure 3.9.

$$C = \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix}$$

<1, 2, 3, 4> cost = 9 + 4 + 1 + 4 = 18
<1, 2, 4, 3> cost = 9 + 4 + 8 + 9 = 30
<1, 3, 2, 4> cost = 9 + 3 + 8 + 4 = 24
<1, 3, 4, 2> cost = 9 + 3 + 8 + 6 = 26
<1, 4, 2, 3> cost = 9 + 7 + 8 + 9 = 33
<1, 4, 3, 2> cost = 9 + 7 + 1 + 6 = 23

etc.

FIGURE 3.9 First few iterations of solving a small instance of the assignment problem by exhaustive search.

****

UNIT-II

Divide-and-Conquer

General Method:

Divide-and-Conquer is the best known algorithm design technique. This algorithm work according to the following general plan:

1. A problem is divided into several sub problems of the same type, ideally of about equal size.
2. The sub problems are solved separately.
3. If necessary, the solutions to the sub problems are combined to get a solution to the original problem.

The Divide-and-Conquer technique diagram

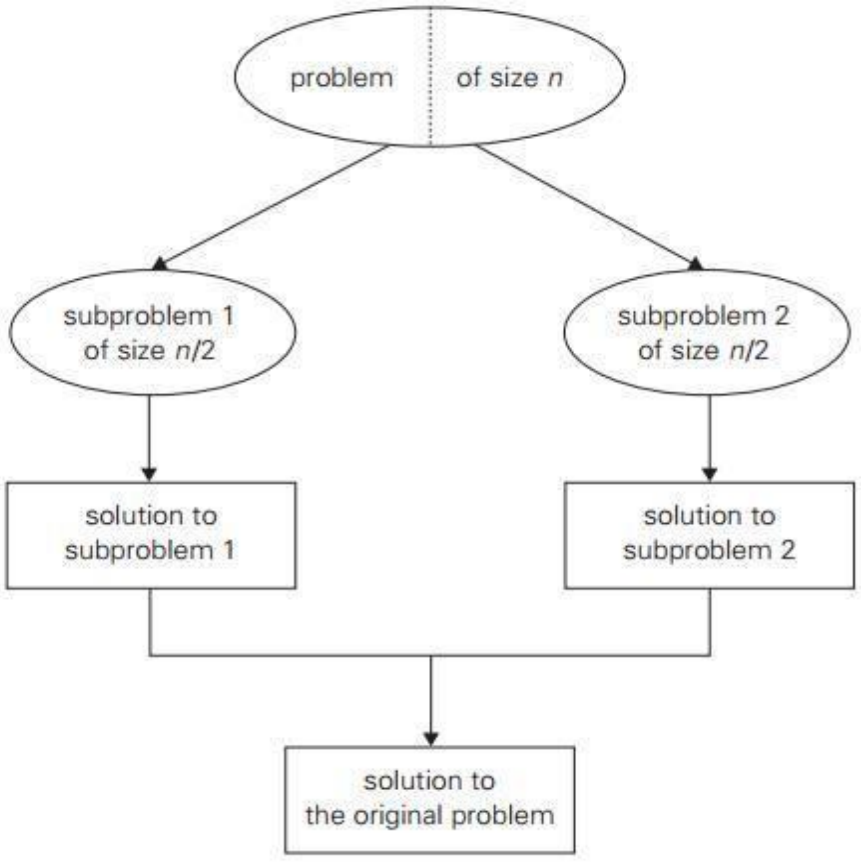


FIGURE 5.1 Divide-and-conquer technique (typical case).

Let us consider the problem of computing the sum of 'n' numbers, $a_0, a_1, .. a_{n-1}$. If n > 1, we can divide the problem into two sub problems, then we compute the first n /2 numbers and then we compute the remaining numbers. Once each of these two sub problems are computed we can add their values to get the solution to the sum of 'n' numbers.

⇨ $a_0 + \ldots\ldots + a_{n-1}$

⇨ $(a_0 + \ldots. + a_{(n/2)-1}) + (a_{n/2} + \ldots. + a_{n-1})$

More generally an instance of size 'n, can be divided into smaller instances.

$$T(n) = \begin{cases} g(n) \\ T(n1) + T(n2) + \ldots. + f(n) \end{cases} \text{for all } n > 1$$

Here T(n) is represented for the time taken in order to divide and conquer of any input of size 'n'.

g(n) represented for the time taken to compute the smaller instances for smaller inputs.

f(n) is denoted for the time taken for dividing the problem an combining the solutions of sub problems.

Complexity:

$$T(n) = \begin{cases} T(n) = T(1) = 1 & \text{where } n = 1 \\ a.T(n/b) + f(n) & \text{for all } n > 1 \end{cases}$$

The above equation is called as the recurrence relation of divide and conquer.
'a' denotes that how many no. of times we are going to find the time complexity.
'b' denotes for how many no. of smaller instances are made.

Master Theorem:

If $f(n) \in \Theta(n^d)$ where $d \geq 0$ in the recurrence equation $T(n) = a.\ T(n/b) + f(n)$, then

The Master Method

Base doesn't matter (only changes leading constants)

$$T(n) = \begin{cases} O(n^d \log n) & \text{if } a = b^d \quad \text{(Case 1)} \\ O(n^d) & \text{if } a < b^d \quad \text{(Case 2)} \\ O(n^{\log_b a}) & \text{if } a > b^d \quad \text{(Case 3)} \end{cases}$$

Base matters

For example, the recurrence equation for the no. of additions A(n) made by the divide and conquer summation algorithm on inputs of size $n = 2^k$ is

$$A(n) = 2.\ A(n/2) + 1$$

Thus, for this example a =2, b =2 and d = 0.
Hence, $a > b^d$ we can represent the order of growth

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$$

according to the Master Theorem.

Applications of Divide and Conquer Approach

Following are some problems, which are solved using divide and conquer approach.

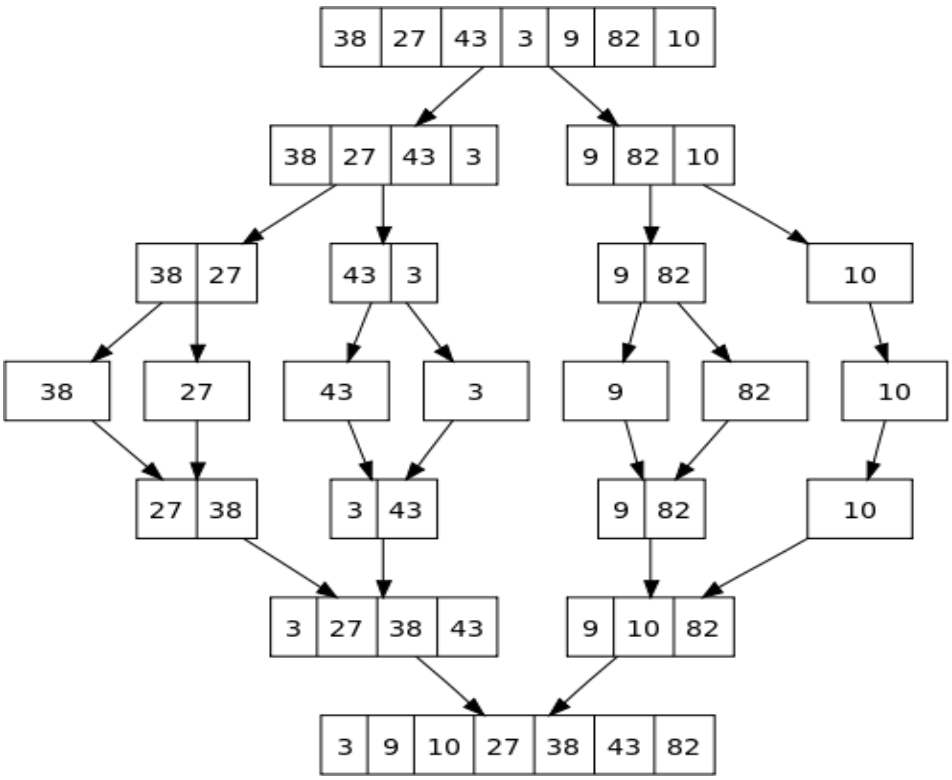
- Finding the maximum and minimum of a sequence of numbers
- Strassen's matrix multiplication
- Merge sort
- Binary search

Merge Sort

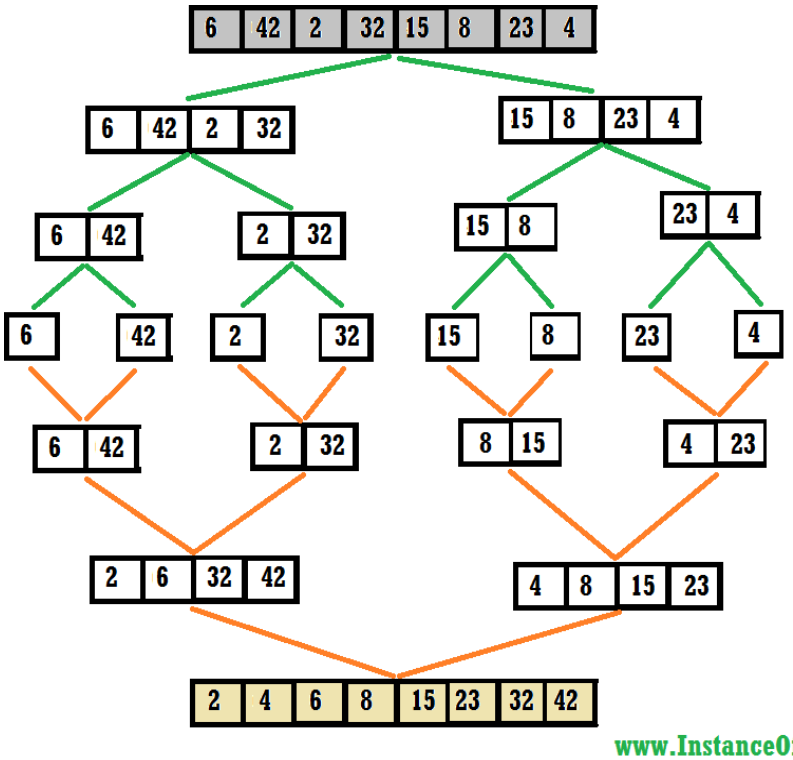
Merge Sort is a perfect example of a successful application of the divide-and-conquer technique. In this technique it sorts the given elements of range A[0,….n-1] by dividing it into two halves A[0,….(n/2)-1] and A[n/2,….n-1], sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted array.

The merging of two sorted arrays can be done as follows:
Two points are initialized to point to the first elements of two arrays, that are being merge. Then the elements pointed to are compared and smaller of them added to new array i.e, being constructed. This operation is continued until one of the two arrays exhausted and remaining elements of other array are copied to the end of the new array.

Example:1 Array size is odd



Example:2 Array size is even



Pseudo code of Merge Sort:

```
ALGORITHM MergeSort( A[0…n-1]
// Sorts array A[0..n-1] by recursive merge sort
// 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…(n/2)-1] to B[0…(n/2)-1]
    copy A[n/2…n-1] to C[n/2…n-1]
    Mergesort(B[0…(n/2)-1])
    Mergesort(C[n/2…n-1])
    Merge(B,C,A)
```

Algorithm for Merge:

```
ALGORITHM Merge(B[0…p-1],C[0…q-1],A[0…p+q-1])
  i ← 0, j ← 0, k ← 0
  While i<p and j<q do
     If  B[i] ≤ C[j]
        A[k]  ← B[i];    i ← i+1
     else A[k]  ← C[j];   j ← j+1
     k ← 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]
```

Recurrence Relation for Merge Sort

Recurrence Relation for the number of key comparisons C(n) is

$$C(n) = 2.\ C(n/2) + C_{merge}(n) \quad \text{for } n > 1,\ C(1) = 0$$

The number of key comparisons perform during the merging stage. At each step, exactly one comparison is made, after which the total number of elements in two arrays still needed to be processed is reduced by one.

In the worst case, neither of the two arrays becomes empty before the other one contains just one element. Therefore, for the worst case, $C_{merge}(n) = n-1$, we have the recurrence

$$C_{worst}(n) = 2.C_{worst}(n/2) + n-1 \quad \text{for } n > 1.\ C_{worst}(1) = 0.$$

Hence, according to Master Theorem,

$$C_{worst}(n) \in \Theta(n \log n)$$

Quick Sort

Quick Sort is the other important sorting algorithm that is based on the divide-and-conquer approach. Merge Sort is done based on the values or elements in the arrays where as Quick Sort is done based on the partitions of the elements in the array.

If we rearrange the elements in the quick sort of the given array A[0..n-1] in order to achieve its partition, a situation is occurred where all the elements before some position S are smaller than or equal to A[S] and all the elements after position S are greater than or equal to A[S].

$$\underbrace{A[0,\ldots..S-1]}_{\leq A[S]}\ A[S]\ \underbrace{A[S+1,\ldots n-1]}_{\geq A[S]}$$

Obviously, after a partition, A[S] will be in its final position in the sorted array, and we can continue sorting the two sub arrays independently (as we have done above).

ALGORITHM Quicksort(A[*l*..r])
// Sorts a sub array by quick sort
// Input: A sub array A[*l*..r] of A[0..n-1], defined by its left and right indices *l* and r
// Output: The sub array A[*l*..r] sorted in non-decreasing order
if *l* < r
    s ← partition(A[*l*..r]) // **s** is a split position
    Quicksort(A[*l*..s-1])
    Quicksort(A[s+1..r])

A partition of A[0..n-1] is more generally of its sub array A[*l*..r] ($0 \leq l < r \leq n-1$)

First we select an element with respect to whose value we are going to divide into sub array. We call that element is 'pivot' element or 'key' element. After several alternative procedures for rearranging elements to achieve a partition, two scans for the sub array will be done from left to right and right to left.

The left to right scan starts at the second element. Since, we want the element greater than the pivot element to be in the first part of the sub array. The elements in the second part of the array must be smaller than the pivot and stops on encountering the first element smaller or equal to pivot element. If the scanning integer i and j have not crossed i.e.m i < j we simply exchange A[i] and A[j] and resume the scans by incrementing 'i' and decrementing 'j'.

		→ i		j ←	
p	all are ≤ p	≥ p	.....	≤ p	all are ≥ p

If we scanning indices have crossed over, i.e., i > j, we have partitioned the array after exchanging the pivot with A[j]:

		j ←	→ i	
p	all are ≤ p	≤ p	≥ p	all are ≥ p

Finally, if the scanning indices stop while pointing to the same element. i.e., i = j

		→ i = j ←	
p	all are ≤ p	= p	all are ≥ p

l ***r***

Example: **44** 33 11 55 77 90 40 60 99 22 88 66

↑ i → i j ← j

Pivot

i < j (swap 55 and 22)

44 33 11 22 77 90 40 60 99 55 88 66

i → i j ← j

i < j (swap 77 and 40)

44 33 11 22 40 90 77 60 99 55 88 66

i → j i ← j

i > j

(here i > j then swap pivot with A[j], i.e., swap 44 and 40)

40 33 11 22 **44** 90 77 60 99 55 88 66

≤ 44 ≥ 44

Similarly same process applied to the two sub arrays.

Algorithm for Partition:

ALGORITHM Partition(A[*l*..r])
// Partitions a sub array by using its first element as a pivot
// Input: A sub array A[*l*..r] of A[0,,n-1] defined by its left and right indices
// *l* and r (*l* < r)
// Output: The sub array A[l..r], sorted in non-decreasing order

```
if l < r
   p ← A[l]
   i ← l; j ← r
   repeat
        repeat i ← i + 1 until A[i] ≥ p
        repeat j ← j - 1 until A[j] ≤ p
        swap (A[i], A[j])
  until i ≥ j
  swap (A[i], A[j])    // undo last swap when i ≥ j
  swap (A[l], A[j])
  return j
```

The recurrence relation for Quick Sort: The recurrence relation for quick sort is

$$C(n) = 2.\ C(n/2) + n \quad \text{for all} \quad n > 1$$

If all the splits happen in the middle of the corresponding sub arrays, we will treated as best case efficiency. The number of key comparisons in the best case will satisfy the following recurrence equation,

$$C_{best}(n) = 2.\ C_{best}(n/2) + n \quad \text{for all } n > 1, \quad C_{best}(1) = 0.$$

According to Master Theorem, $C_{best}(n) = \Theta(n \log_2 n)$

In the worst case, all the splits will be skewed to the extreme, one of the two sub arrays will be empty while the size of the other will be just one less than the size of a sub array being partitioned.

If A[0,...n-1] is a strictly increasing array and we use A[0] as the pivot, the left-to-right scan will stop on A[i] while right-to-left scan will go all the way to reach A[0]. So after making n+1 comparisons to get to the partition and exchanging the pivot A[0] with itself, the algorithm will find itself with strictly increasing array A[1...n-1] to sort. This process continue until the last one A[n-2..n-1]. The total number of key comparisons be equal to

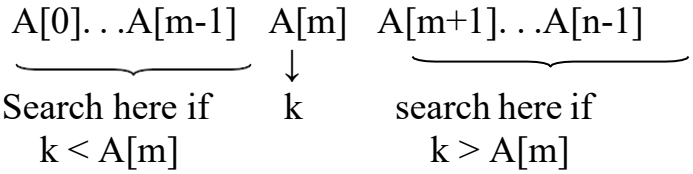
$$C_{worst}(n) = (n+1) + n + (n-1) + \ldots + 3 = \frac{(n+1)\,(n+2)}{2}$$

According to Master Theorem the total number of key comparisons in the worst case = $\Theta(n^2)$.

The average number of key comparisons made by the Quick Sort as a random by ordered array of size 'n'. Assuming that the partition split can happen in each position $S(0 \le S \le n-1)$ with the same probability 1/n, we get the following recurrence relation:

Binary Search

Binary Search is a remarkably efficient algorithm for searching in a sorted array. It works by comparing a search key 'k' with the array's middle element A[m]. If they match, the algorithm stops; otherwise, the same operation is repeated recursively for the first half of the array if k < A[m] and for the second half if k > A[m]:



Example:

As an example, let us apply binary search to searching for k = 70 in the array

3	14	27	31	39	42	55	70	74	81	85	93	98

The iterations of the algorithm are given in the following table:

Index:	0	1	2	3	4	5	6	7	8	9	10	11	12
Value:	3	14	27	31	39	42	55	**70**	74	81	85	93	98

Iteration 1 *l* m r

Iteration 2

70	74	81	85	93	98

l m r

iteration 3

70	74

l, m r (here k = m)

Though binary search is clearly based on a recursive idea, it can be easily implemented as a non recursive algorithm too.

Algorithm for Binary Search:

```
ALGORITHM BinarySearch( A[0..n-1], K)
 // Implements non-recursive binary search
 //  Input: An array A[0..n-1] sorted in ascending order and a search key K
 // Output: An index of the array's element that is equal to K or -1 if there is
//              no such element
 l ← 0; r ← n-1
 while l ≤ r do
       m ← [(l + r) / 2]
       if  K = A[m] return m
       else if  K < A[m]    r ← m-1
       else l ← m+1
  return  -1
```

Recurrence Relation:

The standard way to analyse the efficiency of binary search is to count the number of times the search key is compared with an element of the array.

Let us find the number of key comparisons in the worst case $C_{worst}(n)$.
The worst case inputs include all arrays that do not contain a given search key. Since after one comparison the algorithm faces the same situation but for an array half the size, we get the following recurrence relation for Cworst(n):

$C_{worst}(n) = C_{worst}([n/2]) + 1$ for $n > 1$. $C_{worst}(1) = 1$.

According to Master Theorem, the worst case efficiency of binary search is $\Theta(\log n)$.

The average number of key comparisons made by binary search is only slightly smaller than that in the worst case:

$$C_{avg}(n) \sim \log_2 n$$

Binary Tree Traversals and Related Properties

Binary Tree: 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 T_L and T_R called respectively, the left and right subtree of the root.

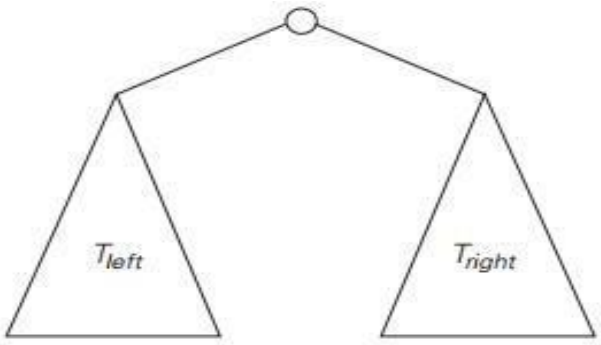


FIGURE 5.4 Standard representation of a binary tree.

Many problems about binary trees can be solved by applying the divide-and-conquer technique. For example, let us consider a recursive algorithm for computing the height of a binary tree. 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. Also note that it is convenient to define the height of the empty tree as −1. Thus, we have the following recursive algorithm.

ALGORITHM *Height(**T**)*

//Computes recursively the height of a binary tree //Input: A binary tree ***T***

//Output: The height of ***T***

if $T = \varnothing$ **return** −1

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

Tree Traversal techniques:

The most important divide-and-conquer algorithms for binary trees are the three classic traversals: preorder, inorder, and 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 that order).

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 (in that order).

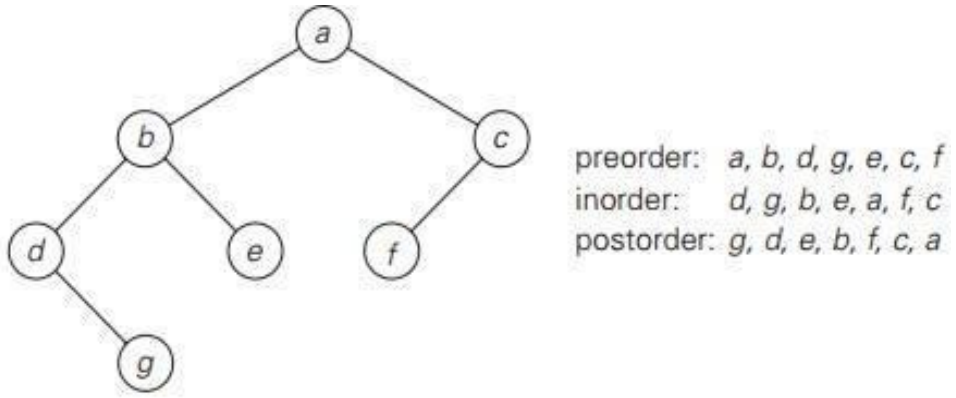


FIGURE 5.6 Binary tree and its traversals.

The number of comparisons made to compute the maximum of two numbers and the number of additions A(n(T)) made by the algorithm are the same. We have the following recurrence relation for A(n(T)):

$$A(n(T)) = A(n(T_{left})) + A(n(T_{right})) + 1 \text{ for } n(T) > 0,$$

$$A(0) = 0.$$

Decrease and Conquer

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 (recursively) or bottom up (without recursion).

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

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

Decrease by a Constant:

In the ***decrease-by-a-constant*** variation, the size of an instance is reduced by the same constant on each iteration of the algorithm. This constant is equal to one. (Eg: Insertion Sort).

Consider, an example of the exponentiation problem of computing a^n for positive integer exponents.

$$f(n) = \begin{cases} f(n-1) \cdot a & \text{if } n > 0, \\ 1 & \text{if } n = 0, \end{cases} \qquad \textbf{(4.1)}$$

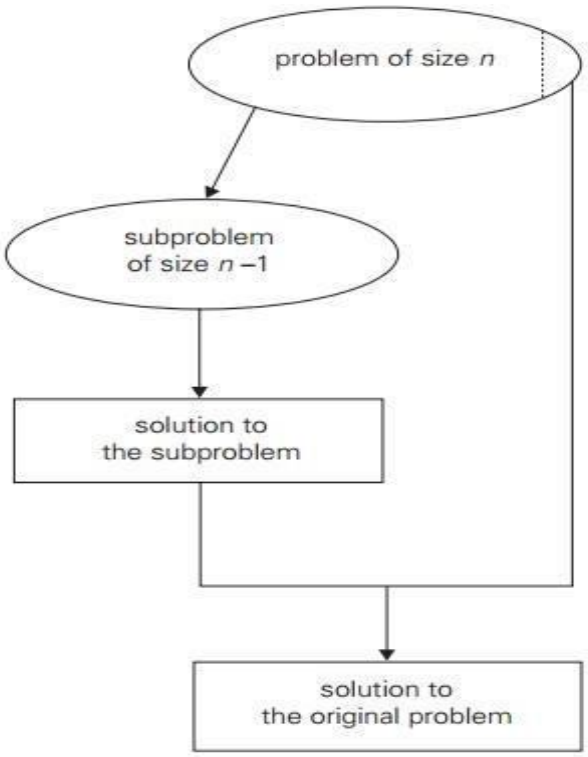


FIGURE 4.1 Decrease-(by one)-and-conquer technique.

Decrease by a Constant factor:

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. (Eg: Binary Search).

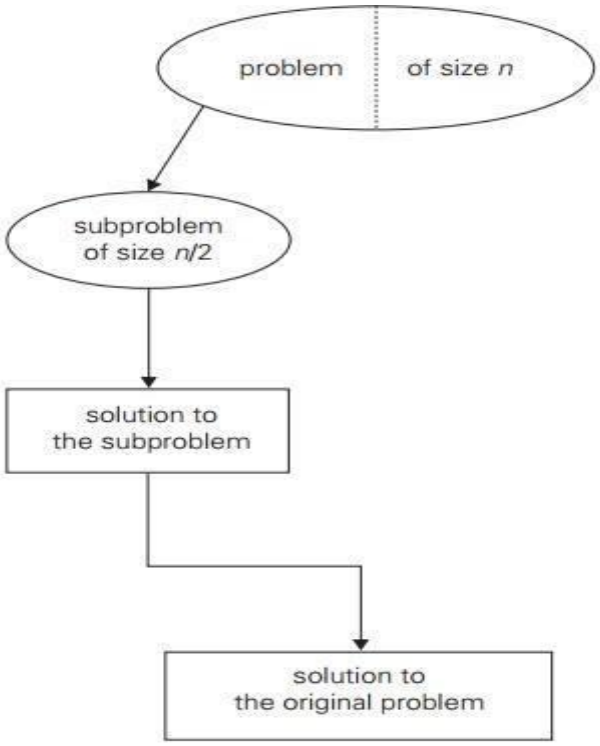


FIGURE 4.2 Decrease-(by half)-and-conquer technique.

For an example, let us revisit the exponentiation problem. If the instance of size n is to compute a^n, the instance of half its size is to compute $a^{n/2}$, with the obvious relationship between the two: $a^n = (a^{n/2})^2$.

To summarize, we have the following formula:

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even and positive,} \\ (a^{(n-1)/2})^2 \cdot a & \text{if } n \text{ is odd,} \\ 1 & \text{if } n = 0. \end{cases} \qquad \textbf{(4.2)}$$

Variable size decrease:

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

Eu-clid's algorithm for computing the greatest common divisor provides a good example of such a situation. (eg: GCD)

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

Insertion Sort

Insertion Sort is 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 $n-1$: $A[0] \le \cdots \le A[n-2]$. 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 subarray 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***.

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 $j \ge 0$ **and** $A[j] > v$ **do**

    $A[j+1] \leftarrow A[j]$

    $j \leftarrow j-1$

  $A[j+1] \leftarrow v$

Though insertion sort is clearly based on a recursive idea, it is more efficient to implement this algorithm bottom up, i.e., iteratively. As shown in Figure 4.3, 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

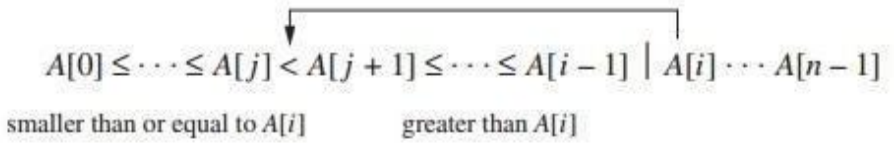


FIGURE 4.3 Iteration of insertion sort: $A[i]$ is inserted in its proper position among the preceding elements previously sorted.

Example:

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.

The basic operation of the algorithm is the key comparison $A[j] > v$. The number of key comparisons in this algorithm obviously depends on the nature of the input.

In the worst case, $A[j] > v$ is executed the largest number of times, i.e., for every $j = i - 1, \ldots, 0$. Since $v = A[i]$, it happens if and only if $A[j] > A[i]$ for $j = i - 1, \ldots, 0$. In other words, the worst-case input is an array of strictly decreasing values. The number of key comparisons for such an input is

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

In the best case, the comparison $A[j] > v$ is executed only once on every iteration of the outer loop. It happens if and only if $A[i-1] \le A[i]$ for every $i = 1, \ldots, n-1$, i.e., if the input array is already sorted in nondecreasing order.

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

On randomly ordered arrays, insertion sort makes on average half as many comparisons as on decreasing arrays, i.e.,

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

Depth-First search

Depth-first search starts a graph's traversal at an arbitrary vertex by marking it as visited. On each iteration, the algorithm proceeds to an unvisited vertex that is adjacent to the one it is currently in. As a practical matter, which of the adjacent unvisited candidates is chosen is dictated by the data structure (Stack) representing the graph.

This process continues until a dead end-a vertex with no adjacent unvisited vertices-is encountered. At a dead end, the algorithm backs up one edge to the vertex it came from and tries to continue visiting unvisited vertices from there. The algorithm eventually halts after backing up to the starting vertex, with the latter being a dead end. By then, all the vertices in the same connected component as the starting vertex have been visited. If unvisited vertices still remain, the depth-first search must be restarted at any one of them.

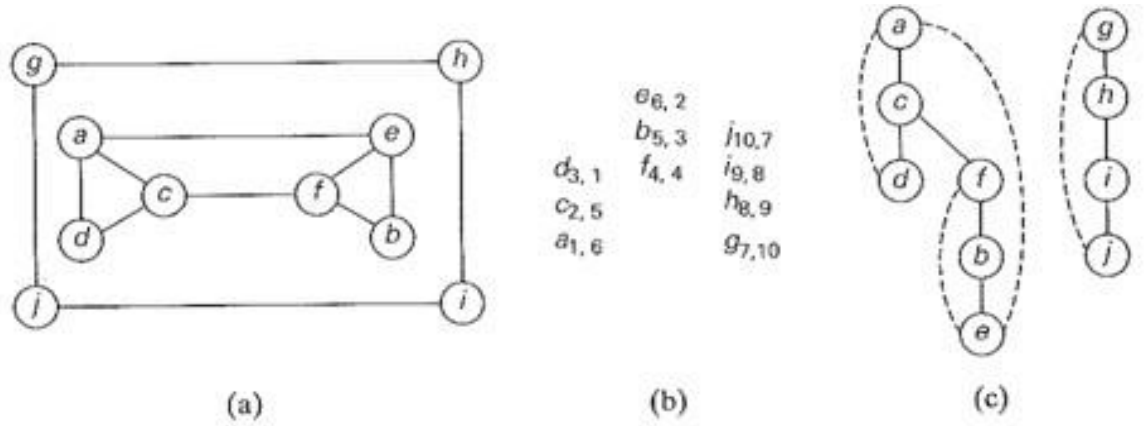


FIGURE 5.5 Example of a DFS traversal. (a) Graph. (b) Traversal's stack (the first subscript number indicates the order in which a vertex was visited, i.e., pushed onto the stack; the second one indicates the order in which it became a dead-end, i.e., popped off the stack). (c) DFS forest (with the tree edges shown with solid lines and the back edges shown with dashed lines).

It is also very useful to accompany a depth-first search traversal by constructing the so-called *depth-first search forest.* We push a vertex onto the stack when the vertex is reached for the first time (i.e., the visit of the vertex starts), and we pop a vertex off the stack when it becomes a dead end (i.e., the visit of the vertex ends).

Pseudocode of the depth-first search

```
ALGORITHM   DFS(G)
 //Implements a depth-first search traversal of a given graph
 //Input: Graph G = (V, E)
 //Output: Graph G with its vertices marked with consecutive integers
//              in the order they've been first encountered by the DFS traversal
mark each vertex in V with 0 as a mark of being "unvisited"
count ← 0
for each vertex v in V do
   if v is marked with 0
      dfs(v)
dfs(v)
 //visits recursively all the unvisited vertices connected to vertex v by a path
 //and numbers them in the order they are encountered
//via global variable count
 count ← count + 1;     mark v with count
 for each vertex w in V adjacent to v do
    if w is marked with 0
       dfs(w)
```

Thus, for the adjacency matrix representation, the traversal's time is in $\Theta(|V|^2)$, and for the adjacency list representation, it is in $\Theta(|V| + |E|)$ where $|V|$ and $|E|$ are the number of the graph's vertices and edges, respectively.

Important elementary applications of DFS include checking connectivity and checking acyclicity of a graph.

Connectivity: If all the nodes in a tree are traversed from the starting vertex and checks after the algorithm halts, whether all the graph vertices are visited or not. If all are visited then the graph is said to be having the property of connectivity.

Articulation point: A vertex of a connected graph is said to be it's articulation point if any edge in the graph is removed then the graph is breaks into disjoint pieces.

Breadth-First Search

Breadth-First Search algorithm proceeds in a concentric manner by visiting first all the vertices that are adjacent to a starting vertex, then all unvisited vertices two edges apart from it, and so on, until all the vertices in the same connected component as the starting vertex are visited. If there still remain unvisited vertices, the algorithm has to be restarted at an arbitrary vertex of another connected component of the graph.

It is convenient to use a queue to trace the operation of breadth-first search. The queue is initialized with the traversal's starting vertex, which is marked as visited. On each iteration, the algorithm identifies all unvisited vertices that are adjacent to the front vertex, marks them as visited, and adds them to the queue; after that, the front vertex is removed from the queue.

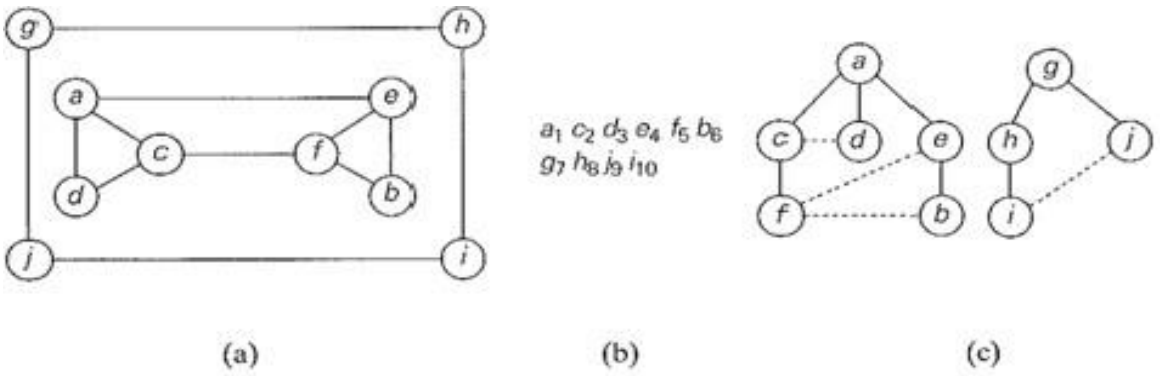


FIGURE 5.6 Example of a BFS traversal. (a) Graph. (b) Traversal's queue, with the numbers indicating the order in which the vertices were visited, i.e., added to (or removed from) the queue. (c) BFS forest (with the tree edges shown with solid lines and the cross edges shown with dotted lines).

Similarly to a DFS traversal, it is useful to accompany a BFS traversal by constructing the so-called *breadth-first search forest.* The traversal's starting vertex serves as the root of the first tree in such a forest. Whenever a new unvisited vertex is reached for the first time, the vertex is attached as a child to the vertex it is being reached from with an edge called a *tree edge.* If an edge leading to a previously visited vertex other than its immediate is encountered, the edge is noted as a *cross edge.*

Pseudocode of the breadth-first search

ALGORITHM *BFS(G)*
//Implements a breadth-first search traversal of a given graph
//Input: Graph G = (V, *E)*
//Output: Graph G with its vertices marked with consecutive integers
//in the order they have been visited by the BFS traversal
mark each vertex in *V* with 0 as a mark of being "unvisited"
***count* +- 0**
for each vertex *v* in *V* do
if *v* is marked with 0
bfs(v)
bfs(v)
//visits all the unvisited vertices connected to vertex *v* by a path
//and assigns them the numbers in the order they are visited
//via global variable *count*
count* +-*count* + 1; mark *v* with *count* and initialize a queue with *v
while the queue is not empty do
for each vertex *w* in *V* adjacent to the front vertex do
if *w* is marked with 0
count* +-- *count* + 1; mark *w* with *count
add *w* to the queue
remove the front vertex from the queue

Breadth-first search has the same efficiency as depth-first search: it is in $\Theta(|V|^2)$ for the adjacency matrix representation and in $\Theta(|V| + |E|)$ for the adjacency list representation.

Differences between DFS and BFS:

TABLE 5.1 Main facts about depth-first search (DFS) and breadth-first search (BFS)

	DFS	BFS
Data structure	stack	queue
No. of vertex orderings	2 orderings	1 ordering
Edge types (undirected graphs)	tree and back edges	tree and cross edges
Applications	connectivity, acyclicity, articulation points	connectivity, acyclicity, minimum-edge paths
Efficiency for adjacent matrix	$\Theta(\|V^2\|)$	$\Theta(\|V^2\|)$
Efficiency for adjacent lists	$\Theta(\|V\| + \|E\|)$	$\Theta(\|V\| + \|E\|)$

Topological Sorting

A ***directed graph***, or ***digraph*** for short, is a graph with directions specified for all its edges. 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.

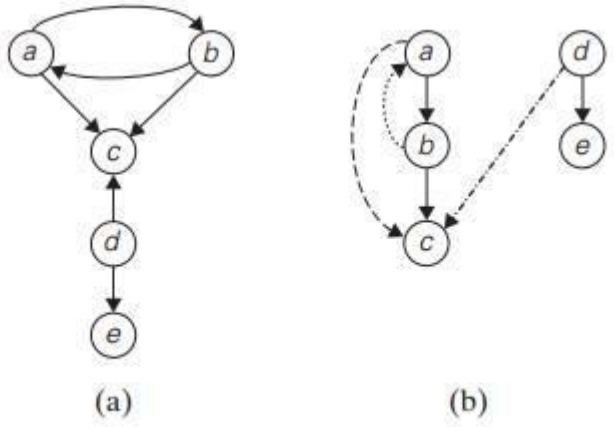


FIGURE 4.5 (a) Digraph. (b) DFS forest of the digraph for the DFS traversal started at *a*.

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.

For example, a set of five required courses {C1, C2, C3, C4, C5} a part-time student has to take in some degree program. The courses can be taken in any order as long as the following course prerequisites are met: C1 and C2 have no prerequisites, C3 requires C1 and C2, C4 requires C3, and C5 requires C3 and C4. The student can take only one course per term. In which order should the student take the courses?

There are two efficient algorithms that both verify whether a digraph is a dag and, if it is, produce an ordering of vertices that solves the topological sorting problem.

- The first algorithm is a simple application of depth-first search: perform a DFS traversal and note the order in which vertices become dead-ends (i.e., popped off the traversal stack).

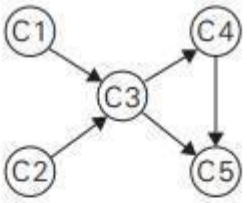


FIGURE 4.6 Digraph representing the prerequisite structure of five courses.

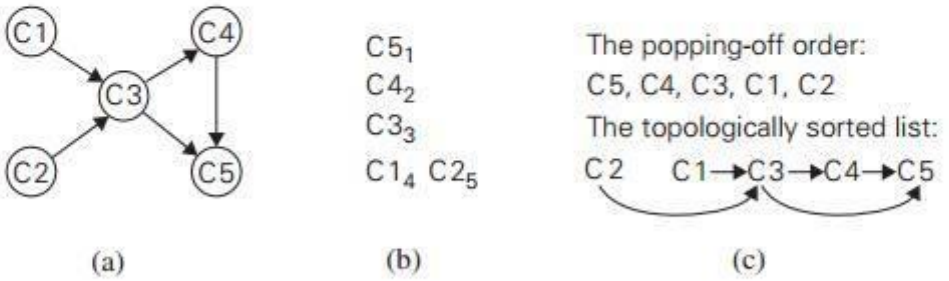


FIGURE 4.7 (a) Digraph for which the topological sorting problem needs to be solved. (b) DFS traversal stack with the subscript numbers indicating the popping-off order. (c) Solution to the problem.

- The second algorithm is based on a direct implementation of the decrease-(by one)-and-conquer technique: repeatedly, identify in a remaining digraph a ***source***, which is a vertex with no incoming edges, and delete it along with all the edges outgoing from it.

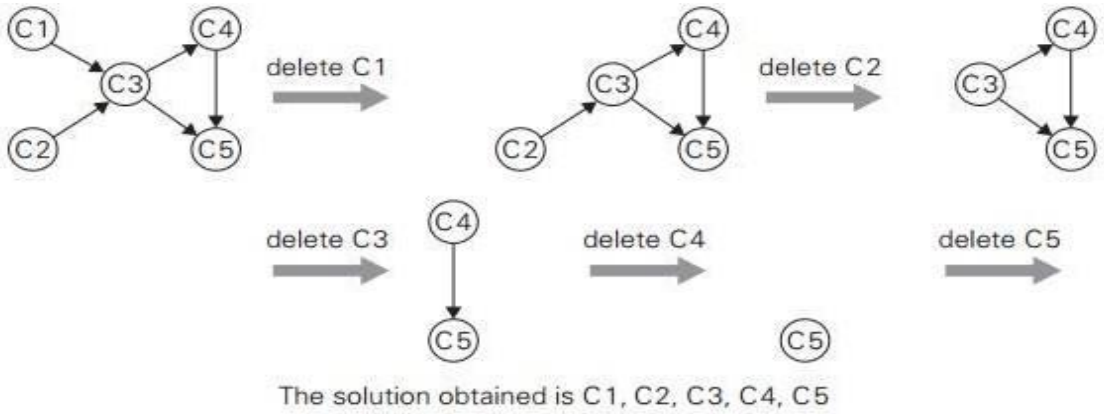


FIGURE 4.8 Illustration of the source-removal algorithm for the topological sorting problem. On each iteration, a vertex with no incoming edges is deleted from the digraph.

Decrease-by-a-Constant-Factor Algorithms

Decrease-by-a-constant-factor is the second major variety of decrease-and-conquer. Decrease by a constant factor algorithms usually run in logarithmic time, and being very efficient, do not happen often, a reduction by a factor other than two is especially rare.

The problems that come under this concept are:

1. Binary Search
2. Fake-coin Problem
3. Josephus Problem

Binary Search: (See Divide-and-Conquer chapter)

Fake-Coin Problem:

It is one of the best method decrease-by-a-constant-factor strategy. Among '*n'* identically looking coins, one is fake. With a balance scale, we can compare any two sets of coins. That is, by tipping to the left, to the right, or staying even. The balance scale will turn according to the weights placed on either sides of the balance scale. The fake-coin is one which is showing either lighter or heavier than genuine one. We always assume a lighter weighted by pile has a fake-coin.

The most natural idea for solving this problem is to divide *n* coins into two piles of (n/2) coins each, leaving one extra coin apart if *n* is odd, and put the two piles on the scale. If the piles weigh the same, the coin put aside must be fake; otherwise, we can proceed in the same manner with the lighter pile, which must be the one with the fake coin.

The recurrence relation for number of weighing:

$$W(n) = W(n/2) + 1 \text{ for } n > 1,$$
$$W(1) = 0 \quad \text{for } n = 1.$$

The order of growth is similar to the binary search, that is,

$$W(n) = \log_2 n.$$

But, the algorithm may not be efficient in all the cases. So, it will be better dividing the coin into 3 piles instead of 2 piles. The order of growth in this case is

$$W(n) = \log_3 n.$$

Transform-and Conquer

We call this general technique ***transform-and-conquer*** because these methods work as two-stage procedures.

1. Transformation Stage
2. Conquering Stage

First, in the transformation stage, the problem's instance is modified to be, for one reason or another, more amenable to solution. Then, in the second or conquering stage, it is solved.

There are three major variations:

1. Transformation to a simpler or more convenient instance of the same problem. This is called ***'instance simplification'***.

2. Transformation to a different representation of the same instance. This is called '***representation change'***.

3. Transformation to an instance of a different problem for which an algorithm is already available. This is called '***problem reduction'***.

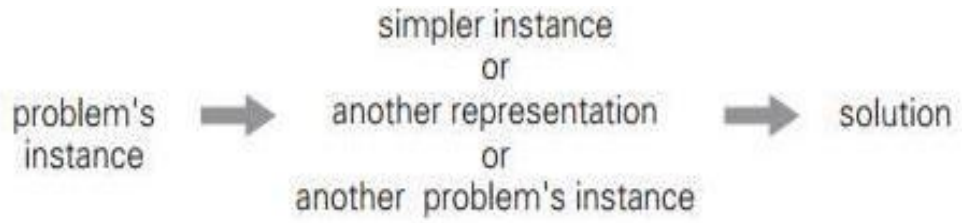


FIGURE 6.1 Transform-and-conquer strategy.

Balanced Search Trees

Binary Search Tree: Binary search tree is one of the principal data structures for implementing dictionaries. It is a binary tree whose nodes contain elements of a set of orderable items, one element per node, so that all elements in the left subtree are smaller than the element in the subtree's root, and all the elements in the right subtree are greater than it.

Note that this transformation from a set to a binary search tree is an example of the representation-change technique.

We gain in the time efficiency of searching, insertion, and deletion, which are all in ***(***log ***n)***, but only in the av-erage case. In the worst case, these operations are in ***(n)*** because the tree can degenerate into a severely unbalanced one with its height equal to ***n*** − 1.

Computer scientists have expended a lot of effort in trying to find a structure that preserves the good properties of the classical binary search tree. They have come up with two approaches.

1. The first approach is of **the instance-simplification** variety: an unbalanced binary search tree is transformed into a balanced one. Because of this, such trees are called ***self-balancing***. An ***AVL tree*** requires the difference between the heights of the left and right subtrees of every node never exceed 1.
2. The second approach is of the **representation-change variety**: allow more than one element in a node of a search tree. Specific cases of such trees are ***2-3 trees***, ***2-3-4 trees***, and more general and important ***B-trees***.

AVL Trees

AVL trees were invented in 1962 by two Russian scientists, G. M. Adelson-Velsky and E. M. Landis [Ade62], after whom this data structure is named.

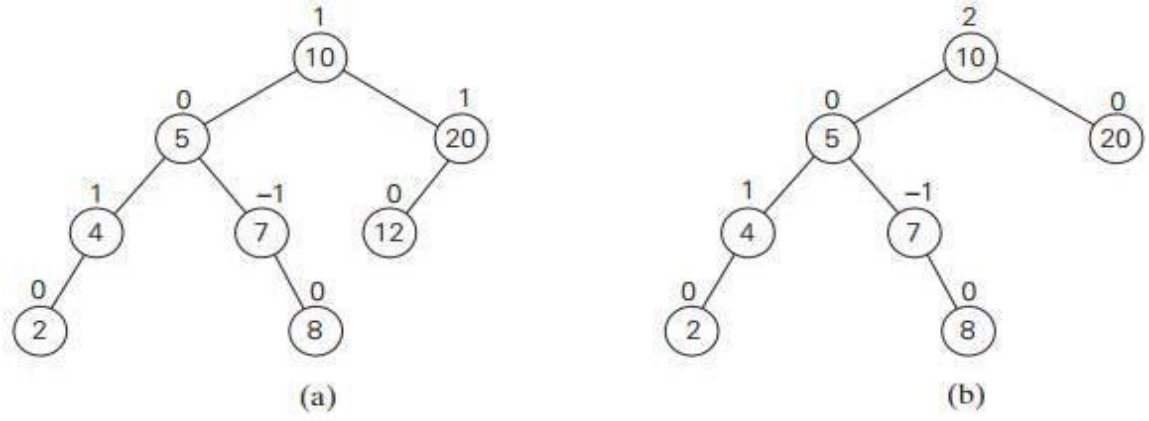


FIGURE 6.2 (a) AVL tree. (b) Binary search tree that is not an AVL tree. The numbers above the nodes indicate the nodes' balance factors.

DEFINITION:

An ***AVL tree*** is a binary search tree in which the ***balance factor*** of every node, which is defined as the difference between the heights of the node's left and right subtrees, is either 0 or +1 or −1. (The height of the empty tree is defined as −1. Of course, the balance factor can also be computed as the difference between the numbers of levels rather than the height difference of the node's left and right subtrees.)

If an insertion of a new node makes an AVL tree unbalanced, we transform the tree by a rotation. A ***rotation*** in an AVL tree is a local transformation of its subtree rooted at a node whose balance has become either +2 or −2. If there are several such nodes, we rotate the tree rooted at the unbalanced node that is the closest to the newly inserted leaf. There are only four types of rotations; in fact, two of them are mirror images of the other two. In their simplest form, the four rotations are shown in Figure.

The first rotation type is called the ***single right rotation***, or ***R-rotation***. (Imagine rotating the edge connecting the root and its left child in the binary tree in Figure (a) to the right). This rotation is performed after a new key is inserted into the left subtree of the left child of a tree whose root had the balance of +1 before the insertion.

The symmetric ***single left rotation***, or ***L-rotation***, is the mirror image of the single ***R***-rotation. It is performed after a new key is inserted into the right subtree of the right child of a tree whose root had the balance of −1 before the insertion.

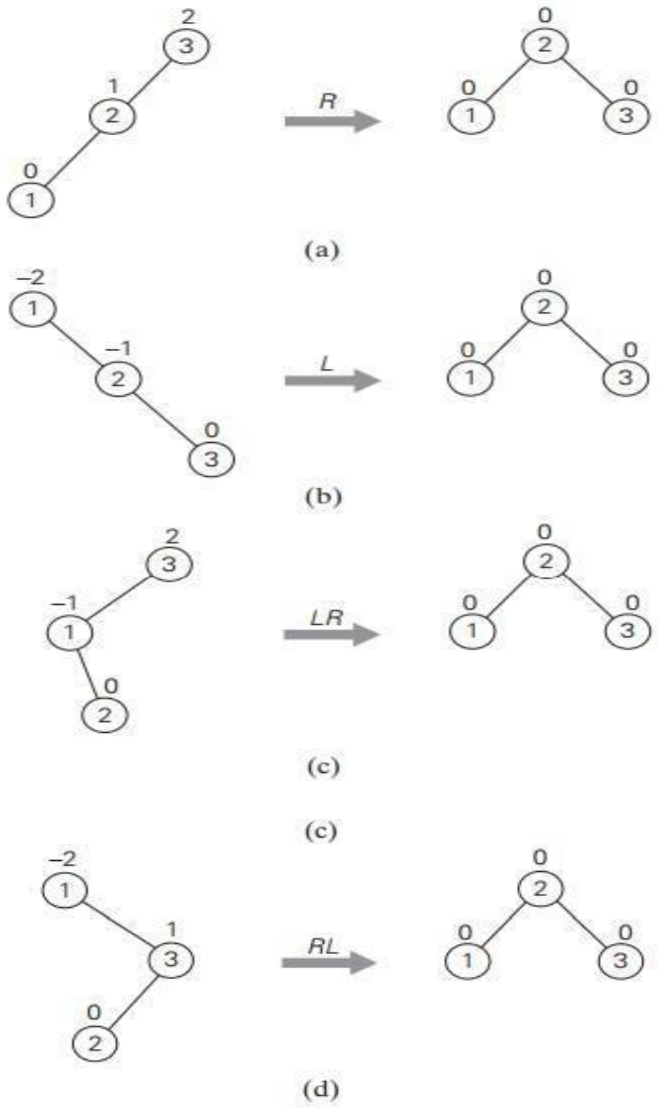


FIGURE 6.3 Four rotation types for AVL trees with three nodes. (a) Single *R*-rotation. (b) Single *L*-rotation. (c) Double *LR*-rotation. (d) Double *RL*-rotation.

The second rotation type is called the ***double left-right rotation*** (***LR-rotation***). It is, in fact, a combination of two rotations: we perform the *L*-rotation of the left subtree of root ***r*** followed by the *R*-rotation of the new tree rooted at ***r*** (Figure 6.5). It is performed after a new key is inserted into the right subtree of the left child of a tree whose root had the balance of +1 before the insertion.

The ***double right-left rotation*** (***RL-rotation***) is the mirror image of the double ***LR***-rotation.

Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertion. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

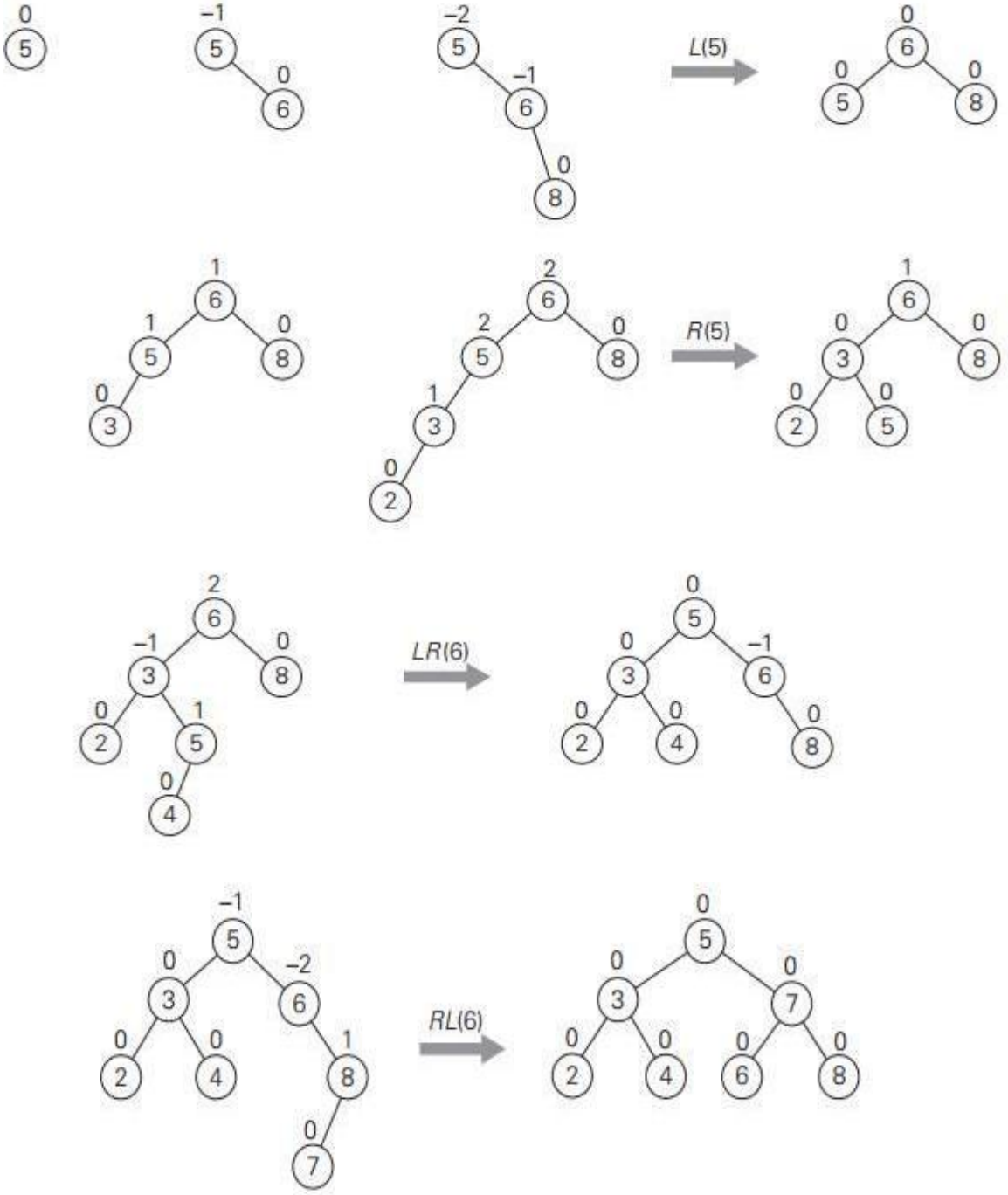


FIGURE 6.6 Construction of an AVL tree for the list 5, 6, 8, 3, 2, 4, 7 by successive insertions. The parenthesized number of a rotation's abbreviation indicates the root of the tree being reorganized.

2-3 Trees

The second idea of balancing a search tree is to allow more than one key in the same node of such a tree.

A ***2-3 tree*** is a tree that can have nodes of two kinds: 2-nodes and 3-nodes.

A ***2-node*** contains a single key ***K*** and has two children: the left subtree keys are less than ***K***, and the right subtree keys are greater than ***K.***

A ***3-node*** contains two ordered keys K_1 and K_2 ($K_1 < K_2$) and has three children. The leftmost subtree with keys less than K_1, the middle subtree with keys between K_1 and K_2, and the rightmost subtree with keys greater than K_2.

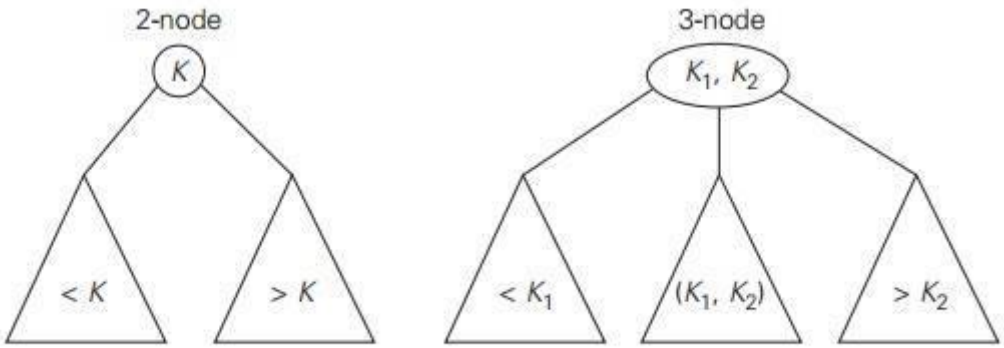


FIGURE 6.7 Two kinds of nodes of a 2-3 tree.

The last requirement of the 2-3 tree is that all its leaves must be on the same level. i.e, A 2-3 tree is always perfectly height-balanced.

Searching for a given key ***K*** in a 2-3 tree is quite straightforward. We start at the root. If the root is a 2-node, we act as if it were a binary search tree. If the root is a 3-node, we know after no more than two key comparisons whether the search can be stopped (if ***K*** is equal to one of the root's keys) or in which of the root's three subtrees it needs to be continued.

Inserting a new key in a 2-3 tree is done as follows. We always insert a new key ***K*** in a leaf, except for the empty tree. The appropriate leaf is found by performing a search for ***K***. If the leaf is a 2-node, we insert ***K*** there as either the first or the second key, depending on whether ***K*** is smaller or larger than the node's old key. If the leaf is a 3-node, we split the leaf in two: the smallest of the three keys (two old ones and the new key) is put in the first leaf, the largest key is put in the second leaf, and the middle key is promoted to the old leaf's parent.

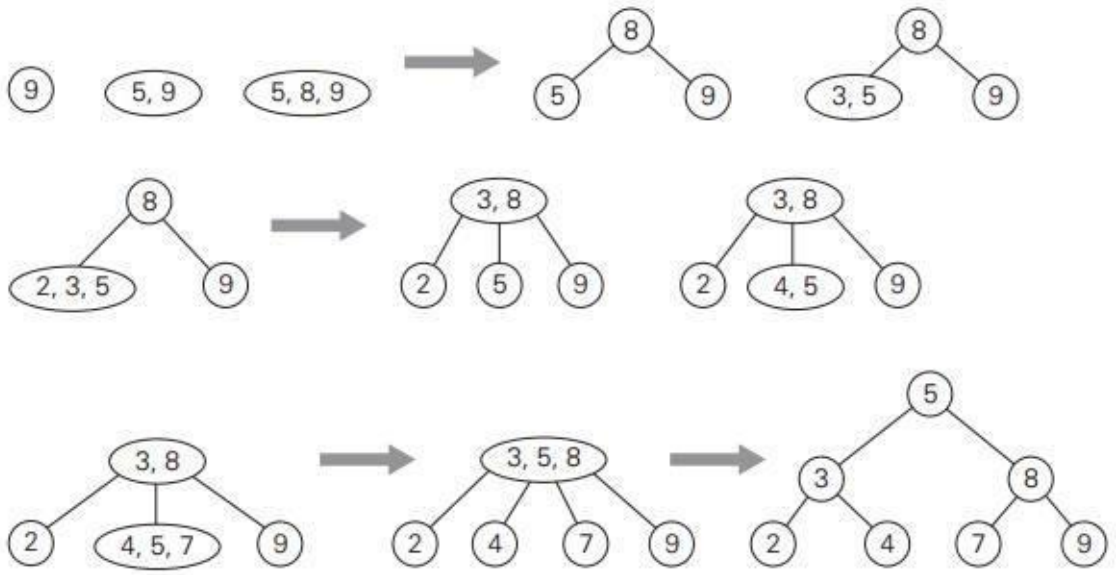


FIGURE 6.8 Construction of a 2-3 tree for the list 9, 5, 8, 3, 2, 4, 7.

As for any search tree, the efficiency of the dictionary operations depends on the tree's height. So let us first find an upper bound for it. A 2-3 tree of height ***h*** with the smallest number of keys is a full tree of 2-nodes. Therefore, for any 2-3 tree of height ***h*** with ***n*** nodes, we get the inequality

$$n \geq 1 + 2 + \cdots + 2^h = 2^{h+1} - 1,$$

and hence

$$h \leq \log_2(n+1) - 1.$$

On the other hand, a 2-3 tree of height ***h*** with the largest number of keys is a full tree of 3-nodes, each with two keys and three children. Therefore, for any 2-3 tree with ***n*** nodes,

$$n \leq 2 \cdot 1 + 2 \cdot 3 + \cdots + 2 \cdot 3^h = 2(1 + 3 + \cdots + 3^h) = 3^{h+1} - 1$$

and hence

$$h \geq \log_3(n+1) - 1.$$

These lower and upper bounds on height ***h,***

$$\log_3(n+1) - 1 \leq h \leq \log_2(n+1) - 1,$$

imply that the time efficiencies of searching, insertion, and deletion are all in $(\log n)$ in both the worst and average case.

Heaps and Heapsort

Heap is a partially ordered data structure that is especially suitable for implementing priority queues. A ***priority queue*** is a multiset of items with an orderable characteristic called an item's ***priority.***

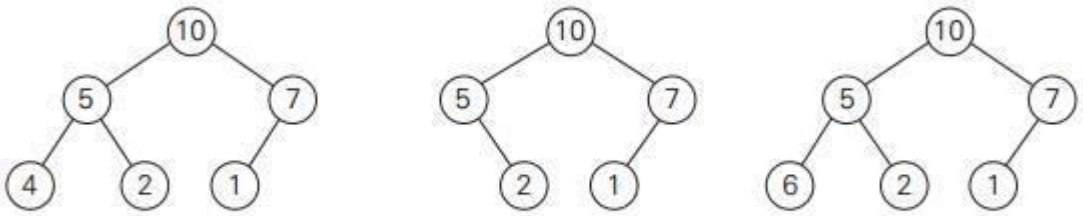


FIGURE 6.9 Illustration of the definition of heap: only the leftmost tree is a heap.

DEFINITION: A ***heap*** can be defined as a binary tree with keys assigned to its nodes, one key per node, provided the following two conditions are met:

1. **The tree's shape property**: The binary tree is ***essentially complete,*** i.e., all its levels are full except possibly the last level, where only some rightmost leaves may be missing.

2. **The parental dominance or heap property**: The key in each node is greater than or equal to the keys in its children. (This condition is considered auto-matically satisfied for all leaves.)

Properties of Heaps:

- The root of a heap always contains its largest element.
- A node of a heap considered with all its descendants is also a heap.
- There exists exactly one essentially complete binary tree with ***n*** nodes. Its height is equal to $\log_2 n$.

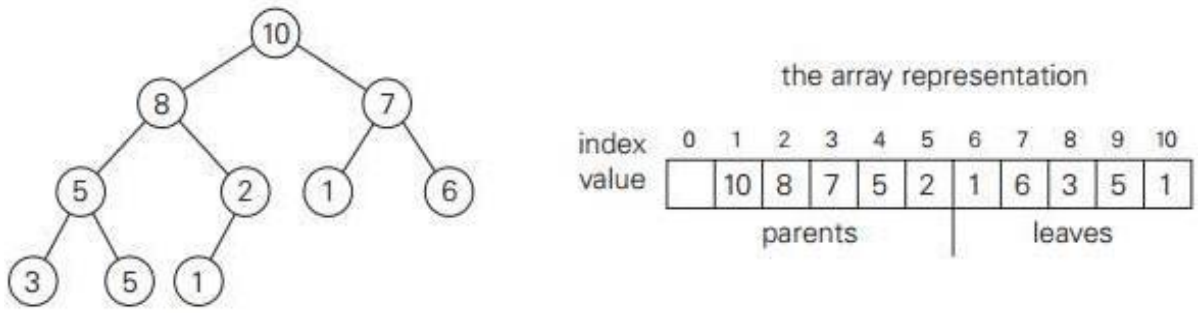


FIGURE 6.10 Heap and its array representation.

- The parental node keys will be in the first $n/2$ positions of the array, while the leaf keys will occupy the last $n/2$ positions.
- The children of a key in the array's parental position i ($1 \le i \le n/2$) will be in positions $2i$ and $2i + 1$, and, correspondingly, the parent of a key in position i ($2 \le i \le n$) will be in position $i/2$.

Heap Sort

Heapsort is an interesting sorting algorithm discovered by J. W. J. Williams [Wil64]. This is a two-stage algorithm that works as follows.

Stage 1 (heap construction): Construct a heap for a given array.

Stage 2 (maximum deletions): Apply the root-deletion operation $n - 1$ times to the remaining heap.

Stage 1: Bottom-up heap construction algorithm:

The list is 2, 9, 7, 6, 5, 8.

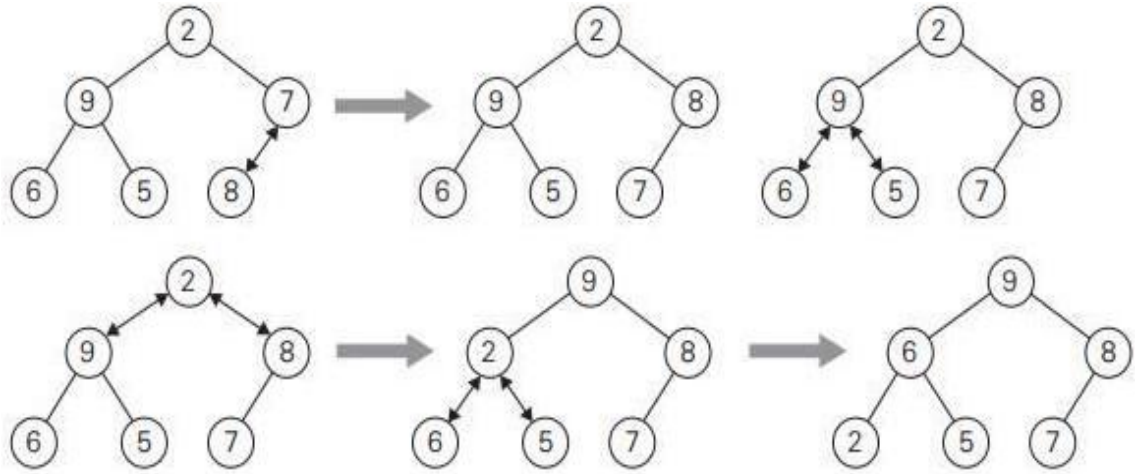


FIGURE 6.11 Bottom-up construction of a heap for the list 2, 9, 7, 6, 5, 8. The double-headed arrows show key comparisons verifying the parental dominance.

ALGORITHM *HeapBottomUp(H [1..n])*

```
//Constructs a heap from elements of a given array
// by the bottom-up algorithm
//Input: An array H [1..n] of orderable items
//Output: A heap H [1..n]
for i ← [n/2] downto 1 do
    k ← i;     v ← H [k]
    heap ← false
    while not heap and 2 * k ≤ n do
        j ← 2 * k
        if j < n //there are two children
            if H [j ] < H [j + 1]    j ← j + 1
        if v ≥ H [j ]
            heap ← true
        else H [k] ← H [j ];      k ← j
    H [k] ← v
```

Efficiency in the worst case: Assume that $n = 2^k - 1$ so that a heap's tree is full, i.e., the largest possible number of nodes occurs on each level. Let h be the height of the tree. i.e, $h = \log_2 n$. Each key on level i of the tree will travel to the leaf level h in the worst case.

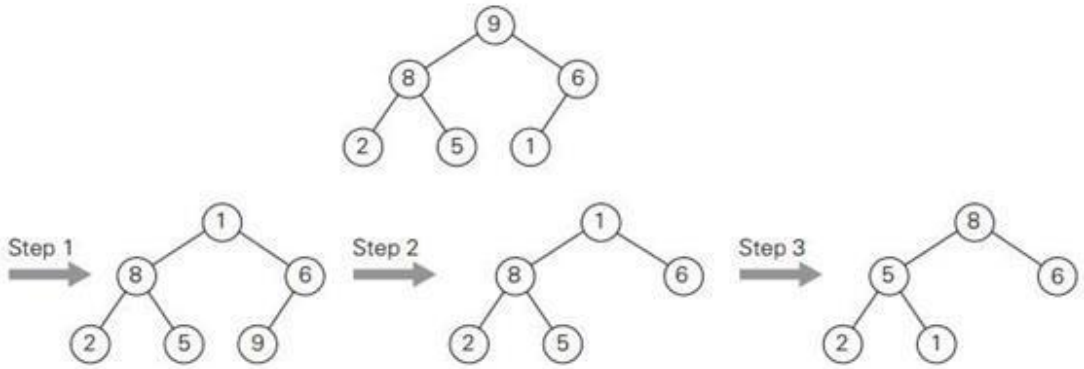
$$C_{worst}(n) = \sum_{i=0}^{h-1} \sum_{\text{level } i \text{ keys}} 2(h-i) = \sum_{i=0}^{h-1} 2(h-i)2^i = 2(n - \log_2(n+1)),$$

Stage 2: Maximum Key Deletion from a heap:

Step 1 Exchange the root's key with the last key ***K*** of the heap.

Step 2 Decrease the heap's size by 1.

Step 3 "Heapify" the smaller tree by sifting ***K*** down the tree exactly in the same way we did it in the bottom-up heap construction algorithm. That is, verify the parental dominance for ***K***: if it holds, we are done; if not, swap ***K*** with the larger of its children and repeat this operation until the parental dominance condition holds for ***K*** in its new position.



Deleting the root's key from a heap. The key to be deleted is swapped with the last key after which the smaller tree is "heapified" by exchanging the new key in its root with the larger key in its children until the parental dominance requirement is satisfied.

Efficiency of second stage: The efficiency of deletion is determined by the number of key comparisons needed to "heapify" the tree after the swap has been made and the size of the tree is decreased by 1. Time efficiency of deletion is in O(log n).

Stage 1 (heap construction)	Stage 2 (maximum deletions)
2 9 **7** 6 5 8	**9** 6 8 2 5 7
2 **9** 8 6 5 7	7 6 8 2 5 | **9**
2 9 8 6 5 7	**8** 6 7 2 5
9 **2** 8 6 5 7	5 6 7 2 | **8**
9 6 8 2 5 7	**7** 6 5 2
	2 6 5 | **7**
	6 2 5
	5 2 | **6**
	5 2
	2 | **5**
	2

FIGURE 6.14 Sorting the array 2, 9, 7, 6, 5, 8 by heapsort.

Problem Reduction

If you need to solve a problem, reduce it ti another problem that you know how to solve. This strategy is called 'Problem Reduction'.

To solve an instance of problem A:

- Transform the instance of problem A into an instance of problem B
- Solve the instance of problem B
- Transform the solution to problem B into a solution of problem A

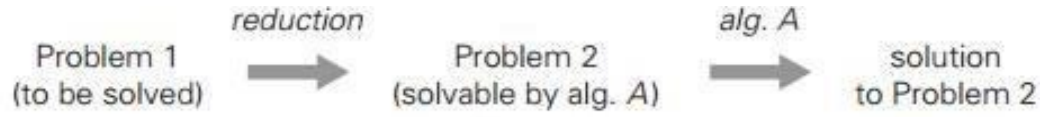


FIGURE 6.15 Problem reduction strategy.

Computing the Least Common Multiple

The ***least common multiple*** of two positive integers ***m*** and ***n***, denoted lcm***(m, n***), is defined as the smallest integer that is divisible by both ***m*** and ***n***.

Given the prime factorizations of ***m*** and ***n***, compute the product of all the common prime factors of ***m*** and ***n,*** all the prime factors of ***m*** that are not in ***n,*** and all the prime factors of ***n*** that are not in ***m***. For example,

$$24 = 2 \cdot 2 \cdot 2 \cdot 3,$$
$$60 = 2 \cdot 2 \cdot 3 \cdot 5,$$
$$\text{lcm}(24, 60) = (2 \cdot 2 \cdot 3) \cdot 2 \cdot 5 = 120.$$

A much more efficient algorithm for computing the least common multiple can be devised by using problem reduction. After all, there is a very efficient algorithm (Euclid's algorithm) for finding the greatest common divisor, which is a product of all the common prime factors of ***m*** and ***n***. It is not difficult to see that the product of lcm***(m, n)*** and gcd***(m, n)*** includes every factor of ***m*** and ***n*** exactly once and hence is simply equal to the product of ***m*** and ***n***. This observation leads to the formula

$$\text{lcm}(m, n) = \frac{m \cdot n}{\text{gcd}(m, n)},$$

where gcd***(m, n)*** can be computed very efficiently by Euclid's algorithm.

****

UNIT-III

Dynamic Programming

Dynamic Programming is one of the algorithm design technique that can be used to solve the problems where the solution of the problem will be given as result of sequence of decisions.

It is invented by a prominent U.S. mathematician, Richard Bellman, in the 1950s as a general method for optimizing multistage decision processes.

Dynamic Programming is a technique for solving problems with overlapping subproblems. Typically, these subproblems arise from a recurrence relating a solution to a given problem with solutions to its smaller subproblems of the same type.

Example: The Fibonacci numbers are the elements of the sequence

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, \ldots ,$$

which can be defined by the simple recurrence

$$F(n) = F(n- 1) + F(n- 2) \text{ for } n > 2$$

and two initial conditions

$$F(0) = 0, \quad F(1) = 1.$$

Dynamic programming design involves 4 major steps.

1) Characterize the structure of optimal solution.

2) Recursively define the value of an optimal solution.

3) Compute the value of an optimum solution in a bottom up fashion.

4) Construct an optimum solution from computed information.

Applications of Dynamic Programming Approach

- Matrix Chain Multiplication
- Longest Common Subsequence
- Travelling Salesman Problem
- Knapsack Problem
- Optimal Binary search Trees

Warshall's and Floyd's Algorithm

Warshall's algorithm for computing the transitive closure of a directed graph and Floyd's algorithm for the all-pairs shortest-paths problem.

Warshall's Algorithm

The adjacency matrix $A = \{a_{i\,j}\}$ of a directed graph is the boolean matrix that has 1 in its i^{th} row and j^{th} column if and only if there is a directed edge from the i^{th} vertex to the j^{th} vertex. We may also be interested in a matrix containing the information about the existence of directed paths of arbitrary lengths between vertices of a given graph.

DEFINITION: The '***transitive closure'*** of a directed graph with *n* vertices can be defined as the *n-by-n* boolean matrix $T = \{t_{i\,j}\}$, in which the element in the i^{th} row ($1 \le i \le n$) and the j^{th} column ($1 \le j \le n$) is 1 if there exists a nontrivial directed path from the i^{th} vertex to the j^{th} vertex; otherwise, $t_{i\,j}$ is 0.

Example:

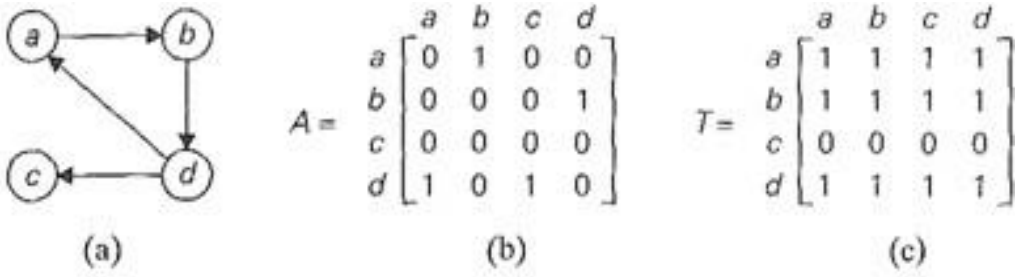


FIGURE 8.2 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

We can generate the transitive closure of a digraph with the help of depth-first search or breadth-first search. Since, by using these methods which traverses the same digraph several times. So, we should go for the better algorithm. i.e, called as Warshall's algorithm.

The transitive closure of a given digraph with *n* vertices through a series of n x n boolean matrices:

$$R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots, R^{(n)}.$$

The element $r_{i\,j}^{(k)}$ in the i^{th} row and j^{th} column of matrix $R^{(k)}$ (k = 0, 1, ... , n) is equal to 1 iff there exists a directed path from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any, numbered not higher than 'k'.

The series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is the adjacency matrix of the digraph.

$R^{(1)}$ contains the information about paths that can given the first vertex as the intermediate vertex. In the same manner this process will be continued for $R^{(n)}$ vertices.

v_n is a list of intermediate vertices not higher tha k, v_j.

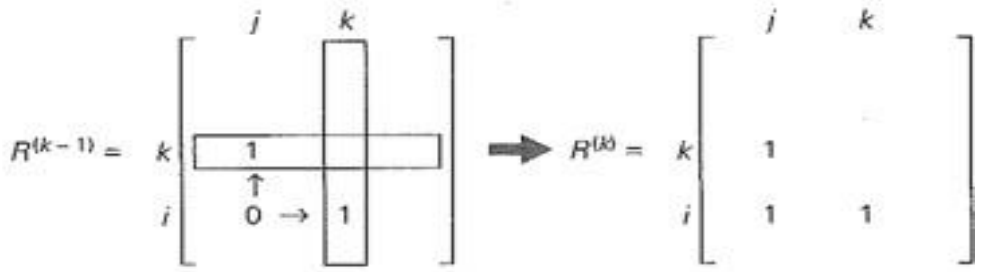


FIGURE 8.3 Rule for changing zeros in Warshall's algorithm

Pseudocode of Warshall's Algorithm:

ALGORITHM *Warshall*$(A[1..n, 1..n])$

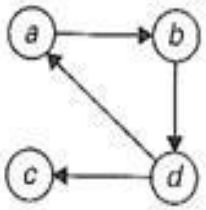
//Implements Warshall's algorithm for computing the transitive closure
//Input: The adjacency matrix A of a digraph with n vertices
//Output: The transitive closure of the digraph
$R^{(0)} \leftarrow A$
for $k \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** n **do**
    for $j \leftarrow 1$ **to** n **do**
      $R^{(k)}[i, j] \leftarrow R^{(k-1)}[i, j]$ **or** $(R^{(k-1)}[i, k]$ **and** $R^{(k-1)}[k, j])$
return $R^{(n)}$

The adjacency matrix $R^{(0)}$ is

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

We have just proved is that if $r_{ij}^{(k)} = 1$, then either $r_{ij}^{(k-1)} = 1$ or both $r_{ik}^{(k-1)} = 1$ and $r_{kj}^{(k-1)} = 1$. It is easy to see that the converse of this assertion is also true. Thus, we have the following formula for generating the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$:

$$r_{ij}^{(k)} = r_{ij}^{(k-1)} \text{ or } \left(r_{ik}^{(k-1)} \text{ and } r_{kj}^{(k-1)}\right).$$



$R^{(0)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	0	1	0

Ones reflect the existence of paths with no intermediate vertices ($R^{(0)}$ is just the adjacency matrix); boxed row and column are used for getting $R^{(1)}$.

$R^{(1)}$ =

	a	b	c	d
a	0	1	0	0
b	0	0	0	1
c	0	0	0	0
d	1	**1**	1	0

Ones reflect the existence of paths with intermediate vertices numbered not higher than 1, i.e., just vertex *a* (note a new path from *d* to *b*); boxed row and column are used for getting $R^{(2)}$.

$R^{(2)}$ =

	a	b	c	d
a	0	1	0	**1**
b	0	0	0	1
c	0	0	0	0
d	1	1	1	**1**

Ones reflect the existence of paths with intermediate vertices numbered not higher than 2, i.e., *a* and *b* (note two new paths); boxed row and column are used for getting $R^{(3)}$.

$R^{(3)}$ =

	a	b	c	d
a	0	1	0	1
b	0	0	0	1
c	0	0	0	0
d	1	1	1	1

Ones reflect the existence of paths with intermediate vertices numbered not higher than 3, i.e., *a*, *b*, and *c* (no new paths); boxed row and column are used for getting $R^{(4)}$.

$R^{(4)}$ =

	a	b	c	d
a	**1**	1	**1**	1
b	**1**	**1**	**1**	1
c	0	0	0	0
d	1	1	1	1

Ones reflect the existence of paths with intermediate vertices numbered not higher than 4, i.e., *a*, *b*, *c*, and *d* (note five new paths).

FIGURE 8.4 Application of Warshall's algorithm to the digraph shown. New ones are in bold.

The time complexity of Warshall's algorithm is $O(n^2)$.

Floyd's Algorithm

The weighted connected graph (undirected or directed), the all-pairs shortest paths problem asks to find the distances from each vertex to all other vertices. It is convenient to record the lengths of shortest paths in an n x n matrix D called the distance matrix.

Weighted matrix: This s a matrix without consideration of any intermediatory nodes.

Distance matrix: The element $d_{i\ j}$ in the i^{th} row and the j^{th} column of this matrix indicates the length of the shortest path from the i^{th} vertex to the j^{th} vertex ($1 \le i, j \le n$).

Example:

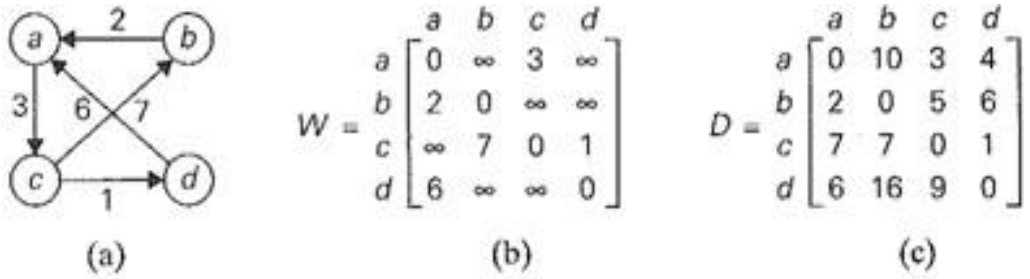


FIGURE 8.5 (a) Digraph. (b) Its weight matrix. (c) Its distance matrix.

We can generate the distance matrix with an algorithm that is very similar to Warshall's algorithm. It is called Floyd's algorithm.

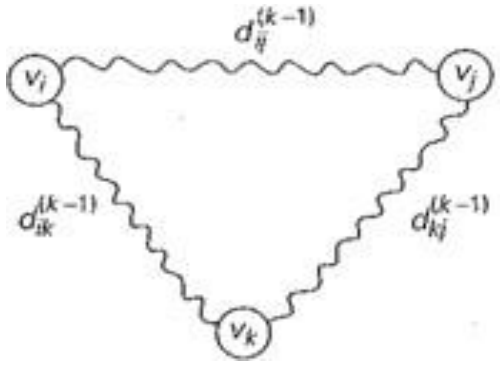
Floyd's algorithm computes the distance matrix of a weighted graph with 'n' vertices through a series of n x n matrices:

$$D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}.$$

Each of these matrices contains the lengths of shortest paths with certain constraints on the paths considered for the matrix in question. Specifically, the element $d^{(k)}_{i\ j}$ in the i^{th} row and the j^{th} column of matrix $D^{(k)}$ ($k = 0, 1, \ldots, n$) is equal to the length of the shortest path among all paths from the i^{th} vertex to the j^{th} vertex with each intermediate vertex, if any, numbered not higher than 'k'.

Here, $D^{(0)}$ is the weight matrix of the graph. The last matrix in the series, $D^{(n)}$ contains the lengths of the shortest paths among all paths.

The idea of Floyd's Algorithm:



The shortest paths in both subsets leads to the following recurrence:

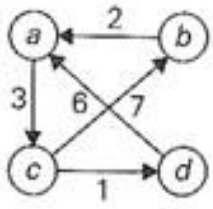
$$d_{ij}^{(k)} = \min\{d_{ij}^{(k-1)},\ d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\} \quad \text{for } k \geq 1, \quad d_{ij}^{(0)} = w_{ij}.$$

The pseudocode of Floyd's algorithm:

ALGORITHM *Floyd*$(W[1..n, 1..n])$

```
//Implements Floyd's algorithm for the all-pairs shortest-paths problem
//Input: The weight matrix W of a graph with no negative-length cycle
//Output: The distance matrix of the shortest paths' lengths
D ← W //is not necessary if W can be overwritten
for k ← 1 to n do
    for i ← 1 to n do
        for j ← 1 to n do
            D[i, j] ← min{D[i, j], D[i, k] + D[k, j]}
return D
```

The time efficiency of Floyd's algorithm is cubic-as is the time efficiency of Warshall's algorithm.



$D^{(0)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	∞	∞
c	∞	7	0	1
d	6	∞	∞	0

Lengths of the shortest paths with no intermediate vertices ($D^{(0)}$ is simply the weight matrix).

$D^{(1)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	**5**	∞
c	∞	7	0	1
d	6	∞	**9**	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 1, i.e. just *a* (note two new shortest paths from *b* to *c* and from *d* to *c*).

$D^{(2)} =$

	a	b	c	d
a	0	∞	3	∞
b	2	0	5	∞
c	**9**	7	0	1
d	6	∞	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 2, i.e. *a* and *b* (note a new shortest path from *c* to *a*).

$D^{(3)} =$

	a	b	c	d
a	0	**10**	3	**4**
b	2	0	5	**6**
c	9	7	0	1
d	6	**16**	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 3, i.e. *a*, *b*, and *c* (note four new shortest paths from *a* to *b*, from *a* to *d*, from *b* to *d*, and from *d* to *b*).

$D^{(4)} =$

	a	b	c	d
a	0	10	3	4
b	2	0	5	6
c	**7**	7	0	1
d	6	16	9	0

Lengths of the shortest paths with intermediate vertices numbered not higher than 4, i.e. *a*, *b*, *c*, and *d* (note a new shortest path from *c* to *a*).

FIGURE 8.7 Application of Floyd's algorithm to the graph shown. Updated elements are shown in bold.

Optimal Binary Search Trees

A Binary Search Tree 'T' is a Binary tree in which each node in the tree contains an identifier.

- All the identifiers in the left sub tree are less than root node 'T'.
- All the identifiers in the right sub tree are greater than the identifier in the root node 'T'.
- To determine an identifier 'x' is represented in Binary search tree first 'x' is compared with root node. If 'x' is less than identifier of the root then the search continues in the left sub tree, Otherwise in the right sub tree.
- If x = identifier of the root node then the search terminates completely.

If probabilities of searching for elements of a set are known an Optimal Binary Search tree for which the average number of comparisons in a search is the smallest possible i.e., to minimizing the average number of comparisons in a successful search.

For example, consider four keys A, B, C, and D to be searched for with probabilities 0.1, 0.2, 0.4, and 0.3, respectively.

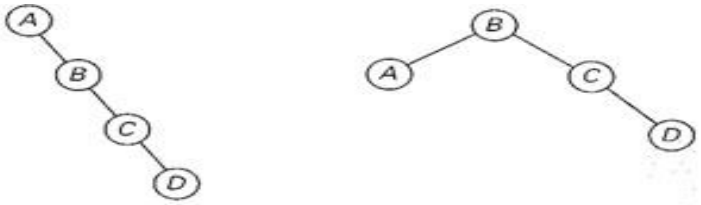
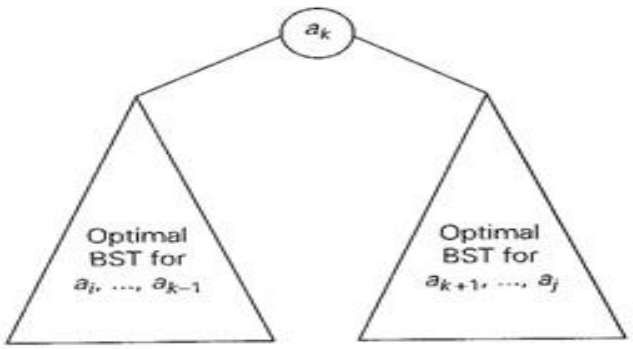


FIGURE 8.8 Two out of 14 possible binary search trees with keys *A*, *B*, *C*, and *D*

These are two out of 14 possible binary search trees containing these keys. The average number of comparisons in a successful search in the first of these trees is $0.1 \cdot 1 + 0.2 \cdot 2 + 0.4\text{-}3 + 0.3\text{-}4 = 2.9$, while for the second one it is $0.1 \cdot 2 + 0.2 \cdot 1 + 0.4\text{-}2 + 0.3\text{-}3 = 2.1$. Neither of these two trees is, in fact, optimal.

Let $a_1, \ldots . a_n$ be distinct keys ordered from the smallest to the largest and let $p_1, \ldots, p_n$ be the probabilities of searching for them. Let C[i, j] be the smallest average number of comparisons made in a successful search in a binary search tree T_i^j made up of keys $a_i, \ldots, a_j$, where i, j are some integer indices, $1 \leq i \leq j \leq n$.

To derive a recurrence underlying the dynamic programming algorithm, we need to consider all possible ways to choose a root a_k among the keys a,, ... , aj. For such a binary search tree, the root contains key $a_{k,}$, the left subtreer T_i^{k-1} contains keys $a_i, \ldots, a_{k-1}$ optimally arranged, and the right subtree T^j_{k+1} contains keys $a_{k+1}, \ldots, a_j$ also optimally arranged.



Binary search tree (BST) with root a_k and two optimal binary search subtrees T_i^{k-1} and T_{k+1}^j

If we count tree levels starting with 1, the following recurrence relation is obtained:

$$C[i, j] = \min_{i \le k \le j} \{C[i, k-1] + C[k+1, j]\} + \sum_{s=i}^{j} p_s \text{ for } 1 \le i \le j \le n.$$

We assume in formula that C[i, i- 1] = 0 for $1 \le i \le n + 1$, which can be interpreted as the number of comparisons in the empty tree. Note that this formula implies that

$$C[i, i] = p_i \quad \text{for } 1 \le i \le n,$$

as it should be for a one-node binary search tree containing a_i.

Pseudocode of the dynamic programming algorithm

```
ALGORITHM OptimalBST(P[l..n])
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[l..n] of search probabilities for a sorted list of n keys
//Output: Average number of comparisons in successful searches in the
//            optimal BST and table R of subtrees' roots in the optimal BST

    for i ← 1 to n do
          C[i, i -1] ← 0
          C[i, i] ← P[i]
          R[i, i] ← i
    C[n + 1, n] ← 0
    for d ← 1 to n - 1 do       //diagonal count
         for i ← 1 to n - d do
              j ← i + d
             minval ← ∞
             for k ← i to j do
                  if C[i, k -1] + C[k + 1, j] < minval
                        minval ← C[i, k- 1] + C[k + 1, j];  kmin ← k
             R[i, j] ← kmin
             sum ← P[i];
             for s ← i + 1 to j do
                 sum ← sum+ P[s]
                 C[i, j] ← minval + sum
return C[1, n ], R
```

EXAMPLE: Let us illustrate the algorithm by applying it to the four-key set we used at the beginning of this section:

key	A	B	C	D
probability	0.1	0.2	0.4	0.3

The initial tables look like this:

main table

	0	1	2	3	4
1	0	0.1			
2		0	0.2		
3			0	0.4	
4				0	0.3
5					0

root table

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

Let us compute $C[1, 2]$:

$$C[1, 2] = \min \begin{matrix} k = 1: C[1, 0] + C[2, 2] + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k = 2: C[1, 1] + C[3, 2] + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{matrix} = 0.4.$$

Thus, out of two possible binary trees containing the first two keys, A and B, the root of the optimal tree has index 2 (i.e., it contains B), and the average number of comparisons in a successful search in this tree is 0.4.

The process continues, then the final tables are:

main table

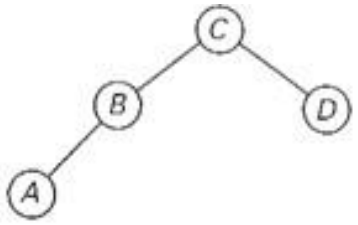
	0	1	2	3	4
1	0	0.1	0.4	1.1	1.7
2		0	0.2	0.8	1.4
3			0	0.4	1.0
4				0	0.3
5					0

root table

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

Thus, the average number of key comparisons in the optimal tree is equal to 1.7. Since R[1, 4] = 3, the root of the optimal tree contains the third key, i.e., C. Its left subtree is made up of keys A and B, and its right subtree contains just key D.

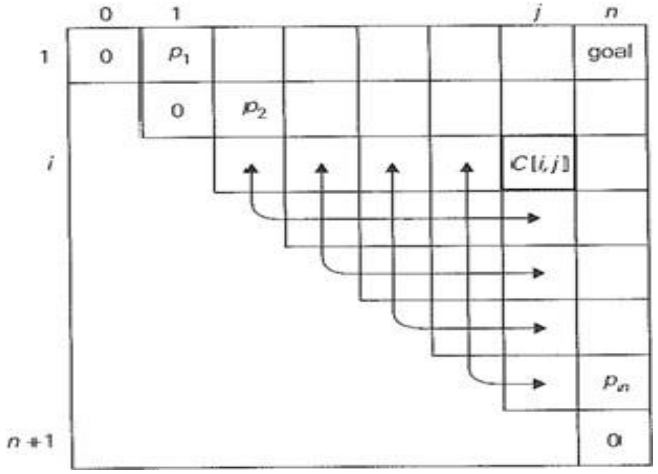
To find the specific structure of these subtrees, we find first their roots by consulting the root table again as follows. Since R[1, 2] = 2, the root of the optimal tree containing A and B is B, with A being its left child (R[1, 1] = 1). Since R[4, 4] = 4, the root of this one-node optimal tree is its only key D.



Optimal binary search tree for the example

The time complexity of this algorithm is cubic $O(n^3)$. The algorithm's space efficiency is quadratic $O(n^2)$.

Table of the dynamic programing algorithm for constructing an optimal binary search tree



Knapsack Problem and Memory Functions

Knapsack Problem

Knapsack problem is another designing technique of the dynamic programming. Here with the given 'n' items of known weights w_1, ... , w_n and values v_1, ... , v_n and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack. The weights and the knapsack's capacity are positive integers.

Let us consider an instance defined by the first 'i' items, $1 \le i \le n$, with weights w_1, ... , w_i, values v_1, ... , v_i, and knapsack capacity 'j', $1 \le j \le W$. Let V[i, j] be the value of an optimal solution to this instance. We can divide all the subsets of the first 'i' items that fit the knapsack of capacity 'j' into two categories:

1. The subsets without including the i^{th} item, then the value of an optimal subset is, V[i - 1, j].
2. The subsets with including the i^{th} item, then the optimal subset is made up of the first (i – 1) items that fit into the knapsack of capacity j- w_i , Then the value of such an optimal subset is v_i + V[i - 1, j - w_i].

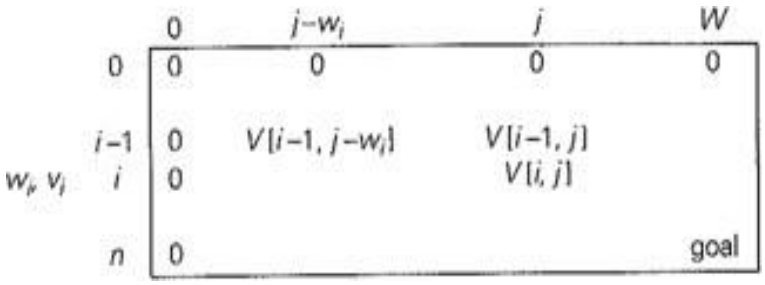
Then the following recurrence relation is used to find out the values:

$$V[i, j] = \begin{cases} \max\{V[i-1, j],\ v_i + V[i-1, j-w_i]\} & \text{if } j - w_i \ge 0 \\ V[i-1, j] & \text{if } j - w_i < 0. \end{cases}$$

For the convenience let us defined the initial conditions as follows:

$$V[0, j] = 0 \text{ for } j \ge 0 \text{ and } V[i, 0] = 0 \text{ for } i \ge 0.$$

Table for solve the Knapsack problem by dynamic programing



Our goal is to find V[n, w], the maximal value of a subset of then given items that fit into the knapsack of capacity W, and an optimal subset itself.

Example:

Let us consider the instance given by the following data:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$

Apply the recurrence relation of the Knapsack inorder to fill the items in to the Knapsack .

$$V[i, j] = \begin{cases} \max\{V[i-1, j],\ v_i + V[i-1, j-w_i]\} & \text{if } j - w_i \geq 0 \\ V[i-1, j] & \text{if } j - w_i < 0. \end{cases}$$

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	10	12	22	22	22
$w_3 = 3, v_3 = 20$	3	0	10	12	22	30	32
$w_4 = 2, v_4 = 15$	4	0	10	15	25	30	37

The maximum value is V[4,5] = $37. Here we are going to use the back tracking computations. Since V[4,5] ≠ V[3,5] it means that by including the item 4, we are getting the optimal solution along with an optimal subset for filling the remaining knapsack which is the capacity of 5 – 2 = 3.

Since V[3,3] ≠ V[2,3], the item 3 is not a part of an optimal subset.

Since V[2,3] ≠ V[1,3], item 2 is a part of optimal solution.

Similarly V[1,2] ≠ V[0,2] item 1 is the final part of optimal solution {item1, item2, item4}.

The time efficiency and space efficiency of this algorithm are both in Θ (n W). The time needed to find the composition of an optimal solution is in O(n + W).

Memory Functions

The dynamic programing deals with problems whose solutions satisfy a recurrence relation with overlapping subproblems.

The Memory Functions technique seeks to combine strengths of the top-down and bottom-up approaches to solving problems with overlapping sub problems. It does this by solving, in the top-down fashion but only once, just necessary subproblems of a given problem and recording their solutions in a table.

Initially, all the table's entries are initialized with a special "null" symbol to indicate that they have not yet been calculated. Thereafter, whenever a new value needs to be calculated, the method checks the corresponding entry in the table first, if this entry is not "null," it is simply retrieved from the table; otherwise, it is computed by the recursive call whose result is then recorded in the table.

The following algorithm implements this idea for the knapsack problem. After initializing the table, the recursive function needs to be called with i = n (the number of items) and j = W (the capacity of the knapsack).

```
ALGORITHM MFKnapsack(i, j)
//Implements the memory function method for the knapsack problem
//Input: A nonnegative integer i indicating the number of the first
//         items being considered and a nonnegative integer j indicating
//         the knapsack's capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights[l .. n ], Values[l .. n ],
//     and table V[0 .. n, 0 .. W] whose entries are initialized with -l's except for
//     row 0 and column 0 initialized with 0's

 If V[i,j] < 0
    if j < Weights[i]
        value← MFKnapsack(i- 1, j)
    else
        value ← max(MFKnapsack(i- 1, j),
                          Values[i] + MFKnapsack(i - 1, j - Weights[i]))
    V[i, j] ← value
  return V[i, j]
```

Example: Let us consider the instance given by the following data:

item	weight	value
1	2	$12
2	1	$10
3	3	$20
4	2	$15

capacity $W = 5$

$$V[i, j] = \begin{cases} \max\{V[i-1, j],\ v_i + V[i-1, j-w_i]\} & \text{if } j - w_i \geq 0 \\ V[i-1, j] & \text{if } j - w_i < 0. \end{cases}$$

Initial Table:

	0	**1**	**2**	**3**	**4**	**5**
0	0	0	0	0	0	0
1	0	-1	-1	-1	-1	-1
2	0	-1	-1	-1	-1	-1
3	0	-1	-1	-1	-1	-1
4	0	-1	-1	-1	-1	-1

Memory function method may be less space-efficient than a space efficient version of a bottom-up algorithm.

Consider V[4,5]

i = 4, j = 5, w4 = 2, v4 = 15

j – w4 = 5 – 2 = 3 > 0

Therefore, V[4,5] = max{V[3,5], 15 + V[3,3]}

Now we find the values of V[3,5] and V[3,3]

Similarly this process continue.

Example of solving an instance of the knapsack problem by the memory function algorithm

		capacity j					
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w_1 = 2, v_1 = 12$	1	0	0	12	12	12	12
$w_2 = 1, v_2 = 10$	2	0	–	12	22	–	22
$w_3 = 3, v_3 = 20$	3	0	–	–	22	–	32
$w_4 = 2, v_4 = 15$	4	0	–	–	–	–	37

Greedy Technique

The approach applied in the opening paragraph to the change-making problem is called '*greedy'*. Computer scientists consider it a general design technique despite the fact that it is applicable to optimization problems only.

The greedy approach suggests constructing a solution through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached. On each step-and this is the central point of this technique, the choice made must be

1. Feasible: it has to satisfy the problem's constraints.
2. Locally optimal: it has to be the best local choice among all feasible choices available on that step.
3. Irrevocable: once made, it cannot be changed on subsequent steps of the algorithm.

As a rule, greedy algorithms are both intuitively appealing and simple. In an algorithm strategy like Greedy, the decision is taken based on the information available. The Greedy method is the most straight forward method. It is popular for obtaining the optimized solutions.

In this we discuss two classic algorithms for minimum spanning tree problem: Prim's algorithm and Kruskal's algorithm. And another classic algorithm, Dijkstra's algorithm for the Shortest-path problem in a weighted graph.

Prim's Algorithm

Prim's Algorithm is applied for minimum spanning tree.

DEFINITION: A **spanning tree** of a connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. A **minimum spanning tree** of a weighted connected graph is its spanning tree of the smallest weight, where the weight of a tree is defined as the sum of the weights on all its edges.

The minimum spanning tree problem is the problem of finding a minimum spanning tree for a given weighted connected graph.

Example:

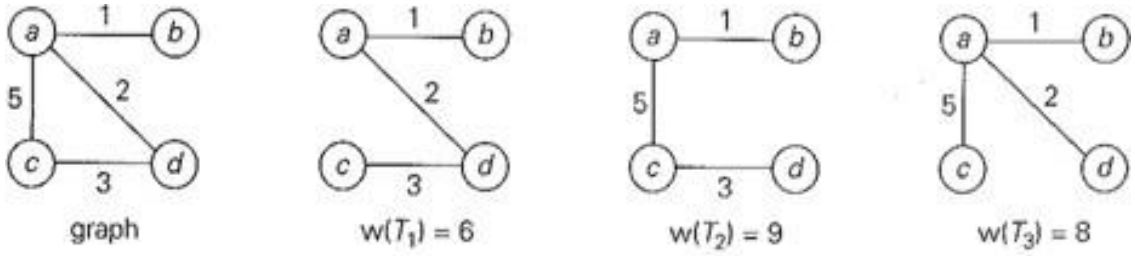


FIGURE 9.1 Graph and its spanning trees; T_1 is the minimum spanning tree

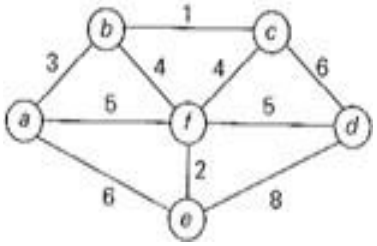
Prim's algorithm constructs a minimum spanning tree through a sequence of expanding subtrees. The initial subtree in such a sequence consists of a single vertex selected arbitrarily from the set V of the graph's vertices. On each iteration, we expand the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. By the nearest vertex, we mean a vertex not in the tree connected to a vertex in the tree by an edge of the smallest weight. Ties can be broken arbitrarily. The algorithm stops after all the graph's vertices have been included in the tree being constructed.

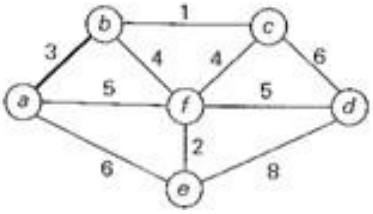
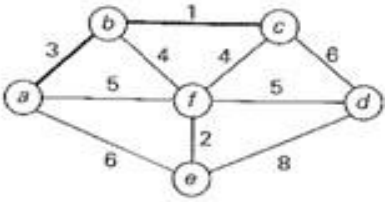
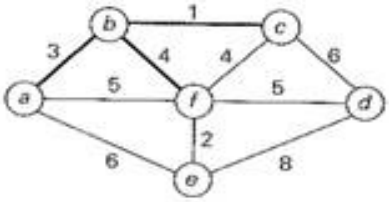
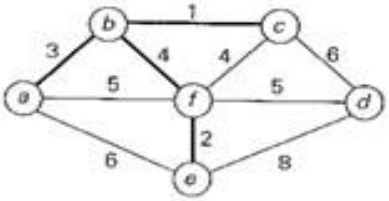
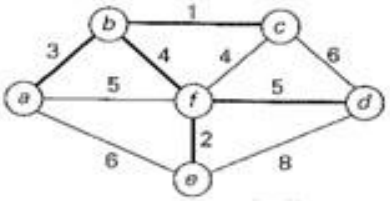
In this the total number of iterations is n-1, where 'n' is the number of vertices in the graph.

Pseudocode of this algorithm.

ALGORITHM Prim(G)
//Prim's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = (V, E)
//Output: E_T, the set of edges composing a minimum spanning tree of G
$V_T \leftarrow \{v_0\}$ //the set of tree vertices can be initialized with any vertex
$E_T \leftarrow \phi$
for $\leftarrow$ 1 to |V| - 1 do
find a minimum-weight edge e* = (v*, u') among all the edges (v, u)
such that v is in V_T and u is in $V - V_T$
$V_T \leftarrow V_T \cup \{u^*\}$
$E_T \leftarrow E_T \cup \{e^*\}$
return E_T

Example: Application of Prim's algorithm. The parenthesized labels of a vertex in the middle column indicate the nearest tree vertex and edge weight; selected vertices and edges are shown in bold.



Tree vertices	Remaining vertices	Illustration
a(–, –)	**b(a, 3)** c(–, ∞) d(–, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(–, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

We can split the vertices that are not in the tree into two sets, the "fringe" and the "unseen." The fringe contains only the vertices that are not in the tree but are adjacent to at least one tree vertex. These are the candidates from which the next tree vertex is selected. The unseen vertices are all the other vertices of the graph, called "unseen" because they are yet to be affected by the algorithm.

After we have identified a vertex u* to be added to the tree, we need to perform two operations:

1. Move u* from the set $V - V_T$ to the set of tree vertices V_T.
2. For each remaining vertex u in $V - V_T$ that is connected to u* by a shorter edge than the u's current distance label, update its labels by u* and the weight of the edge between u* and u, respectively.

The algorithm spends most of its in finding the smallest edge. So, time of the algorithm basically depends on how do we search this edge. Therefore Prim's algorithm runs in $O(n^2)$ time.

Kruskal's Algorithm

There is another greedy algorithm for the minimum spanning tree problem that also always yields an optimal solution. It is named Kruskal's algorithm.

Kruskal's algorithm looks at a minimum spanning tree for a weighted connected graph G = (V, E*)* as an acyclic subgraph with |V| - 1 edges for which the sum of the edge weights is the smallest.

The algorithm begins by sorting the graph's edges in nondecreasing order of their weights. Then, starting with the empty subgraph, it scans this sorted list adding the next edge on the list to the current sub graph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

Pseudocode of this algorithm.

ALGORITHM Kruskal(G)

//Kruskal's algorithm for constructing a minimum spanning tree
//Input: A weighted connected graph G = (V, E)
//Output: : E_T, the set of edges composing a minimum spanning tree of G

Sort E in nondecreasing order of the edge weights $w(e_1) \leq \ldots \leq w(e_{i\,|E|})$

$E_T \leftarrow \phi$; ecounter $\leftarrow$ 0 //initialize the set of tree edges and its size

k $\leftarrow$ 0 //initialize the number of processed edges

while ecounter < |V| - 1 do

    k $\leftarrow$ k + l

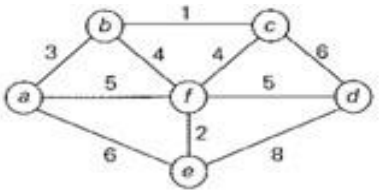
    if E_T U $\{e_{i\,k}\}$ is acyclic

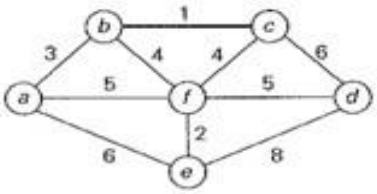
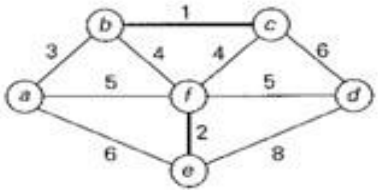
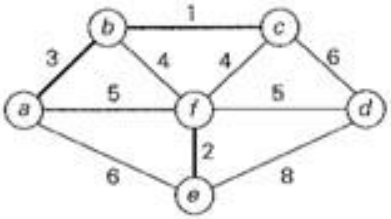
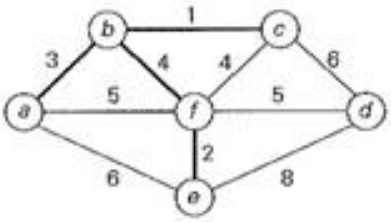
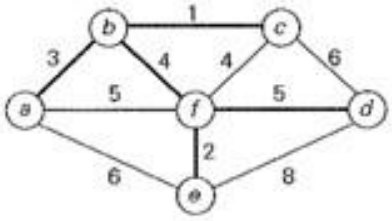
        $E_T \leftarrow E_T$ U $\{e_{i\,k}\}$; ecounter $\leftarrow$ ecounter + 1

return E_T

The running time of Kruskal's algorithm will be dominated by the time needed for sorting the edge weights of a given graph. Hence, with an efficient sorting algorithm, the time efficiency of Kruskal's algorithm is O(|E| log |E|).

Example: Application of Kruskal's algorithm. Selected edges are shown in bold.



Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de **1** 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 **2** 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 **3** 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 **4** 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df** ae cd de 1 2 3 4 4 5 **5** 6 6 8	
df 5		

Dijkstra's Algorithm

We consider the single-source shortest-paths problem: for a given vertex called the source in a weighted connected graph, find shortest paths to all its other vertices. This problem is called Dijkstra's algorithm. This algorithm is applicable to graphs with nonnegative weights only.

Dijkstra's algorithm finds the shortest paths to a graph's vertices in order of their distance from a given source. First, it finds the shortest path from the source to a vertex nearest to it, then to a second nearest, and so on.

In general, before its ith iteration commences, the algorithm has already identified the shortest paths to *i* - 1 other vertices nearest to the source. These vertices, the source, and the edges of the shortest paths leading to them from the source form a subtree T_i, of the given graph. The next vertex nearest to the source can be found among the vertices adjacent to the vertices of T_i. The set of vertices adjacent to the vertices in T_i are called "fringe vertices".

To identify the ith nearest vertex, the algorithm computes, for every fringe vertex *u,* the sum of the distance to the nearest tree vertex *v* and the length d_v of the shortest path from the source to v, and then selects the vertex with the smallest such sum. The fact that it suffices to compare the lengths of such special paths is the central insight of Dijkstra's algorithm.

After we have identified a vertex *u** to be added to the tree, we need to perform two operations:

1. Move u* from the fringe to the set of tree vertices.
2. For each remaining fringe vertex u that is connected to u* by an edge of weight w(u*, u) such that d_{u^*}.+ w(u*, u) < d_u , update the labels of u by u* and d_{u^*} + w(u*, u), respectively.

Pseudocode for Dijkstra's Algorithm:

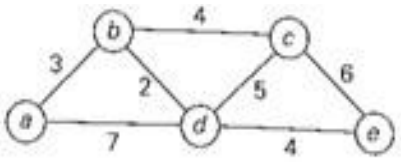
ALGORITHM Dijkstra(G, s)

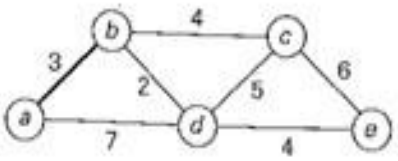
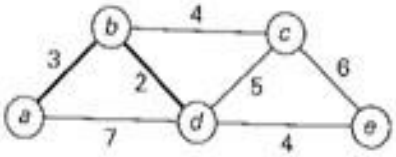
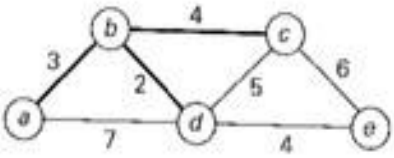
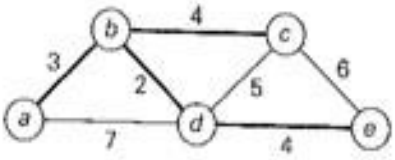
//Dijkstra's algorithm for single-source shortest paths
//Input: A weighted connected graph G = (V, E) with nonnegative weights
// and its vertex s
//Output: The length d_v of a shortest path from s to v
// and its penultimate vertex p_v for every vertex v in V

```
lnitialize(Q)          //initialize vertex priority queue to empty
for every vertex v in V do
    dv ← ∞;   pv ← null
    Insert(Q, v, dv)          //initialize vertex priority in the priority queue
ds ← 0;   Decrease(Q, s, ds)    //update priority of s with ds
VT ← 0
for  i ← 0 to |V| - 1do
    u* ← DeleteMin(Q)    //delete the minimum priority element
   VT ← VT U {u*}
   for every vertex  u in V - VT that is adjacent to u* do
       if du* + w(u*, u) < du
          du ← du*  + w(u*, u);  pu ← u*
          Decrease(Q, u, du)
```

The time efficiency of Dijkstra's algorithm depends on the data structures used for implementing the priority queue and for representing an input graph itself. It is O(|E| log |V|).

Example: Application of Dijkstra's algorithm. The next closest vertex is shown in bold.



Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

The shortest paths and their lengths are:

from a to b :	$a - b$	of length 3
from a to d :	$a - b - d$	of length 5
from a to c :	$a - b - c$	of length 7
from a to e :	$a - b - d - e$	of length 9

****

UNIT-IV

Limitations of Algorithm Power

A fair assessment of algorithms as problem-solving tools is inescapable: they are very powerful instruments, especially when they are executed by modern computers. But the power of algorithms is not unlimited, and its limits are discussed.

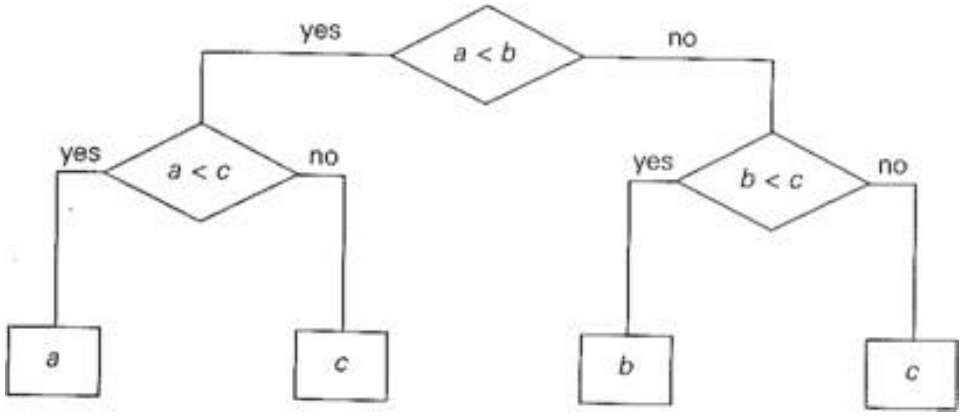
Some problems cannot be solved by any algorithm. Some problems can be solved algorithmically but not in polynomial time. And some problems can be solved in polynomial time, there are usually lower bounds on the efficiency of the algorithms.

Given a class of algorithms for solving a particular problem, a lower bound indicates the best possible efficiency any algorithm from this class can have.

Decision Trees

Many important algorithms, especially those for sorting and searching, we can comparing items of their inputs. Such algorithms with a device called the **'decision tree'.**

Example: Decision tree for finding a minimum of three numbers.



Each leaf node represents a possible outcome of the algorithm's run on some input of size 'n'. An important point is that the number of leaves must be at least as large as the number of possible outcomes. Hence, the number of comparisons in the worst case is equal to the height of the algorithm's decision tree.

The central idea behind this model is that a tree with a given number of leaves, which is dictated by the number of possible outcomes. So it is not difficult to prove that for any binary tree with '*l*' leaves and height 'h',

$$h \geq \log_2 l$$

Hence, the largest number of leaves in such a tree is 2^h.

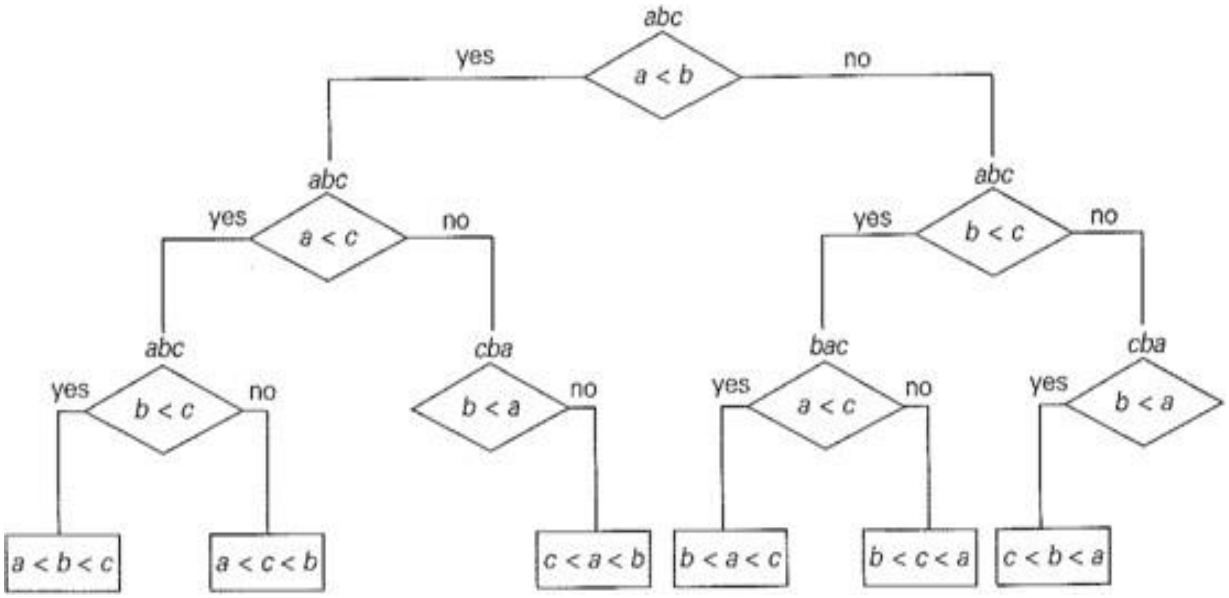
This formula puts a lower bound on the heights of binary decision trees. Such a bound is called the information- theoretic lower bound. This technique illustrate two important problems: sorting and searching in a sorted array.

1. Decision Trees for Sorting Algorithms

Most sorting algorithms are comparison based, i.e., they work by comparing elements in a list to be sorted. By studying properties of decision trees for comparison-based sorting algorithms, we can derive important lower bounds on time efficiencies of such algorithms.

We can interpret an outcome of a sorting algorithm as finding a permutation of the element indices of an input list that puts the list's elements in ascending order. For example, for the outcome a < c < b obtained by sorting a list a, b, c, the permutation in question is 1, 3, 2. Hence, the number of possible outcomes for sorting an arbitrary n-element list is equal to n!.

Figure: Decision tree for three element Selection sort



A triple above a node indicates the state of the array being sorted. Note the two redundant comparisons b <a with a single possible outcome because of the results of some previously made comparisons.

Hence the worst -case number of comparisons made by such an algorithm cannot be less than [$\log_2$ n!l:

$$C_{worst}(n) \geq \lceil \log_2 n! \rceil. \qquad \textbf{(11.2)}$$

Using Stirling's formula for $n!$, we get

$$\lceil \log_2 n! \rceil \approx \log_2 \sqrt{2\pi n}(n/e)^n = n \log_2 n - n \log_2 e + \frac{\log_2 n}{2} + \frac{\log_2 2\pi}{2} \approx n \log_2 n.$$

So 'n $\log_2$ n' comparisons are necessary to sort an arbitrary n-element list by any comparison – based sorting algorithms.

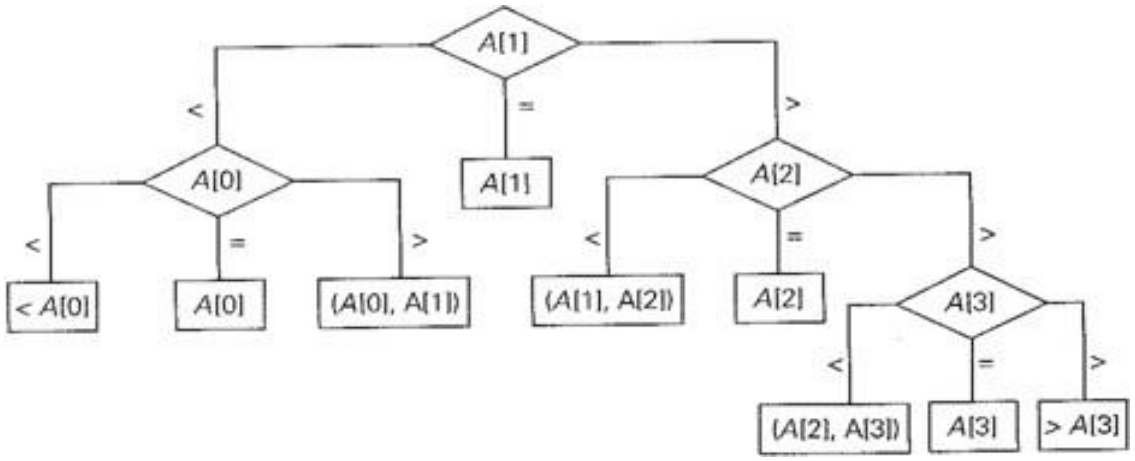
2. Decision Trees for Searching a Sorted Array

In this , decision trees can be used for establishing lower bounds on the number of key comparisons in searching a sorted array of **n** keys: A[0] < A[l] < ... < A[n -1]. The principal algorithm for this problem is binary search in the worst case, C_{worst} bs(n), is given by the formula:

$$C_{worst}^{bs}(n) = \lfloor \log_2 n \rfloor + 1 = \lceil \log_2 (n+1) \rceil.$$

We are dealing with three-way comparisons in which search key K is compared with some element A[i] to see whether K < A[i], K = A[i], or K > A[i], it is natural to try using ternary decision trees.

Figure: Ternary decision tree for binary search in a 4-element array.



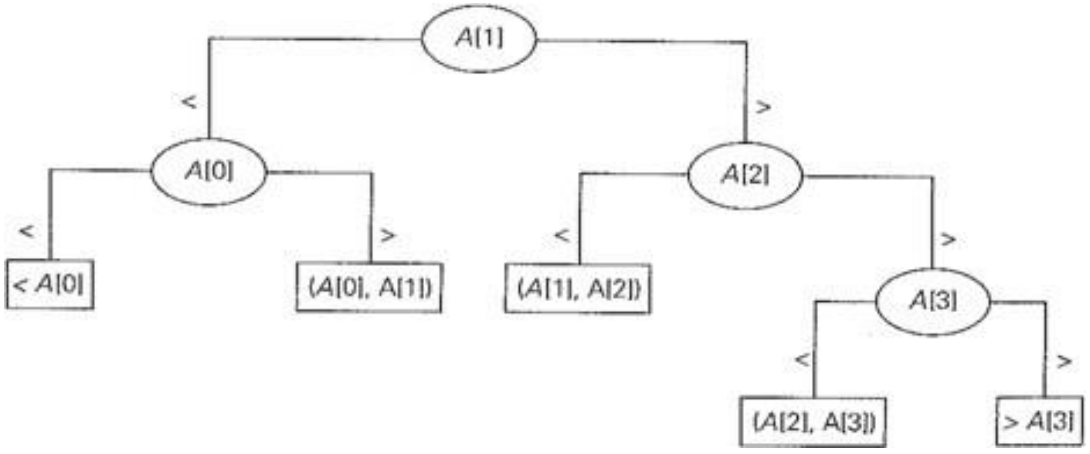
The internal nodes of that tree indicate the array's elements being compared with the search key. The leaves indicate either a matching element in the case of a successful search or a found interval that the search key belongs to in the case of an unsuccessful search.

For an array of ***n*** elements, all such decision trees will have 2n +1 leaves. Since the minimum height ***h*** of a ternary tree with ***l*** leaves is $[\log_3 l]$, we get the following lower bound on the number of worst-case comparisons:

$$C_{worst}(n) \geq \lceil \log_3(2n+1) \rceil.$$

To obtain a better lower bound, we should consider binary decision trees.

Figure: Binary decision tree for binary search in 4-element array.



Internal nodes are same as ternary tree. Leaves represent only unsuccessful searches, and there are n+1 for searching an n-element array. Comparison of above 2 trees, in the binary tree is simply the ternary tree with all the middle subtrees eliminated. Applying inequality $[h \geq \log_2 l]$:

$$C_{worst}(n) \geq \lceil \log_2(n+1) \rceil.$$

P, NP and NP – Complete problems

Complexity theory seeks to classify problems according to their computational complexity. Problems that can be solved in polynomial time are called **tractable,** problems that cannot be solved in polynomial time are called **intractable.**

Most problems can be solved in polynomial time by some algorithm, they are computing the product and the greatest common divisor of two integers, sorting, searching, acyclicity of a graph, finding a minimum spanning tree, and finding the shortest paths in a weighted graph. The problems that can be solved in polynomial time as the set that computer science theoreticians call **P**. A more formal definition includes in **P** only **decision problems,** which are problems with yes/no answers.

DEFINITION: Class **P** is a class of decision problems that can be solved in polynomial time by (deterministic) algorithms. This class of problems is called **polynomial.**

The restriction of **P** to decision problems can be justified by the following Reasons:

1) it is sensible to exclude problems not solvable in polynomial time because of their exponentially large output. Eg: generating subsets of a given set.

2) many important problems that are not decision problems in their most natural formulation can be reduced to a series of decision problems that are easier to study. Eg: coloring problem.

Some decision problems cannot be solved at all by any algorithm. Such problems are called '**undecidable'**. The **halting problem** is an example of undecidable decision problem. Given a computer program and an input to it, determine whether the program will halt on that input or continue working indefinitely on it.

Assume that '**A'** is an algorithm that solves the halting problem. That is, for any program **'P'** and input **'I'**,

$$A(P, I) = \begin{cases} 1, & \text{if program } P \text{ halts on input } I; \\ 0, & \text{if program } P \text{ does not halt on input } I. \end{cases}$$

Best examples of this category:

1. **Hamiltonian circuit:** Determine whether a given graph has a Hamiltonian circuit (a path that starts and ends at the same vertex and passes through all the other vertices exactly once).
2. **Traveling salesman:** Find the shortest tour through n cities with known positive integer distances between them (find the shortest Hamiltonian circuit in a complete graph with positive integer weights).
3. **Knapsack problem:** Find the most valuable subset of n items of given positive integer weights and values that fit into a knapsack of a given positive integer capacity.
4. **Graph coloring:** For a given graph, find its chromatic number (the smallest number of colors that need to be assigned to the graph's vertices so that no two adjacent vertices are assigned the same color).

Nondeterministic algorithm: is a two-stage procedure that takes as its input an instance **I** of a decision problem and does the following.

Nondeterministic ("guessing") stage: An arbitrary string **S** is generated that can be thought of as a candidate solution to the given instance I.

Deterministic ("verification") stage: A deterministic algorithm takes both **I** and **S** as its input and outputs yes if **S** represents a solution to instance **I**.

Finally, a nondeterministic algorithm is said to be **nondeterministic polynomial** if the time efficiency of its verification stage is polynomial.

DEFINITION: Class '**NP**' is the class of decision problems that can be solved by nondeterministic polynomial algorithms. This class of problems is called **'nondeterministic polynomial'**.

Most decision problems are in NP. First of all, this class includes all the problems in **P**:

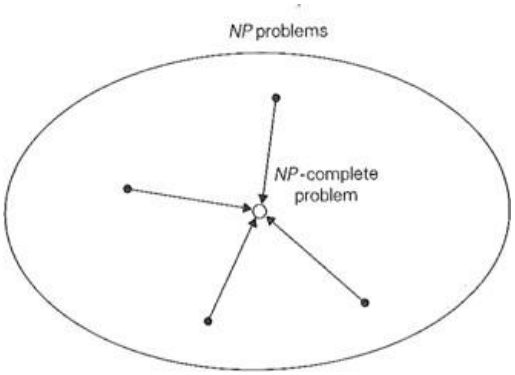
$$P \subseteq NP.$$

This is true because, if a problem is in **P**, we can use the deterministic polynomial time algorithm that solves it in the verification-stage of a nondeterministic algorithm that simply ignores string **S** generated in its nondeterministic ("guessing") stage. But **N P** also contains the Hamiltonian circuit problem, the partition problem, traveling salesman, the knapsack, graph coloring and many hundreds of other difficult combinatorial optimization problems.

NP-Complete Problems

DEFINITION: A decision problem **D** is said to be NP-Complete if
1. it belongs to class NP;
2. every problem in NP is polynomially reducible to **D**.

Notion of an NP-Complete problem polynomial time reductions of NP problems to an NP-Complete problem are shown by arrows.



The fact that closely related decision problems are polynomially reducible to each other is not very surprising. For example, let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

The definition of NP-Completeness immediately implies that if we find a deterministic polynomial time algorithm for just one NP-complete problem then every problem in NP can be solved in polynomial time by a deterministic algorithm, and hence **P = NP**.

The first proof of a problem's NP- completeness was published by S.Cook for the CNF satisfiability problem: three Boolean variables x_1, x_2, x_3 and their negations denoted $\bar{x}_1$, $\bar{x}_2$, $\bar{x}_3$ respectively.

$$(x_1 \vee \bar{x}_2 \vee \bar{x}_3)\&(\bar{x}_1 \vee x_2)\&(\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3).$$

The CNF-satisfiability problem asks whether or not we can assign values ***true*** and ***false*** to variables of a given boolean expression in its CNF form to make the entire expression ***true.***

(if ***x1 =true, x2 =true,*** **and** ***x3 =false,*** **the entire expression is** ***true.)***

Coping with the limitations of Algorithm Power

This approach makes it possible to solve some large instances of difficult combinatorial problems, in the worst case. Two algorithm techniques, **Back-tracking** and **Branch-and-bound** make it possible to solving at least some large instances of difficult combinatorial problems. Both strategies can be considered an improvement over exhaustive search. These are based on the construction of a **'State-space-tree'** whose nodes reflect specific choices made for a solution's components.

Backtracking

Backtracking is one of the k=most general technique. In this, we search for the set of solutions or optimal solution which satisfies some constraints. Backtracking is a variation of exhaustive search, where the search is refined by eliminating certain possibilities. Backtracking is usually faster method than an exhaustive search.

In this method we construct a tree, called state-space tree. Its root represents an initial state before the search for a solution begins. The nodes of the first level in the tree represents the choices made for the first component of a solution, the second level nodes represent the choices for the second component, and so on.

A node in a state-space tree is said to be '**promising'** if it corresponds to a partially constructed solution that may still lead to a complete solution; otherwise, it is called '**nonpromising'**. Leaves represent either nonpromising dead ends or complete solutions found by the algorithm.

A state-space tree for a backtracking algorithm is constructed in the manner of depth- first search.

If the current node is promising, its child is generated by adding the first remaining legitimate option for the next component of a solution.

If the current node is nonpromising, the algorithm backtracks to the node's parent to consider the next possible option for its last component; if there is no such option, it backtracks one more level up the tree, and so on.

Finally, if the algorithm reaches a complete solution to the problem, it either stops or continues searching for other possible solutions.

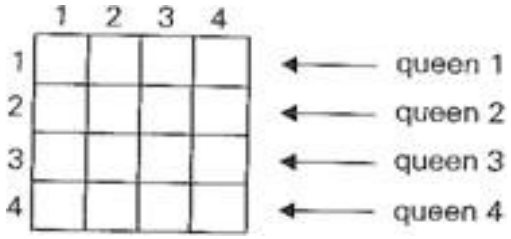
General Algorithm

AlGORITHM Backtrack(X[l..i])
//Gives a template of a generic backtracking algorithm
//Input: X[1..i] specifies first i promising components of a solution
//Output: All the tuples representing the problem's solutions
if X[1..i] is a solution **write** X[l..i]
else
for each element x ϵ S_{i+1} consistent with X[1..i] and the constraints **do**
X[i + 1] ← X
Backtrack(X[1..i + 1])

n-Queens Problem

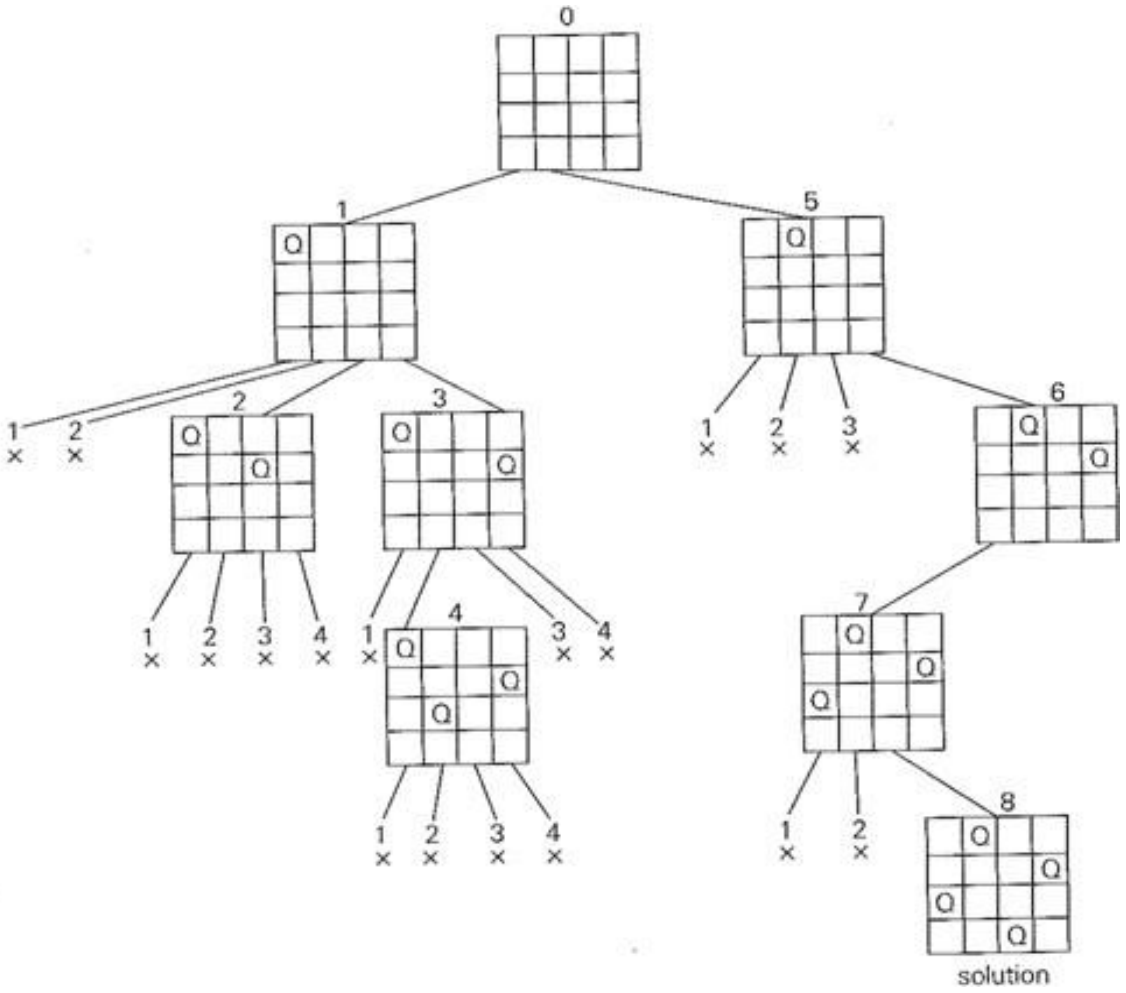
The problem is to place '**n**' queens on an **n-by-n** chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal. For $n = 1$, the problem has a trivial solution, and it is easy to see that there is no solution for $n = 2$ and $n = 3$.

So let us consider the four-queens problem and solve it by the backtracking technique. Since each of the four queens has to be placed in its own row, all we need to do is to assign a column for each queen on the board.



We start with the empty board and then place queen 1 in the first possible position of its row, which is in column 1 of row 1. Then we place queen 2, after trying unsuccessfully colunms 1 and 2, in the first acceptable position for it, which is square (2,3), the square in row 2 and column 3. This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2,4). Then queen 3 is placed at (3,2), which proves to be another dead end. The algorithm then backtracks all the way to queen 1 and moves it to (1,2). Queen 2 then goes to (2,4), queen 3 to (3,1), and queen 4 to (4,3), which is a solution to the problem.

State-Space tree of solving 4-Queen problem by Backtracking.



Here ‘x’ denotes an unsuccessful attempt to place a queen in the indicated column. The numbers above the nodes indicate the order in which the nodes are generated.

Hamiltonian Circuit Problem

A path that starts and ends at the same vertex and passes through all the other vertices exactly once.

Example:

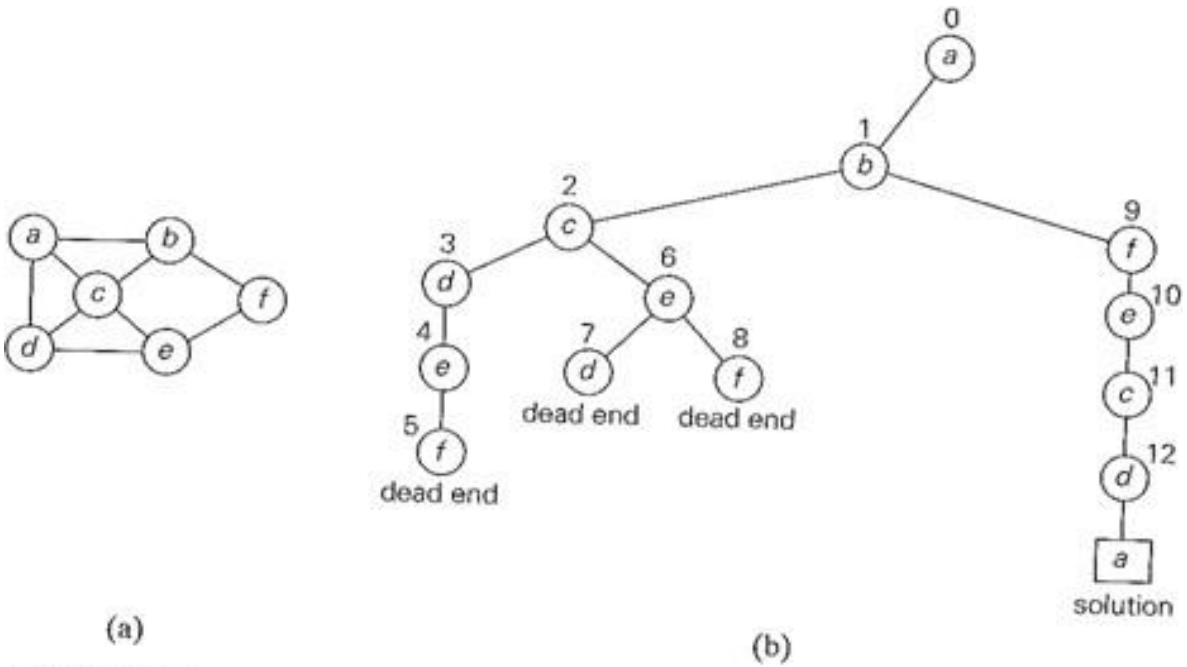


FIGURE 12.3 (a) Graph. (b) State-space tree for finding a Hamiltonian circuit. The numbers above the nodes of the tree indicate the order in which the nodes are generated.

Path: a → b → f → e → c → d → a

We can assume that if a Hamiltonian circuit exists, it starts at vertex 'a'. We make vertex 'a' the root of the state-space tree. Using the alphabet order to break the three-way tie among the vertices adjacent to 'a', we select vertex 'b'. From 'b', the algorithm proceeds to' 'c, then to 'd', then to 'e', and finally to 'f', which proves to be a dead end. So the algorithm backtracks from ' f' to 'e', then to 'd', and then to 'c', which provides the first alternative for the algorithm to pursue. Going from 'c' to 'e' eventually proves useless, and the algorithm has to backtrack from 'e' to 'c' and then to 'b'. From there, it goes to the vertices f, e, c, and d, from which it can legitimately return to 'a', yielding the Hamiltonian circuit **a, b, f, e, c, d, a.** If we wanted to find another Hamiltonian circuit, we could continue this process by backtracking from the leaf of the solution found.

Subset-Sum Problem

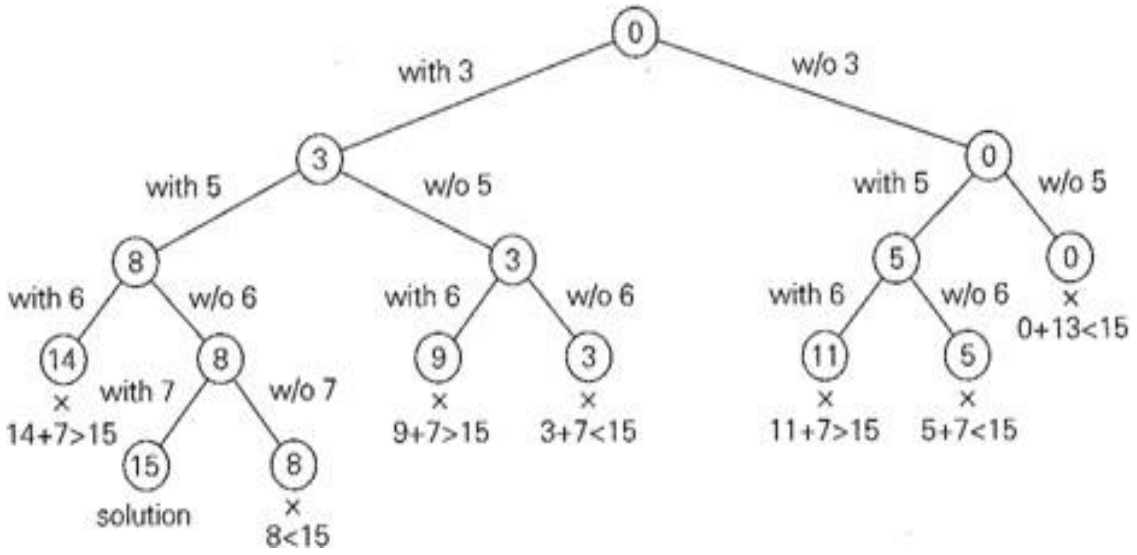
Fmd a subset of a given set S= {s1,s2, ... , sn) of n positive integers whose sum is equal to a given positive integer 'd'.

For example, for S = {1, 2, 5, 6, 8] and d = 9, there are two solutions: {1,2, 6} and {1, 8}.

It is convenient to sort the set's elements in increasing order. So we will assume that

$$s1 \leq s2 \leq \ldots\ldots \leq sn$$

The state-space tree can be constructed as a binary tree for the instance S= (3, 5, 6, 7} and d = 15.



Here the number inside a node is the sum of the elements already included in subsets represented by the node.

The root of the tree represents the starting point, with no decisions about the given elements. Its left and right children represent, respectively, inclusion and exclusion of s1 in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of s2, while going to the right corresponds to its exclusion, and so on.

We record the value of **s'**, the sum of these numbers, in the node. If **s'** is equal to **d**, we have a solution to the problem. If **s'** is not equal to **d**, we can terminate the node as non promising if either of the following two inequalities holds:

$$s' + s_{i+1} > d \text{ (the sum } s' \text{ is too large)}$$

$$s' + \sum_{j=i+1}^{n} s_j < d \text{ (the sum } s' \text{ is too small).}$$

Branch-and-Bound

Branch-and-Bound is a general algorithm for finding optimal solutions of various optimization problems. Branch-and-Bound is general optimization technique that applies where the greedy method and dynamic programming fail.

Note that in the standard terminology of optimization problems, a '**feasible solution'** is a point in the problem's search space that satisfies all the problem's constraints, while an '**optimal solution'** is a feasible solution with the best value of the objective function.

Compared to backtracking, branch-and-bound requires two additional items:

1. A way to provide, for every node of a state-space tree, a bound on the best value of the objective function on any solution that can be obtained by adding further components to the partial solution represented by the node.

2. The value of the best solution seen so far.

If this information is available, we can compare a node's bound value with the value of the best solution seen so far: if the bound value is not better than the best solution, because no solution obtained from it can yield a better solution than the one already available. This is the principal idea of the branch-and-bound technique.

In general, we terminate a search path at the current node in a state-space tree of a branch-and-bound algorithm for any one of the following three reasons:

1. The value of the node's bound is not better than the value of the best solution seen so far.
2. The node represents no feasible solutions because the constraints of the problem are already violated.
3. The subset of feasible solutions represented by the node consists of a single point.

The Knapsack Problem

In this problem, given **n** items of known weights w_i and values v_i, i = 1, 2, ...n , and a knapsack of capacity **W**, find the most valuable subset of the items that fit in the knapsack. It is convenient to order the items of a given instance in descending order by their value-to-weight ratios. Then the first item gives the best payoff per weight unit and the last one gives the worst payoff per weight unit, with ties resolved arbitrarily:

$$v_1/w_1 \geq v_2/w_2 \geq \ldots \geq v_n/w_n.$$

We record the total weight **w** and the total value **v** of this selection in the node, along with some upper bound **ub** on the value of any subset that can be obtained by adding zero or more items to this selection.

A simple way to compute the upper bound **ub** is to add to **v**, the total value of the items already selected, the product of the remaining capacity of the knapsack **W - w** and the best per unit payoff among the remaining items, which is v_{i+1}/ w_{i+1}:

$$ub = v + (W - w)(v_{i+1}/w_{i+1}).$$

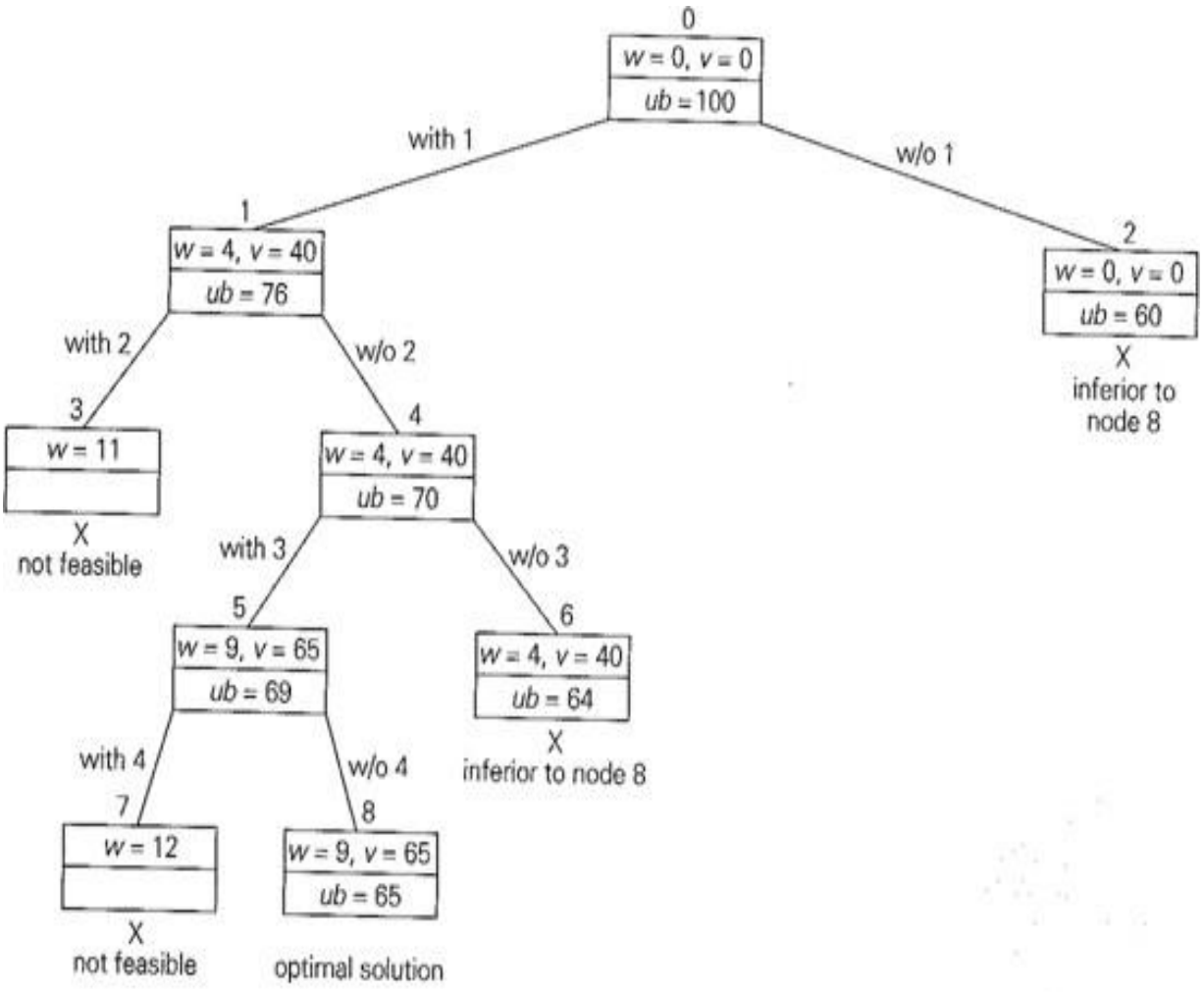
Example:

item	weight	value	value / weight
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The knapsack's capacity W is 10.

At the root of the state-space tree (in figure), no items have been selected. Hence, both the total weight of the items already selected **w** and their total value **v** are equal to '**0**'. The value of the upper bound computed by above formula is $100. Node 1, the left child of the root, represents the subsets that include item 1. The total weight and value of the items already included are 4 and $40, the value of the upper bound is 40 + (10 - 4) * 6 = $76. Node 2 represents the subsets that do not include item 1. Accordingly, **w** = 0, **v** = $0, and **ub** = 0 + (10- 0) * 6 = $60. Since node 1 has a larger upper bound than the upper bound of node 2, it is more promising for this maximization problem, and we branch from node 1 first. Its children, nodes 3 and 4 represent subsets with item 1 and with and without item 2. Since the total weight **w** of every subset represented by node 3 exceeds the knapsack's capacity, node 3 can be terminated immediately.

State-space tree of the branch-and-bound algorithm for the instance of the knapsack problem:



Traveling Salesman Problem

To apply the branch-and-bound technique to instances of the traveling salesman problem if we come up with a reasonable lower bound on tour lengths. One very simple lower bound can be obtained by finding the smallest element in the intercity distance matrix **D** and multiplying it by the number of cities **n**.

Example:

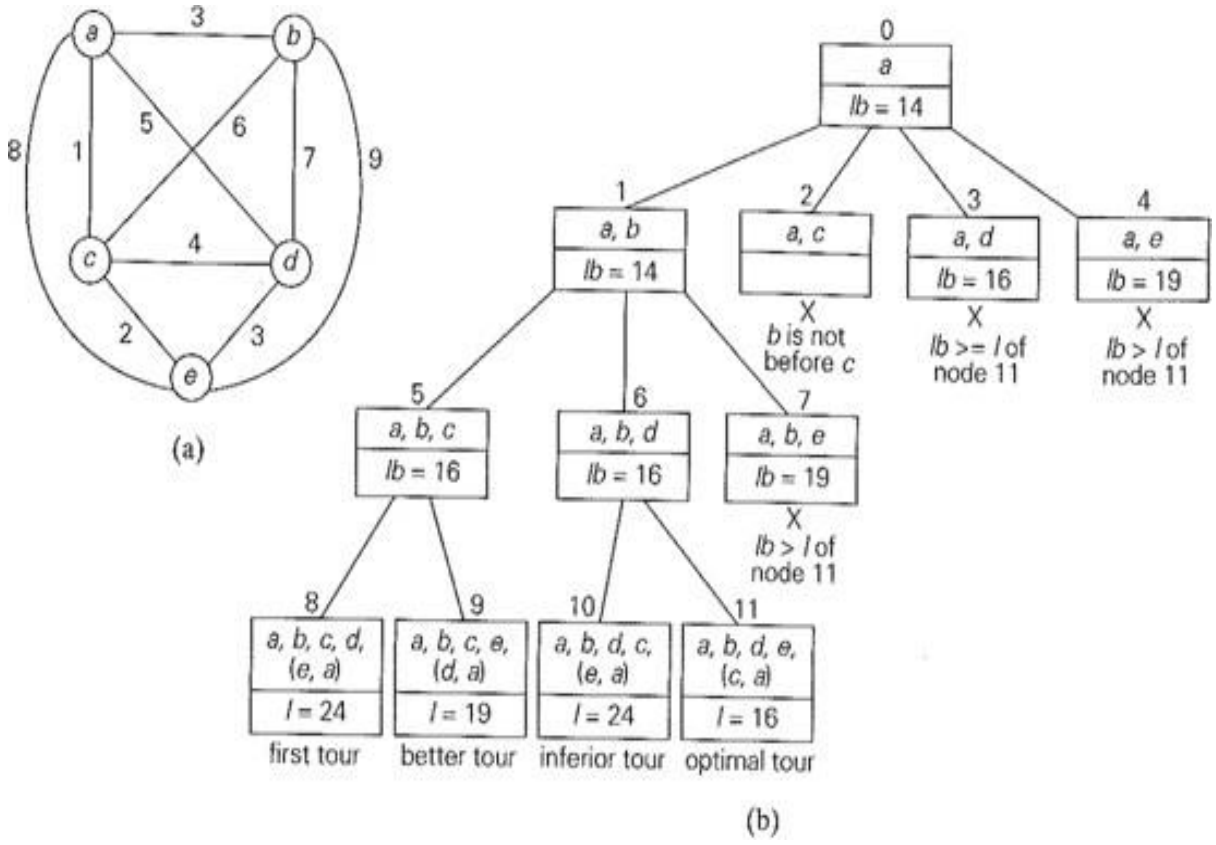


Figure: (a) Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find the shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

For each city **i**, $1 \le i \le n$, find the sum $\mathbf{S_i}$ of the distances from city **i** to the two nearest cities; compute the sum **S** of these **n** numbers; divide the result by 2; and, if all the distances are integers, round up the result to the nearest integer:

$$lb = [S/2]$$

For example, for the instance of the above graph, formula yields

$$lb = [(1+3) + (3+6) + (1+2) + (3+4) + (2+3)] / 2 = 14.$$

For example, for all the Hamiltonian circuits of the graph, that must include edge (a, d), we get the following lower bound by summing the lengths of the two shortest edges with each of the vertices, with the edges (a, d) and (d, a):

$$[(1+ 5) + (3 + 6) + (1+ 2) + (3 + 5) + (2 + 3)] / 2 = 16.$$

Now apply the branch-and-bound algorithm, with the bounding function given by formula, to find the shortest Hamiltonian circuit for the graph of the given graph.

First we start at 'a'. Second, because our graph is undirected, we can generate only tours in which **b** is visited before **c**. In addition, after visiting n - 1 = 4 cities, a tour has no choice but to visit the remaining unvisited city and return to the starting one.

Approximation Algorithms for NP-hard Problems

The optimization versions of such difficult combinatorial problems fall in the class of ***NP-hard*** *problems-problems* that are at least as hard as NP-complete problems. Hence, there are no known polynomial-time algorithms for these problems.

If we use an algorithm whose output is just an approximation of the actual optimal solution, we can quantify the accuracy of an approximate solution $\mathbf{s_a}$ to a problem minimizing some function **f** by the size of the relative error of this approximation

$$re(s_a) = \frac{f(s_a) - f(s^*)}{f(s^*)},$$

where s^* is an exact solution to the problem. Alternatively, since $re(s_a) = f(s_a)/f(s^*) - 1$, we can simply use the ***accuracy ratio***

$$r(s_a) = \frac{f(s_a)}{f(s^*)}$$

as a measure of accuracy of s_a. Note that for the sake of scale uniformity, the accuracy ratio of approximate solutions to maximization problems is often computed as

$$r(s_a) = \frac{f(s^*)}{f(s_a)}$$

to make this ratio greater than or equal to 1, as it is for minimization problems.

The closer $\mathbf{r(s_a)}$ is to 1, the better the approximate solution. The best upper bound of possible $\mathbf{r(s_a)}$ values taken over all instances of the problem is called the **'performance ratio'**.

Approximation Algorithms for the Traveling Salesman Problem

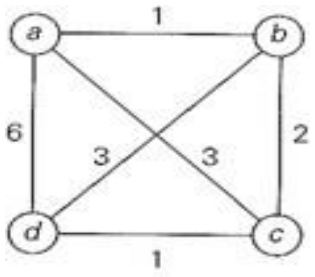
Nearest-neighbor algorithm: The following simple greedy algorithm is based on the *nearest-neighbor* heuristic: the idea of always going to the nearest unvisited city next.

Step 1: Choose an arbitrary city as the start.

Step 2: Repeat the following operation until all the cities have been visited: go to the unvisited city nearest the one visited last (ties can be broken arbitrarily).

Step 3: Return to the starting city.

Example:



For the instance represented by the graph in Figure 12.10, with a as the starting vertex, the nearest-neighbor algorithm yields the tour (Hamiltonian circuit) s_a: $a - b - c - d - a$ of length 10.

The optimal solution, as can be easily checked by exhaustive search, is the tour s^*: $a - b - d - c - a$ of length 8. Thus, the accuracy ratio of this approximation is

$$r(s_a) = \frac{f(s_a)}{f(s^*)} = \frac{10}{8} = 1.25$$

i.e, tour $\mathbf{s_a}$ is 25% longer then the optimal tour s*

Twice-around-the-tree algorithm: This algorithm exploits a connection between Hamiltonian Circuits and spanning trees of the same graph.

Step 1: Construct a minimum spanning tree of the graph corresponding to a given instance of the traveling salesman problem.

Step 2: Starting at an arbitrary vertex, perform a walk around the minimum spanning tree recording all the vertices passed by.

Step 3: Scan the vertex list obtained in Step 2 and eliminate from it all repeated occurrences of the same vertex except the starting one at the end of the list. The vertices remaining on the list will form a Hamiltonian circuit, which is the output of the algorithm.

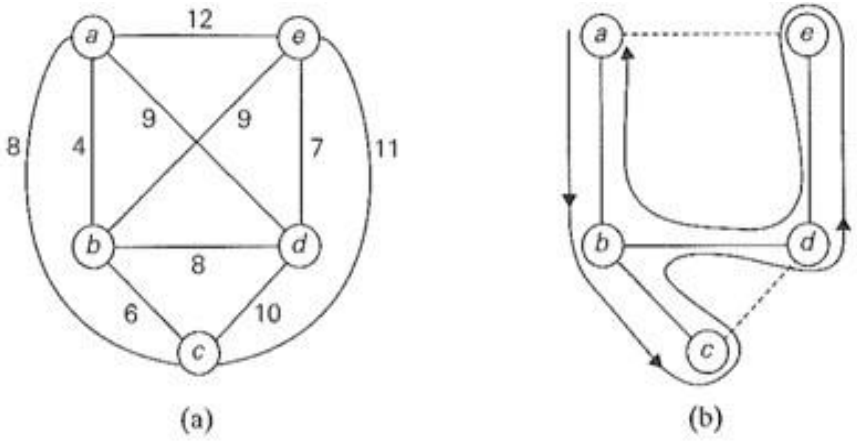


FIGURE 12.11 Illustration of the twice-around-the-tree algorithm. (a) Graph. (b) Walk around the minimum spanning tree with the shortcuts.

Let us apply this algorithm to the. The minimum spanning tree of this graph is made up of edges (a, b), (b, c), (b, d), and (d, e) (Fig. 12.1lb). A twice-around-the-tree walk that starts and ends at **a** is

a, b, c, b, d, e, d, **b, a.**

Eliminating the second **b** (a shortcut from c to d), the second **d**, and the third **b** (a shortcut from e to a) then yields the Hamiltonian circuit,

a, b, c, d, e, a

of length 41.

Approximation Algorithms for the Knapsack Problem

The knapsack problem is another well-known NP-hard problem. Given **n** items of known weights **w1, ... , wn** and values **v1, ... , vn**, and a knapsack of weight capacity **W**, find the most valuable sub-set of the items that fits into the knapsack.

Greedy algorithms for the knapsack problem: In this select the items in decreasing order of their weights; however, heavier items may not be the most valuable in the set. Alternatively, if we pick up the items in decreasing order of their value, there is no guarantee that the knapsack's capacity will efficiently.

Greedy algorithm for the discrete knapsack problem

Step 1: Compute the value-to-weight ratios $r_i = v_i / w_i$, $i = 1, \ldots, n$, for the items given.

Step 2: Sort the items in nonincreasing order of the ratios computed in Step1. (Ties can be broken arbitrarily.)

Step 3: Repeat the following operation until no item is left in the sorted list: if the current item on the list fits into the knapsack, place it in the knapsack; otherwise, proceed to the next item.

EXAMPLE: Let us consider the instance of the knapsack problem with the knapsack's capacity equal to 10 and the item information as follows:

item	weight	value
1	7	$42
2	3	$12
3	4	$40
4	5	$25

Computing the value-to-weight ratios and sorting the items in nonincreasing order of these efficiency ratios yields

item	weight	value	$\frac{\text{value}}{\text{weight}}$
1	4	$40	10
2	7	$42	6
3	5	$25	5
4	3	$12	4

The greedy algorithm will select the first item of weight 4, skip the next item of weight 7, select the next item of weight 5, and skip the last item of weight 3. The solution obtained happens to be optimal for this instance.

****