UNIT I INTRODUCTION

1.1 NOTION OF AN ALGORITHM

An ***algorithm*** is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

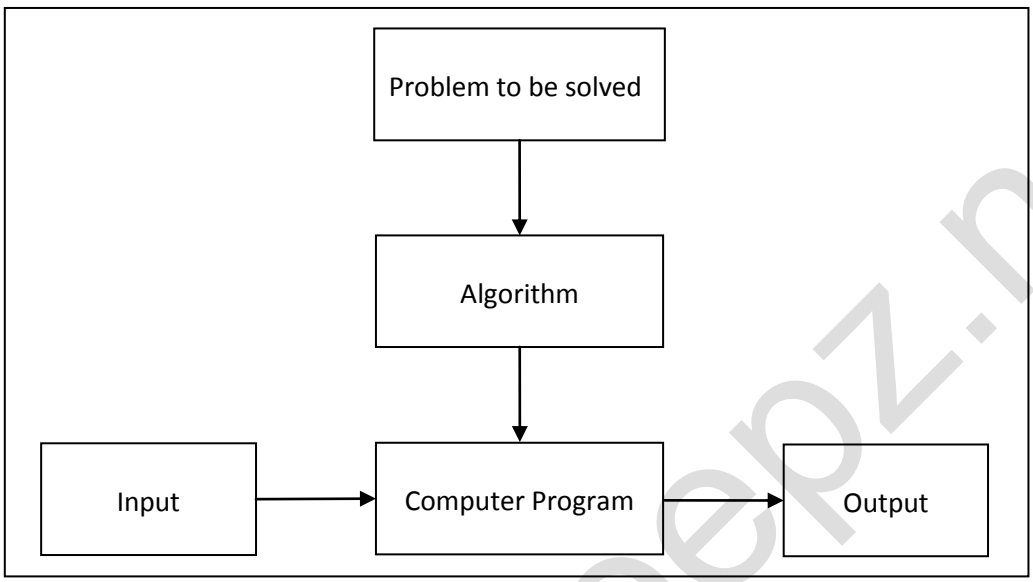


FIGURE 1.1 The notion of the algorithm.

It is a step by step procedure with the input to solve the problem in a finite amount of time to obtain the required output.

The notion of the algorithm illustrates some important points:

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

Characteristics of an algorithm:

Input: Zero / more quantities are externally supplied**.**

Output: At least one quantity is produced.

Definiteness: Each instruction is clear and unambiguous.

Finiteness: If the instructions of an algorithm is traced then for all cases the algorithm must terminates after a finite number of steps.

Efficiency: Every instruction must be very basic and runs in short time.

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 some statements like assignments are used.
4. The compound statements may be enclosed with **{** and **}** brackets. **if**, **for**, **while** can be closed 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 letters after 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.

Example:

The greatest common divisor(GCD) of two nonnegative integers *m* and *n* (not-both-zero), denoted gcd*(m, n)*, is defined as the largest integer that divides both *m* and *n* evenly, i.e., with a remainder of zero.

Euclid's algorithm is based on applying repeatedly the equality gcd*(m, n)* = gcd*(n, m* mod *n),* where *m* mod *n* is the remainder of the division of *m* by *n*, until *m* mod *n* is equal to 0. Since gcd*(m,* 0*)* = *m*, the last value of *m* is also the greatest common divisor of the initial *m* and *n*.
gcd*(*60*,* 24*)* can be computed as follows:gcd*(*60*,* 24*)* = gcd*(*24*,* 12*)* = gcd*(*12*,* 0*)* = 12.

Euclid's algorithm for computing gcd*(m, n)* in simple steps

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*. Go to Step 1.

Euclid's algorithm for computing gcd*(m, n)* expressed in pseudocode

ALGORITHM *Euclid_gcd(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*

1.2 FUNDAMENTALS OF ALGORITHMIC PROBLEM SOLVING

A sequence of steps involved in designing and analyzing an algorithm is shown in the figure below.

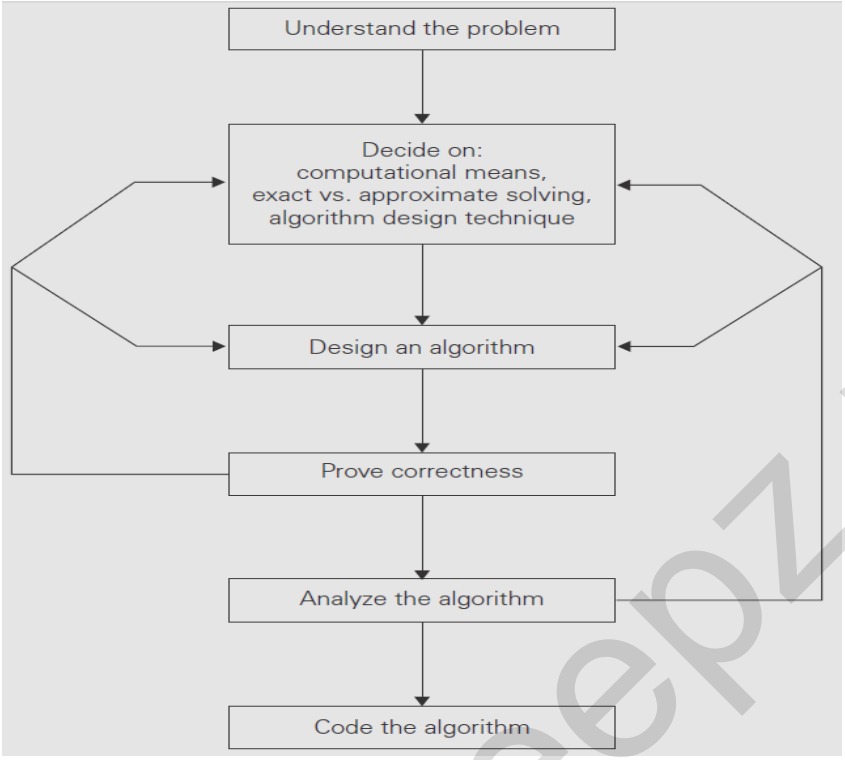


FIGURE 1.2 Algorithm design and analysis process.

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

(b) Choosing between Exact and Approximate Problem Solving

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

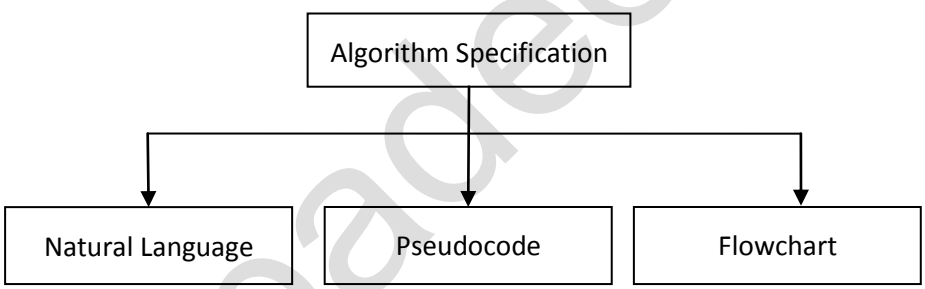


FIGURE 1.3 Algorithm Specifications

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.

Example: An algorithm to perform addition of two numbers.

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

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

b. Pseudocode

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

ALGORITHM *Sum(a,b)*

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

This specification is more useful for implementation of any language.

c. Flowchart

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

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

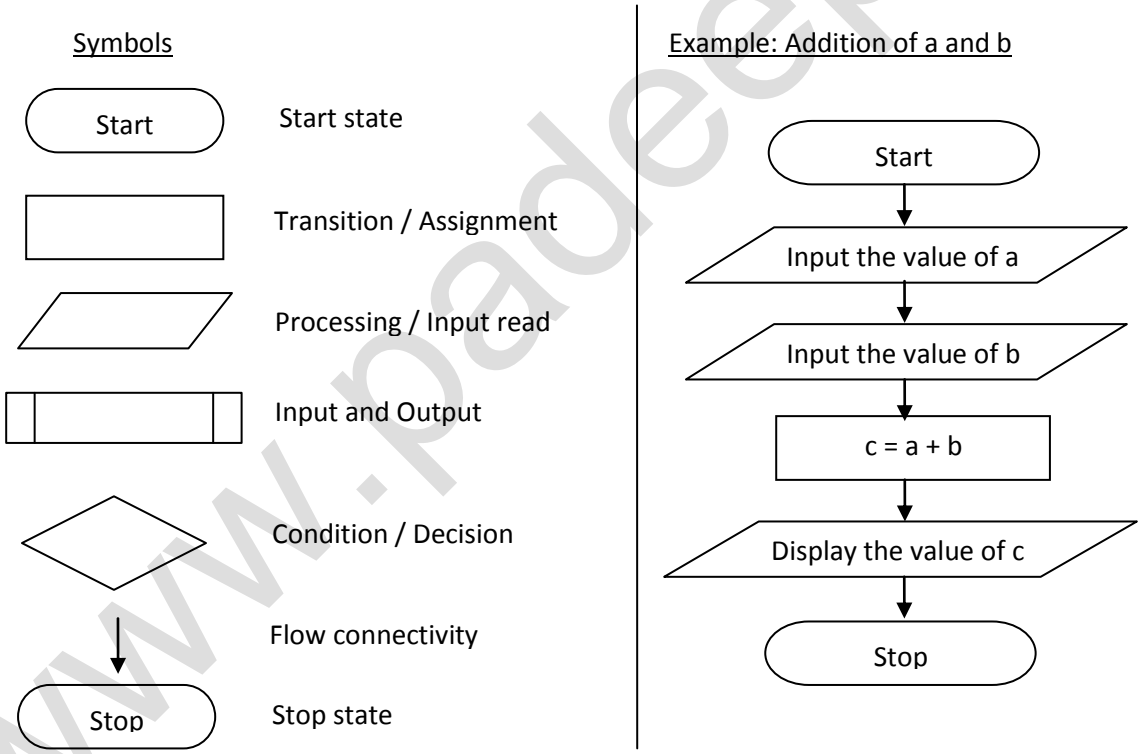


FIGURE 1.4 Flowchart symbols and Example for two integer addition.

(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 of time.

- For example, the correctness of Euclid's algorithm for computing the greatest common divisor stems from the correctness of the equality gcd*(m, n)* = gcd*(n, m* mod *n)*.
- A common technique for proving correctness is to use **mathematical induction** because an algorithm's iterations provide a natural sequence of steps needed for such proofs.
- The notion of correctness for approximation algorithms is less straightforward than it is for exact algorithms. The **error** produced by the algorithm should not exceed a predefined limit.

(v) Analyzing an Algorithm

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

(vi) Coding an Algorithm

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

1.3 IMPORTANT PROBLEM TYPES

The most important problem types are:

(i). Sorting
(ii). Searching
(iii). String processing
(iv). Graph problems
(v). Combinatorial problems
(vi). Geometric problems
(vii). 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 student grade-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 equal elements in its input.

(ii) Searching

- The ***searching problem*** deals with finding a given value, called a ***search key***, in a given set.
- 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 line segments 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 a permutation, a combination, or a subset that satisfies certain constraints.
- A desired combinatorial object may also be required to have some additional property such s 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.

1.4 FUNDAMENTALS OF THE ANALYSIS OF ALGORITHM EFFICIENCY

The efficiency of an algorithm can be in terms of time and space. The algorithm efficiency can 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.

1.5 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

- An algorithm's efficiency is defined as a function of some parameter n indicating the algorithm's input size. In most cases, selecting such a parameter is quite straightforward. For example, it will be the size of the list for problems of sorting, searching.
- For the problem of evaluating a polynomial $p(x) = a_n x^n + \ldots + a_0$ of degree n, the size of the parameter will be the polynomial's degree or the number of its coefficients, which is larger by 1 than its degree.
- In computing the product of two $n \times n$ matrices, the choice of a parameter indicating an input size does matter.
- Consider a spell-checking algorithm. If the algorithm examines individual characters of its input, then the size is measured by the number of characters.
- In measuring input size for algorithms solving problems such as checking primality of a positive integer n. the input is just one number.
- The input size by the number b of bits in the n's binary representation is $b=(\log_2 n)+1$.

(ii) Units for Measuring Running Time

Some standard unit of time measurement such as a second, or millisecond, and so on can be used to measure the running time of a program after implementing the algorithm.

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.

So, we need metric to measure an *algorithm*'s efficiency that does not depend on these extraneous factors.

One possible approach is to ***count the number of times each of the algorithm's operations is 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, and Average-Case Efficiencies

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 found at 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*** of an algorithm is its efficiency for the worst case input of size *n*.
- The algorithm runs the longest among all possible inputs of that size.
- For the input of size *n*, the running time is $C_{worst}(n) = n$.

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 to a search key), then the running time is $C_{best}(n) = 1$

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 list is the same for every *i*.

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

Yet another type of efficiency is called ***amortized efficiency***. It applies not to a single run of an algorithm but rather to a sequence of operations performed on the same data structure.

1.6 ASYMPTOTIC NOTATIONS AND ITS PROPERTIES

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

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

To compare and rank such orders of growth, computer scientists use three notations, 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 1:

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

where $g(n) = n^2$.

(i) O - Big oh notation

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

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

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

O = Asymptotic upper bound = Useful for worst case analysis = Loose bound

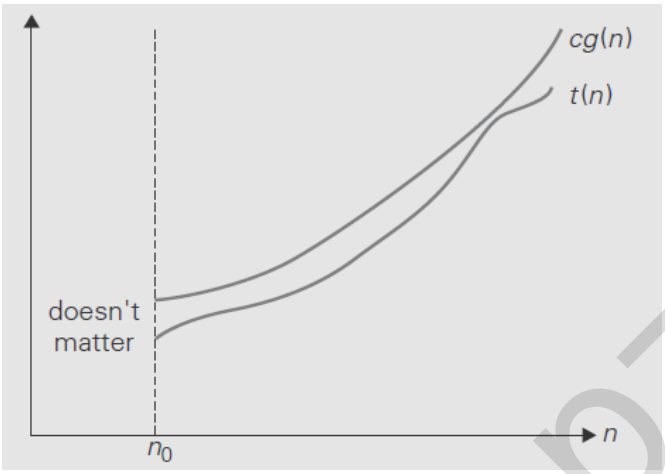


FIGURE 1.5 Big-oh notation: $t\ (n)\ \in\ O(g(n))$.

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

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

$= 101n$

$\leq 101n^2$ ($\because\ n \leq n^2$)

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 3: 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., $t(n) \leq cg(n)$

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

(ii) Ω - Big omega notation

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

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

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

Ω = Asymptotic lower bound = Useful for best case analysis = Loose bound

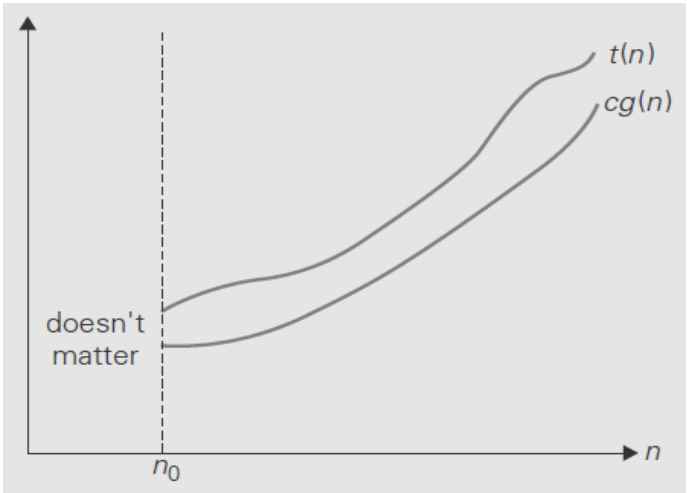


FIGURE 1.6 Big-omega notation: *t (n)* ∈ Ω *(g(n))*.

Example 4: Prove the assertions $n^3+10n^2+4n+2 \in \Omega(n^2)$.
Proof: $n^3+10n^2+4n+2 \geq n^2$ (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

A function t(n) is said to be in Θ*(g(n)),* denoted $t(n) \in \Theta(g(n))$, 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.
Θ = Asymptotic tight bound = Useful for average case analysis

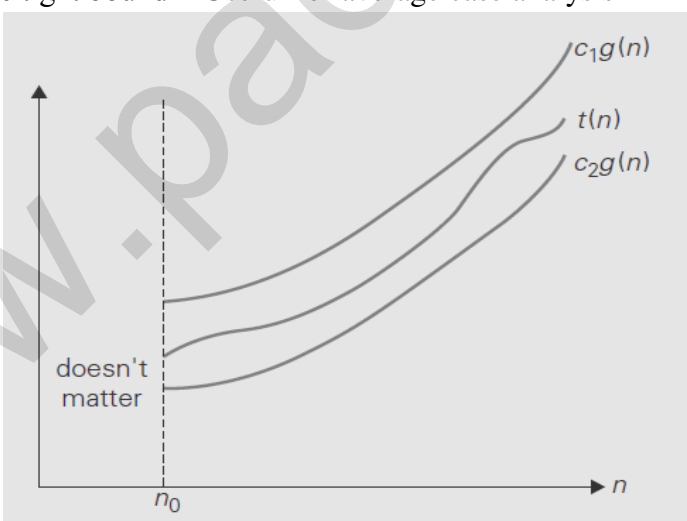


FIGURE 1.7 Big-theta notation: *t (n)* ∈ Θ*(g(n))*.

Example 5: Prove the assertions $\frac{1}{2}n(n-1) \in \Theta(n^2)$.
Proof: First prove the right inequality (the upper bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \leq \frac{1}{2}n^2 \quad \text{for all } n \geq 0.$$

Second, we prove the left inequality (the lower bound):

$$\frac{1}{2}n(n-1) = \frac{1}{2}n^2 - \frac{1}{2}n \geq \frac{1}{2}n^2 - \left[\frac{1}{2}n\right]\left[\frac{1}{2}n\right] \quad \text{for all } n \geq 2.$$

$\therefore \quad \frac{1}{2}n(n-1) \geq \frac{1}{4}n^2$

i.e., $\frac{1}{4}n^2 \leq \frac{1}{2}n(n-1) \leq \frac{1}{2}n^2$

Hence, $\frac{1}{2}n(n-1) \in \Theta(n^2)$, where c2= $\frac{1}{4}$, c1= $\frac{1}{2}$ and n_0=2

Note: asymptotic notation can be thought of as "relational operators" for functions similar to the corresponding relational operators for values.

$= \Rightarrow \Theta()$, $\leq \Rightarrow O()$, $\geq \Rightarrow \Omega()$, $< \Rightarrow o()$, $> \Rightarrow \omega()$

Useful Property Involving the Asymptotic Notations

The following property, in particular, is useful in analyzing algorithms that comprise two consecutively executed parts.

THEOREM: If $t_1(n) \in O(g_1(n))$ and $t_2(n) \in O(g_2(n))$, then $t_1(n) + t_2(n) \in O(\max\{g_1(n), g_2(n)\})$. (The analogous assertions are true for the Ω and Θ notations as well.)

PROOF: The proof extends to orders of growth the following simple fact about four arbitrary real numbers a_1, b_1, a_2, b_2: if $a_1 \leq b_1$ and $a_2 \leq b_2$, then $a_1 + a_2 \leq 2 \max\{b_1, b_2\}$.

Since $t_1(n) \in O(g_1(n))$, there exist some positive constant c_1 and some nonnegative integer n_1 such that

$$t_1(n) \leq c_1 g_1(n) \text{ for all } n \geq n_1.$$

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

$$t_2(n) \leq c_2 g_2(n) \text{ for all } n \geq n_2.$$

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

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

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

The property implies that the algorithm's overall efficiency will be determined by the part with a higher order of growth, i.e., its least efficient part.

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

Basic rules of sum manipulation

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

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

Summation formulas

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

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

1.7 MATHEMATICAL ANALYSIS FOR 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 $n! = 1 \cdot \ldots \cdot (n-1) \cdot n = (n-1)! \cdot n$, for $n \geq 1$ and 0!= 1 by definition, we can compute $F(n) = F(n-1) \cdot n$ with the following recursive algorithm. **(ND 2015)**

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

$$M(n) = \underbrace{M(n-1)}_{\text{To compute F(n-1)}} + \underbrace{1}_{\text{To multiply F(n-1) by n}} \quad \text{for } n > 0$$

$M(n-1)$ multiplications are spent to compute $F(n-1)$, and one more multiplication is needed to multiply the result by *n*.

Recurrence relations

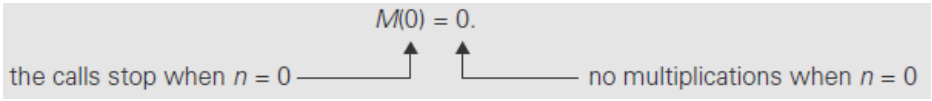
The last equation defines the sequence M(n) that we need to find. This equation defines M(n) not explicitly, i.e., as a function of n, but implicitly as a function of its value at another point, namely n − 1. Such equations are called *recurrence relations* or *recurrences*.

Solve the recurrence relation $M(n) = M(n-1) + 1$, i.e., to find an explicit formula for *M(n)* in terms of *n* only.

To determine a solution uniquely, we need an initial condition that tells us the value with which the sequence starts. We can obtain this value by inspecting the condition that makes the algorithm stop its recursive calls:

if $n = 0$ **return** 1.

This tells us two things. First, since the calls stop when n = 0, the smallest value of n for which this algorithm is executed and hence M(n) defined is 0. Second, by inspecting the pseudocode's exiting line, we can see that when n = 0, the algorithm performs no multiplications.



Thus, the recurrence relation and initial condition for the algorithm's number of multiplications *M(n)*:

M(n) = M(n − 1) + 1 for n > 0,

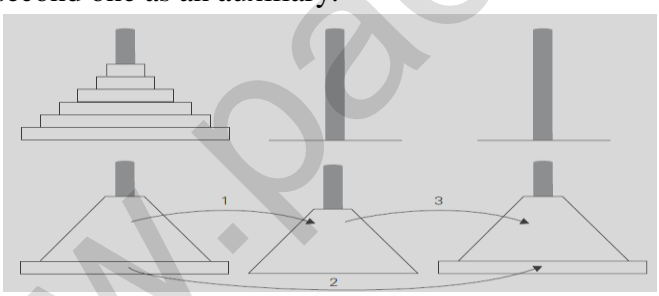
M(0) = 0 for n = 0.

Method of backward substitutions

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.



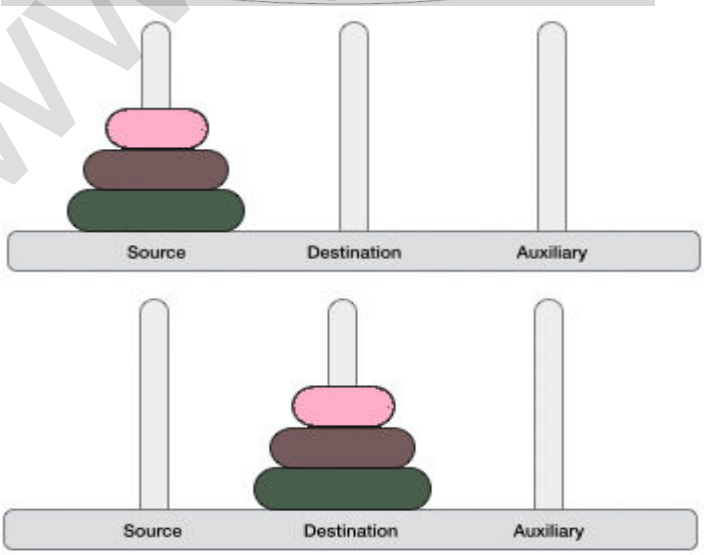


FIGURE 1.8 Recursive solution to the Tower of Hanoi puzzle.

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 recurrence equation 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 the number 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. $M(n-3) = 2M(n-4) + 1$

$= 2^4M(n-4) + 2^3 + 2^2 + 2 + 1$

…

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

…

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

EXAMPLE 3: An investigation of a recursive version of the algorithm which finds the number of binary digits in the **binary representation** of a positive decimal integer.

ALGORITHM *BinRec(n)*

//Input: A positive decimal integer *n*

//Output: The number of binary digits in *n*'s binary representation

if $n = 1$ **return** 1

else return *BinRec*($\lfloor n/2 \rfloor$)+ 1

Algorithm analysis

The number of additions made in computing *BinRec*($\lfloor n/2 \rfloor$) is $A(\lfloor n/2 \rfloor)$, plus one more addition is made by the algorithm to increase the returned value by 1. This leads to the recurrence $A(n) = A(\lfloor n/2 \rfloor) + 1$ for $n > 1$.

Since the recursive calls end when *n* is equal to 1 and there are no additions made

then, the initial condition is $A(1) = 0$.

The standard approach to solving such a recurrence is to solve it only for $n = 2^k$

$A(2^k) = A(2^{k-1}) + 1$ for $k > 0$,

$A(2^0) = 0$.

backward substitutions

$A(2^k) = A(2^{k-1}) + 1$ substitute $A(2^{k-1}) = A(2^{k-2}) + 1$

$= [A(2^{k-2}) + 1] + 1 = A(2^{k-2}) + 2$ substitute $A(2^{k-2}) = A(2^{k-3}) + 1$

$= [A(2^{k-3}) + 1] + 2 = A(2^{k-3}) + 3$ . . .

. . .

$= A(2^{k-i}) + i$

. . .

$= A(2^{k-k}) + k$.

Thus, we end up with $A(2^k) = A(1) + k = k$, or, after returning to the original variable $n = 2^k$ and hence $k = \log_2 n$,

$A(n) = \log_2 n \in \Theta(\log_2 n)$.

1.8 MATHEMATICAL ANALYSIS FOR NON-RECURSIVE ALGORITHMS

General Plan for Analyzing the Time Efficiency of Nonrecursive Algorithms

1. Decide on a *parameter* (or parameters) indicating an input's size.
2. Identify the algorithm's *basic operation* (in the innermost loop).
3. Check whether the *number of times the basic operation is executed* depends only on the size of an input. If it also depends on some additional property, the worst-case, average-case, and, 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 n numbers**. 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].

- The comparison operation is considered as the algorithm's basic operation, because the comparison is executed on each repetition of the loop and not the assignment.
- The number of comparisons will be the same for all arrays of size n; therefore, there is no need to distinguish among the worst, average, and best cases here.
- Let C(n) denotes the number of times this comparison is executed. The algorithm makes one 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, inclusive. Therefore, the sum for C(n) is calculated as follows:

$$c(n) = \sum_{i=1}^{n-1} 1$$

i.e., Sum up 1 in repeated n-1 times

$$c(n) = \sum_{i=1}^{n-1} 1 = n - 1 \in \Theta(n)$$

EXAMPLE 2: Consider the **element uniqueness problem**: check whether all the Elements in a given 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.

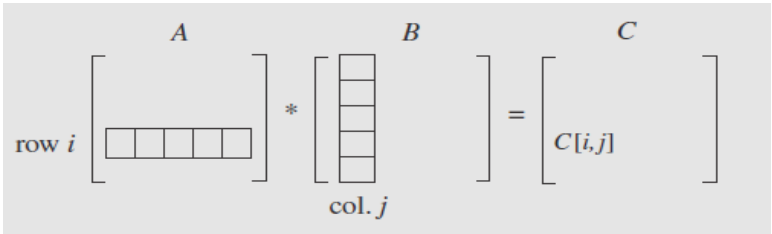
$$C_{worst}(n) = \sum_{i=0}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=0}^{n-2} [(n-1) - (i+1) + 1] = \sum_{i=0}^{n-2} (n-1-i)$$

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

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

EXAMPLE 3: Consider matrix multiplication. Given two n × n matrices A and B, find the time efficiency of the definition-based algorithm for computing their product C = AB. By definition, C

is an n × n matrix whose elements are computed as the scalar (dot) products of the rows of matrix A and 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 of indices 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

- An input's size is matrix order n.
- There are two arithmetical operations (multiplication and addition) in the innermost loop. But we consider multiplication as the basic operation.
- Let us set up a sum for the total number of multiplications M(n) executed by the algorithm. Since this count depends only on the size of the input matrices, we do not have to investigate the worst-case, average-case, and best-case efficiencies separately.
- There is just one multiplication executed on each repetition of the algorithm's innermost loop, which is governed by the variable k ranging from the lower bound 0 to the upper bound n − 1.
- Therefore, the number of multiplications made for every pair of specific values of variables i and j is

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

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

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

Now, we can compute this sum by using formula (S1) and rule (R1)

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

.

The running time of the algorithm on a particular machine m, we can do it by the product

$$T(n) \approx c_m M(n) = c_m n^3,$$

If we consider, time spent on the additions too, then the total time on the machine is

$$T(n) \approx c_m M(n) + c_a A(n) = c_m n^3 + c_a n^3 = (c_m + c_a) n^3$$

EXAMPLE 4 The following algorithm finds the number of binary digits in the **binary representation** of a positive decimal integer.

ALGORITHM Binary(n)

```
//Input: A positive decimal integer n
//Output: The number of binary digits in n's binary representation
count ←1
while n > 1 do
        count ←count + 1
        n←⌊n/2⌋
return count
```

Algorithm analysis

- An input's size is n.
- The loop variable takes on only a few values between its lower and upper limits.
- Since the value of n is about halved on each repetition of the loop, the answer should be about $\log_2 n$.
- The exact formula for the number of times.
- The comparison $n > 1$ will be executed is actually $\lfloor \log_2 n \rfloor + 1$.

UNIT II BRUTE FORCE AND DIVIDE-AND-CONQUER

2.1 BRUTE FORCE

Brute force is a straightforward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts involved.

Selection Sort, Bubble Sort, Sequential Search, String Matching, Depth-First Search and Breadth-First Search, Closest-Pair and Convex-Hull Problems can be solved by Brute Force.

Examples:

1. Computing a^n : a * a * a * … * a (n times)
2. Computing n! : The n! can be computed as n*(n-1)* … *3*2*1
3. Multiplication of two matrices : C=AB
4. Searching a key from list of elements (Sequential search)

Advantages:

1. Brute force is applicable to a very wide variety of problems.
2. It is very useful for solving small size instances of a problem, even though it is inefficient.
3. The brute-force approach yields reasonable algorithms of at least some practical value with no limitation on instance size for sorting, searching, and string matching.

Selection Sort

- First scan 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 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 in the sorted list.
- Generally, on the ***i***th 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:

$$A_0 \le A_1 \le \ldots \le A_{i-1} \mid A_i, \ldots, A_{min}, \ldots, A_{n-1}$$

in their final positions | *the last n – i elements*

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

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*]

| 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

The sorting of list 89, 45, 68, 90, 29, 34, 17 is illustrated with the selection sort algorithm.

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:

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

Thus, selection sort is a $\Theta(n^2)$ algorithm on all inputs.

Note: The number of key swaps is only **Θ(n),** or, more precisely n – 1.

Bubble Sort

The bubble sorting algorithm 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: $A_0, \ldots, A_j \overset{?}{\leftrightarrow} A_{j+1}, \ldots, A_{n-i-1} \mid A_{n-i} \le \ldots \le A_{n-1}$

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* ← 0 **to** n − 2 **do**

for *j* ← 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.

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.

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:

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

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.

$C_{worst}(n) \in \Theta(n^2)$

2.2 CLOSEST-PAIR AND CONVEX-HULL PROBLEMS

We consider a straight forward approach (Brute Force) to two well-known problems dealing with a finite set of points in the plane. These problems are very useful in important applied areas like computational geometry and operations research.

Closest-Pair Problem

The closest-pair problem finds the two closest points in a set of n points. It is the simplest of a variety of problems in computational geometry that deals with proximity of points in the plane or higher-dimensional spaces.

Consider the two-dimensional case of the closest-pair problem. The points are specified in a standard fashion by their (x, y) Cartesian coordinates and that the distance between two points $p_i(x_i, y_i)$ and $p_j(x_j, y_j)$ is the standard Euclidean distance.

$$d(p_i, p_j) = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$$

The following algorithm computes the distance between each pair of distinct points and finds a pair with the smallest distance.

ALGORITHM *BruteForceClosestPair(P)*

//Finds distance between two closest points in the plane by brute force

//Input: A list *P* of *n* $(n \geq 2)$ points $p_1(x_1, y_1), \ldots, p_n(x_n, y_n)$

//Output: The distance between the closest pair of points

$d \leftarrow \infty$

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

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

    $d \leftarrow \min(d, sqrt((x_i - x_j)^2 + (y_i - y_j)^2))$ //*sqrt* is square root

return d

The basic operation of the algorithm will be squaring a number. The number of times it will be executed can be computed as follows:

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

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

$$= 2[(n-1) + (n-2) + \ldots + 1]$$

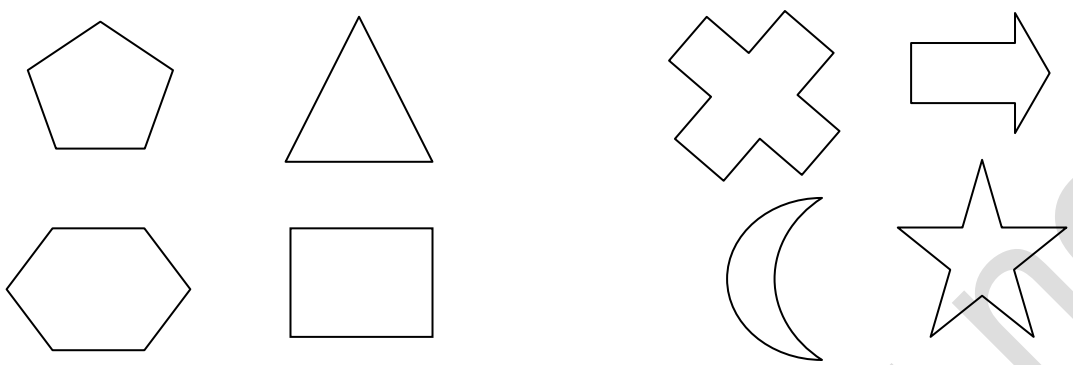
$$= (n-1)n \in \Theta(n^2).$$

Of course, speeding up the innermost loop of the algorithm could only decrease the algorithm's running time by a constant factor, but it cannot improve its asymptotic efficiency class.

Convex-Hull Problem

Convex Set

A set of points (finite or infinite) in the plane is called ***convex*** if for any two points p and q in the set, the entire line segment with the endpoints at p and q belongs to the set.



(a) (b)

FIGURE 2.1 (a) Convex sets. (b) Sets that are not convex.

All the sets depicted in Figure 2.1 (a) are convex, and so are a straight line, a triangle, a rectangle, and, more generally, any convex polygon, a circle, and the entire plane.

On the other hand, the sets depicted in Figure 2.1 (b), any finite set of two or more distinct points, the boundary of any convex polygon, and a circumference are examples of sets that are not convex.

Take a rubber band and stretch it to include all the nails, then let it snap into place. The convex hull is the area bounded by the snapped rubber band as shown in Figure 2.2

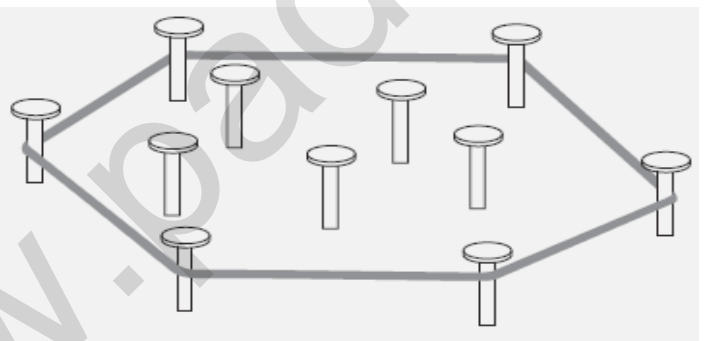


FIGURE 2.2 Rubber-band interpretation of the convex hull.

Convex hull

The ***convex hull*** of a set S of points is the smallest convex set containing S. (The smallest convex hull of S must be a subset of any convex set containing S.)

If S is convex, its convex hull is obviously S itself. If S is a set of two points, its convex hull is the line segment connecting these points. If S is a set of three points not on the same line, its convex hull is the triangle with the vertices at the three points given; if the three points do lie on the same line, the convex hull is the line segment with its endpoints at the two points that are farthest apart. For an example of the convex hull for a larger set, see Figure 2.3.

THEOREM

The convex hull of any set *S* of $n>2$ points not all on the same line is a convex polygon with the vertices at some of the points of *S*. (If all the points do lie on the same line, the polygon degenerates to a line segment but still with the endpoints at two points of *S*.)

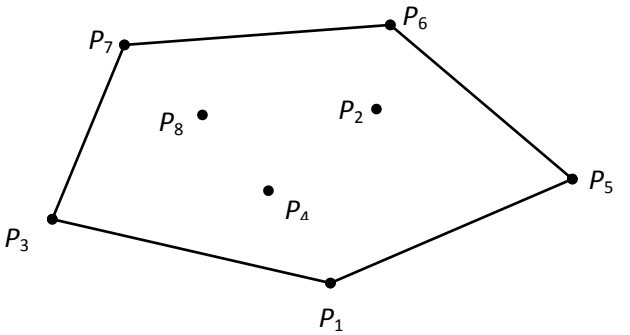


FIGURE 2.3 The convex hull for this set of eight points is the convex polygon with vertices at p_1, p_5, p_6, p_7, and p_3.

The ***convex-hull problem*** is the problem of constructing the convex hull for a given set *S* of *n* points. To solve it, we need to find the points that will serve as the vertices of the polygon in question. Mathematicians call the vertices of such a polygon "extreme points." By definition, an ***extreme point*** of a convex set is a point of this set that is *not a middle point of any line segment with endpoints in the set*. For example, the extreme points of a triangle are its three vertices, the extreme points of a circle are all the points of its circumference, and the extreme points of the convex hull of the set of eight points in Figure 2.3 are p_1, p_5, p_6, p_7, and p_3.

Application

Extreme points have several special properties other points of a convex set do not have. One of them is exploited by the ***simplex method***, This algorithm solves ***linear programming*** **Problems.**

We are interested in extreme points because their identification solves the convex-hull problem. Actually, to solve this problem completely, we need to know a bit more than just which of *n* points of a given set are extreme points of the set's convex hull. we need to know which pairs of points need to be connected to form the boundary of the convex hull. Note that this issue can also be addressed by listing the extreme points in a clockwise or a counterclockwise order.

We can solve the convex-hull problem by brute-force manner. The convex hull problem is one with no obvious algorithmic solution. there is a simple but inefficient algorithm that is based on the following observation about line segments making up the boundary of a convex hull: a line segment connecting two points *pi* and *pj* of a set of *n* points is a part of the convex hull's boundary if and only if all the other points of the set lie on the same side of the straight line through these two points. Repeating this test for every pair of points yields a list of line segments that make up the convex hull's boundary.

Facts

A few elementary facts from analytical geometry are needed to implement the above algorithm.

- First, the straight line through two points $(x_1, y_1), (x_2, y_2)$ in the coordinate plane can be defined by the equation $ax + by = c$, where $a = y_2 - y_1$, $b = x_1 - x_2$, $c = x_1y_2 - y_1x_2$.
- Second, such a line divides the plane into two half-planes: for all the points in one of them, $ax + by > c$, while for all the points in the other, $ax + by < c$. (For the points on the line itself, of course, $ax + by = c$.) Thus, to check whether certain points lie on the same side of the line, we can simply check whether the expression $ax + by - c$ has the same sign for each of these points.

Time efficiency of this algorithm.

Time efficiency of this algorithm is in $O(n^3)$: for each of $n(n-1)/2$ pairs of distinct points, we may need to find the sign of $ax + by - c$ for each of the other $n - 2$ points.

2.3 EXHAUSTIVE SEARCH

For discrete problems in which no efficient solution method is known, it might be necessary to test each possibility sequentially in order to determine if it is the solution. Such ***exhaustive*** examination of all possibilities is known as ***exhaustive search, complete search or*** **direct *search.***

Exhaustive search is simply a brute force approach to combinatorial problems (Minimization or maximization of optimization problems and constraint satisfaction problems).

Reason to choose brute-force / ***exhaustive search*** approach as an important algorithm design strategy

1. First, unlike some of the other strategies, brute force is applicable to a very wide variety of problems. In fact, it seems to be the only **general approach** for which it is more difficult to point out problems it *cannot* tackle.
2. Second, for some important problems, e.g., sorting, searching, matrix multiplication, string matching the brute-force approach yields reasonable algorithms of at least some practical value **with no limitation on instance size**.
3. Third, the expense of designing a more efficient algorithm may be unjustifiable if only a few instances of a problem need to be solved and a brute-force algorithm can solve those instances with **acceptable speed**.
4. Fourth, even if too **inefficient** in general, a brute-force algorithm can still be **useful for solving small-size instances** of a problem.

Exhaustive Search is applied to the important problems like

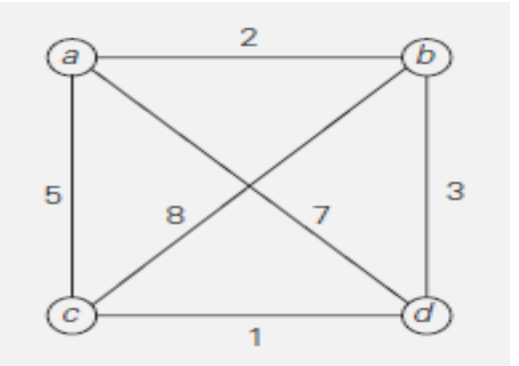
- Traveling Salesman Problem
- Knapsack Problem
- Assignment Problem.

2.4 TRAVELING SALESMAN PROBLEM

The ***traveling salesman problem (TSP)*** is one of the combinatorial problems. 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).

A Hamiltonian circuit can also be defined as a sequence of $n + 1$ adjacent vertices $vi_0, vi_1, \ldots, vi_{n-1}, vi_0$, where the first vertex of the sequence is the same as the last one and all the other $n - 1$ vertices are distinct. All circuits start and end at one particular vertex. Figure 2.4 presents a small instance of the problem and its solution by this method.



Tour	Length
a ---> b ---> c ---> d ---> a	I = 2 + 8 + 1 + 7 = 18
a ---> b ---> d ---> c ---> a	**I = 2 + 3 + 1 + 5 = 11 optimal**
a ---> c ---> b ---> d ---> a	I = 5 + 8 + 3 + 7 = 23
a ---> c ---> d ---> b ---> a	**I = 5 + 1 + 3 + 2 = 11 optimal**
a ---> d ---> b ---> c ---> a	I = 7 + 3 + 8 + 5 = 23
a ---> d ---> c ---> b ---> a	I = 7 + 1 + 8 + 2 = 18

FIGURE 2.4 Solution to a small instance of the traveling salesman problem by exhaustive search.

Time efficiency

- We can get all the tours by generating all the permutations of n − 1 intermediate cities from a particular city.. i.e. **(n - 1)!**
- Consider two intermediate vertices, say, *b* and *c*, and then only permutations in which *b* precedes *c*. (This trick implicitly defines a tour's direction.)
- An inspection of Figure 2.4 reveals three pairs of tours that differ only by their direction. Hence, we could cut the number of vertex permutations by **half** because cycle total lengths in both directions are same.
- The total number of permutations needed is still $\frac{1}{2}(n-1)!$, which makes the exhaustive-search approach impractical for large n. It is useful for very small values of *n*.

2.5 KNAPSACK PROBLEM

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.

Real time examples:

- A Thief who wants to steal the most valuable loot that fits into his knapsack,
- A transport plane that has to deliver the most valuable set of items to a remote location without exceeding the plane's capacity.

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.

FIGURE 2.5 Instance of the knapsack problem.

Subset	Total weight	Total value
Φ	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 (Maximum-Optimum)**
{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

FIGURE 2.6 knapsack problem's solution by exhaustive search. The information about the optimal selection is in bold.

Time efficiency: As given in the example, the solution to the instance of Figure 2.5 is given in Figure 2.6. Since the *number of subsets of an n-element set is* 2^n, the exhaustive search leads to a $\Omega(2^n)$ algorithm, no matter how efficiently individual subsets are generated.

Note: Exhaustive search of both the traveling salesman and knapsack problems leads to extremely inefficient algorithms on every input. In fact, these two problems are the best-known examples of ***NP-hard problems***. **No polynomial-time** algorithm is known for any *NP*-hard problem. Moreover, most computer scientists believe that such algorithms do not exist. some sophisticated approaches like **backtracking** and **branch-and-bound** enable us to solve some instances but not all instances of these in less than exponential time. Alternatively, we can use one of many **approximation algorithms.**

2.6 ASSIGNMENT PROBLEM.

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 jth 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.

Assignment problem solved by exhaustive search is illustrated with an example as shown in figure 2.8. 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

FIGURE 2.7 Instance of an Assignment problem.

An instance of the assignment problem is completely specified by its cost matrix *C*.

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

The problem is to select one element in each row of the matrix so that all selected elements are in different columns and the total sum of the selected elements is the smallest possible.

We can describe feasible solutions to the assignment problem as *n*-tuples <j_1, . . . , j_n > in which the *i*th component, $i = 1, \ldots, n$, indicates the column of the element selected in the *i*th row (i.e., the job number assigned to the *i*th 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. Similarly we can have $4! = 4 \cdot 3 \cdot 2 \cdot 1 = 24, i.e., 24$ permutations.

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 given below.

<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

<2, 1, 3, 4>	**cost = 2 + 6 + 1 + 4 = 13 (Min)**
<2, 1, 4, 3>	cost = 2 + 6 + 8 + 9 = 25
<2, 3, 1, 4>	cost = 2 + 3 + 5 + 4 = 14
<2, 3, 4, 1>	cost = 2 + 3 + 8 + 7 = 20
<2, 4, 1, 3>	cost = 2 + 7 + 5 + 9 = 23
<2, 4, 3, 1>	cost = 2 + 7 + 1 + 7 = 17, etc

FIGURE 2.8 First few iterations of solving a small instance of the assignment problem by exhaustive search.

Since the number of permutations to be considered for the general case of the assignment problem is ***n!,*** exhaustive search is impractical for all but very small instances of the problem. Fortunately, there is a much more efficient algorithm for this problem called the ***Hungarian method.***

2.7 DIVIDE AND CONQUER METHODOLOGY

A **divide and conquer algorithm** works by recursively breaking down a problem into two or more sub-problems of the same (or related) type (**divide**), until these become simple enough to be solved directly (**conquer**).

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

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

The divide-and-conquer technique as shown in Figure 2.9, which depicts the case of dividing a problem into two smaller subproblems, then the subproblems solved separately. Finally solution to the original problem is done by combining the solutions of subproblems.

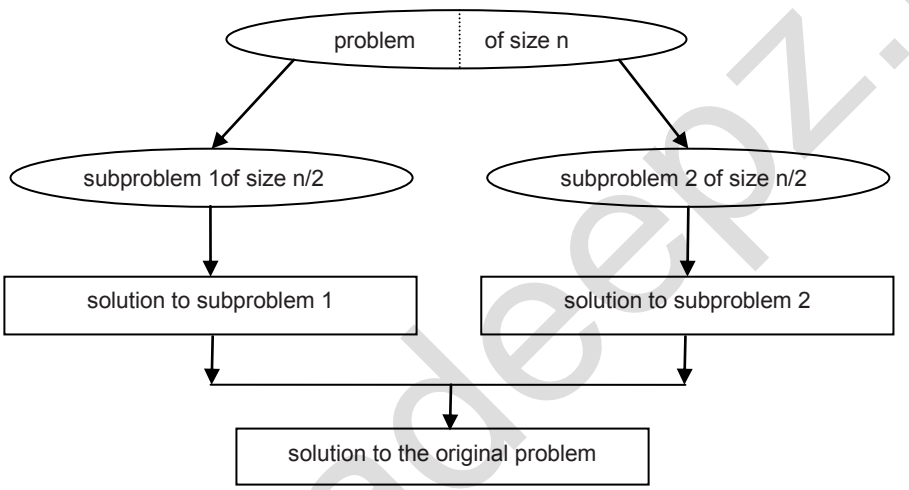


FIGURE 2.9 Divide-and-conquer technique.

Divide and conquer methodology can be easily applied on the following problem.

1. Merge sort
2. Quick sort
3. Binary search

2.8 MERGE SORT

Mergesort is based on divide-and-conquer technique. It sorts a given array $A[0..n-1]$ by dividing it into two halves $A[0..\lfloor n/2 \rfloor - 1]$ and $A[\lfloor n/2 \rfloor..n-1]$, sorting each of them recursively, and then merging the two smaller sorted arrays into a single sorted one.

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

//Sorts array $A[0..n-1]$ by recursive mergesort

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

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

if $n > 1$

copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$

copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$

Mergesort($B[0..\lfloor n/2 \rfloor - 1]$*)*

Mergesort($C[0..\lceil n/2 \rceil - 1]$*)*

Merge(B, C, A) //see below

The ***merging*** of two sorted arrays can be done as follows. Two pointers (array indices) are initialized to point to the first elements of the arrays being merged. The elements pointed to are compared, and the smaller of them is added to a new array being constructed; after that, the index of the smaller element is incremented to point to its immediate successor in the array it was copied from. This operation is repeated until one of the two given arrays is exhausted, and then the remaining elements of the other array are copied to the end of the new array.

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

//Merges two sorted arrays into one sorted array

//Input: Arrays *B*[0..$p-1$] and *C*[0..$q-1$] both sorted

//Output: Sorted array *A*[0..$p+q-1$] of the elements of *B* and *C*

$i \leftarrow 0$; $j \leftarrow 0$; $k \leftarrow 0$

while $i < p$ **and** $j < q$ **do**

  if $B[i] \leq C[j]$

    $A[k] \leftarrow B[i]$; $i \leftarrow i+1$

  else $A[k] \leftarrow C[j]$; $j \leftarrow j+1$

  $k \leftarrow k+1$

if $i = p$

  copy *C*[$j..q-1$] to *A*[$k..p+q-1$]

else copy *B*[$i..p-1$] to *A*[$k..p+q-1$]

The operation of the algorithm on the list 8, 3, 2, 9, 7, 1, 5, 4 is illustrated in Figure 2.10.

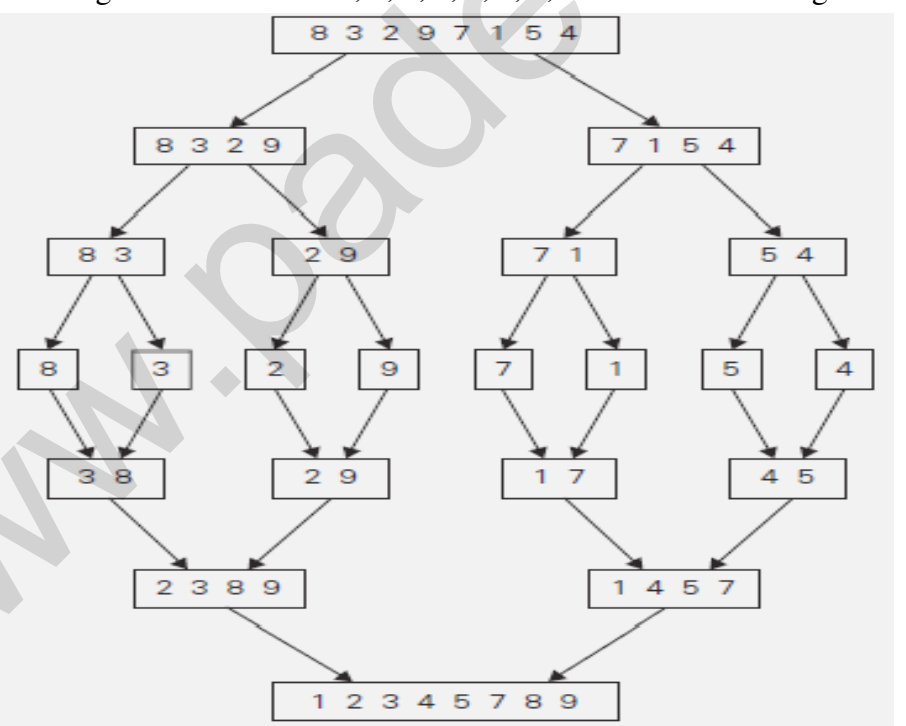


FIGURE 2.10 Example of mergesort operation.

The recurrence relation for the number of key comparisons $C(n)$ is

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

In the worst case, $C_{merge}(n) = n - 1$, and we have the recurrence

$Cworst(n) = 2C_{worst}(n/2) + n - 1$ for $n > 1$, $C_{worst}(1) = 0$.

By Master Theorem, $C_{worst}(n) \in \Theta(n \log n)$
the exact solution to the worst-case recurrence for $n = 2^k$
$C_{worst}(n) = n \log_2 n - n + 1$.

For large *n,* the number of comparisons made by this algorithm in the average case turns out to be about 0.25*n* less and hence is also in **Θ *(n* log *n).***

First, the algorithm can be implemented bottom up by merging pairs of the array's elements, then merging the sorted pairs, and so on. This avoids the time and space overhead of using a stack to handle recursive calls. Second, we can divide a list to be sorted in more than two parts, sort each recursively, and then merge them together. This scheme, which is particularly useful for sorting files residing on secondary memory devices, is called ***multiway mergesort***.

2.9 QUICK SORT

Quicksort is the other important sorting algorithm that is based on the divide-and-conquer approach. quicksort divides input elements according to their value. A partition is an arrangement of the array's elements so that all the elements to the left of some element $A[s]$ are less than or equal to $A[s]$, and all the elements to the right of $A[s]$ are greater than or equal to it:

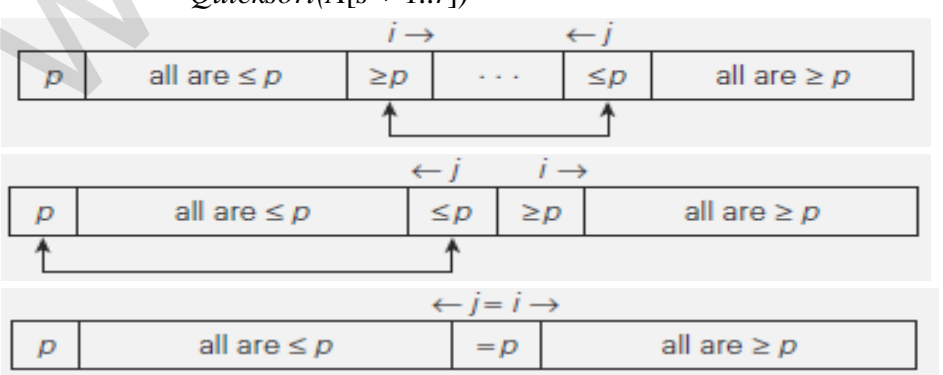
$$\underbrace{A[0] \ldots A[s-1]}_{\text{all are} \le A[s]} \qquad A[s] \qquad \underbrace{A[s+1] \ldots A[n-1]}_{\text{all are} \ge A[s]}$$

Sort the two subarrays to the left and to the right of $A[s]$ independently. No work required to combine the solutions to the subproblems.

Here is pseudocode of quicksort: call *Quicksort(*$A[0..n-1]$*)* where As a partition algorithm use the *HoarePartition*

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

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



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

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

0	1	2	3	4	5	6	7
	i						*j*
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	9	8	2	4	7
			i			*j*	
5	3	1	4	8	2	9	7
			i		*j*		
5	3	1	4	8	2	9	7
			i		*j*		
5	3	1	4	2	8	9	7
			j		*i*		
5	3	1	4	2	8	9	7
2	3	1	4	**5**	8	9	7
	i		*j*				
2	3	1	4				
	i	*j*					
2	3	1	4				
	i	*j*					
2	1	3	4				
	j	*i*					
2	1	3	4				
1	**2**	3	4				
1							
			i j				
		3	4				
		j	*i*				
		3	4				
			4				
						i	*j*
					8	9	7
						i	*j*
					8	7	9
						j	*i*
					8	7	9
					7	**8**	9
					7		
							9

FIGURE 2.11 Example of quicksort operation of Array with pivots shown in bold.

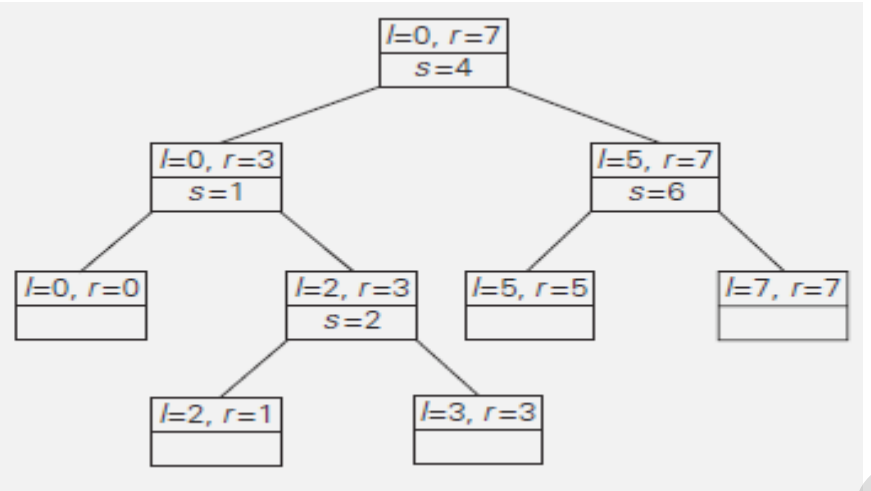


FIGURE 2.12 Tree of recursive calls to *Quicksort* with input values *l* and *r* of subarray bounds and split position *s* of a partition obtained.

The number of key comparisons in the best case satisfies the recurrence

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

By Master Theorem, $C_{best}(n) \in \Theta(n \log_2 n)$; solving it exactly for $n = 2^k$ yields $C_{best}(n) = n \log_2 n$.

The total number of key comparisons made will be equal to

$Cworst(n) = (n + 1) + n + \ldots + 3 = ((n + 1)(n + 2))/2 - 3 \in \Theta(n^2)$.

$$C_{avg}(n) = \frac{1}{n}\sum_{s=0}^{n-1}[(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1,$$

$$C_{avg}(0) = 0, \quad C_{avg}(1) = 0.$$

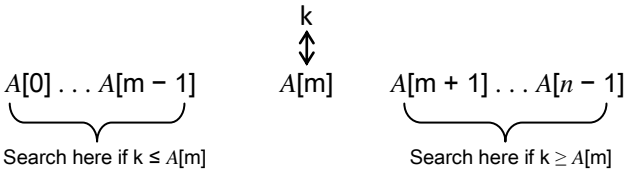
$$C_{avg}(n) \approx 2n \ln n \approx 1.39n \log_2 n.$$

2.10 BINARY SEARCH

A binary search is efficient algorithm to find the position of a target (key) value within a sorted array.

- The binary search algorithm begins by comparing the target value to the value of the middle element of the sorted array. If the target value is equal to the middle element's value, then the position is returned and the search is finished.
- If the target value is less than the middle element's value, then the search continues on the lower half of the array.
- if the target value is greater than the middle element's value, then the search continues on the upper half of the array.
- This process continues, eliminating half of the elements, and comparing the target value to the value of the middle element of the remaining elements - until the target value is either found (position is returned).

Binary search is a remarkably efficient algorithm for searching in a sorted array (Say A). 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]:



Though binary search is clearly based on a recursive idea, it can be easily implemented as a nonrecursive algorithm, too. Here is pseudocode of this nonrecursive version.

ALGORITHM *BinarySearch(A*[0..*n* − 1]*, K)*

//Implements nonrecursive 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

$l \leftarrow 0$; $r \leftarrow n - 1$

while $l \leq r$ **do**

$m \leftarrow \lfloor (l + r)/2 \rfloor$

if $K = A[m]$ **return** m

else if $K < A[m]$

$r \leftarrow m - 1$

else $l \leftarrow m + 1$

return −1

The standard way to analyze the efficiency of binary search is to count the number of times the search key is compared with an element of the array (three-way comparisons). One comparison of K with A[m], the algorithm can determine whether K is smaller, equal to, or larger than A[m].

As an example, let us apply binary search to searching for $K = 70$ in the array. 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								*l*		*m*			*r*
iteration 3								*l*,*m*	*r*				

The worst-case inputs include all arrays that do not contain a given search key, as well as some successful searches. Since after one comparison the algorithm faces the same situation but for an array half the size,

The number of key comparisons in the worst case $C_{worst}(n)$ by recurrence relation.

$$C_{worst}(n) = C_{worst}\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + 1 \; for \; n > 1, C_{worst}(1) = 1.$$

$\therefore C_{worst}(n) = \lfloor log_2 \, n \rfloor + 1 = \lceil log_2(n + 1) \rceil$ $\because C_{worst}(2^k) = (k + 1) = log_2 \, k + 1$ for $n = 2^k$

- First, The worst-case time efficiency of binary search is in Θ(log n).
- Second, the algorithm simply reduces the size of the remaining array by half on each iteration, the number of such iterations needed to reduce the initial size *n* to the final size 1 has to be about $\log_2 n$.

- Third, the logarithmic function grows so slowly that its values remain small even for very large values of *n*.

The average case slightly smaller than that in the worst case

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

The average number of comparisons in a successful is

$\boldsymbol{C_{avg}(n) \approx \log_2 n - 1}$

The average number of comparisons in an unsuccessful is

$\boldsymbol{C_{avg}(n) \approx \log_2(n + 1).}$

2.11 MULTIPLICATION OF LARGE INTEGERS

Some applications like modern cryptography require manipulation of integers that are over 100 decimal digits long. Since such integers are too long to fit in a single word of a modern computer, they require special treatment.

In the conventional pen-and-pencil algorithm for multiplying two n-digit integers, each of the n digits of the first number is multiplied by each of the n digits of the second number for the total of n^2 digit multiplications.

The divide-and-conquer method does the above multiplication in less than n^2 digit multiplications.

Example: $23 * 14 = (2 \cdot 10^1 + 3 \cdot 10^0) * (1 \cdot 10^1 + 4 \cdot 10^0)$

$= (2 * 1)10^2 + (2 * 4 + 3 * 1)10^1 + (3 * 4)10^0$

$= 2 \cdot 10^2 + 11 \cdot 10^1 + 12 \cdot 10^0$

$= 3 \cdot 10^2 + 2 \cdot 10^1 + 2 \cdot 10^0$

$= 322$

The term $(2 * 1 + 3 * 4)$ computed as $2 * 4 + 3 * 1 = (2 + 3) * (1 + 4) - (2 * 1) - (3 * 4)$. Here $(2 * 1)$ and $(3 * 4)$ are already computed used. So only one multiplication only we have to do.

For any pair of two-digit numbers $a = a_1a_0$ and $b = b_1b_0$, their product c can be computed by the formula $c = a * b = c_2 10^2 + c_1 10^1 + c_0$,

where

$c_2 = a_1 * b_1$ is the product of their first digits,

$c_0 = a_0 * b_0$ is the product of their second digits,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$ is the product of the sum of the

a's digits and the sum of the b's digits minus the sum of c_2 and c_0.

Now we apply this trick to multiplying two *n*-digit integers *a* and *b* where *n* is a positive even number. Let us divide both numbers in the middle to take advantage of the divide-and-conquer technique. We denote the first half of the *a*'s digits by a_1 and the second half by a_0; for *b*, the notations are b_1 and b_0, respectively. In these notations, $a = a_1a_0$ implies that $a = a_1 10^{n/2} + a_0$ and $b = b_1b_0$ implies that $b = b_1 10^{n/2} + b_0$. Therefore, taking advantage of the same trick we used for two-digit numbers, we get

$C = a * b = (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0)$

$= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0)$

$= c_2 10^n + c_1 10^{n/2} + c_0,$

where

$c_2 = a_1 * b_1$ is the product of their first halves,

$c_0 = a_0 * b_0$ is the product of their second halves,

$c_1 = (a_1 + a_0) * (b_1 + b_0) - (c_2 + c_0)$

If $n/2$ is even, we can apply the same method for computing the products c_2, c_0, and c_1. Thus, if n is a power of 2, we have a recursive algorithm for computing the product of two n-digit integers. In its pure form, the recursion is stopped when n becomes 1. It can also be stopped when we deem n small enough to multiply the numbers of that size directly.

The multiplication of n-digit numbers requires three multiplications of $n/2$-digit numbers, the recurrence for the number of multiplications $M(n)$ is $M(n) = 3M(n/2)$ for $n > 1$, $M(1) = 1$. Solving it by backward substitutions for $n = 2^k$ yields

$$
\begin{aligned}
M(2^k) &= 3M(2^{k-1}) \\
&= 3[3M(2^{k-2})] \\
&= 3^2M(2^{k-2}) \\
&= \ldots \\
&= 3^iM(2^{k-i}) \\
&= \ldots \\
&= 3^kM(2^{k-k}) \\
&= 3^k.
\end{aligned}
$$

(Since $k = \log_2 n$)

$$\boldsymbol{M(n) = 3^{\log_2 n} = n^{\log_2 3} \approx n^{1.585}.}$$

(On the last step, we took advantage of the following property of logarithms: $a^{\log_b c} = c^{\log_b a}$.)

Let $A(n)$ be the number of digit additions and subtractions executed by the above algorithm in multiplying two n-digit decimal integers. Besides $3A(n/2)$ of these operations needed to compute the three products of $n/2$-digit numbers, the above formulas require five additions and one subtraction. Hence, we have the recurrence

$A(n) = 3 \cdot A(n/2) + cn$ for $n > 1$, $A(1) = 1$.

By using Master Theorem, we obtain $\boldsymbol{A(n) \in \Theta(n^{\log_2 3})}$,

which means that the total number of additions and subtractions have the same asymptotic order of growth as the number of multiplications.

Example: For instance: a = 2345, b = 6137, i.e., n=4.

Then C = a * b = $(23*10^2+45)*(61*10^2+37)$

$$
\begin{aligned}
C = a * b &= (a_1 10^{n/2} + a_0) * (b_1 10^{n/2} + b_0) \\
&= (a_1 * b_1)10^n + (a_1 * b_0 + a_0 * b_1)10^{n/2} + (a_0 * b_0) \\
&= (23 * 61)10^4 + (23 * 37 + 45 * 61)10^2 + (45 * 37) \\
&= 1403 \cdot 10^4 + 3596 \cdot 10^2 + 1665 \\
&= 14391265
\end{aligned}
$$

2.12 STRASSEN'S MATRIX MULTIPLICATION

The Strassen's Matrix Multiplication find the product C of two 2×2 matrices A and B with just seven multiplications as opposed to the eight required by the brute-force algorithm.

$$\begin{bmatrix} c_{00} & c_{01} \\ c_{10} & c_{11} \end{bmatrix} = \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} * \begin{bmatrix} b_{00} & b_{01} \\ b_{10} & b_{11} \end{bmatrix}$$
$$= \begin{bmatrix} m_1 + m_4 - m_5 + m_7 & m_3 + m_5 \\ m_2 + m_4 & m_1 + m_3 - m_2 + m_6 \end{bmatrix}$$

where

$$\begin{aligned} m_1 &= (a_{00} + a_{11}) * (b_{00} + b_{11}), \\ m_2 &= (a_{10} + a_{11}) * b_{00}, \\ m_3 &= a_{00} * (b_{01} - b_{11}), \\ m_4 &= a_{11} * (b_{10} - b_{00}), \\ m_5 &= (a_{00} + a_{01}) * b_{11}, \\ m_6 &= (a_{10} - a_{00}) * (b_{00} + b_{01}), \\ m_7 &= (a_{01} - a_{11}) * (b_{10} + b_{11}). \end{aligned}$$

Thus, to multiply two 2×2 matrices, Strassen's algorithm makes 7 multiplications and 18 additions/subtractions, whereas the brute-force algorithm requires 8 multiplications and 4 additions. These numbers should not lead us to multiplying 2×2 matrices by Strassen's algorithm. Its importance stems from its *asymptotic* superiority as matrix order n goes to infinity.

Let A and B be two $n \times n$ matrices where n is a power of 2. (If n is not a power of 2, matrices can be padded with rows and columns of zeros.) We can divide A, B, and their product C into four $n/2 \times n/2$ submatrices each as follows:

$$\left[\begin{array}{c|c} C_{00} & C_{01} \\ \hline C_{10} & C_{11} \end{array}\right] = \left[\begin{array}{c|c} A_{00} & A_{01} \\ \hline A_{10} & A_{11} \end{array}\right] * \left[\begin{array}{c|c} B_{00} & B_{01} \\ \hline B_{10} & B_{11} \end{array}\right]$$

The value C_{00} can be computed either as $A_{00} * B_{00} + A_{01} * B_{10}$ or as $M_1 + M_4 - M_5 + M_7$ where M_1, M_4, M_5, and M_7 are found by Strassen's formulas, with the numbers replaced by the corresponding submatrices. The seven products of $n/2 \times n/2$ matrices are computed recursively by Strassen's matrix multiplication algorithm.

The asymptotic efficiency of Strassen's matrix multiplication algorithm

If M(n) is the number of multiplications made by Strassen's algorithm in multiplying two n×n matrices, where n is a power of 2, The recurrence relation is M(n) = 7M(n/2) for n > 1, M(1)=1.

Since $n = 2^k$,

$$\begin{aligned} M(2^k) &= 7M(2^{k-1}) \\ &= 7[7M(2^{k-2})] \\ &= 7^2 M(2^{k-2}) \\ &= \ldots \end{aligned}$$

$= 7^i M(2^{k-i})$

$= \ldots$

$= 7^k M(2^{k-k}) = 7^k M(2^0) = 7^k M(1) = 7^k(1)$ (Since M(1)=1)

$M(2^k) = 7^k$.

Since $k = \log_2 n$,

$M(n) = 7^{\log_2 n}$

$= n^{\log_2 7}$

$\approx \boldsymbol{n^{2.807}}$

which is smaller than n^3 required by the brute-force algorithm.

Since this savings in the number of multiplications was achieved at the expense of making extra additions, we must check the number of additions $A(n)$ made by Strassen's algorithm. To multiply two matrices of order $n>1$, the algorithm needs to multiply seven matrices of order $n/2$ and make 18 additions/subtractions of matrices of size $n/2$; when $n = 1$, no additions are made since two numbers are simply multiplied. These observations yield the following recurrence relation:

$A(n) = 7A(n/2) + 18(n/2)^2$ for $n > 1$, $A(1) = 0$.

By closed-form solution to this recurrence and the Master Theorem, $A(n) \in \Theta(n^{\log_2 7})$. which is a better efficiency class than $\Theta(n^3)$of the brute-force method.

Example: Multiply the following two matrices by Strassen's matrix multiplication algorithm.

$$\mathbf{A} = \begin{bmatrix} 1 & 0 & 2 & 1 \\ 4 & 1 & 1 & 0 \\ 0 & 1 & 3 & 0 \\ 5 & 0 & 2 & 1 \end{bmatrix} \qquad \mathbf{B} = \begin{bmatrix} 0 & 1 & 0 & 1 \\ 2 & 1 & 0 & 4 \\ 2 & 0 & 1 & 1 \\ 1 & 3 & 5 & 0 \end{bmatrix}$$

Answer:

$$\mathbf{C} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \mathbf{x} \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix}$$

Where $\mathbf{A_{00}} = \begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix}$ $\mathbf{A_{01}} = \begin{bmatrix} 2 & 1 \\ 1 & 0 \end{bmatrix}$ $\mathbf{A_{10}} = \begin{bmatrix} 0 & 1 \\ 5 & 0 \end{bmatrix}$ $\mathbf{A_{11}} = \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}$

$\mathbf{B_{00}} = \begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix}$ $\mathbf{B_{01}} = \begin{bmatrix} 0 & 1 \\ 0 & 4 \end{bmatrix}$ $\mathbf{B_{10}} = \begin{bmatrix} 2 & 0 \\ 1 & 3 \end{bmatrix}$ $\mathbf{B_{11}} = \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}$

$M_1 = (A_{00}+A_{11})*(B_{00}+B_{11}) = \left(\begin{bmatrix} 1 & 0 \\ 4 & 1 \end{bmatrix} + \begin{bmatrix} 3 & 0 \\ 2 & 1 \end{bmatrix}\right) * \left(\begin{bmatrix} 0 & 1 \\ 2 & 1 \end{bmatrix} + \begin{bmatrix} 1 & 1 \\ 5 & 0 \end{bmatrix}\right) = \begin{bmatrix} 4 & 0 \\ 6 & 2 \end{bmatrix} * \begin{bmatrix} 1 & 2 \\ 7 & 1 \end{bmatrix} = \begin{bmatrix} 4 & 8 \\ 20 & 14 \end{bmatrix}$

Similarly apply Strassen's matrix multiplication algorithm to find the following.

$M_2 = \begin{bmatrix} 2 & 4 \\ 2 & 8 \end{bmatrix}$, $M_3 = \begin{bmatrix} -1 & 0 \\ -9 & 4 \end{bmatrix}$, $M_4 = \begin{bmatrix} 6 & -3 \\ 3 & 0 \end{bmatrix}$, $M_5 = \begin{bmatrix} 8 & 3 \\ 10 & 5 \end{bmatrix}$, $M_6 = \begin{bmatrix} 2 & -3 \\ -2 & -3 \end{bmatrix}$, $M_7 = \begin{bmatrix} 3 & 2 \\ -9 & -4 \end{bmatrix}$

$C_{00} = \begin{bmatrix} 5 & 4 \\ 4 & 5 \end{bmatrix}$, $C_{01} = \begin{bmatrix} -7 & 3 \\ 1 & 9 \end{bmatrix}$, $C_{10} = \begin{bmatrix} 8 & 1 \\ 5 & 8 \end{bmatrix}$, $C_{11} = \begin{bmatrix} 3 & 7 \\ 7 & 7 \end{bmatrix}$

$$\mathbf{C} = \begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} 5 & 4 & 7 & 3 \\ 4 & 5 & 1 & 9 \\ 8 & 1 & 3 & 7 \\ 5 & 8 & 7 & 7 \end{bmatrix}$$

2.13 Closest-Pair and Convex-Hull Problems.

The two-dimensional versions of the closest-pair problem and the convex-hull problem problems can be solved by brute-force algorithms in $\theta(n^2)$ and $O(n^3)$ time, respectively. The divide-and-conquer technique provides sophisticated and asymptotically more efficient algorithms to solve these problems.

The Closest-Pair Problem

Let P be a set of $n > 1$ points in the Cartesian plane. The points are ordered in nondecreasing order of their x coordinate. It will also be convenient to have the points sorted (by merge sort) in a separate list in nondecreasing order of the y coordinate and denote such a list by Q.

If $2 \le n \le 3$, the problem can be solved by the obvious brute-force algorithm. If $n > 3$, we can divide the points into two subsets Pl and Pr of $\lceil n/2 \rceil$ and $\lfloor n/2 \rfloor$ points, respectively, by drawing a vertical line through the median m of their x coordinates so that $\lceil n/2 \rceil$ points lie to the left of or on the line itself, and $\lceil n/2 \rceil$ points lie to the right of or on the line. Then we can solve the closest-pair problem recursively for subsets P_l and P_r. Let d_l and d_r be the smallest distances between pairs of points in Pl and Pr, respectively, and let $d = \min\{d_l, d_r\}$.

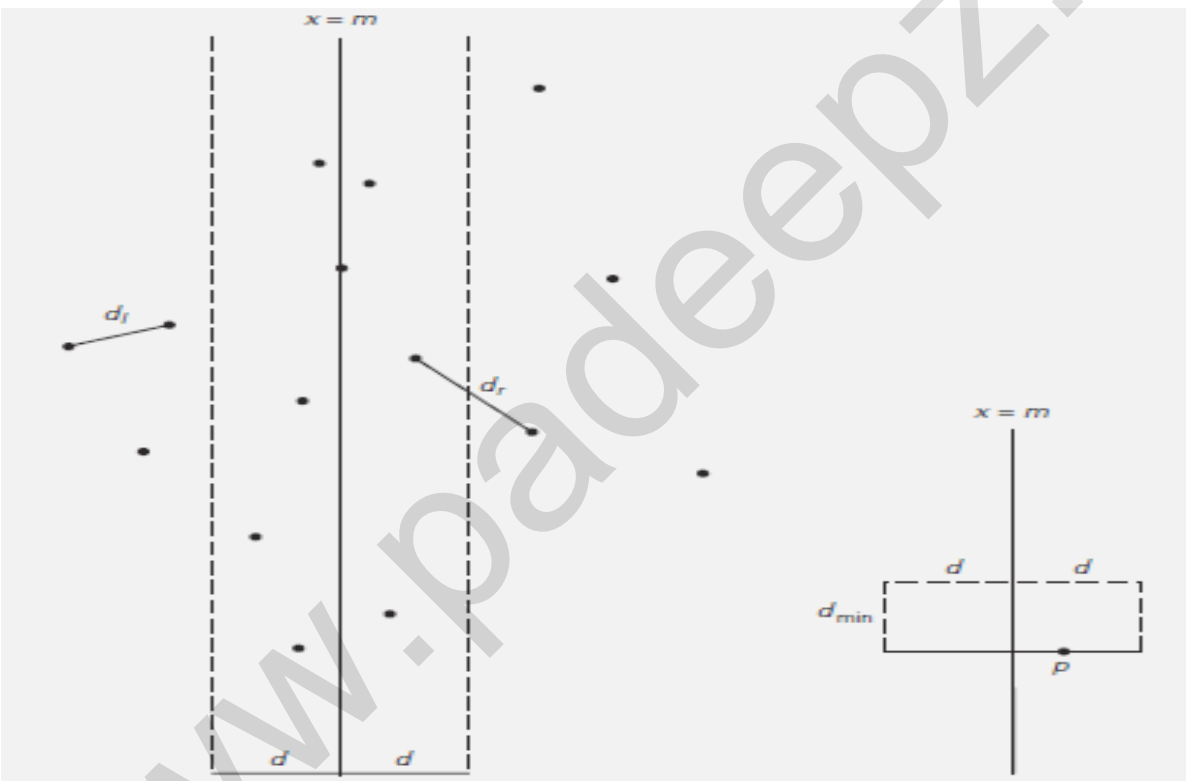


FIGURE 2.13 (a) Idea of the divide-and-conquer algorithm for the closest-pair problem. (b) Rectangle that may contain points closer than d_{min} to point p.

Note that d is not necessarily the smallest distance between all the point pairs because points of a closer pair can lie on the opposite sides of the separating line. Therefore, as a step combining the solutions to the smaller subproblems, we need to examine such points. Obviously, we can limit our attention to the points inside the symmetric vertical strip of width $2d$ around the separating line, since the distance between any other pair of points is at least d (Figure 2.13a).

Let S be the list of points inside the strip of width 2d around the separating line, obtained from Q and hence ordered in nondecreasing order of their y coordinate. We will scan this list, updating the information about dmin, the minimum distance seen so far, if we encounter a closer

pair of points. Initially, $d_{min} = d$, and subsequently $d_{min} \leq d$. Let p(x, y) be a point on this list. For a point p (x, y) to have a chance to be closer to p than d_{min}, the point must follow p on list S and the difference between their y coordinates must be less than d_{min}.

Geometrically, this means that p must belong to the rectangle shown in Figure 2.13b. The principal insight exploited by the algorithm is the observation that the rectangle can contain just a few such points, because the points in each half (left and right) of the rectangle must be at least distance d apart. It is easy to prove that the total number of such points in the rectangle, including p, does not exceed 8. A more careful analysis reduces this number to 6. Thus, the algorithm can consider no more than five next points following p on the list S, before moving up to the next point.

Here is pseudocode of the algorithm. We follow the advice given in to avoid computing square roots inside the innermost loop of the algorithm.

ALGORITHM *EfficientClosestPair(P, Q)*

//Solves the closest-pair problem by divide-and-conquer
//Input: An array P of $n \geq 2$ points in the Cartesian plane sorted in nondecreasing
// order of their x coordinates and an array Q of the same points sorted in
// nondecreasing order of the y coordinates
//Output: Euclidean distance between the closest pair of points
if $n \leq 3$
    return the minimal distance found by the brute-force algorithm
else
    copy the first $\lceil n/2 \rceil$ points of P to array P_l
    copy the same $\lceil n/2 \rceil$ points from Q to array Q_l
    copy the remaining $\lfloor n/2 \rfloor$ points of P to array P_r
    copy the same $\lfloor n/2 \rfloor$ points from Q to array Q_r
    $d_l \leftarrow$*EfficientClosestPair*(P_l, Q_l)
    $d_r \leftarrow$*EfficientClosestPair*(P_r, Q_r)
    $d \leftarrow \min\{d_l, d_r\}$
    $m \leftarrow P[\lceil n/2 \rceil - 1].x$
    copy all the points of Q for which $|x - m| < d$ into array $S[0..num - 1]$
    $dminsq \leftarrow d^2$
    for $i \leftarrow 0$ **to** $num - 2$ **do**
        $k \leftarrow i + 1$
        while $k \leq num - 1$ **and** $(S[k].y - S[i].y)^2 < dminsq$
            $dminsq \leftarrow \min((S[k].x - S[i].x)^2 + (S[k].y - S[i].y)^2, dminsq)$
            $k \leftarrow k + 1$
return *sqrt(dminsq)*

The algorithm spends linear time both for dividing the problem into two problems half the size and combining the obtained solutions. Therefore, assuming as usual that n is a power of 2, we have the following recurrence for the running time of the algorithm:

$$T(n) = 2T(n/2) + f(n),$$

where $f(n) \in \Theta(n)$. Applying the Master Theorem (with $a = 2$, $b = 2$, and $d = 1$), we get $T(n) \in \Theta(n \log n)$. The necessity to presort input points does not change the overall efficiency class if sorting is done by a $O(n \log n)$ algorithm such as mergesort. In fact, this is the best efficiency

UNIT III DYNAMIC PROGRAMMING AND GREEDY TECHNIQUE

3.1 COMPUTING A BINOMIAL COEFFICIENT

Dynamic Programming Binomial Coefficients

Dynamic Programming was invented by Richard Bellman, 1950. It is a very general technique for solving optimization problems.

Dynamic Programming requires:

1. Problem divided into overlapping sub-problems
2. Sub-problem can be represented by a table
3. Principle of optimality, recursive relation between smaller and larger problems

Compared to a brute force recursive algorithm that could run exponential, the dynamic programming algorithm runs typically in quadratic time. The recursive algorithm ran in exponential time while the iterative algorithm ran in linear time.

Computing a Binomial Coefficient

Computing binomial coefficients is non optimization problem but can be solved using dynamic programming.

Binomial coefficients are represented by $C(n, k)$ =n! / (k! (n-k)!) or $\binom{n}{k}$ and can be used to represent the coefficients of a binomial:

$(a + b)^n = C(n, 0)a^n b^0 + \ldots + C(n, k)a^{n-k}b^k + \ldots + C(n, n)\, a^0 b^n$

The recursive relation is defined by the prior power

$C(n, k) = C(n\text{-}1, k\text{-}1) + C(n\text{-}1, k)$ for $n > k > 0$ with initial Condition $C(n, 0) = C(n, n) = 1$

Dynamic algorithm constructs a *nxk* table, with the first column and diagonal filled out using the Initial Condition. Construct the table:

		k					
		0	**1**	**2**	**...**	***k*-1**	***k***
	0	1					
	1	1	1				
	2	1	2	1			
n	**. . .**						
	k	1					1
	. . .						
	***n*-1**	1				$C(n\text{-}1, k\text{-}1)$	$C(n\text{-}1, k)$
	n	1					$C(n, k)$

The table is then filled out iteratively, row by row using the recursive relation.

Algorithm *Binomial*(n, k)

for $i \leftarrow 0$ **to** n **do** // fill out the table row wise

for $i = 0$ **to** $\min(i, k)$ **do**

if $j==0$ or $j==i$ **then** $C[i, j] \leftarrow 1$ // Initial Condition

else $C[i, j] \leftarrow C[i\text{-}1, j\text{-}1] + C[i\text{-}1, j]$ // recursive relation

return $C[n, k]$

The cost of the algorithm is filing out the table. Addition is the basic operation. Because $k \leq n$, the sum needs to be split into two parts because only the half the table needs to be filled out for $i < k$ and remaining part of the table is filled out across the entire row.

$A(n, k)$ = sum for upper triangle + sum for the lower rectangle

$= \sum_{i=1}^{k} \sum_{j=1}^{i-1} 1 + \sum_{i=1}^{n} \sum_{j=1}^{k} 1$

$= \sum_{i=1}^{k} (i-1) + \sum_{i=1}^{n} k$

$= (k-1)k/2 + k(n-k) \in \Theta(nk)$

Time efficiency: $\Theta(nk)$

Space efficiency: $\Theta(nk)$

Example: Relation of binomial coefficients and pascal's triangle.

A formula for computing binomial coefficients is this:

$$\binom{n}{m} = \frac{n!}{(n-m)!m!}$$

Using an identity called Pascal's Formula a recursive formulation for it looks like this:

$$\binom{n}{m} = \begin{cases} 1 & \text{if } m = 0 \\ 1 & \text{if } n = m \\ \binom{n-1}{m} + \binom{n-1}{m-1} & \text{otherwise} \end{cases}$$

This construction forms Each number in the triangle is the sum of the two numbers directly above it.

n	$\binom{n}{0}$	$\binom{n}{1}$	$\binom{n}{2}$	$\binom{n}{3}$	$\binom{n}{4}$	$\binom{n}{5}$	$\binom{n}{6}$	$\binom{n}{7}$
0	1							
1	1	1						
2	1	2	1					
3	1	3	3	1				
4	1	4	6	4	1			
5	1	5	10	10	5	1		
6	1	6	15	20	15	6	1	
7	1	7	21	35	35	21	7	1

Finding a binomial coefficient is as simple as a lookup in Pascal's Triangle.

Example: $(x+y)^7 = 1 \cdot x^7y^0 + 7 \cdot x^6y^1 + 21 \cdot x^5y^2 + 35 \cdot x^4y^3 + 35 \cdot x^3y^4 + 21 \cdot x^2y^5 + 7 \cdot x^1y^6 + 1 \cdot x^0y^7$

$= x^7 + 7x^6y + 21x^5y^2 + 35x^4y^3 + 35x^3y^4 + 21x^2y^5 + 7xy^6 + y^7$

3.2 WARSHALL'S AND FLOYD' ALGORITHM

Warshall's and Floyd's Algorithms: Warshall's algorithm for computing the transitive closure (there is a path between any two nodes) of a directed graph and Floyd's algorithm for the all-pairs shortest-paths problem. These algorithms are based on dynamic programming.

WARSHALL'S ALGORITHM (All-Pairs Path Existence Problem)

A **directed graph** (or digraph) is a graph, or set of vertices connected by edges, where the edges have a direction associated with them.

An **Adjacency matrix** A = $\{a_{ij}\}$ of a directed graph is the boolean matrix that has 1 in its ith row and jth column if and only if there is a directed edge from the ith vertex to the jth vertex.

The **transitive closure** of a directed graph with n vertices can be defined as the n x n boolean matrix T = $\{t_{ij}\}$, in which the element in the ith row and the jth column is 1 if there exists a nontrivial path (i.e., directed path of a positive length) from the ith vertex to the jth vertex; otherwise, tij is 0.

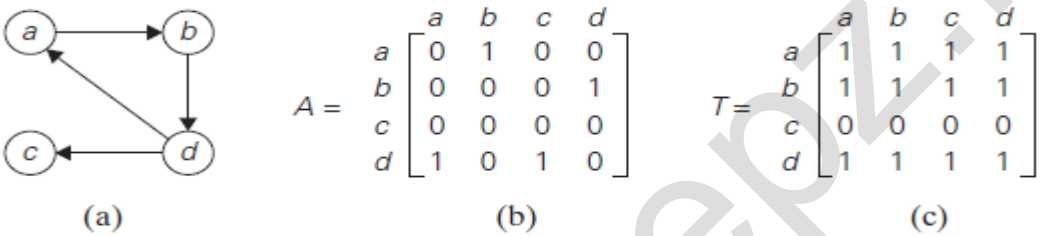


FIGURE 3.1 (a) Digraph. (b) Its adjacency matrix. (c) Its transitive closure.

The transitive closure of a digraph can be generated with the help of depth-first search or breadth-first search. Every vertex as a starting point yields the transitive closure for all.

Warshall's algorithm constructs the transitive closure through a series of n × n boolean matrices: $R^{(0)}, \ldots, R^{(k-1)}, R^{(k)}, \ldots R^{(n)}$.

The element $r_{ij}^{(k)}$ in the *i*th row and *j*th column of matrix $R^{(k)}$ ($i, j = 1, 2, \ldots, n, k = 0, 1, \ldots, n$) is equal to 1 if and only if there exists a directed path of a positive length from the *i*th vertex to the *j*th vertex with each intermediate vertex, if any, numbered not higher than *k*.

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

- The series starts with $R^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $R^{(0)}$ is nothing other than the adjacency matrix of the digraph.
- $R^{(1)}$ contains the information about paths that can use the first vertex as intermediate. it may contain more 1's than $R^{(0)}$.
- The last matrix in the series, R(n), reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.
- In general, each subsequent matrix in series has one more vertex to use as intermediate for its paths than its predecessor.
- The last matrix in the series, $R^{(n)}$, reflects paths that can use all n vertices of the digraph as intermediate and hence is nothing other than the digraph's transitive closure.

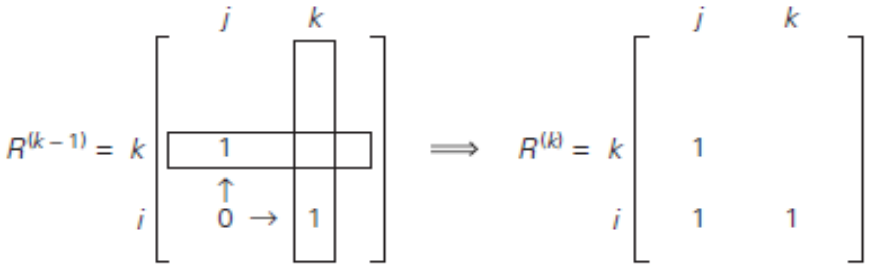


FIGURE 3.2 Rule for changing zeros in Warshall's algorithm.

All the elements of each matrix $R^{(k)}$ is computed from its immediate predecessor $R^{(k-1)}$. Let $r_{ij}^{(k)}$, the element in the ith row and jth column of matrix $R^{(k)}$, be equal to 1. This means that there exists a path from the ith vertex v_i to the jth vertex v_j with each intermediate vertex numbered not higher than k.

The first part of this representation means that there exists a path from v_i to v_k with each intermediate vertex numbered not higher than $k - 1$ (hence, $r_{ik}^{(k-1)} = 1$), and the second part means that there exists a path from v_k to v_j with each intermediate vertex numbered not higher than $k - 1$ (hence, $r_{kj}^{(k-1)} = 1$).

Thus the following formula generas the elements of matrix $R^{(k)}$ from the elements of matrix $R^{(k-1)}$:

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

ApplyingWarshall's algorithm by hand:

- If an element r_{ij} is 1 in $R^{(k-1)}$, it remains 1 in $R^{(k)}$.
- If an element r_{ij} is 0 in $R^{(k-1)}$, it has to be changed to 1 in $R^{(k)}$ if and only if the element in its row i and column k and the element in its column j and row k are both 1's in$R^{(k-1)}$.

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

//ImplementsWarshall'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←1 **to** n **do**

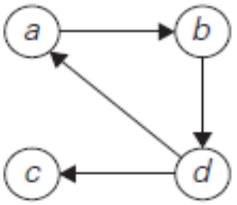
  for i ←1 **to** n **do**

    for j ←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 $R^{(n)}$

Warshall's algorithm's time efficiency is only $\Theta(n^3)$. Space efficiency is $\Theta(n^2)$. i.e matrix size.



$R^{(0)}$ =

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

1's 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

1's 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**

1's 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

1's 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

1's 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 3.3 Application of Warshall's algorithm to the digraph shown. New 1's are in bold.

FLOYD'S ALGORITHM (All-Pairs Shortest-Paths Problem)

Floyd'sl algorithm is an algorithm for finding shortest paths for all pairs in a weighted connected graph (undirected or directed) with (+/-) edge weights.

A **distance matrix** is a matrix (two-dimensional array) containing the distances, taken pairwise, between the vertices of graph.

The lengths of shortest paths in an n × n matrix D called the distance matrix: the element d_{ij} in the ith row and the jth column of this matrix indicates the length of the shortest path from the ith vertex to the jth vertex.

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

Floyd's algorithm computes the distance matrix of a weighted graph with *n* vertices through a series of $n \times n$ matrices:

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

The element $d_{ij}^{(k)}$ in the *i*th row and the *j*th column of matrix $D^{(k)}$ *(i, j* = 1, 2, . . . , *n, k* = 0, 1, . . . , *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.*

Steps to compute $D^{(0)}, \ldots, D^{(k-1)}, D^{(k)}, \ldots, D^{(n)}$

- The series starts with $D^{(0)}$, which does not allow any intermediate vertices in its paths; hence, $D^{(0)}$ is simply the weight matrix of the graph.
- As in Warshall's algorithm, we can compute all the elements of each matrix $D^{(k)}$ from its immediate predecessor $D^{(k-1)}$.
- The last matrix in the series, $D^{(n)}$, contains the lengths of the shortest paths among all paths that can use all n vertices as intermediate and hence is nothing other than the distance matrix.

Let $d_{ij}^{(k)}$ be the element in the ith row and the jth column of matrix $D^{(k)}$. This means that $d_{ij}^{(k)}$ is equal to the length of the shortest path among all paths from the ith vertex v_i to the jth vertex v_j with their intermediate vertices numbered not higher than k.

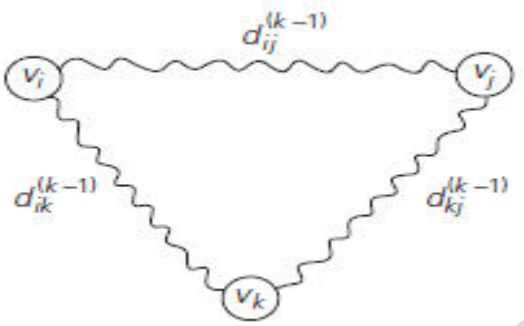


FIGURE 3.4 Underlying idea of Floyd's algorithm.

The length of the shortest path can be computed by the following recurrence:

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

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**

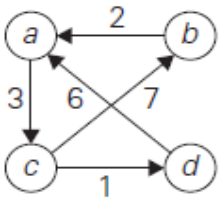
  for i ←1 **to** n **do**

    for j ←1 **to** n **do**

      D[i, j]←min{D[i, j], D[i, k]+ D[k, j]}

return D

Floyd's Algorithm's time efficiency is only $\Theta(n^3)$. Space efficiency is $\Theta(n^2)$. i.e matrix size.



$D^{(0)} =$

	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 3.5 Application of Floyd's algorithm to the digraph shown. Updated elements are shown in bold.

3.3 OPTIMAL BINARY SEARCH TREES

A binary search tree is one of the most important data structures in computer science. One of its principal applications is to implement a dictionary, a set of elements with the operations of searching, insertion, and deletion.

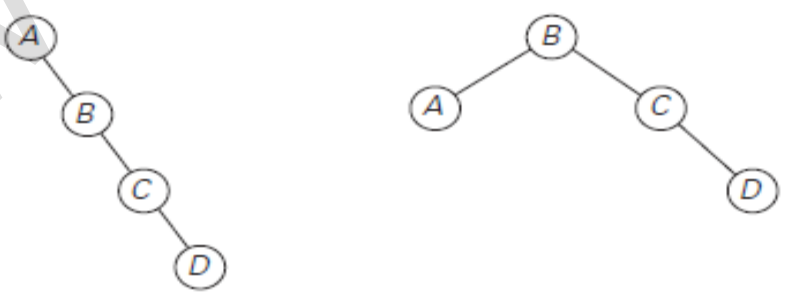


FIGURE 3.6 Two out of 14 possible binary search trees with keys *A, B, C,* and *D*.

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 3.6 depicts 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 . 1+ 0.2 . 2 + 0.4 . 3+ 0.3 . 4 = 2.9, and for the second one it is 0.1 . 2 + 0.2 . 1+ 0.4 . 2 + 0.3 . 3= 2.1. Neither of these two trees is optimal.

The total number of binary search trees with n keys is equal to the nth **Catalan number**,

$$c(n) = \frac{1}{n+1}\binom{2n}{n} \quad \text{for } n > 0, \quad c(0) = 1$$

c(n)=(2n)!/(n+1)!n!

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, $1 \le i \le j \le n$.

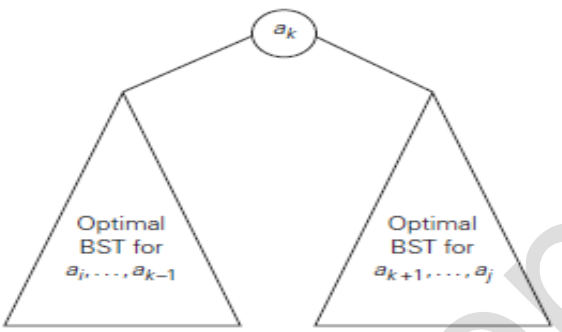


FIGURE 3.7 Binary search tree (BST) with root a_k and two optimal binary search subtrees $T_i^{\,k-1}$ and $T_{k+1}^{\,j}$.

Consider all possible ways to choose a root a_k among the keys $a_i, \ldots, a_j$. For such a binary search tree (Figure 3.7), the root contains key a_k, the left subtree $T_i^{\,k-1}$ contains keys $a_i, \ldots, a_{k-1}$ optimally arranged, and the right subtree $T_{k+1}^{\,j}$ contains keys $a_{k+1}, \ldots, a_j$ also optimally arranged.

If we count tree levels starting with 1 to make the comparison numbers equal the keys' levels, the following recurrence relation is obtained:

$$C(i, j) = \min_{i \le k \le j} \{p_k \cdot 1 + \sum_{s=i}^{k-1} p_s \cdot (\text{level of } a_s \text{ in } T_i^{k-1} + 1) + \sum_{s=k+1}^{j} p_s \cdot (\text{level of } a_s \text{ in } T_{k+1}^{j} + 1)\}$$

$$= \min_{i \le k \le j} \{\sum_{s=i}^{k-1} p_s \cdot \text{level of } a_s \text{ in } T_i^{k-1} + \sum_{s=k+1}^{j} p_s \cdot \text{level of } a_s \text{ in } T_{k+1}^{j} + \sum_{s=i}^{j} p_s\}$$

$$= \min_{i \le k \le j} \{C(i, k-1) + C(k+1, j)\} + \sum_{s=i}^{j} p_s.$$

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

We assume in above 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 $C(i, i) = p_i$ for $1 \le i \le n$, as it should be for a one-node binary search tree containing a_i.

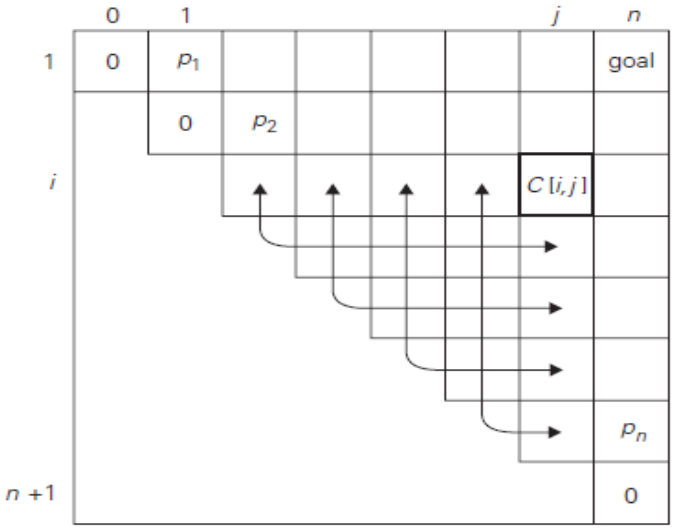


FIGURE 3.8 Table of the dynamic programming algorithm for constructing an optimal binary search tree.

The two-dimensional table in Figure 3.8 shows the values needed for computing C(i, j). They are in row i and the columns to the left of column j and in column j and the rows below row i. The arrows point to the pairs of entries whose sums are computed in order to find the smallest one to be recorded as the value of C(i, j). This suggests filling the table along its diagonals, starting with all zeros on the main diagonal and given probabilities p_i, $1 \le i \le n$, right above it and moving toward the upper right corner.

ALGORITHM OptimalBST(P [1..n])

```
//Finds an optimal binary search tree by dynamic programming
//Input: An array P[1..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

The algorithm's space efficiency is clearly quadratic, ie, **:** $\Theta(n^3)$; the time efficiency of this version of the algorithm is cubic. It is Possible to reduce the running time of the algorithm to $\Theta(n^2)$ by taking advantage of monotonicity of entries in the root table, i.e., R[i,j] is always in the range between R[i,j-1] and R[i+1,j]

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 are:

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 $C(1, 2)$:

$$C(1, 2) = \min \left\{ \begin{array}{ll} k = 1: & C(1, 0) + C(2, 2) + \sum_{s=1}^{2} p_s = 0 + 0.2 + 0.3 = 0.5 \\ k = 2: & C(1, 1) + C(3, 2) + \sum_{s=1}^{2} p_s = 0.1 + 0 + 0.3 = 0.4 \end{array} \right\}$$
$$= 0.4.$$

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.

We arrive at the following final tables:

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 (and the root of the one node tree: R(1, 1) = 1). Since R(4, 4) = 4, the root of this one-node optimal tree is its only key D. Figure 3.10 presents the optimal tree in its entirety.

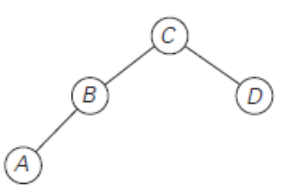


FIGURE 3.10 Optimal binary search tree for the above example.

3.4 KNAPSACK PROBLEM AND MEMORY FUNCTIONS

Designing a dynamic programming algorithm for the knapsack problem:

Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of capacity W, find the most valuable subset of the items that fit into the knapsack.

Assume that all the weights and the knapsack capacity are positive integers; the item values do not have to be integers.

0 / 1 knapsack problem means, the chosen item should be either null or whole.

Recurrence relation that expresses a solution to an instance of the knapsack problem

Let us consider an instance defined by the first i items, $1 \le i \le n$, with weights $w_1, \ldots, w_i$, values $v_1, \ldots, v_i$, and knapsack capacity j, $1 \le j \le W$. Let $F(i, j)$ be the value of an optimal solution to this instance, i.e., the value of the most valuable subset of the first i items that fit into the knapsack of capacity j. We can divide all the subsets of the first i items that fit the knapsack of capacity j into two categories: those that do not include the ith item and those that do. Note the following:

1. Among the subsets that do not include the ith item, the value of an optimal subset is, by definition, $F(i-1, j)$.
2. Among the subsets that do include the ith item (hence, $j - w_i \ge 0$), an optimal subset is made up of this item and an optimal subset of the first $i-1$ items that fits into the knapsack of capacity $j - w_i$. The value of such an optimal subset is $v_i + F(i-1, j-w_i)$.

Thus, the value of an optimal solution among all feasible subsets of the first I items is the maximum of these two values. Of course, if the ith item does not fit into the knapsack, the value of an optimal subset selected from the first i items is the same as the value of an optimal subset selected from the first $i-1$ items. These observations lead to the following recurrence:

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

It is convenient to define the initial conditions as follows:

$F(0, j) = 0$ for $j \ge 0$ and $F(i, 0) = 0$ for $i \ge 0$.

Our goal is to find $F(n, W)$, the maximal value of a subset of the n given items that fit into the knapsack of capacity W, and an optimal subset itself.

*For F(i, j),*compute the maximum of the entry in the previous row and the same column and the sum of v_i and the entry in the previous row and w_i columns to the left. The table can be filled either row by row or column by column.

ALGORITHM DPKnapsack($w[1..n]$, $v[1..n]$, W)

var $V[0..n,0..W]$, $P[1..n,1..W]$: int

for $j := 0$ to W **do**

$V[0,j] := 0$

for $i := 0$ to n **do**

$V[i,0] := 0$

for $i := 1$ to n **do**

for $j := 1$ to W **do**

if $w[i] \le j$ and $v[i] + V[i-1,j-w[i]] > V[i-1,j]$ then
$V[i,j] := v[i] + V[i-1,j-w[i]]$; $P[i,j] := j-w[i]$
else
$V[i,j] := V[i-1,j]$; $P[i,j] := j$
return $V[n,W]$ and the optimal subset by backtracing

Note: Running time and space: O(nW).

Table3.1 for solving the knapsack problem by dynamic programming.

		0	$j-w_i$	j	W
	0	0	0	0	0
	$i-1$	0	$F(i-1, j-w_i)$	$F(i-1, j)$	
w_i, v_i	i	0		$F(i, j)$	
	n	0			goal

EXAMPLE 1 Let us consider the instance given by the following data:
Table 3.2 An instance of the knapsack problem:

item	**weight**	**value**	capacity
1	2	$12	
2	1	$10	$W = 5$
3	3	$20	
4	2	$15	

The maximal value is $F(4, 5) = \$37$. We can find the composition of an optimal subset by **backtracing** (Back tracing finds the actual optimal subset, i.e. solution), the computations of this entry in the table. Since $F(4, 5) > F(3, 5)$, item 4 has to be included in an optimal solution along with an optimal subset for filling $5 - 2 = 3$ remaining units of the knapsack capacity. The value of the latter is $F(3, 3)$. Since $F(3, 3) = F(2, 3)$, item 3 need not be in an optimal subset. Since $F(2, 3) > F(1, 3)$, item 2 is a part of an optimal selection, which leaves element $F(1, 3 - 1)$ to specify its remaining composition. Similarly, since $F(1, 2) > F(0, 2)$, item 1 is the final part of the optimal solution {item 1, item 2, item 4}.

Table 3.3 Solving an instance of the knapsack problem by the dynamic programming algorithm.

	Capacity j						
	i	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w1 = 2$, $v1 = 12$	1	0	0	12	12	12	12
$w2 = 1$, $v2 = 10$	2	0	10	12	22	22	22
$w3 = 3$, $v3 = 20$	3	0	10	12	22	30	32
$w4 = 2$, $v4 = 15$	4	0	10	15	25	30	**37**

Memory Functions

The direct top-down approach to finding a solution to such a recurrence leads to an algorithm that solves common subproblems more than once and hence is very inefficient.

The bottom up fills a table with solutions to all smaller subproblems, but each of them is solved only once. An unsatisfying aspect of this approach is that solutions to some of these smaller subproblems are often not necessary for getting a solution to the problem given.

Since this drawback is not present in the top-down approach, it is natural to try to combine the strengths of the top-down and bottom-up approaches. The goal is to get a method that solves only subproblems that are necessary and does so only once. Such a method exists; it is based on using **memory functions.**

This method solves a given problem in the top-down manner but, in addition, maintains a table of the kind that would have been used by a bottom-up dynamic programming algorithm.

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 knapsack capacity).

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 capacity
//Output: The value of an optimal feasible subset of the first i items
//Note: Uses as global variables input arrays Weights [1..n], Values[1..n],
//        and table F[0..n, 0..W ] whose entries are initialized with −1's except for
//        row 0 and column 0 initialized with 0's
if F[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]))
        F[i, j ]←value
return F[i, j ]
```

EXAMPLE 2 Let us apply the memory function method to the instance considered in Example 1.

	Capacity j						
	I	0	1	2	3	4	5
	0	0	0	0	0	0	0
$w1 = 2, v1 = 12$	1	0	0	12	12	12	12
$w2 = 1, v2 = 10$	2	0	-	12	22	-	22
$w3 = 3, v3 = 20$	3	0	-	-	22	-	32
$w4 = 2, v4 = 15$	4	0	-	-	-	-	37

Only 11 out of 20 nontrivial values (i.e., not those in row 0 or in column 0) have been computed. Just one nontrivial entry, V (1, 2), is retrieved rather than being recomputed. For larger instances, the proportion of such entries can be significantly larger.

3.5 GREEDY TECHNIQUE

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:

- *feasible*, i.e., it has to satisfy the problem's constraints
- *locally optimal*, i.e., it has to be the best local choice among all feasible choices available on that step
- *irrevocable*, i.e., once made, it cannot be changed on subsequent steps of the algorithm

Greedy Technique algorithms are:

- Prim's algorithm
- Kruskal's Algorithm
- Dijkstra's Algorithm
- Huffman Trees

Two classic algorithms for the minimum spanning tree problem: Prim's algorithm and Kruskal's algorithm. They solve the same problem by applying the greedy approach in two different ways, and both of them always yield an optimal solution.

Another classic algorithm named Dijkstra's algorithm used to find the shortest-path in a weighted graph problem solved by Greedy Technique . Huffman codes is an important data compression method that can be interpreted as an application of the greedy technique.

The first way is one of the common ways to do the proof for Greedy Technique is by **mathematical induction**.

The second way to prove optimality of a greedy algorithm is to show that on each step it does at least as well as any other algorithm could in **advancing** toward the problem's goal.

Example: find the minimum number of moves needed for a chess knight to go from one corner of a 100 × 100 board to the diagonally opposite corner. (The knight's moves are L-shaped jumps: two squares horizontally or vertically followed by one square in the perpendicular direction.)

A greedy solution is clear here: jump as close to the goal as possible on each move. Thus, if its start and finish squares are (1,1) and (100, 100), respectively, a sequence of 66 moves such as (1, 1) − (3, 2) − (4, 4) − . . . − (97, 97) − (99, 98) − (100, 100) solves the problem(The number k of two-move advances can be obtained from the equation 1+ 3k = 100).

Why is this a minimum-move solution? Because if we measure the distance to the goal by the Manhattan distance, which is the sum of the difference between the row numbers and the difference between the column numbers of two squares in question, the greedy algorithm decreases it by 3 on each move.

The third way is simply to show that the final result obtained by a greedy algorithm is optimal based on the **algorithm's output** rather than the way it operates.

Example: Consider the problem of placing the maximum number of chips on an 8 × 8 board so that no two chips are placed on the same or adjacent vertically, horizontally, or diagonally.

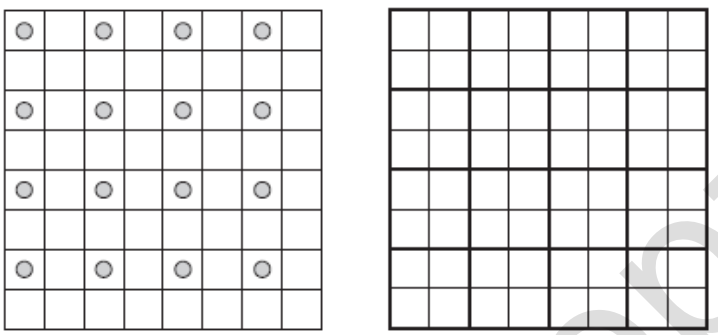


FIGURE 3.12 (a) Placement of 16 chips on non-adjacent squares. (b) Partition of the board proving impossibility of placing more than 16 chips.

It is impossible to place more than one chip in each of these squares, which implies that the total number of nonadjacent chips on the board cannot exceed 16.

3.6 PRIM'S ALGORITHM

A ***spanning tree*** of an undirected connected graph is its connected acyclic subgraph (i.e., a tree) that contains all the vertices of the graph. If such a graph has weights assigned to its edges, a ***minimum spanning tree*** 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.

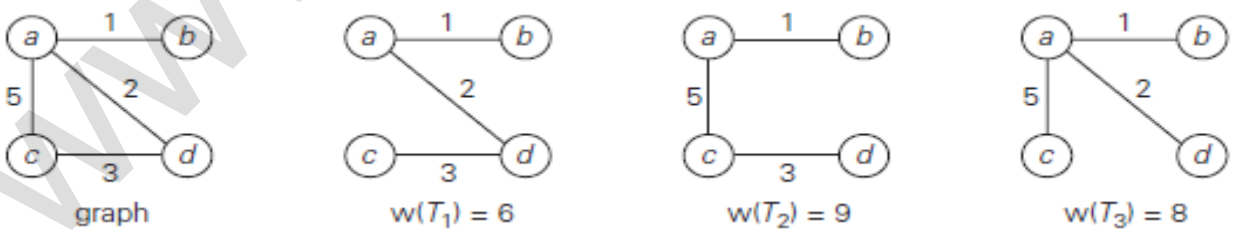


FIGURE 3.13 Graph and its spanning trees, with *T*1 being the minimum spanning tree.

The minimum spanning tree is illustrated in Figure 3. If we were to try constructing a minimum spanning tree by exhaustive search, we would face two serious obstacles. First, the number of spanning trees grows exponentially with the graph size (at least for dense graphs). Second, generating all spanning trees for a given graph is not easy; in fact, it is more difficult than finding a minimum spanning tree for a weighted graph.

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, the algorithm expands the current tree in the greedy manner by simply attaching to it the nearest vertex not in that tree. The algorithm stops after all the graph's vertices have been included in the tree being constructed.

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 $i \leftarrow 1$ **to** $|V| - 1$ **do**

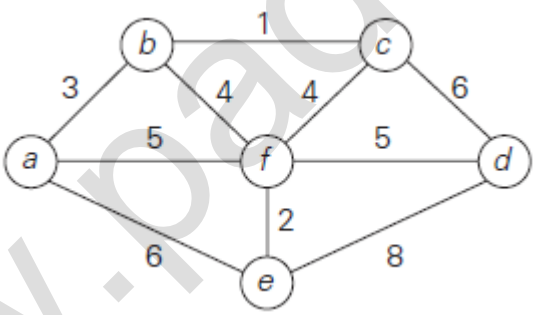
find a minimum-weight edge $e^* = (v^*, u^*)$ among all the edges (v, u) such that v is in V_T and u is in $V - V_T$

$V_T \leftarrow V_T \cup \{u^*\}$

$E_T \leftarrow E_T \cup \{e^*\}$

return E_T

If a graph is represented by its adjacency lists and the priority queue is implemented as a min-heap, the running time of the algorithm is $O(|E| \log |V|)$ in a connected graph, where $|V| - 1 \leq |E|$.



 www.padeepz.net

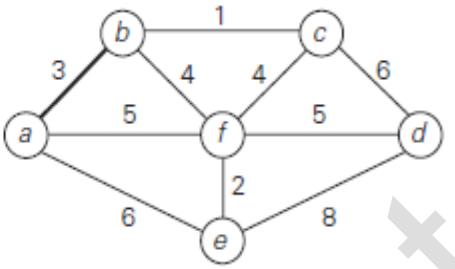
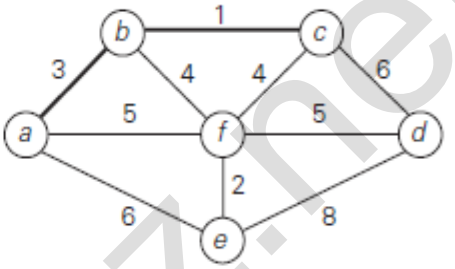
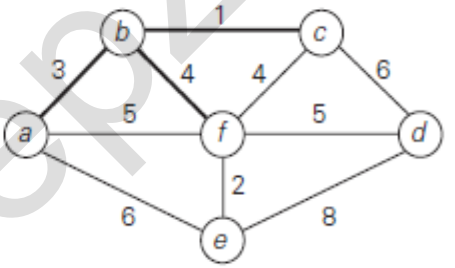
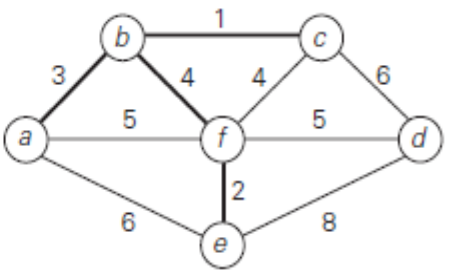
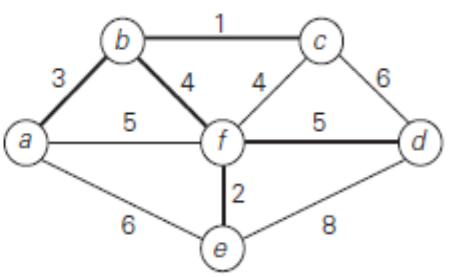
Tree vertices	Remaining vertices	Illustration
a(−, −)	**b(a, 3)** c(−, ∞) d(−, ∞) e(a, 6) f(a, 5)	
b(a, 3)	**c(b, 1)** d(−, ∞) e(a, 6) f(b, 4)	
c(b, 1)	d(c, 6) e(a, 6) **f(b, 4)**	
f(b, 4)	d(f, 5) **e(f, 2)**	
e(f, 2)	**d(f, 5)**	
d(f, 5)		

FIGURE 3.14 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 in bold.

 www.padeepz.net

3.7 KRUSKAL'S ALGORITHM

Kruskal's algorithm looks at a minimum spanning tree of 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 constructs a minimum spanning tree as an expanding sequence of subgraphs that are always acyclic but are not necessarily connected on the intermediate stages of the algorithm.

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 subgraph if such an inclusion does not create a cycle and simply skipping the edge otherwise.

Kruskal's algorithm looks at a minimum spanning tree of 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.

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 $w(e_{i1}) \le \ldots \le w(e_{i\,|E|})$

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

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

while $ecounter < |V| - 1$ **do**

$k \leftarrow k + 1$

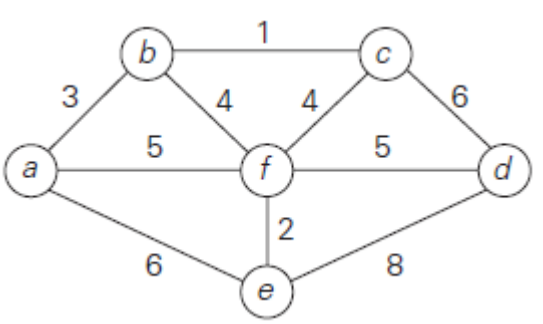
if $E_T \cup \{e_{ik}\}$ is acyclic

$E_T \leftarrow E_T \cup \{e_{ik}\}$; $ecounter \leftarrow ecounter + 1$

return E_T

The initial forest consists of |V | trivial trees, each comprising a single vertex of the graph. The final forest consists of a single tree, which is a minimum spanning tree of the graph. On each iteration, the algorithm takes the next edge (u, v) from the sorted list of the graph's edges, finds the trees containing the vertices u and v, and, if these trees are not the same, unites them in a larger tree by adding the edge (u, v).

Fortunately, there are efficient algorithms for doing so, including the crucial check for whether two vertices belong to the same tree. They are called union-find algorithms. With an efficient union-find algorithm, the running time of Kruskal's algorithm will be $O(|E| \log |E|)$.



 www.padeepz.net

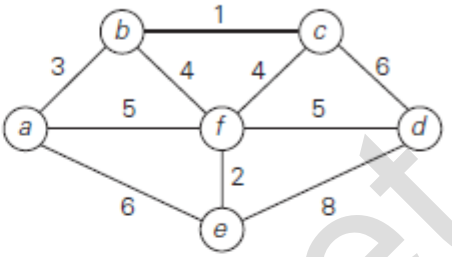
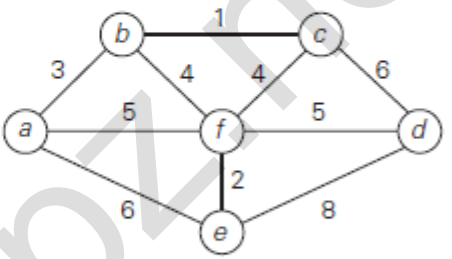
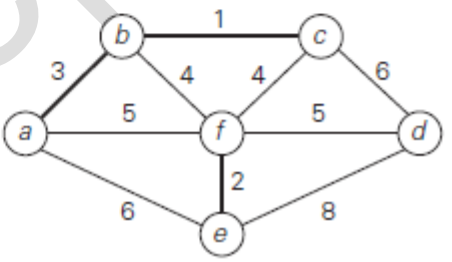
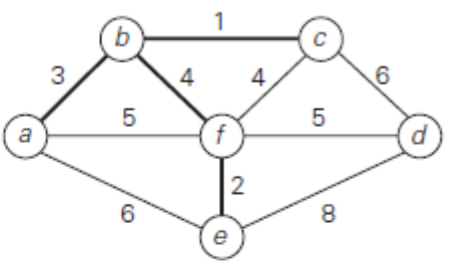
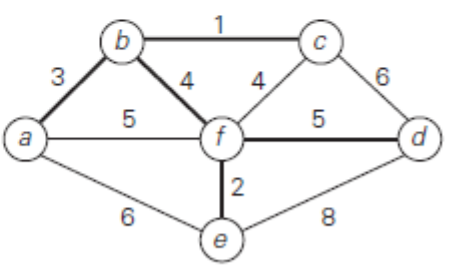
Tree edges	Sorted list of edges	Illustration
	bc ef ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bc 1	bc **ef** ab bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ef 2	bc ef **ab** bf cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
ab 3	bc ef ab **bf** cf af df ae cd de 1 2 3 4 4 5 5 6 6 8	
bf 4	bc ef ab bf cf af **df**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

FIGURE 3.15 Application of Kruskal's algorithm. Selected edges are shown in bold.

 www.padeepz.net

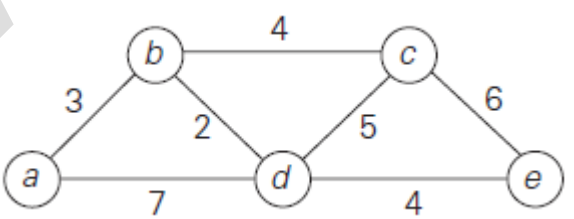
3.8 DIJKSTRA'S ALGORITHM

- Dijkstra's Algorithm solves 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.
- The single-source shortest-paths problem asks for a family of paths, each leading from the source to a different vertex in the graph, though some paths may, of course, have **edges in common**.
- The most widely used **applications** are transportation planning and packet routing in communication networks including the Internet.
- It also includes **finding shortest paths** in social networks, speech recognition, document formatting, robotics, compilers, and airline crew scheduling.
- In the world of **entertainment**, one can mention pathfinding in video games and finding best solutions to puzzles using their state-space graphs.
- Dijkstra's algorithm is the best-known algorithm for the single-source shortest-paths problem.

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 dv of a shortest path from s to v and its penultimate vertex pv for every
// vertex v in V
Initialize(Q) //initialize priority queue to empty
for every vertex v in V
    $dv \leftarrow \infty$; $pv \leftarrow$ **null**
    Insert (Q, v, dv) //initialize vertex priority in the priority queue
$Ds \leftarrow 0$; *Decrease(Q, s, d_s)* //update priority of s with d_s
$V_T \leftarrow \Phi$
for $i \leftarrow 0$ **to** $|V| - 1$ **do**
    $u^* \leftarrow$ *DeleteMin(Q)* //delete the minimum priority element
    $V_T \leftarrow V_T \cup \{u^*\}$
    for every vertex u in $V - VT$ that is adjacent to u^* **do**
        if $d_u^* + w(u^*, u) < d_u$
        $d_u \leftarrow d_u^* + w(u^*, u)$; $p_u \leftarrow u^*$
        Decrease(Q, u, d_u)

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 in Θ ($|V|^2$) for graphs represented by their weight matrix and the priority queue implemented as an unordered array. For graphs represented by their adjacency lists and the priority queue implemented as a min-heap, it is in O(|E| log |V |).



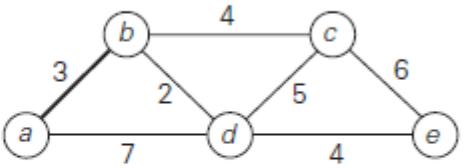
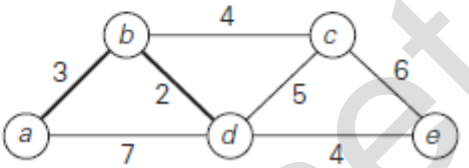
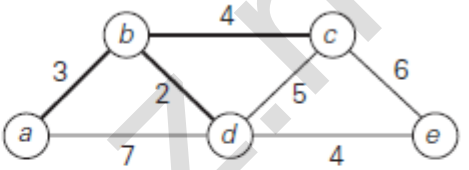
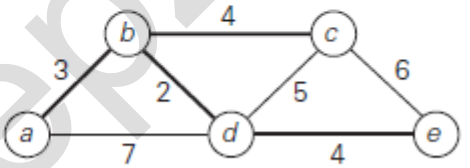
Tree vertices	Remaining vertices	Illustration
a(−, 0)	**b(a, 3)** c(−, ∞) d(a, 7) e(−, ∞)	
b(a, 3)	c(b, 3 + 4) **d(b, 3 + 2)** e(−, ∞)	
d(b, 5)	**c(b, 7)** e(d, 5 + 4)	
c(b, 7)	**e(d, 9)**	
e(d, 9)		

FIGURE 3.16 Application of Dijkstra's algorithm. The next closest vertex is shown in bold

The shortest paths (identified by following nonnumeric labels backward from a destination vertex in the left column to the source) and their lengths (given by numeric labels of the tree vertices) are as follows:

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

3.9 HUFFMAN TREES

To encode a text that comprises symbols from some *n*-symbol alphabet by assigning to each of the text's symbols some sequence of bits called the ***codeword***. For example, we can use a ***fixed-length encoding*** that assigns to each symbol a bit string of the same length *m* ($m \geq \log 2\ n$). This is exactly what the standard ASCII code does.

Variable-length encoding, which assigns codewords of different lengths to different symbols, introduces a problem that fixed-length encoding does not have. Namely, how can we tell how many bits of an encoded text represent the first (or, more generally, the *i*th) symbol? To avoid this complication, we can limit urselvesto the so-called ***prefix-free*** (or simply ***prefix***) ***codes***.

In a prefix code, no codeword is a prefix of a codeword of another symbol. Hence, with such an encoding, we can simply scan a bit string until we get the first group of bits that is a codeword for some symbol, replace these bits by this symbol, and repeat this operation until the bit string's end is reached.

Huffman's algorithm

Step 1 Initialize n one-node trees and label them with the symbols of the alphabet given. Record the frequency of each symbol in its tree's root to indicate the tree's ***weight***. (More generally, the weight of a tree will be equal to the sum of the frequencies in the tree's leaves.)

Step 2 Repeat the following operation until a single tree is obtained. Find two trees with the smallest weight (ties can be broken arbitrarily, but see Problem 2 in this section's exercises). Make them the left and right subtree of a new tree and record the sum of their weights in the root of the new tree as its weight.

A tree constructed by the above algorithm is called a **Huffman tree**. It defines in the manner described above is called a **Huffman code**.

EXAMPLE Consider the five-symbol alphabet {A, B, C, D, _} with the following occurrence frequencies in a text made up of these symbols:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15

The Huffman tree construction for this input is shown in Figure 3.18

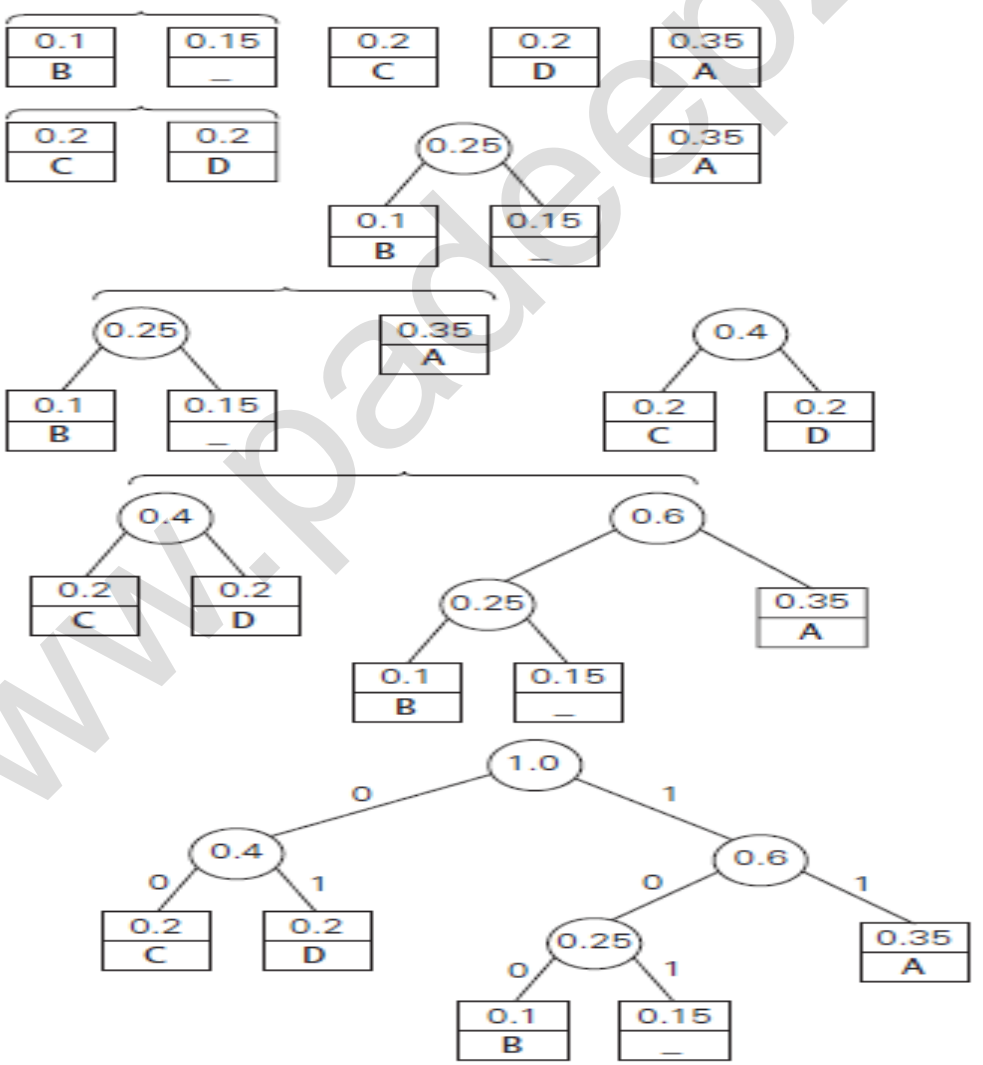


FIGURE 3.18 Example of constructing a Huffman coding tree.

The resulting codewords are as follows:

symbol	A	B	C	D	_
frequency	0.35	0.1	0.2	0.2	0.15
codeword	11	100	00	01	101

Hence, DAD is encoded as 011101, and 10011011011101 is decoded as BAD_AD. With the occurrence frequencies given and the codeword lengths obtained, the average number of bits per symbol in this code is 2 . 0.35 + 3 . 0.1+ 2 . 0.2 + 2 . 0.2 + 3 . 0.15 = 2.25.

We used a fixed-length encoding for the same alphabet, we would have to use at least 3 bits per each symbol. Thus, for this toy example, Huffman's code achieves the ***compression ratio -*** a standard measure of a compression algorithm's effectiveness of $(3- 2.25) / 3 \cdot 100\% =$ **25%**. In other words, Huffman's encoding of the text will use 25% less memory than its fixed-length encoding.

Running time is O(*n log n*), as each priority queue operation takes time O(*log n*).

Applications of Huffman's encoding

1. Huffman's encoding is a variable length encoding, so that number of bits used are lesser than fixed length encoding.
2. Huffman's encoding is very useful for file compression.
3. Huffman's code is used in transmission of data in an encoded format.
4. Huffman's encoding is used in decision trees and game playing.

UNIT IV ITERATIVE IMPROVEMENT

4.1 THE SIMPLEX METHOD

Linear Programming

Linear programming problem (LPP) is to optimize a linear function of several variables subject to linear constraints:

maximize (or minimize) $c_1 x_1 + ... + c_n x_n$

subject to $a_{i\,1} x_1 + ... + a_{in} x_n \leq (\text{or} \geq \text{or} =)\ b_i,\ i = 1,...,m$ $x_1 \geq 0, ... , x_n \geq 0$

The function $z = c_1 x_1 + ... + c_n x_n$ is called the *objective function*;

constraints $x_1 \geq 0, ... , x_n \geq 0$ are called *nonnegativity constraints*

Example

maximize $3x + 5y$

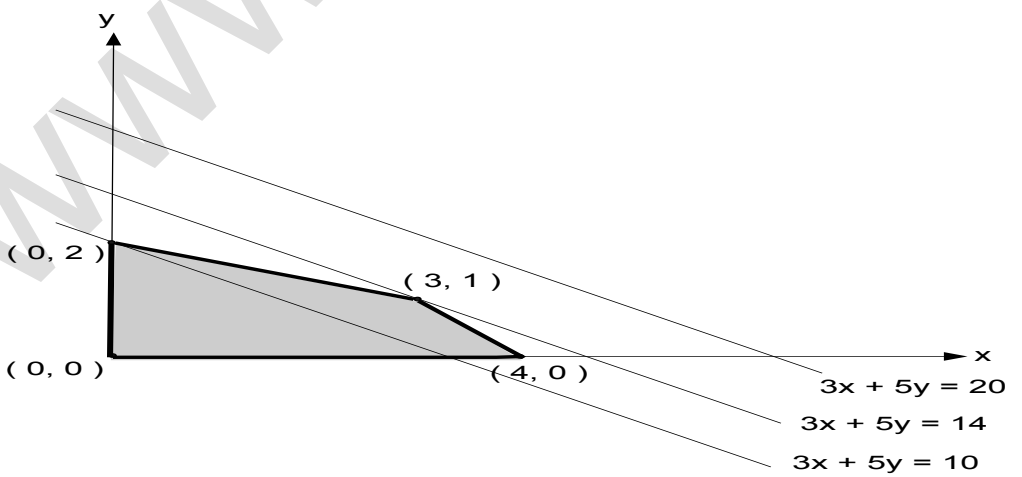
subject to $x + y \leq 4$

$x + 3y \leq 6$

$x \geq 0,\ y \geq 0$

Feasible region is the set of points defined by the constraints

Geometric solution



Optimal solution: x = 3, y = 1

Extreme Point Theorem Any LP problem with a nonempty bounded feasible region has an optimal solution; moreover, an optimal solution can always be found at an *extreme point* of the problem's feasible region.

Three possible outcomes in solving an LP problem

- has a finite optimal solution, which may not be unique
- *unbounded*: the objective function of maximization (minimization) LP problem is unbounded from above (below) on its feasible region
- *infeasible*: there are no points satisfying all the constraints, i.e. the constraints are contradictory

The Simplex Method

- The classic method for solving LP problems; one of the most important algorithms ever invented.
- Invented by George Dantzig in 1947.
- Based on the iterative improvement idea.
- Generates a sequence of adjacent points of the problem's feasible region with improving values of the objective function until no further improvement is possible.

Standard form of LP problem

- Must be a **maximization** problem
- All constraints (except the nonnegativity constraints) must be in the form of linear equations
- All the variables must be required to be nonnegative
- Thus, the general linear programming problem in standard form with m constraints and n unknowns ($n \geq m$) is
- Maximize $c_1 x_1 + \ldots + c_n x_n$
- Subject to $a_{i\,1} x_1 + \ldots + a_{in} x_n = b_i$, $i = 1,\ldots,m$, $x_1 \geq 0, \ldots, x_n \geq 0$

Example

maximize $3x + 5y$

subject to $x + y \leq 4$

$x + 3y \leq 6$

$x \geq 0,\ y \geq 0$

maximize $3x + 5y + 0u + 0v$

subject to $x + y + u = 4$

$x + 3y + v = 6$

$x \geq 0,\ y \geq 0,\ u \geq 0,\ v \geq 0$

Variables u and v, transforming inequality constraints into equality constrains, are called *slack variables*

Basic feasible solutions

A *basic solution* to a system of m linear equations in n unknowns ($n \geq m$) is obtained by setting $n - m$ variables to 0 and solving the resulting system to get the values of the other m variables. The variables set to 0 are called *nonbasic*; the variables obtained by solving the system are called *basic*.

A basic solution is called *feasible* if all its (basic) variables are nonnegative.

Example $x + y + u = 4$

$x + 3y + v = 6$

(0, 0, 4, 6) is basic feasible solution

(x, y are nonbasic; u, v are basic)

There is a 1-1 correspondence between extreme points of LP's feasible region and its basic feasible solutions.

Simplex Tableau

maximize $z = 3x + 5y + 0u + 0v$

subject to $x + y + u = 4$

$x + 3y + v = 6$

$x \geq 0,\ y \geq 0,\ u \geq 0,\ v \geq 0$

	x	y	u	v	
u	1	1	1	0	4
v	1	3	0	1	6
objective row	−3	−5	0	0	0

basic variables = u,v

basic feasible solution = (0, 0, 4, 6)

value of z at (0, 0, 4, 6) =0

Outline of the Simplex Method

Step 0 [Initialization] Present a given LP problem in standard form and set up initial tableau.

Step 1 [Optimality test] If all entries in the objective row are nonnegative then stop: the tableau represents an optimal solution.

Step 2 [Find entering variable] Select the most negative entry in the objective row. Mark its column to indicate the entering variable and the **pivot column**.

Step 3 [Find departing (leaving) variable] For each positive entry in the pivot column, calculate the θ-ratio by dividing that row's entry in the rightmost column (solution) by its entry in the pivot column. (If there are no positive entries in the pivot column then stop: the problem is unbounded.) Find the row with the smallest θ-ratio, mark this row to indicate the departing variable and the **pivot row.**

Step 4 [Form the next tableau] Divide all the entries in the pivot row by its entry in the pivot column. Subtract from each of the other rows, including the objective row, the new pivot row multiplied by the entry in the pivot column of the row in question. Replace the label of the pivot row by the variable's name of the pivot column and go back to Step 1.

Example of Simplex Method Application

	x	y	u	v	
u	1	1	1	0	4
← v	1	3	0	1	6
	−3	−5	0	0	0
	↑				

basic feasible sol. (0, 0, 4, 6) $z = 0$

	x	y	u	v	
← u	$\frac{2}{3}$	0	1	$-\frac{1}{3}$	2
y	$\frac{1}{3}$	1	0	$\frac{1}{3}$	2
	$-\frac{4}{3}$	0	0	$\frac{5}{3}$	10
	↑				

basic feasible sol. (0, 2, 2, 0) $z = 10$

	x	y	u	v	
x	1	0	3/2	−1/3	3
y	0	1	−1/2	1/2	1
	0	0	2	1	14

basic feasible sol. (3, 1, 0, 0) $z = 14$

Notes on the Simplex Method

- Finding an initial basic feasible solution may pose a problem.
- Theoretical possibility of cycling.
- Typical number of iterations is between m and 3m, where m is the number of equality constraints in the standard form.
- Worse-case efficiency is **exponentia**l.
- More recent interior-point algorithms such as Karmarkar's algorithm (1984) have polynomial worst-case efficiency and have performed competitively with the simplex method in empirical tests.

Example 1:

Use Simplex method to solve the formers problem given below.

A farmer has a 320 acre farm on which she plants two crops: corn and soybeans. For each acre of corn planted, her expenses are $50 and for each acre of soybeans planted, her expenses are $100. Each acre of corn requires 100 bushels of storage and yields a profit of $60; each acre of

soybeans requires 40 bushels of storage and yields a profit of $90. If the total amount of storage space available is 19,200 bushels and the farmer has only $20,000 on hand, how many acres of each crop should she plant in order to maximize her profit? What will her profit be if she follows this strategy?

Solution

Linear Programming Problem Formulation

	Corn	Soybean	**Total**
Expenses	$50	$100	$20,000
Storage(bushels)	100	40	19,200
Profit	60	90	Maximize profit

A farmer has a 320 acre farm is unwanted data but c+s<=320.

c = corn planted acres and s = soybean planted acres

$50c + 100s \leq 20{,}000$

$100c + 40s \leq 19{,}200$

Maximize: $60c + 90s = P$

Canonical form of LPP

Maximize: $60c + 90s$

Subject to $50c + 100s = 20000$

$100c + 40s = 19200$

$c \geq 0, s \geq 0$

Solving by algebra (Intersection of lines)

Maximize: $60c + 90s$

Subject to $50c + 100s = 20000$ (1)

$100c + 40s = 19200$ (2)

(1)/50 => $c + 2s = 400$

(2)/20 => $5c + 2s = 960$

(2) – (1) => $4c = 560$

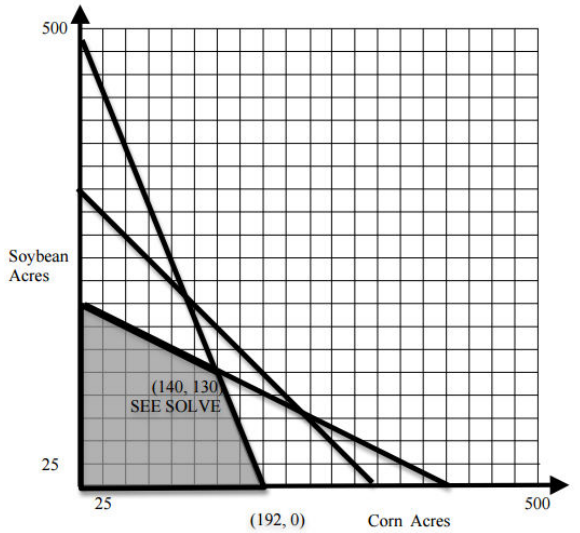
$c = 140$

Substitute c = 140 in (1) then s = 130

Profit: $p = 60c + 90s = 60(140) + 90(130) = \$20{,}100$

She should plant 140 acres corn and 130 acres of soybean for $20,100.

Solving by Graphical method



Profit at (0, 200) = 60c + 90s = 60(0) +90(200) = $18,000

Profit at (192, 0) = 60c + 90s = 60(192) +90(0) = $11,520

Profit at (140, 130) = 60c + 90s = 60(140) +90(130) = $20,100

She should plant 140 acres corn and 130 acres of soybean for $20,100.

Solving by Simplex method

Canonical form of LPP

Maximize: 60x + 90y

Subject to $50x + 100y + s_1 = 20000$

$100x + 40y + s_2 = 19200$

$x \geq 0, y \geq 0$

Iteration I

Basic	z	x	y	s_1	s_2	Solution
s_1	0	50	**100**	1	0	**20000**
s_2	0	100	40	0	1	19200
z	1	-60	**-90**	0	0	0

CPR

Select least ratio ɵ

Solution/pivot elements

20000/100 =200 √

Select the most negative value in row z.

Pivot element : Intersection of pivot row and pivot column: 100

Basic variables : s_1, s_2, z

Non Basic variables : x, y

Enter variable : y

Leave variable : s_1

Initial solution at $(x, y, s_1, s_2) = (0, 0, 20000, 19200)$

Initial solution z = 0

Pivot row:

Replace the **leaving variable** in basic column with the **entering variable**.

New Pivot Row = Current Pivot Row / Pivot Element

All other rows including z:

New Row = Current Row – (Its Pivot column coefficient)* New Pivot Row

Row y

New Pivot Row = Current Pivot Row / Pivot Element

= (0, 50, 100, 1, 0, 20000) / 100

= $(0, \frac{1}{2}, 1, \frac{1}{100}, 0, 200)$

Row s_2

New Row = Current Row – (Its Pivot column coefficient)* New Pivot Row

= (0, 100, 40, 0, 1, 19200) - (40)*($0, \frac{1}{2}, 1, \frac{1}{100}, 0, 200$)

= $(0, 80, 0, \frac{-4}{10}, 1, 96)$

Row z

New Row = Current Row – (Its Pivot column coefficient)* New Pivot Row

= (1, -60, -90, 0, 0, 0) - (-90)*($0, \frac{1}{2}, 1, \frac{1}{100}, 0, 200$)

= (1, -60, -90, 0, 0, 0) + (90)*($0, \frac{1}{2}, 1, \frac{1}{100}, 0, 200$)

= $(1, -15, 0, \frac{9}{10}, 0, 18000)$

Iteration II

Basic	z	x	y	s_1	s_2	Solution
y	0	1/2	1	1/100	0	200
s_2	0	80	0	-4/10	1	96
z	1	-15	0	9/10	0	18000

NPR

Basic	z	x	y	s_1	s_2	Solution
y	0	1/2	1	1/100	0	200
s_2	0	**80**	0	-4/10	1	11200
z	1	-15	0	9/10	0	18000

CPR

Select least ratio θ

Solution/pivot elements

200/(1/2) =400

Select the most negative value in row z.

Pivot element : Intersection of pivot row and pivot column: 80

Basic variables : y, s_2, z

Non Basic variables : x, s_1

Enter variable : x

Leave variable : s_2

Second solution at (x, y, s_1, s_2) = (0, 200, 0, 11200)

Second solution z = 18000 (Improved solution)

Row x

New Pivot Row = Current Pivot Row / Pivot Element

= $(0, 80, 0, \frac{-4}{10}, 1, 11200) / 80$

= $(0, 1, 0, \frac{-1}{200}, \frac{1}{80}, 140)$

Row y

New Row = Current Row – (Its Pivot column coefficient)* New Pivot Row

= (0, 1/2, 1, 1/100, 0, 200) - $(\frac{1}{2})$*(0, 1, 0, $\frac{-1}{200}$, $\frac{1}{80}$, 140)

= (0, 0, 1, $\frac{1}{80}$, $\frac{-1}{160}$, 130)

Row z

New Row = Current Row – (Its Pivot column coefficient)* New Pivot Row

= (1, -15, 0, $\frac{9}{10}$, 0, 18000) - (-15)*(0, 1, 0, $\frac{-1}{200}$, $\frac{1}{80}$, 140)

= (1, -15, 0, $\frac{9}{10}$, 0, 18000) + (15)*(0, 1, 0, $\frac{-1}{200}$, $\frac{1}{80}$, 140)

= (1, 0, 0, $\frac{33}{40}$, $\frac{15}{80}$, 20100)

Iteration III

Basic	z	x	y	s_1	s_2	Solution
y	0	0	1	1/80	-1/160	130
x	0	1	0	-1/200	1/80	140
z	1	0	0	33/40	15/80	20100

The above table has no negative values in row z.

Therefore, the above table is optimum table.

Profit at (140, 130) = 60c + 90s = 60(140) +90(130) = $20,100

Final solution at (x, y, s_1, s_2) = (130, 140, 0, 0)

Final solution z = $20,100 (Optimized solution)

Primal to Dual conversion (Dual to Primal)

[Primal = Dual of Dual]

Primal

Maximize

$$z = \sum_{j=1}^{n} c_j x_j ,$$

subject to:

$$\sum_{j=1}^{n} a_{ij} x_j \leq b_i \quad (\mathrm{i} = 1, 2, \ldots, \mathrm{m}),$$

$$x_j \geq 0 \quad (\mathrm{j} = 1, 2, \ldots, \mathrm{n}).$$

Dual

Minimize

$$z' = \sum_{i=1}^{m} b_i y_i ,$$

subject to:

$$\sum_{i=1}^{m} a_{ij} y_i \geq c_j \quad (\mathrm{j} = 1, 2, \ldots, \mathrm{n}),$$

$$y_i \geq 0 \quad (\mathrm{i} = 1, 2, \ldots, \mathrm{m}).$$

Example:

The Primal problem

Minimize $4x_1 + 2x_2 - x_3$

subject to $x_1 + x_2 + 2x_3 \geq 3$

$2x_1 - 2x_2 + 4x_3 \leq 5$

$x_1, x_2, x_3 \geq 0.$

The dual problem

Maximize $3y_1 + 5y2$

subject to $y_1 + 2y_2 \leq 4$

$y_1 - 2y_2 \leq 2$

$2y_1 + 4y_2 \leq -1$

$y_1 \geq 0, y_2 \geq 0$

4.2 THE MAXIMUM-FLOW PROBLEM

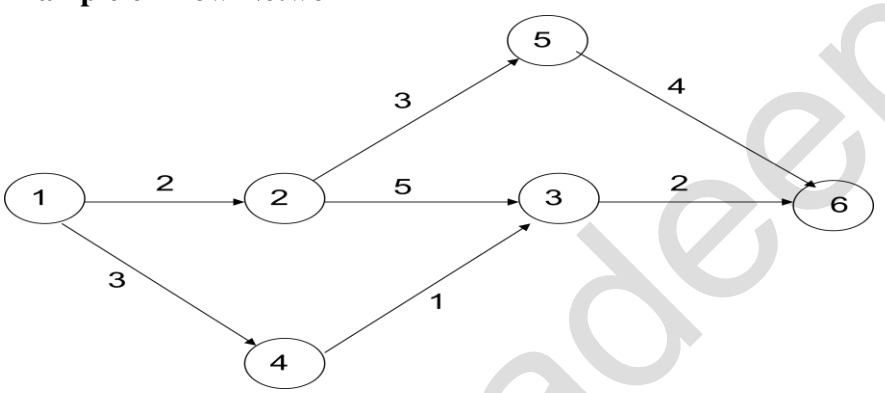
Maximum Flow Problem

Problem of maximizing the flow of a material through a transportation network (e.g., pipeline system, communications or transportation networks)

Formally represented by a connected weighted digraph with n vertices numbered from 1 to n with the following properties:

- Contains exactly one vertex with no entering edges, called the ***source*** (numbered 1)
- Contains exactly one vertex with no leaving edges, called the ***sink*** (numbered n)
- Has positive integer weight u_{ij} on each directed edge (i,j), called the ***edge capacity***, indicating the upper bound on the amount of the material that can be sent from i to j through this edge.
- A digraph satisfying these properties is called a **flow network** or simply a network.

Example of Flow Network



Node (1) = source

Node(6) = sink

Definition of a Flow

A *flow* is an assignment of real numbers x_{ij} to edges (i,j) of a given network that satisfy the following:

- *flow-conservation requirements*

  The total amount of material entering an intermediate vertex must be equal to the total amount of the material leaving the vertex

- *capacity constraints*

  $$0 \le x_{ij} \le u_{ij} \quad \text{for every edge } (i,j) \in E$$

Flow value and Maximum Flow Problem

Since no material can be lost or added to by going through intermediate vertices of the network, the total amount of the material leaving the source must end up at the sink:

$$\sum_{j:\,(1,j)\in E} x_{1j} \;=\; \sum_{j:\,(j,n)\in E} x_{jn}$$

The *value* of the flow is defined as the total outflow from the source (= the total inflow into the sink). The *maximum flow problem* is to find a flow of the largest value (maximum flow) for a given network.

Maximum-Flow Problem as LP problem

Maximize $v = \sum_{j:\,(1,j)\in E} x_{1j}$

subject to

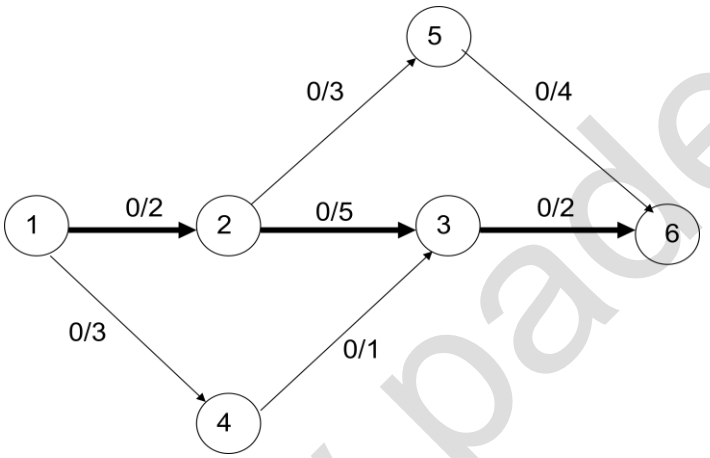
$$\sum_{j:\,(j,i)\in E} x_{ji} - \sum_{j:\,(i,j)\in E} x_{ij} = 0 \quad \text{for } i = 2, 3, \ldots, n\text{-}1$$

$0 \le x_{ij} \le u_{ij}$ for every edge $(i,j) \in E$

Augmenting Path (Ford-Fulkerson) Method

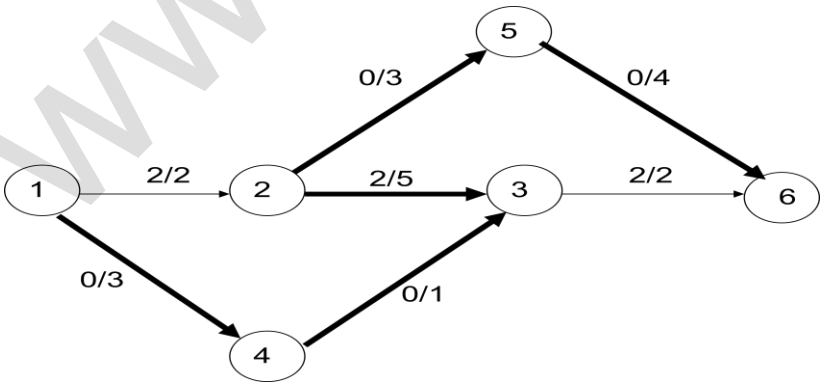
- Start with the zero flow ($x_{ij} = 0$ for every edge).
- On each iteration, try to find a *flow-augmenting path* from source to sink, which a path along which some additional flow can be sent.
- If a flow-augmenting path is found, adjust the flow along the edges of this path to get a flow of increased value and try again.
- If no flow-augmenting path is found, the current flow is maximum.

Example 1



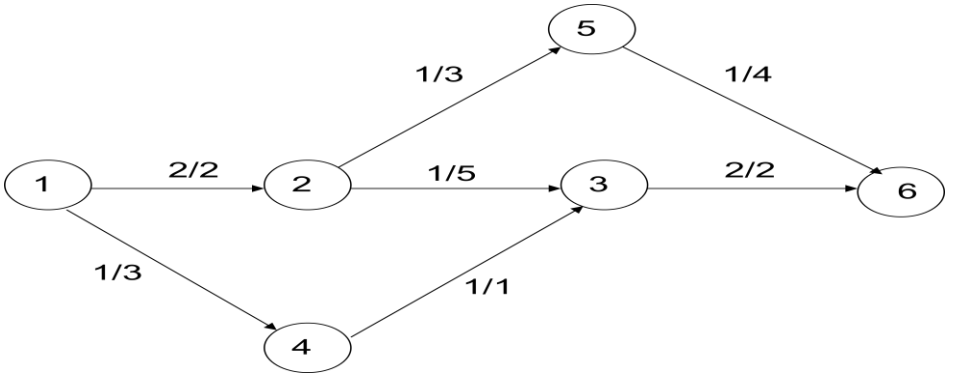
Augmenting path: 1→2 →3 →6

x_{ij}/u_{ij}



Augmenting path: 1 →4 →3←2 →5 →6

Example 1 (maximum flow)



Finding a flow-augmenting path

To find a flow-augmenting path for a flow x, consider paths from source to sink in the underlying undirected graph in which any two consecutive vertices i,j are either:

- connected by a directed edge (i to j) with some positive unused capacity $r_{ij} = u_{ij} - x_{ij}$
  - known as *forward edge* (→)

OR

- connected by a directed edge (j to i) with positive flow x_{ji}
  - known as *backward edge* (←)

If a flow-augmenting path is found, the current flow can be increased by r units by increasing x_{ij} by r on each forward edge and decreasing x_{ji} by r on each backward edge, where

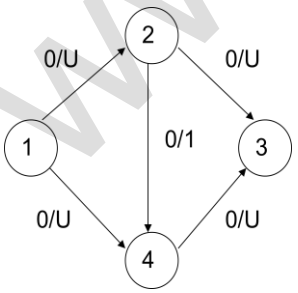
r = min {r_{ij} on all forward edges, x_{ji} on all backward edges}

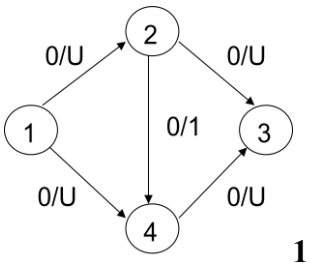
- Assuming the edge capacities are integers, r is a positive integer
- On each iteration, the flow value increases by at least 1
- Maximum value is bounded by the sum of the capacities of the edges leaving the source; hence the augmenting-path method has to stop after a finite number of iterations
- The final flow is always maximum, its value doesn't depend on a sequence of augmenting paths used

Performance degeneration of the method

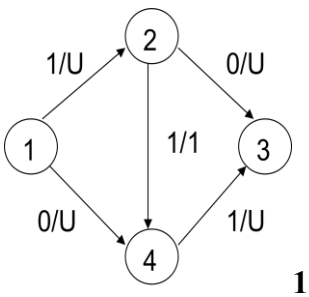
- The augmenting-path method doesn't prescribe a specific way for generating flow-augmenting paths
- Selecting a bad sequence of augmenting paths could impact the method's efficiency

Example 2

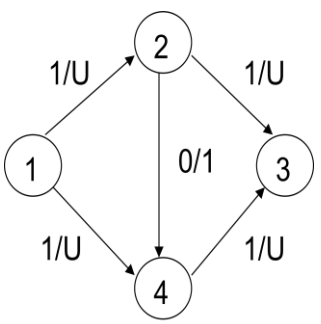




1→2→4→3

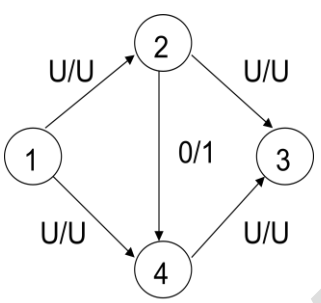


1→4←2→3 **V=1**



V=2

…



V=2U

Requires 2U iterations to reach maximum flow of value 2U

Shortest-Augmenting-Path Algorithm

Generate augmenting path with the least number of edges by BFS as follows.

Starting at the source, perform BFS traversal by marking new (unlabeled) vertices with two labels:

- first label – indicates the amount of additional flow that can be brought from the source to the vertex being labeled
- second label – indicates the vertex from which the vertex being labeled was reached, with "+" or "–" added to the second label to indicate whether the vertex was reached via a forward or backward edge

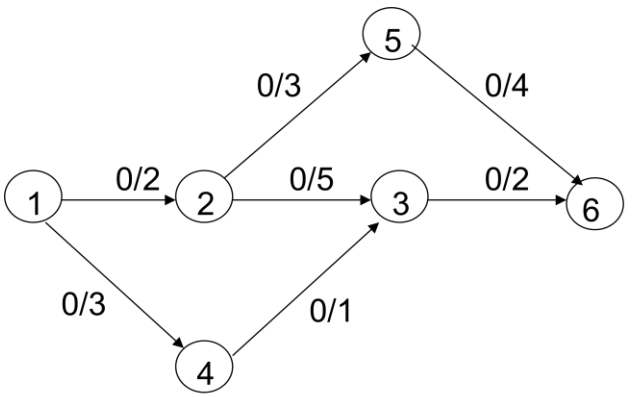
Vertex labeling

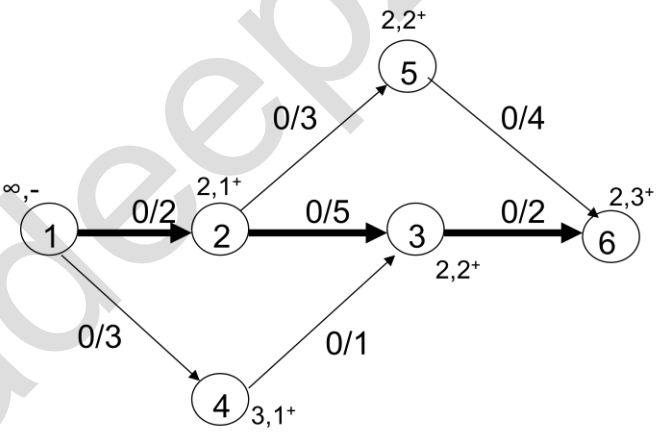
- The source is always labeled with ∞,-
- All other vertices are labeled as follows:
  - If unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from i to j with positive unused capacity $r_{ij} = u_{ij} - x_{ij}$ (forward edge), vertex j is labeled with l_j, i^+, where $l_j = \min\{l_i, r_{ij}\}$

- If unlabeled vertex j is connected to the front vertex i of the traversal queue by a directed edge from j to i with positive flow x_{ji} (backward edge), vertex j is labeled l_j, i^-, where $l_j = \min\{l_i, x_{ji}\}$

- If the sink ends up being labeled, the current flow can be augmented by the amount indicated by the sink's first label.
- The augmentation of the current flow is performed along the augmenting path traced by following the vertex second labels from sink to source; the current flow quantities are increased on the forward edges and decreased on the backward edges of this path.
- If the sink remains unlabeled after the traversal queue becomes empty, the algorithm returns the current flow as maximum and stops.

Example: Shortest-Augmenting-Path Algorithm

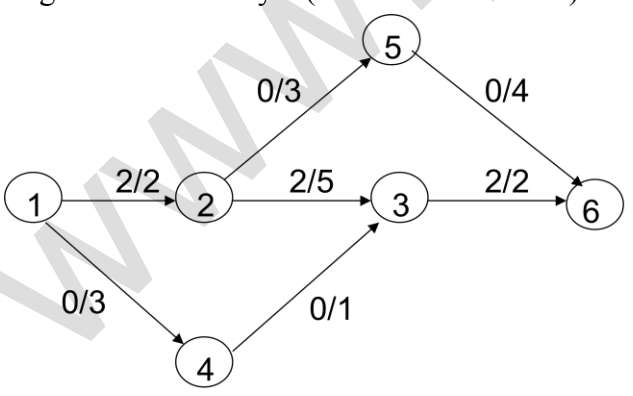


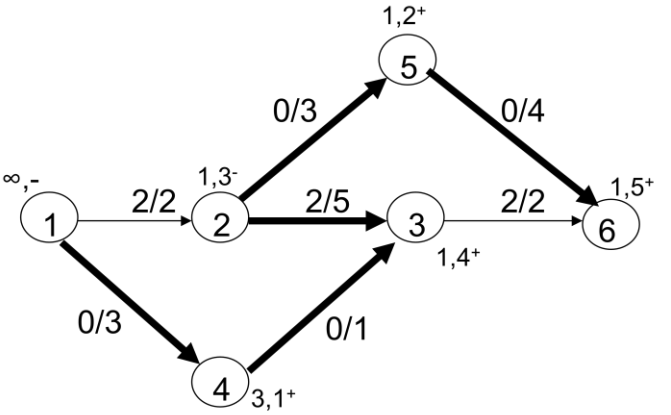


Queue: 1 2 4 3 5 6

↑ ↑ ↑ ↑

Augment the flow by 2 (the sink's first label) along the path 1→2→3→6

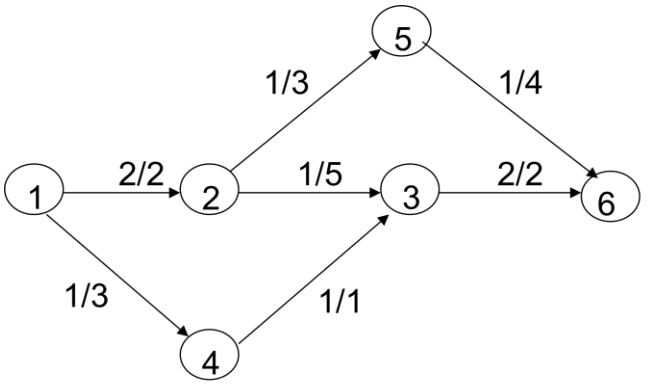


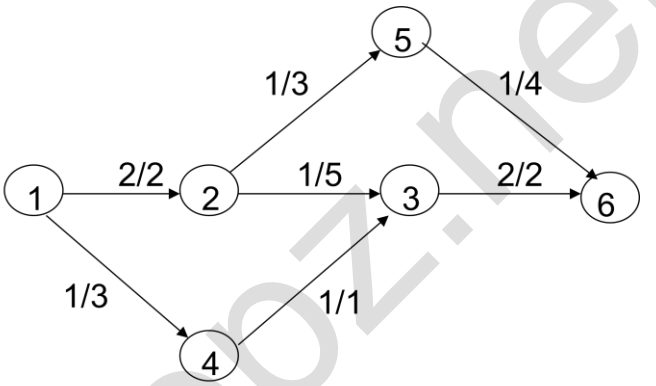


Queue: 1 4 3 2 5 6

↑ ↑ ↑ ↑ ↑

Augment the flow by 1 (the sink's first label) along the path 1→4→3←2→5→6





Queue: 1 4

↑ ↑

No augmenting path (the sink is unlabeled) the current flow is maximum

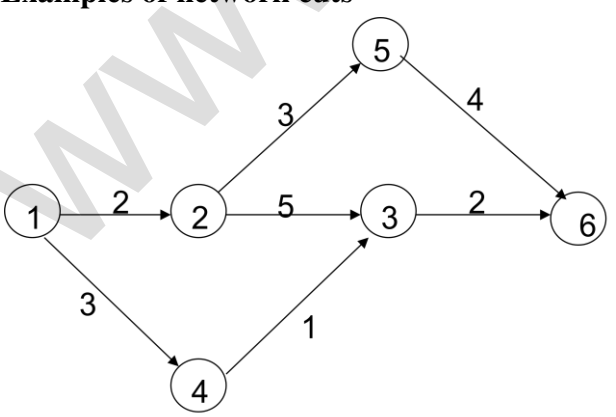
Definition of a Cut

Let X be a set of vertices in a network that includes its source but does not include its sink, and let X, the complement of X, be the rest of the vertices including the sink. The *cut* induced by this partition of the vertices is the set of all the edges with a tail in X and a head in X.

Capacity of a cut is defined as the sum of capacities of the edges that compose the cut.

- We'll denote a cut and its capacity by C(X,X) and c(X,X)
- Note that if all the edges of a cut were deleted from the network, there would be no directed path from source to sink
- *Minimum cut* is a cut of the smallest capacity in a given network

Examples of network cuts



If X = {1} and X = {2,3,4,5,6}, C(X,X) = {(1,2), (1,4)}, c = 5

If X ={1,2,3,4,5} and X = {6}, C(X,X) = {(3,6), (5,6)}, c = 6

If X = {1,2,4} and X ={3,5,6}, C(X,X) = {(2,3), (2,5), (4,3)}, c = 9

Max-Flow Min-Cut Theorem

1. The value of maximum flow in a network is equal to the capacity of its minimum cut
2. The shortest augmenting path algorithm yields both a maximum flow and a minimum cut:
    - Maximum flow is the final flow produced by the algorithm
    - Minimum cut is formed by all the edges from the labeled vertices to unlabeled vertices on the last iteration of the algorithm.
    - All the edges from the labeled to unlabeled vertices are full, i.e., their flow amounts are equal to the edge capacities, while all the edges from the unlabeled to labeled vertices, if any, have zero flow amounts on them.

ALGORITHM *ShortestAugmentingPath(G)*

//Implements the shortest-augmenting-path algorithm
//Input: A network with single source 1, single sink n, and positive integer capacities u_{ij} on
// its edges (i, j)
//Output: A maximum flow x
assign $x_{ij} = 0$ to every edge (i, j) in the network
label the source with $\infty, -$ and add the source to the empty queue Q
while not *Empty(Q)* **do**
  $i \leftarrow$ *Front(Q)*; *Dequeue(Q)*
  for every edge from i to j **do** //forward edges
    if j is unlabeled
      $r_{ij} \leftarrow u_{ij} - x_{ij}$
      if $r_{ij} > 0$
        $lj \leftarrow \min\{l_i, r_{ij}\}$; label j with $l_j, i+$
        Enqueue(Q, j)
  for every edge from j to i **do** //backward edges
    if j is unlabeled
      if $x_{ji} > 0$
        $l_j \leftarrow \min\{l_i, x_{ji}\}$; label j with $l_j, i-$
        Enqueue(Q, j)
  if the sink has been labeled
    //augment along the augmenting path found
    $j \leftarrow n$ //start at the sink and move backwards using second labels
    while $j \neq 1$ //the source hasn't been reached
      if the second label of vertex j is $i+$
        $x_{ij} \leftarrow x_{ij} + l_n$
      else //the second label of vertex j is $i-$
        $x_{ij} \leftarrow x_{ij} - l_n$
      $j \leftarrow i$; $i \leftarrow$ the vertex indicated by i's second label
    erase all vertex labels except the ones of the source
    reinitialize Q with the source
  return x //the current flow is maximum

Time Efficiency

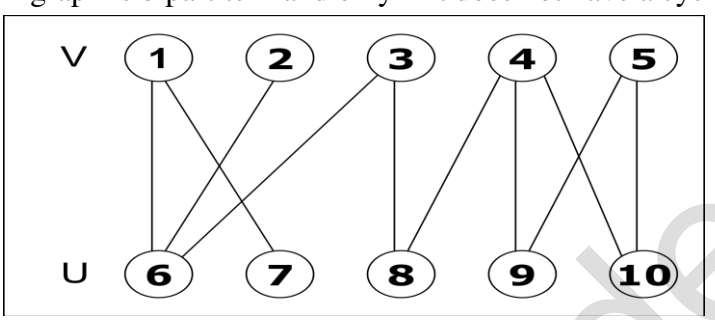
- The number of augmenting paths needed by the shortest-augmenting-path algorithm never exceeds $nm/2$, where n and m are the number of vertices and edges, respectively.
- Since the time required to find shortest augmenting path by breadth-first search is in $O(n+m)=O(m)$ for networks represented by their adjacency lists, the time efficiency of the shortest-augmenting-path algorithm is in $O(nm^2)$ for this representation.
- More efficient algorithms have been found that can run in close to $O(nm)$ time, but these algorithms don't fall into the iterative-improvement paradigm.

4.3 MAXIMUM MATCHING IN BIPARTITE GRAPHS

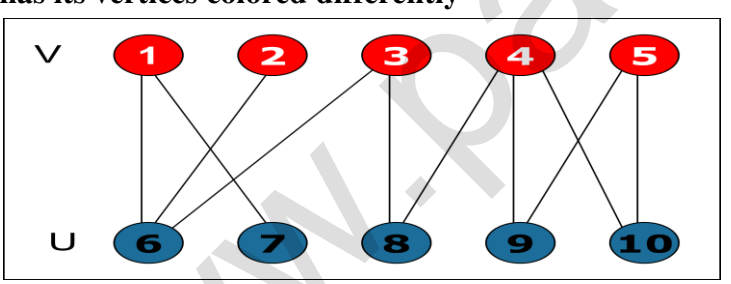
Bipartite Graphs

Bipartite graph: a graph whose vertices can be partitioned into two disjoint sets V and U, not necessarily of the same size, so that every edge connects a vertex in V to a vertex in U.

A graph is bipartite if and only if it does not have a cycle of an odd length.

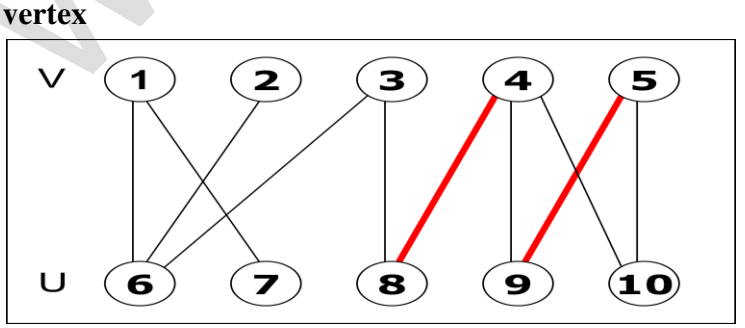


A bipartite graph is *2-colorable*: the vertices can be colored in two colors so that every edge has its vertices colored differently



Matching in a Graph

A *matching* in a graph is a subset of its edges with the property that no two edges share a vertex



a matching in this graph M = {(4,8), (5,9)}

A *maximum* (or *maximum cardinality*) *matching* is a matching with the largest number of edges

- always exists
- not always unique

Free Vertices and Maximum Matching

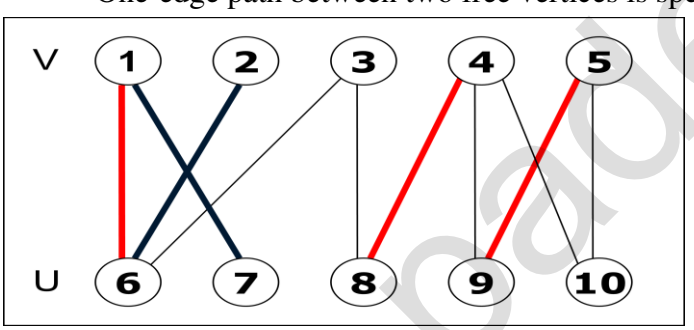
For a given matching M, a vertex is called *free* (or *unmatched*) if it is not an end point of any edge in M; otherwise, a vertex is said to be *matched*

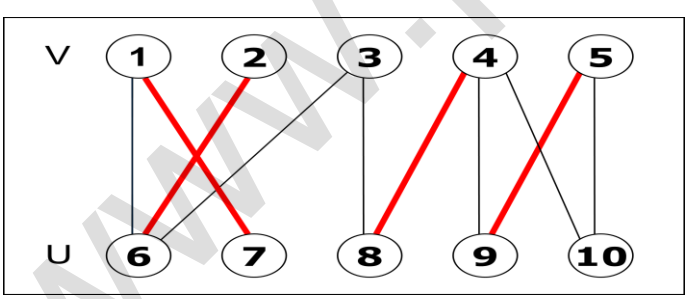
- If every vertex is matched, then M is a maximum matching
- If there are unmatched or free vertices, then M may be able to be improved
- We can immediately increase a matching by adding an edge connecting two free vertices (e.g., (1,6) above)
- Matched vertex = 4, 5, 8, 9. Free vertex = 1, 2, 3, 6, 7, 10.

Augmenting Paths and Augmentation

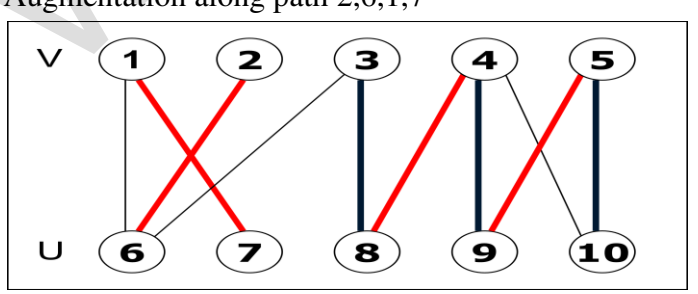
An *augmenting path* for a matching M is a path from a free vertex in V to a free vertex in U whose edges alternate between edges not in M and edges in M

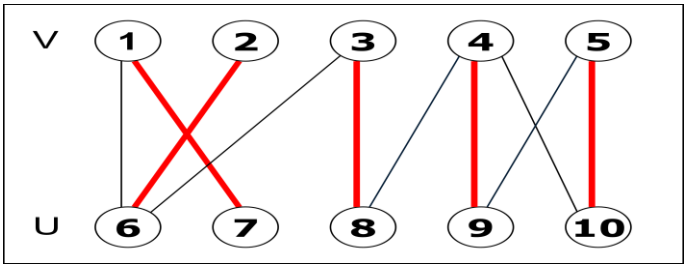
- The length of an augmenting path is always odd
- Adding to M the odd numbered path edges and deleting from it the even numbered path edges increases the matching size by 1 (*augmentation*)
- One-edge path between two free vertices is special case of augmenting path





Augmentation along path 2,6,1,7





Augmentation along 3, 8, 4, 9, 5, 10

Matching on the right is maximum (*perfect* matching).

<u>Theorem:</u> A matching M is maximum if and only if there exists no augmenting path with respect to M.

Augmenting Path Method (template)

- Start with some initial matching . e.g., the empty set
- Find an augmenting path and augment the current matching along that path. e.g., using breadth-first search like method
- When no augmenting path can be found, terminate and return the last matching, which is maximum

4.4 THE STABLE MARRIAGE PROBLEM.

Stable Marriage Problem

- There is a set Y = $\{m_1,\ldots,m_n\}$ of n men and a set X = $\{w_1,\ldots,w_n\}$ of n women. Each man has a ranking list of the women, and each woman has a ranking list of the men (with no ties in these lists).
- A *marriage matching* M is a set of n pairs (m_i, w_j).
- A pair (m, w) is said to be a *blocking pair* for matching M if man m and woman w are not matched in M but prefer each other to their mates in M.
- A marriage matching M is called *stable* if there is no blocking pair for it; otherwise, it's called *unstable*.
- The *stable marriage problem* is to find a stable marriage matching for men's and women's given preferences.

Instance of the Stable Marriage Problem

An instance of the stable marriage problem can be specified either by two sets of preference lists or by a ranking matrix, as in the example below.

<u>men's preferences</u>

	1st	2nd	3rd
Bob:	**Lea**	**Ann**	**Sue**
Jim:	**Lea**	**Sue**	**Ann**
Tom:	**Sue**	**Lea**	**Ann**

<u>women's preferences</u>

	1st	2nd	3rd
Ann:	**Jim**	**Tom**	**Bob**
Lea:	**Tom**	**Bob**	**Jim**
Sue:	**Jim**	**Tom**	**Bob**

<u>ranking matrix</u>

	Ann	Lea	Sue
Bob	**2,3**	**1,2**	**3,3**
Jim	**3,1**	**1,3**	**2,1**
Tom	**3,2**	**2,1**	**1,2**

Data for an instance of the stable marriage problem. (a) Men's preference lists; (b) women's preference lists. (c) Ranking matrix (with the boxed cells composing an unstable matching).

Stable Marriage Algorithm (Gale-Shapley)

Step 0 Start with all the men and women being free

Step 1 While there are free men, arbitrarily select one of them and do the following:

- *Proposal* The selected free man *m* proposes to *w*, the next woman on his preference list
- *Response* If *w* is free, she accepts the proposal to be matched with *m*. If she is not free, she compares *m* with her current mate. If she prefers *m* to him, she accepts *m*'s proposal, making her former mate free; otherwise, she simply rejects *m*'s proposal, leaving *m* free

Step 2 Return the set of *n* matched pairs

Example

Free men: Bob, Jim, Tom

	Ann	Lea	Sue
Bob	2,3	**1,2**	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bob proposed to Lea, Lea accepted Bob

Free men: Jim, Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Jim proposed to Lea, Lea rejected

Free men: Jim, Tom

	Ann	Lea	Sue
Bob	2,3	**1,2**	3,3
Jim	3,1	1,3	**2,1**
Tom	3,2	2,1	1,2

Jim proposed to Sue, Sue accepted

Free men: Tom

	Ann	Lea	Sue
Bob	2,3	**1,2**	3,3
Jim	3,1	1,3	**2,1**
Tom	3,2	2,1	1,2

Tom proposed to Sue, Sue rejected

Free men: Tom

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	**2,1**
Tom	3,2	**2,1**	1,2

Tom proposed to Lea, Lea replaced Bob with Tom

Free men: Bob

	Ann	Lea	Sue
Bob	2,3	1,2	3,3
Jim	3,1	1,3	2,1
Tom	3,2	2,1	1,2

Bob proposed to Ann, Ann accepted

An accepted proposal is indicated by a boxed cell; a rejected proposal is shown by an underlined cell.

Analysis of the Gale-Shapley Algorithm

- The algorithm terminates after no more than n^2 iterations with a stable marriage output.
- The stable matching produced by the algorithm is always *man-optimal*: each man gets the highest rank woman on his list under any stable marriage. One can obtain the *woman-optimal* matching by making women propose to men.
- A man (woman) optimal matching is unique for a given set of participant preferences.
- The stable marriage problem has practical applications such as matching medical-school graduates with hospitals for residency training.

UNIT V - COPING WITH THE LIMITATIONS OF ALGORITHM POWER

5.1 LIMITATIONS OF ALGORITHM POWER

There are many algorithms for solving a variety of different problems. They are very powerful instruments, especially when they are executed by modern computers.

The power of algorithms is limited because of the following reasons:

- There are some problems cannot be solved by any algorithm.
- There are some problems can be solved algorithmically but not in polynomial time.
- There are some problems can be solved in polynomial time by some algorithms, but they are usually lower bounds on their efficiency.

Algorithms limits are identified by the following:

- Lower-Bound Arguments
- Decision Trees
- P, NP and NP-Complete Problems

5.2 LOWER-BOUND ARGUMENTS

We can look at the efficiency of an algorithm two ways. We can establish its **asymptotic efficiency class** (say, for the worst case) and see where this class stands with respect to the **hierarchy of efficiency classes**.

For example, selection sort, whose efficiency is quadratic, is a reasonably fast algorithm, whereas the algorithm for the Tower of Hanoi problem is very slow because its efficiency is exponential.

Lower bounds means estimating the minimum amount of work needed to solve the problem. We present several methods for establishing lower bounds and illustrate them with specific examples.

1. Trivial Lower Bounds
2. Information-Theoretic Arguments
3. Adversary Arguments
4. Problem Reduction

In analyzing the efficiency of specific algorithms in the preceding, we should distinguish between a lower-bound class and a minimum number of times a particular operation needs to be executed.

Trivial Lower Bounds

The simplest method of obtaining a lower-bound class is based on counting the number of items in the problem's **input** that must be **processed** and the number of **output** items that need to be **produced**.

Since any algorithm must at least "read" all the items it needs to process and "write" all its outputs, such a count yields a **trivial lower bound**.

For example, any algorithm for generating all permutations of n distinct items must be in $\Omega(n!)$ because the size of the output is n!. And this bound is **tight** because good algorithms for generating permutations spend a constant time on each of them except the initial one.

Consider the problem of **evaluating a polynomial of degree n** at a given point x, given its coefficients $a_n, a_{n-1}, \ldots, a_0$. $p(x) = a_n x^n + a_{n-1}x^{n-1} + \ldots + a_0$. All the coefficients have to be processed by any polynomial-evaluation algorithm. i.e **$\Omega(n)$.** This is tight lower bound.

Similarly, a trivial lower bound for computing **the product of two n × n matrices** is **$\Omega(n^2)$** because any such algorithm has to process $2n^2$ elements in the input matrices and generate n^2 elements of the product. It is still unknown, however, whether this bound is tight.

The trivial bound for the **traveling salesman problem** is $\Omega(n^2)$, because its input is *n(n-1)/2* intercity distances and its output is a list of n + 1 cities making up an optimal tour. But this bound is useless because there is no known algorithm with the running time being a polynomial function.

Determining the lower bound lies in **which part of an input must be processed** by any algorithm solving the problem. For example, searching for an element of a given value in a sorted array does not require processing all its elements.

Information-Theoretic Arguments

The information-theoretical approach seeks to establish a lower bound based on **the amount of information it has to produce** by algorithm.

Consider an example "**Game of guessing number**", the well-known game of deducing a positive integer between 1 and n selected by somebody by asking that person questions with yes/no answers. The amount of uncertainty that any algorithm solving this problem has to resolve can be measured by $\lceil \log_2 n \rceil$.

The number of bits needed to specify a particular number among the n possibilities. Each answer to the question gives information about each bit.

1. Is the first bit zero? → No→ first bit is 1
2. Is the second bit zero? → Yes→ second bit is 0
3. Is the third bit zero? → Yes→ third bit is 0
4. Is the forth bit zero? → Yes→ forth bit is 0

The number in binary is 1000, i.e.8 in decimal value.

The above approach is called the ***information-theoretic argument*** because of its connection to information theory. This is useful for finding ***information-theoretic lower bounds*** for many problems involving comparisons, including sorting and searching.

Its underlying idea can be realized the mechanism of ***decision trees***. Because

Adversary Arguments

Adversary Argument is a method of proving by **playing a role of adversary (opponent)** in which algorithm has to work more for **adjusting input** consistently.

Consider the Game of guessing number between positive integer 1 and n by asking a person (Adversary) with yes/no type answers for questions. After each question at least one-half of the numbers reduced. If an algorithm stops before the size of the set is reduced to 1, the adversary can exhibit a number.

Any algorithm needs $\lceil \log_2 n \rceil$ iterations to shrink an n-element set to a one-element set by halving and rounding up the size of the remaining set. Hence, at least $\lceil \log_2 n \rceil$ questions need to be asked by any algorithm in the worst case. This example illustrates the ***adversary method*** for establishing lower bounds.

Consider the problem of **merging two sorted lists** of size n $a_1 < a_2 < \ldots < a_n$ and $b_1 < b_2 < \ldots < b_n$ into a single sorted list of size 2n. For simplicity, we assume that all the a's and b's are distinct, which gives the problem a unique solution.

Merging is done by repeatedly comparing the first elements in the remaining lists and outputting the smaller among them. The number of key comparisons (lower bound) in the worst case for this algorithm for merging is $\mathbf{2n - 1}$.

Problem Reduction

Problem reduction is a method in which a difficult unsolvable problem P is reduced to another solvable problem B which can be solved by a known algorithm.

A similar reduction idea can be used for finding a lower bound. To show that problem P is at least as hard as another problem Q with a known lower bound, we need to reduce Q to P (not P to Q!). In other words, we should show that an arbitrary instance of problem Q can be transformed to an instance of problem P, so any algorithm solving P would solve Q as well. Then a lower bound for Q will be a lower bound for P. Table 5.1 lists several important problems that are often used for this purpose.

TABLE 5.1 Problems often used for establishing lower bounds by problem reduction

Problem	Lower bound	Tightness
Sorting	Ω(n log n)	yes
searching in a sorted array	Ω (log n)	yes
element uniqueness problem	Ω (n log n)	yes
multiplication of n-digit integers	Ω(n)	unknown
multiplication of n × n matrices	Ω(n2)	unknown

Consider the Euclidean minimum spanning tree problem as an example of establishing a lower bound by reduction:

Given n points in the Cartesian plane, construct a tree of minimum total length whose vertices are the given points. As a problem with a known lower bound, we use the element uniqueness problem.

We can transform any set x1, x2, . . . , xn of n real numbers into a set of n points in the Cartesian plane by simply adding 0 as the points' y coordinate: (x1, 0), (x2, 0), . . . , (xn, 0). Let T be a minimum spanning tree found for this set of points. Since T must contain a shortest edge, checking whether T contains a zero length edge will answer the question about uniqueness of the given numbers. This reduction implies that Ω (n log n) is a lower bound for the Euclidean minimum spanning tree problem,

Note: Limitations of algorithm can be studied by obtaining lower bound efficiency.

5.3 DECISION TREES

Important algorithms like sorting and searching are based on comparing items of their inputs. The study of the performance of such algorithm is called a **decision tree**. As an example, Figure 5.1 presents a decision tree of an algorithm for finding a minimum of three numbers. Each internal node of a binary decision tree represents a key comparison indicated in the node.

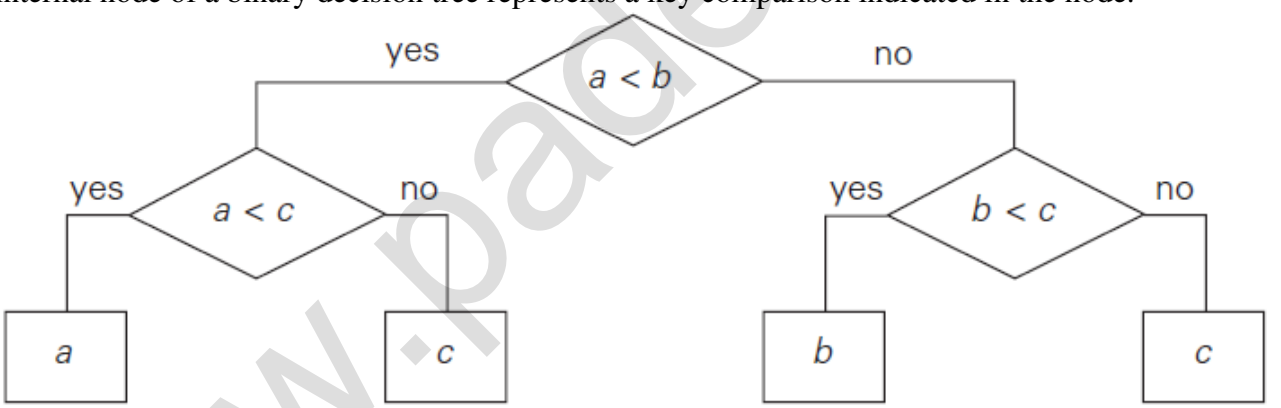
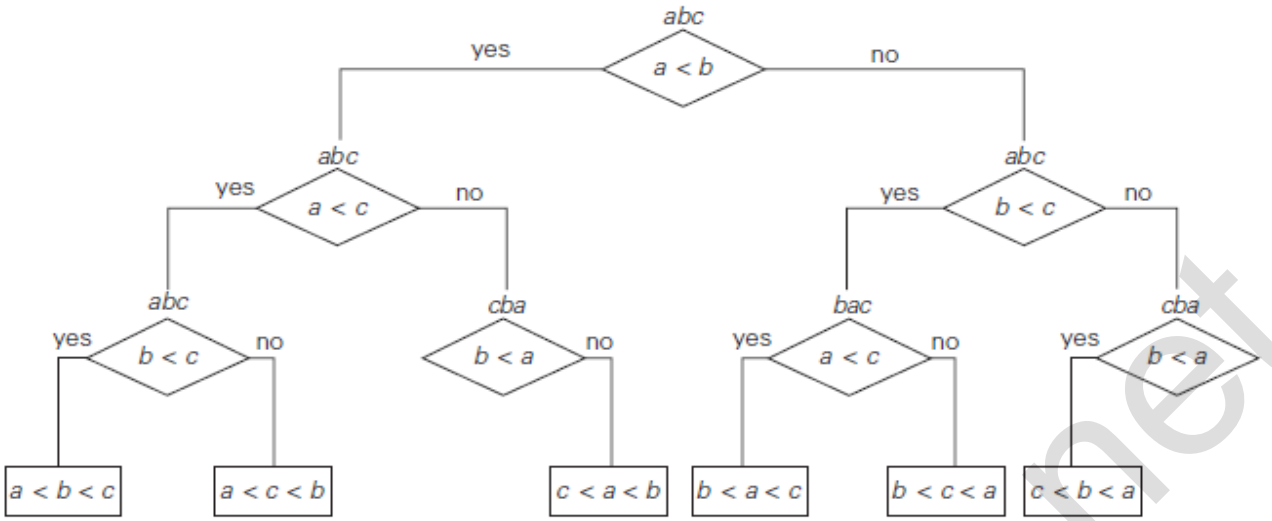


FIGURE 5.1 Decision tree for finding a minimum of three numbers.

Consider a binary decision tree with height h and leaves n. and height h, then $h \geq \lceil \log_2 n \rceil$. A binary tree of height h with the largest number of leaves on the last level is 2^h. In other words, $2^h \geq n$, which puts a lower bound on the heights of binary decision trees. Hence the worst-case number of comparisons made by any comparison-based algorithm for the problem is called the information theoretic lower bound.

Decision Trees for Sorting



C b a
1 2 3

FIGURE 5.2 Decision tree for the tree-element selection sort.

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

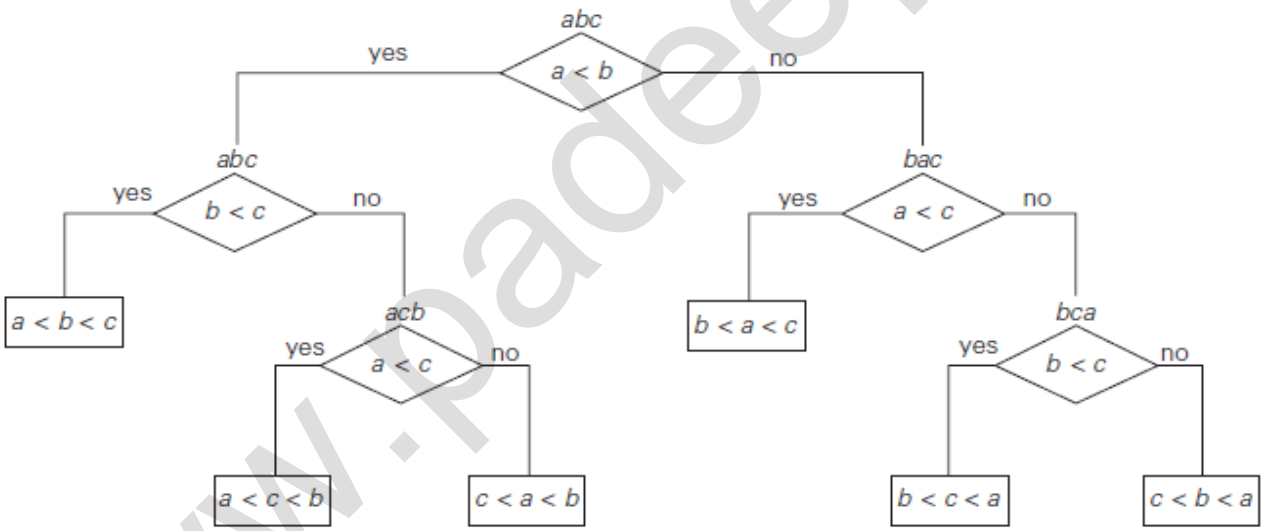


FIGURE 5.3 Decision tree for the three-element insertion sort.

The three-element insertion sort whose decision tree is given in Figure 5.3, this number is (2 + 3 + 3 + 2 + 3 + 3)/6 = 2.66. Under the standard assumption that all n! outcomes of sorting are equally likely, the following lower bound on the average number of comparisons C_{avg} made by any comparison-based algorithm in sorting an n-element list has been proved:

$$C_{avg}(n) \geq \log_2 n!.$$

Decision tree is a convenient model of algorithms involving comparisons in which

- internal nodes represent comparisons
- leaves represent outcomes (or input cases)

Decision Trees and Sorting Algorithms

- Any comparison-based sorting algorithm can be represented by a decision tree (for each fixed *n*)
- Number of leaves (outcomes) ≥ *n*!

- Height of binary tree with $n!$ leaves $\geq \lceil \log_2 n! \rceil$
- Minimum number of comparisons in the worst case $\geq \lceil \log_2 n! \rceil$ for any comparison-based sorting algorithm, since the longest path represents the worst case and its length is the height
- $\lceil \log_2 n! \rceil \approx n \log_2 n$ (by Sterling approximation)
- This lower bound is tight (mergesort or heapsort)

Decision Trees for Searching a Sorted Array

Decision trees can be used for establishing lower bounds on the number of key comparisons in searching a sorted array of n keys: $A[0]<A[1]<\ldots<A[n-1]$.

The principal algorithm for this problem is binary search. The number of comparisons made by binary search in the worst case, $C_{worst}(n)$, is given by the formula

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

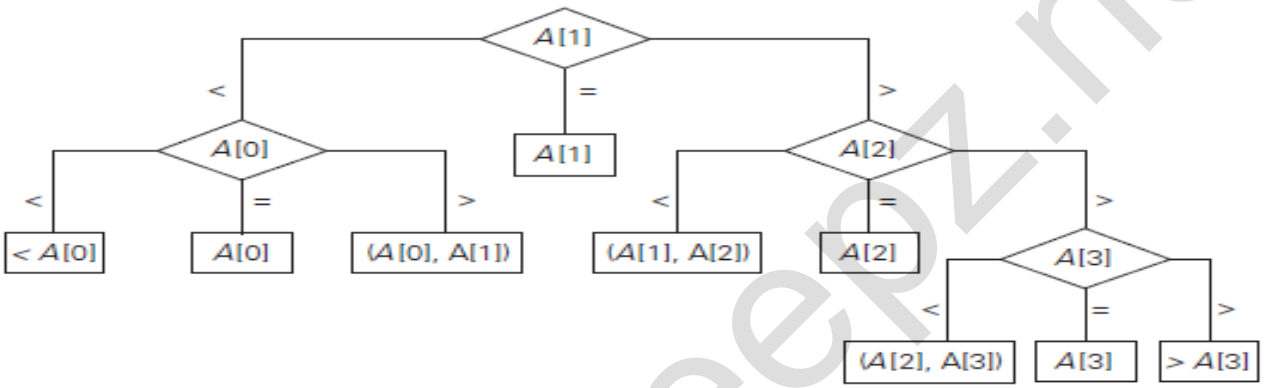


FIGURE 5.4 Ternary decision tree for binary search in a four-element array.

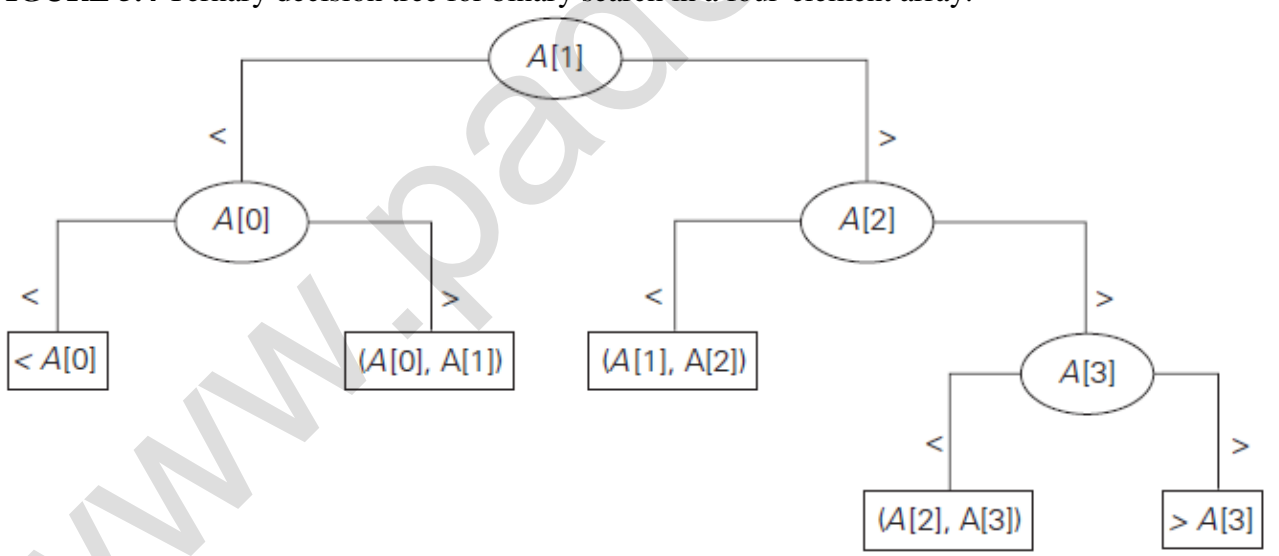


FIGURE 5.5 Binary decision tree for binary search in a four-element array.

As comparison of the decision trees in the above illustrates, the binary decision tree is simply the ternary decision tree with all the middle subtrees eliminated. Applying inequality to such binary decision trees immediately yields $C_{worst}(n) \geq \lceil \log_2 (n+1) \rceil$

5.4 P, NP AND NP-COMPLETE PROBLEMS

Problems that can be solved in polynomial time are called ***tractable***, and problems that cannot be solved in polynomial time are called ***intractable***.

There are several **reasons for intractability**.

- **First**, we **cannot solve** arbitrary instances of intractable problems in a reasonable amount of time unless such **instances are very small**.
- **Second**, although there might be a huge difference between the running times in *O(p(n))* for polynomials of **drastically different degrees**. where p(n) is a polynomial of the problem's input size n.
- **Third**, polynomial functions possess many convenient properties; in particular, both the sum and composition of two polynomials are **always polynomials too**.
- **Fourth**, the choice of this class has led to a development of an extensive theory called ***computational complexity***.

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 class***.

- 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.
- The class of decision problems that are solvable in O($p(n)$) **polynomial time**, where $p(n)$ is a polynomial of problem's input size n

  Examples:
  - Searching
  - Element uniqueness
  - Graph connectivity
  - Graph acyclicity
  - Primality testing (finally proved in 2002)
- **The restriction of P** to decision problems can be justified by the following reasons.
  - First, it is sensible to **exclude problems not solvable in polynomial time** because of their exponentially large output. e.g., generating subsets of a given set or all the permutations of n distinct items.
  - Second, **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. For example, instead of asking about the minimum number of colors needed to color the vertices of a graph so that no two adjacent vertices are colored the same color. Coloring of the graph's vertices with no more than m colors for m = 1, 2, ... (The latter is called the **m-coloring problem**.)
  - So, every decision problem can not be solved in polynomial time. Some **decision** problems cannot be solved at all by any algorithm. Such problems are called **undecidable**, as opposed to **decidable** problems that can be solved by an algorithm (**Halting problem**).
- **Non polynomial-time algorithm:** There are many important problems, however, for which no polynomial-time algorithm has been found.
  - ***Hamiltonian circuit problem*:** 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.
  - ***Traveling salesman problem*:** 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).

- ***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.
- ***Partition problem*:** Given n positive integers, determine whether it is possible to partition them into two disjoint subsets with the same sum.
- ***Bin-packing problem*:** Given n items whose sizes are positive rational numbers not larger than 1, put them into the smallest number of bins of size 1.
- ***Graph-coloring problem*:** For a given graph, find its chromatic number, which is 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.
- ***Integer linear programming problem*:** Find the maximum (or minimum) value of a linear function of several integer-valued variables subject to a finite set of constraints in the form of linear equalities and inequalities.

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

1. **Nondeterministic ("guessing") stage:** An arbitrary string S is generated that can be thought of as a candidate solution to the given instance.
2. **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. (If S is not a solution to instance I , the algorithm either returns no or is allowed not to halt at all.)

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:

$$\mathbf{P \subseteq NP}$$

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 NP also contains the Hamiltonian circuit problem, the partition problem, decision versions of the traveling salesman, the knapsack, graph coloring, and many hundreds of other difficult combinatorial optimization. The halting problem, on the other hand, is among the rare examples of decision problems that are known not to be in NP.

Note that P = NP would imply that each of many hundreds of difficult combinatorial decision problems can be solved by a polynomial-time algorithm.

Definition: A decision problem $D1$ is said to be ***polynomially reducible*** to a decision problem $D2$, if there exists a function t that transforms instances of $D1$ to instances of $D2$ such that:

1. t maps all yes instances of $D1$ to yes instances of $D2$ and all no instances of $D1$ to no instances of $D2$.

2. t is computable by a polynomial time algorithm.

This definition immediately implies that if a problem D1 is polynomially reducible to some problemD2 that can be solved in polynomial time, then problem D1 can also be solved in polynomial time

Definition: A decision problem D is said to be ***NP-complete*** if it is hard as any problem in NP.

1. It belongs to class *NP*

2. Every problem in *NP* is polynomially reducible to D

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.

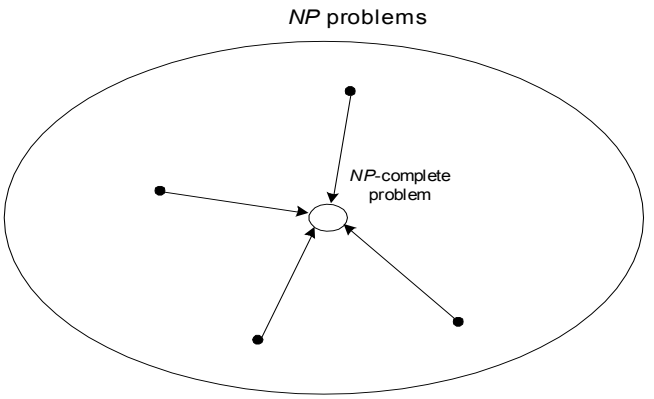


FIGURE 5.6 Polynomial-time reductions of *NP* problems to an *NP*-complete problem

Theorem: A decision problem is said to be *NP-complete* if it is hard as any problem in NP.

Proof: Let us prove that the Hamiltonian circuit problem is polynomially reducible to the decision version of the traveling salesman problem.

We can map a graph G of a given instance of the Hamiltonian circuit problem to a complete weighted graph G′ representing an instance of the traveling salesman problem by assigning 1 as the weight to each edge in G and adding an edge of weight 2 between any pair of nonadjacent vertices in G. As the upper bound m on the Hamiltonian circuit length, we take $m = n$, where n is the number of vertices in G (and G′). Obviously, this transformation can be done in polynomial time.

Let G be a yes instance of the Hamiltonian circuit problem. Then G has a Hamiltonian circuit, and its image in G′ will have length n, making the image a yes instance of the decision traveling salesman problem.

Conversely, if we have a Hamiltonian circuit of the length not larger than n in G′, then its length must be exactly n and hence the circuit must be made up of edges present in G, making the inverse image of the yes instance of the decision traveling salesman problem be a yes instance of the Hamiltonian circuit problem.

This completes the proof.

Theorem: **State and prove Cook's theorem.**

Prove that CNF-sat is *NP*-complete.

Satisfiability of boolean formula for three conjuctive normal form is NP-Complete.

NP problems obtained by polynomial-time reductions from a *NP*-complete problem

Proof: The notion of *NP*-completeness requires, however, polynomial reducibility of *all* problems in *NP,* both known and unknown, to the problem in question. Given the bewildering variety of decision problems, it is nothing short of amazing that specific examples of *NP*-complete problems have been actually found.

Nevertheless, this mathematical feat was accomplished independently by Stephen Cook in the United States and Leonid Levin in the former Soviet Union. In his 1971 paper, Cook [Coo71] showed that the so-called ***CNF-satisfiability problem*** is *NP*complete.

x_1	x_2	x_3	$\bar{x}_1$	$\bar{x}_2$	$\bar{x}_3$	$x_1 \vee \bar{x}_2 \vee \bar{x}_3$	$\bar{x}_1 \vee x_2$	$\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3$	$(x_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_2) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3)$
T	T	T	F	F	F	T	T	F	F
T	**T**	**F**	F	F	T	T	T	T	**T**
T	F	T	F	T	F	T	F	T	F
T	F	F	F	T	T	T	F	T	F

F	T	T	T	F	F	F	T	T	F
F	T	F	T	F	T	T	T	T	T
F	F	T	T	T	F	T	T	T	T
F	F	F	T	T	T	T	T	T	T

The CNF-satisfiability problem deals with boolean expressions. Each boolean expression can be represented in conjunctive normal form, such as the following expression involving three boolean variables x_1, x_2, and x_3 and their negations denoted $\bar{x}_1$, $\bar{x}_2$, and $\bar{x}_3$ 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 one can assign values *true* and *false* to variables of a given boolean expression in its CNF form to make the entire expression *true*. (It is easy to see that this can be done for the above formula: if x_1 = *true*, x_2 = *true*, and x_3 = *false*, the entire expression is *true*.)

Since the Cook-Levin discovery of the first known *NP*-complete problems, computer scientists have found many hundreds, if not thousands, of other examples. In particular, the well-known problems (or their decision versions) mentioned above—Hamiltonian circuit, traveling salesman, partition, bin packing, and graph coloring—are all *NP*-complete. It is known, however, that if *P* != *NP* there must exist *NP* problems that neither are in *P* nor are *NP*-complete.

Showing that a decision problem is *NP*-complete can be done in two steps.

1. First, one needs to show that the problem in question is in *NP*; i.e., a randomly generated string can be checked in polynomial time to determine whether or not it represents a solution to the problem. Typically, this step is easy.
2. The second step is to show that every problem in NP is reducible to the problem in questionin polynomial time. Because of the transitivity of polynomial reduction, this step can be done by showing that a known NP-complete problem can be transformed to the problem in question in polynomial time.

The definition of *NP*-completeness immediately implies that if there exists 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*.

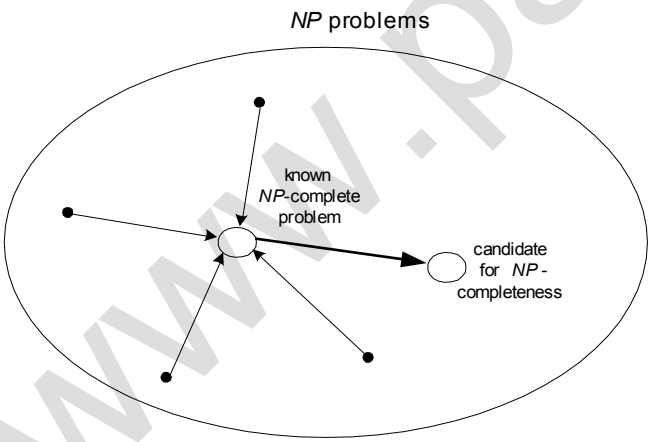


FIGURE 5.7 NP-completeness by reduction

Examples: TSP, knapsack, partition, graph-coloring and hundreds of other problems of combinatorial nature P = NP would imply that every problem in NP, including all NP-complete problems, could be solved in polynomial time If a polynomial-time algorithm for just one NP-complete problem is discovered, then every problem in NP can be solved in polynomial time, i.e. P = NP Most but not all researchers believe that P != NP , i.e. P is a proper subset of NP. If P != NP, then the NP-complete problems are not in P, although many of them are very useful in practice.

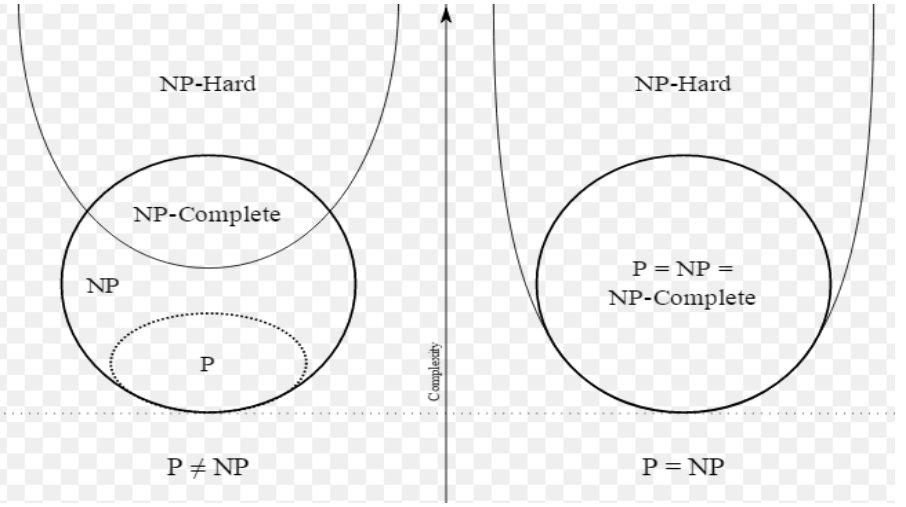


FIGURE 5.8 Relation among P, NP, NP-hard and NP Complete problems

5.5 COPING WITH THE LIMITATIONS OF ALGORITHM POWER

There are some problems that are difficult to solve algorithmically. At the same time, few of them are so important, we must solve by some other technique. Two algorithm design techniques ***backtracking*** and ***branch-and-bound*** that often make it possible to solve at least some large instances of difficult combinatorial problems.

Both backtracking and branch-and-bound are based on the construction of a state-space tree whose nodes reflect specific choices made for a solution's components. Both techniques terminate a node as soon as it can be guaranteed that no solution to the problem can be obtained by considering choices that correspond to the node's descendants

We consider a few approximation algorithms for solving the Assignment Problem, traveling salesman and knapsack problems. There are three classic methods like the bisection method, the method of false position, and Newton's method for approximate root finding.

Exact Solution Strategies are given below:

Exhaustive search **(brute force)-**

- useful only for small instances

Dynamic programming

- applicable to some problems (e.g., the knapsack problem)

Backtracking

- eliminates some unnecessary cases from consideration
- yields solutions in reasonable time for many instances but worst case is still exponential

Branch-and-bound

- further refines the backtracking idea for optimization problems

Coping with the Limitations of Algorithm Power are given below:

Backtracking

- *n*-Queens Problem
- Hamiltonian Circuit Problem
- Subset-Sum Problem

Branch-and-Bound
- Assignment Problem
- Knapsack Problem
- Traveling Salesman Problem

Approximation Algorithms for *NP*-Hard Problems
- Approximation Algorithms for the Traveling Salesman Problem
- Approximation Algorithms for the Knapsack Problem

Algorithms for Solving Nonlinear Equations
- Bisection Method
- False Position Method
- Newton's Method

5.6 BACKTRACKING

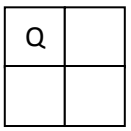
- Backtracking is a more intelligent variation approach.
- The principal idea is to construct solutions one component at a time and evaluate such partially constructed candidates as follows.
- If a partially constructed solution can be developed further without violating the problem's constraints, it is done by taking the first remaining legitimate option for the next component.
- If there is no legitimate option for the next component, no alternatives for any remaining component need to be considered. In this case, the algorithm backtracks to replace the last component of the partially constructed solution with its next option.
- It is convenient to implement this kind of processing by constructing a tree of choices being made, called **the 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 represent the choices made for the first component of a solution, the nodes of the second level 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. In the majority of cases, a statespace tree for a backtracking algorithm is constructed in the manner of depthfirst 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, and the processing moves to this child. If the current node turns out to be 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 (if just one solution is required) or continues searching for other possible solutions.
- Backtracking techniques are applied to solve the following problems
  - *n*-Queens Problem
  - Hamiltonian Circuit Problem
  - Subset-Sum Problem

5.7 N-QUEENS PROBLEM

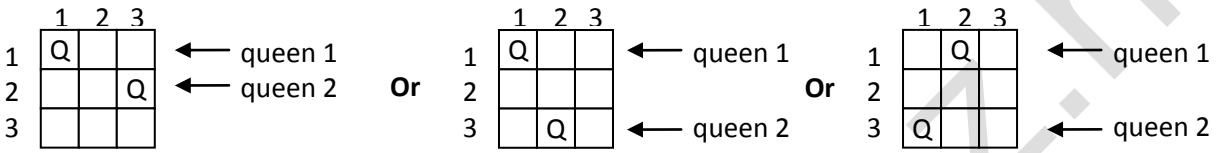
The problem is to place n queens on an $n \times n$ chessboard so that no two queens attack each other by being in the same row or in the same column or on the same diagonal.

For n = 1, the problem has **a trivial solution**.

For n = 2, it is easy to see that there is **no solution** to place 2 queens in 2 × 2 chessboard.

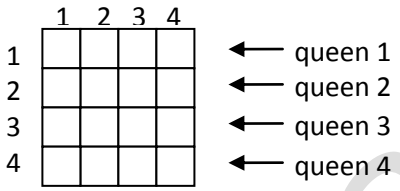


For n = 3, it is easy to see that there is **no solution** to place 3 queens in 3 × 3 chessboard.

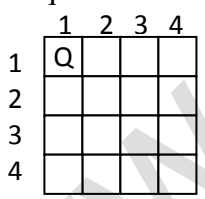


For n = 4, There is **solution** to place 4 queens in 4 × 4 chessboard. the four-queens problem solved by the backtracking technique.

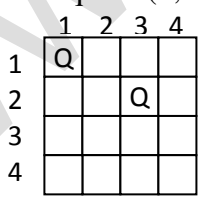
Step 1: Start with the empty board



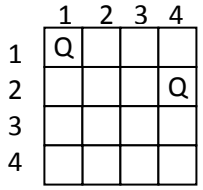
Step 2: Place queen 1 in the first possible position of its row, which is in column 1 of row 1.



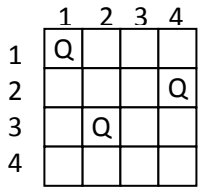
Step 3: place queen 2, after trying unsuccessfully columns 1 and 2, in the first acceptable position for it, which is square (2, 3), the square in row 2 and column 3.



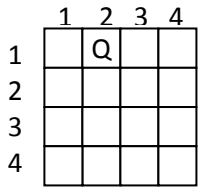
Step 4: This proves to be a dead end because there is no acceptable position for queen 3. So, the algorithm backtracks and puts queen 2 in the next possible position at (2, 4).



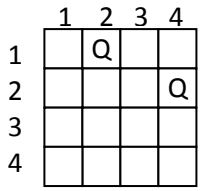
Step 5: Then queen 3 is placed at (3, 2), which proves to be another dead end.



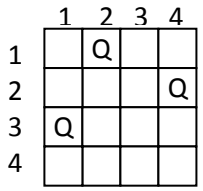
Step 6: The algorithm then backtracks all the way to queen 1 and moves it to (1, 2).



Step 7: The queen 2 goes to (2, 4).



Step 8: The queen 3 goes to (3, 1).



Step 9: The queen 3 goes to (4, 3). This is a solution to the problem.

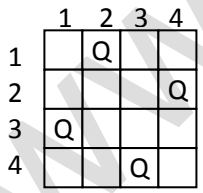


FIGURE 5.9 Solution four-queens problem in 4x4 Board.

The state-space tree of this search is shown in Figure 12.2

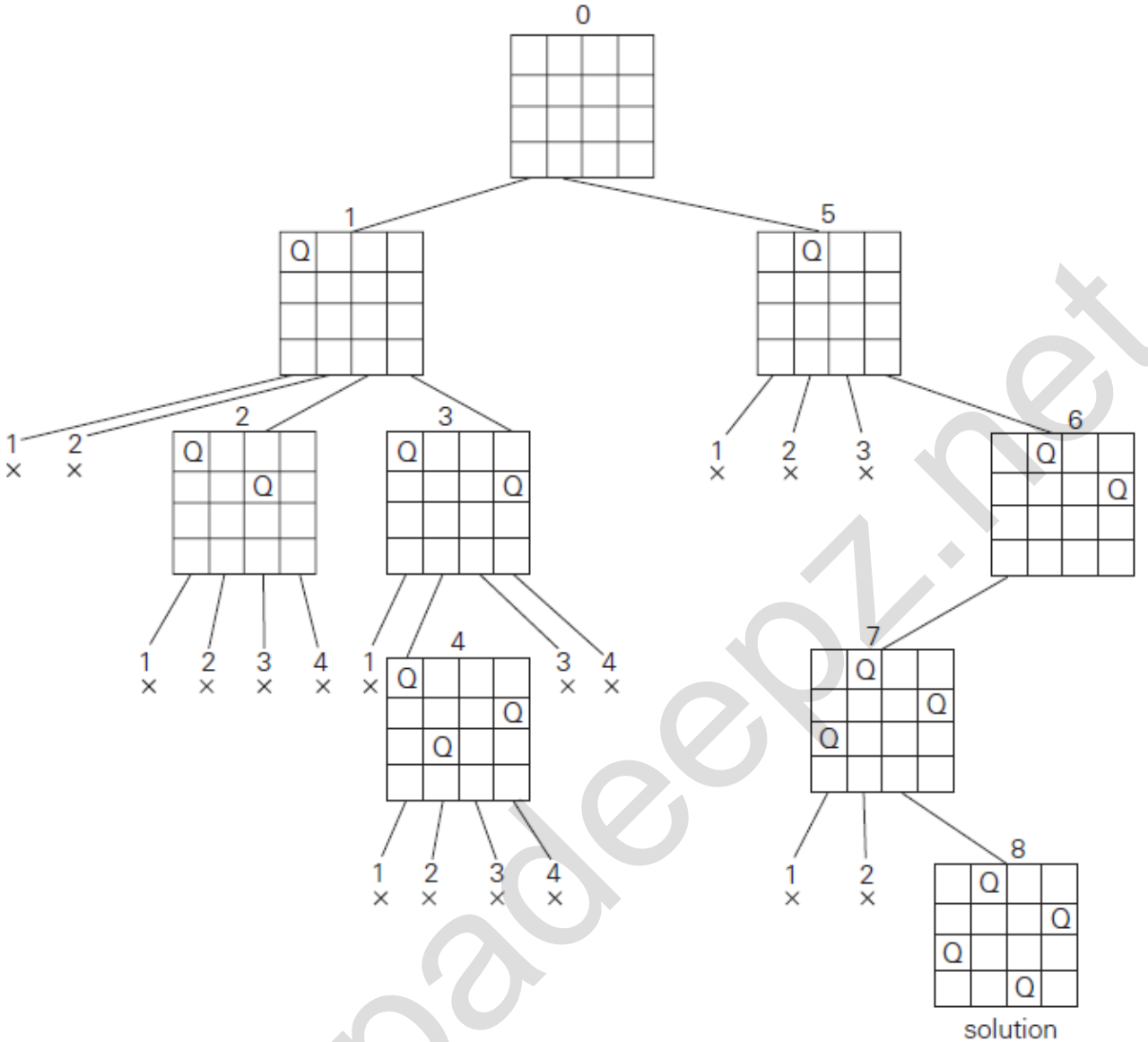


FIGURE 5.10 State-space tree of solving the four-queens problem by backtracking. × denotes an unsuccessful attempt to place a queen.

For *n* = 8, There is **solution** to place 8 queens in 8 × 8 chessboard.

	1	2	3	4	5	6	7	8
1				Q				
2						Q		
3								Q
4			Q					
5	Q							
6							Q	
7					Q			
8		Q						

FIGURE 5.11 Solution 8-queens problem in 8x8 Board.

5.8 HAMILTONIAN CIRCUIT PROBLEM

A **Hamiltonian circuit** (also called a **Hamiltonian cycle**, **Hamilton cycle**, or **Hamilton circuit**) is a graph **cycle** (i.e., closed loop) through a graph that visits each node exactly once. A graph possessing a **Hamiltonian cycle** is said to be a **Hamiltonian** graph.

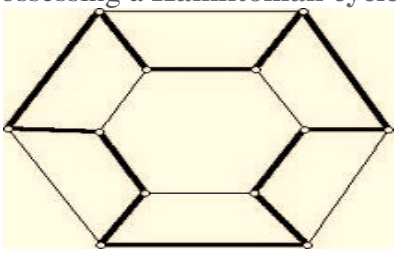


FIGURE 5.12 Graph contains Hamiltonian circuit

Let us consider the problem of finding a Hamiltonian circuit in the graph in Figure 5.13.

Example: Find Hamiltonian circuit starts at vertex *a.*

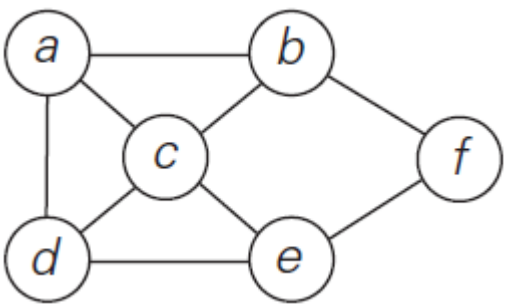


FIGURE 5. 13 Graph.

Solution:

- Assume that if a Hamiltonian circuit exists, it starts at vertex *a.* accordingly, we make vertex *a* the root of the state-space tree as in Figure 5.14.
- In a Graph G, Hamiltonian cycle begins at some vertex $V_1 \in G$, and the vertices are visited only once in the order $V_1, V_2, \ldots, V_n$. (V_i are distinct except for V_1 and V_{n+1} which are equal).
- The first component of our future solution, if it exists, is a first intermediate vertex of a Hamiltonian circuit to be constructed. 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.

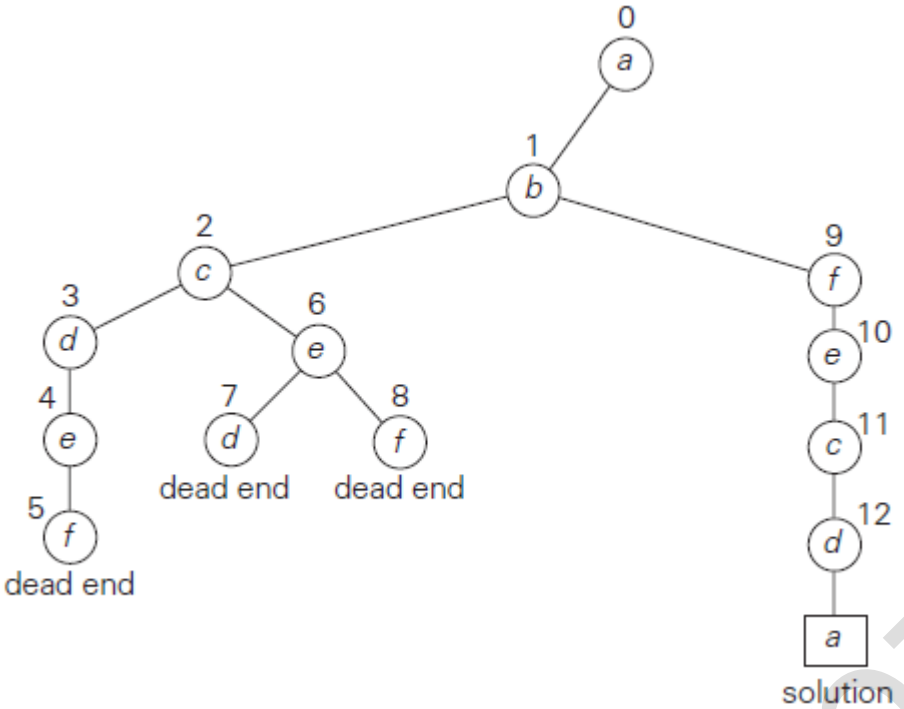


FIGURE 5.14 State-space tree for finding a Hamiltonian circuit.

5.9 SUBSET SUM PROBLEM

The ***subset-sum problem*** finds a subset of a given set $A = \{a1, \ldots, an\}$ of n positive integers whose sum is equal to a given positive integer d. For example, for $A = \{1, 2, 5, 6, 8\}$ and $d = 9$, there are two solutions: {1, 2, 6} and {1, 8}. Of course, some instances of this problem may have no solutions.

It is convenient to sort the set's elements in increasing order. So, we will assume that $a1 < a2 < \ldots < an$.

A = {3, 5, 6, 7} and d = 15 of the subset-sum problem. The number inside a node is the sum of the elements already included in the subsets represented by the node. The inequality below a leaf indicates the reason for its termination.

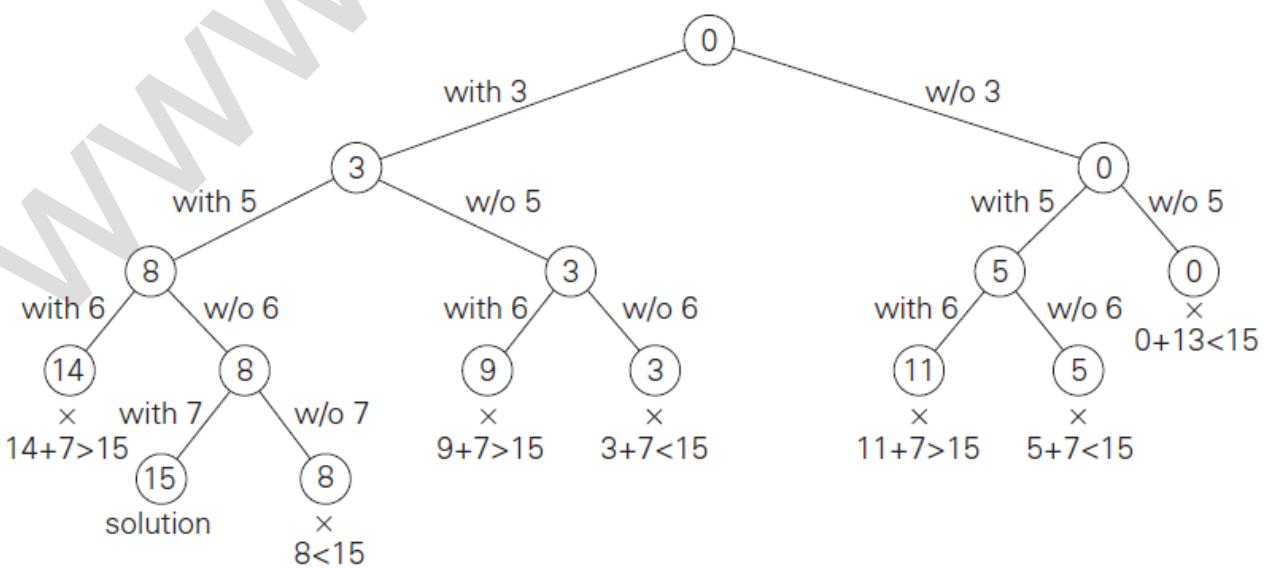


FIGURE 5.15 Complete state-space tree of the backtracking algorithm applied to the instance

Example:

- The state-space tree can be constructed as a binary tree like that in Figure 5.15 for the instance $A = \{3, 5, 6, 7\}$ and $d = 15$.
- The root of the tree represents the starting point, with no decisions about the given elements made as yet.
- Its left and right children represent, respectively, inclusion and exclusion of a_1 in a set being sought. Similarly, going to the left from a node of the first level corresponds to inclusion of a_2 while going to the right corresponds to its exclusion, and so on.
- Thus, a path from the root to a node on the *i*th level of the tree indicates which of the first *I* numbers have been included in the subsets represented by that node.
- 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. We can either report this result and stop or, if all the solutions need to be found, continue by backtracking to the node's parent.
- If *s* is not equal to *d,* we can terminate the node as nonpromising if either of the following two inequalities holds:

$$s + a_{i+1} > d \text{ (the sum } s \text{ is too large),}$$
$$s + \sum_{j=i+1}^{n} a_j < d \text{ (the sum } s \text{ is too small).}$$

General Remarks

From a more general perspective, most backtracking algorithms fit the following escription. An output of a backtracking algorithm can be thought of as an *n*-tuple $(x_1, x_2, \ldots, x_n)$ where each coordinate *xi* is an element of some finite lin early ordered set *Si* . For example, for the *n*-queens problem, each *Si* is the set of integers (column numbers) 1 through *n.*

A backtracking algorithm generates, explicitly or implicitly, a state-space tree; its nodes represent partially constructed tuples with the first i coordinates defined by the earlier actions of the algorithm. If such a tuple $(x_1, x_2, \ldots, x_i)$ is not a solution, the algorithm finds the next element in S_{i+1} that is consistent with the values of $((x_1, x_2, \ldots, x_i)$ and the problem's constraints, and adds it to the tuple as its (i + 1)st coordinate. If such an element does not exist, the algorithm backtracks to consider the next value of xi, and so on.

ALGORITHM *Backtrack*(*X* [1..*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*[1..*i*]
else //see Problem this section
for each element $x \in Si+1$ consistent with *X*[1..*i*] and the constraints **do**
$X[i + 1] \leftarrow x$
Backtrack(*X*[1..*i* + 1])

5.10 BRANCH AND BOUND

An optimization problem seeks to minimize or maximize some objective function, usually subject to some constraints. 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 (e.g., a Hamiltonian circuit in the travelling salesman problem or a subset of items whose total weight does not exceed the knapsack's capacity in the knapsack problem), whereas an ***optimal solution*** is a feasible solution with the best value of the objective function (e.g., the shortest Hamiltonian circuit or the most valuable subset of items that fit the knapsack).

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 function1 on any solution that can be obtained by adding further components to the partially constructed 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 value of the best solution seen so far—i.e., not smaller for a minimization problem and not larger for a maximization problem—the node is nonpromising and can be terminated (some people say the branch is "pruned"). Indeed, 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 (and hence no further choices can be made)—in this case, we compare the value of the objective function for this feasible solution with that of the best solution seen so far and update the latter with the former if the new solution is better.

Some problems can be solved by Branch-and-Bound are:

1. Assignment Problem
2. Knapsack Problem
3. Traveling Salesman Problem

5.11 ASSIGNMENT PROBLEM

Let us illustrate the branch-and-bound approach by applying it to the problem of assigning n people to n jobs so that the total cost of the assignment is as small as possible. An instance of the assignment problem is specified by an $n \times n$ cost matrix C.

$$C = \begin{array}{c} \begin{array}{cccc} \text{job 1} & \text{job 2} & \text{job 3} & \text{job 4} \end{array} \\ \begin{bmatrix} 9 & 2 & 7 & 8 \\ 6 & 4 & 3 & 7 \\ 5 & 8 & 1 & 8 \\ 7 & 6 & 9 & 4 \end{bmatrix} \end{array} \begin{array}{l} \text{person } a \\ \text{person } b \\ \text{person } c \\ \text{person } d \end{array}$$

We have to find a lower bound on the cost of an optimal selection without actually solving the problem. We can do this by several methods. For example, it is clear that the cost of any solution, including an optimal one, cannot be smaller than the sum of the smallest elements in each of the matrix's rows. For the instance here, this sum is 2 + 3+ 1+ 4 = 10. It is important to stress that this is not the cost of any legitimate selection (3 and 1 came from the same column of the matrix); it is just a lower bound on the cost of any legitimate selection. We can and will apply the same thinking to partially constructed solutions. For example, for any legitimate selection that selects 9 from the first row, the lower bound will be 9 + 3 + 1+ 4 = 17.

It is sensible to consider a node with the best bound as most promising, although this does not, of course, preclude the possibility that an optimal solution will ultimately belong to a different branch of the state-space tree. This variation of the strategy is called the ***best-first branch-and-bound***.

The lower-bound value for the root, denoted *lb*, is 10. The nodes on the first level of the tree correspond to selections of an element in the first row of the matrix, i.e., a job for person *a as shown in* Figure 5.15.

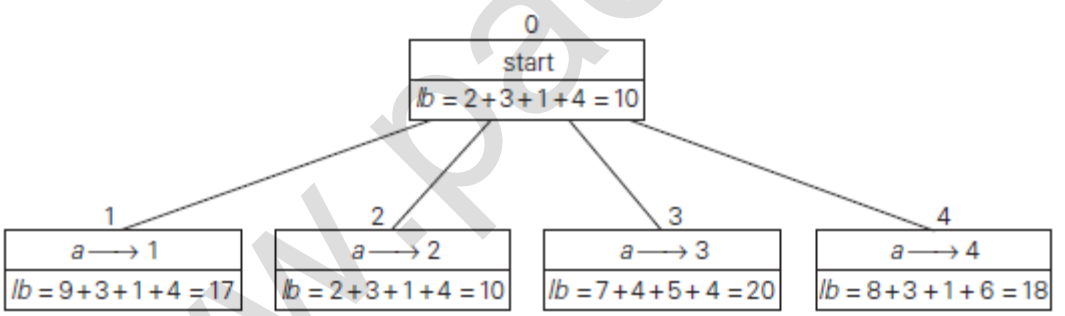


FIGURE 5.15 Levels 0 and 1 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm. The number above a node shows the order in which the node was generated. A node's fields indicate the job number assigned to person *a* and the lower bound value, *lb,* for this node.

.

So we have four live leaves (promising leaves are also called ***live***) —nodes 1 through 4— that may contain an optimal solution. The most promising of them is node 2 because it has the smallest lowerbound value. Following our best-first search strategy, we branch out from that node first by considering the three different ways of selecting an element from the second row and not in the second column—the three different jobs that can be assigned to person *b* (Figure 5.16).

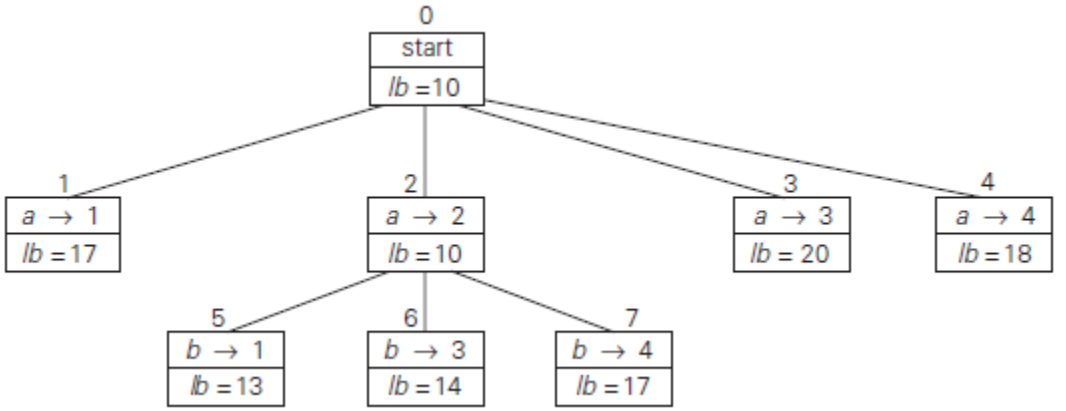


FIGURE 5.16 Levels 0, 1, and 2 of the state-space tree for the instance of the assignment problem being solved with the best-first branch-and-bound algorithm.

Of the six live leaves—nodes 1, 3, 4, 5, 6, and 7—that may contain an optimal solution, we again choose the one with the smallest lower bound, node 5. First, we consider selecting the third column's element from *c*'s row (i.e., assigning person *c* to job 3); this leaves us with no choice but to select the element from the fourth column of *d*'s row (assigning person *d* to job 4). This yields leaf 8 (Figure 5.17), which corresponds to the feasible solution $\{a \rightarrow 2, b \rightarrow 1, c \rightarrow 3, d \rightarrow 4\}$ with the total cost of 13. Its sibling, node 9, corresponds to the feasible solution $\{a \rightarrow 2, b \rightarrow 1, c \rightarrow 4, d \rightarrow 3\}$ with the total cost of 25. Since its cost is larger than the cost of the solution represented by leaf 8, node 9 is simply terminated. (Of course, if its cost were smaller than 13, we would have to replace the information about the best solution seen so far with the data provided by this node.)

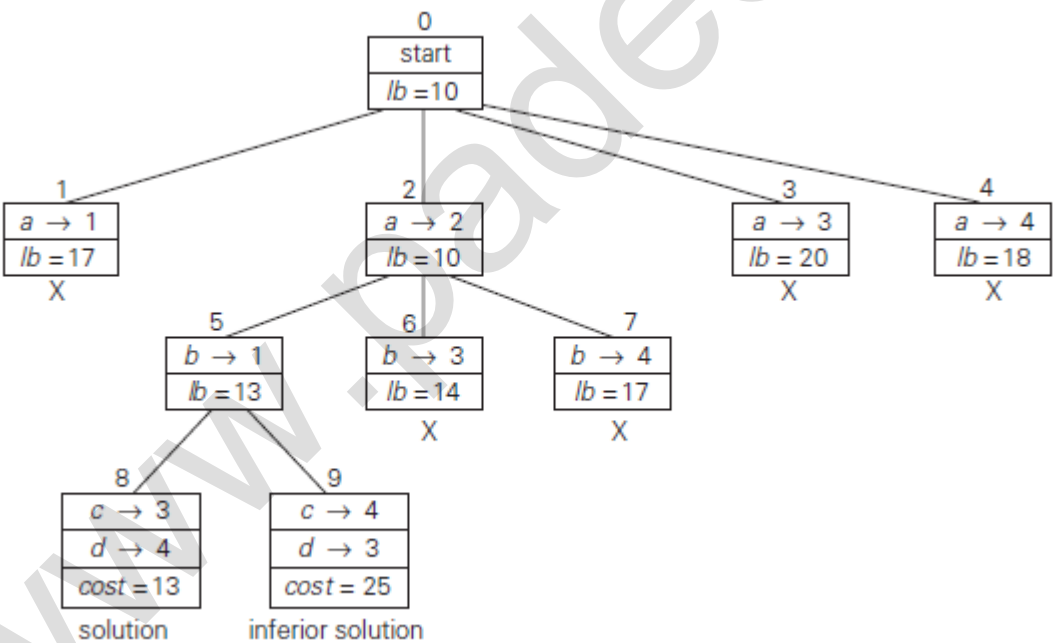


FIGURE 5.17 Complete state-space tree for the instance of the assignment problem solved with the best-first branch-and-bound algorithm.

Now, as we inspect each of the live leaves of the last state-space tree—nodes 1, 3, 4, 6, and 7 in Figure 5.17—we discover that their lower-bound values are not smaller than 13, the value of the best selection seen so far (leaf 8). Hence, we terminate all of them and recognize the solution represented by leaf 8 as the optimal solution to the problem.

5.12 KNAPSACK PROBLEM

Let us now discuss how we can apply the branch-and-bound technique to solving the knapsack problem. Given n items of known weights w_i and values v_i , $i = 1, 2, \ldots, n$, 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:

$$v1/w1 \geq v2/w2 \geq \ldots \geq vn/wn.$$

It is natural to structure the state-space tree for this problem as a binary tree constructed as follows. Each node on the ith level of this tree, $0 \leq i \leq n$, represents all the subsets of n items that include a particular selection made from the first i ordered items. This particular selection is uniquely determined by the path from the root to the node: a branch going to the left indicates the inclusion of the next item, and a branch going to the right indicates its exclusion. 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.

Item	Weight	value	value / weight	capacity
1	4	$40	10	$W = 10$
2	7	$42	6	
3	5	$25	5	
4	3	$12	4	
	w=19	v=119	$v_{i+1}/w_{i+1}=25$	

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}:

$$ub = v + (W - w)(v_{i+1}/w_{i+1}).$$
$$= 0+(10-0)\ (10)$$
$$= 100$$

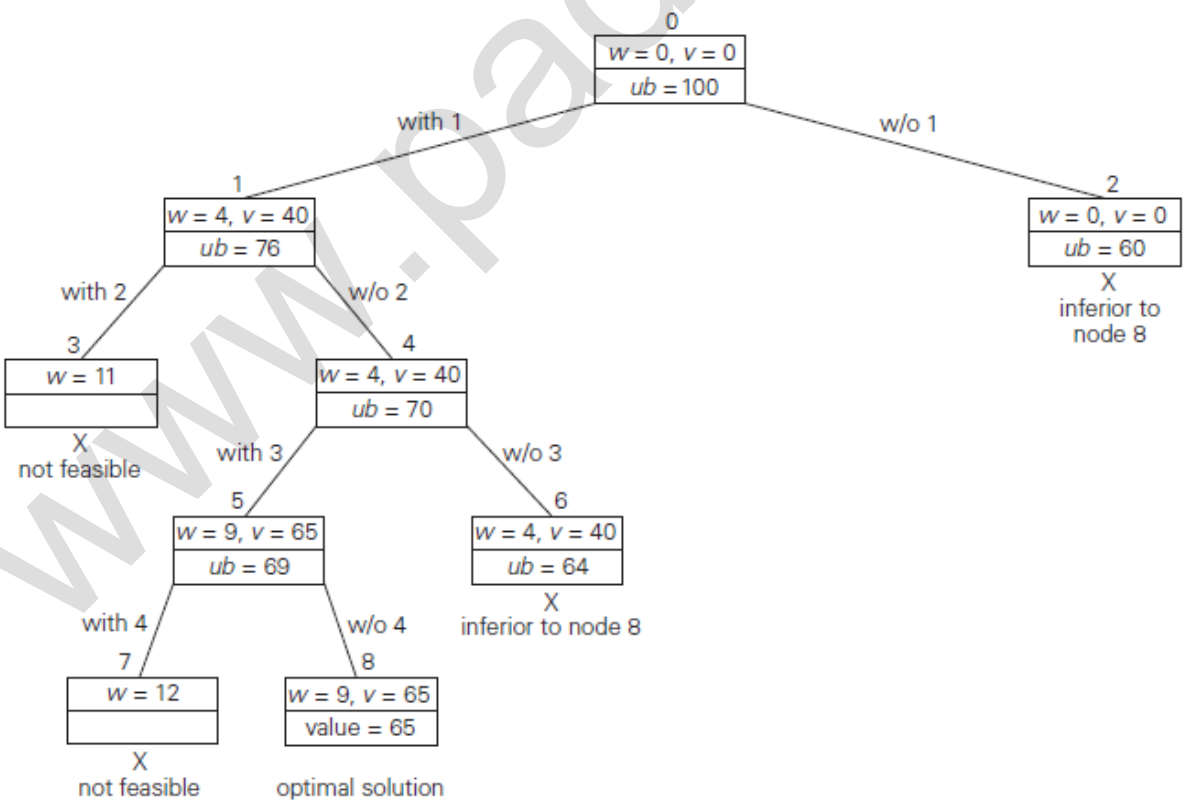


FIGURE 5.18 State-space tree of the best-first branch-and-bound algorithm for the instance of the knapsack problem.

At the root of the state-space tree (see Figure 5.18), no items have been selected as yet. 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 formula (12.1) 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, respectively; 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, respectively.

Since the total weight w of every subset represented by node 3 exceeds the knapsack's capacity, node 3 can be terminated immediately. Node 4 has the same values of w and v as its parent; the upper bound ub is equal to $40 + (10 - 4) * 5 = \$70$. Selecting node 4 over node 2 for the next branching (why?), we get nodes 5 and 6 by respectively including and excluding item 3. The total weights and values as well as the upper bounds for these nodes are computed in the same way as for the preceding nodes. Branching from node 5 yields node 7, which represents no feasible solutions, and node 8, which represents just a single subset {1, 3} of value \$65. The remaining live nodes 2 and 6 have smaller upper-bound values than the value of the solution represented by node 8. Hence, both can be terminated making the subset {1, 3} of node 8 the optimal solution to the problem.

Solving the knapsack problem by a branch-and-bound algorithm has a rather unusual characteristic. Typically, internal nodes of a state-space tree do not define a point of the problem's search space, because some of the solution's components remain undefined. If we had done this for the instance investigated above, we could have terminated nodes 2 and 6 before node 8 was generated because they both are inferior to the subset of value \$65 of node 5.

5.13 TRAVELING SALESMAN PROBLEM

We will be able to apply the branch-and-bound technique to instances of the travelling 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. But there is a less obvious and more informative lower bound for instances with symmetric matrix D, which does not require a lot of work to compute. It is not difficult to show (Problem 8 in this section's exercises) that we can compute a lower bound on the length l of any tour as follows. For each city i, $1 \leq i \leq n$, find the sum si 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:

$$lb = \lceil s/2 \rceil$$

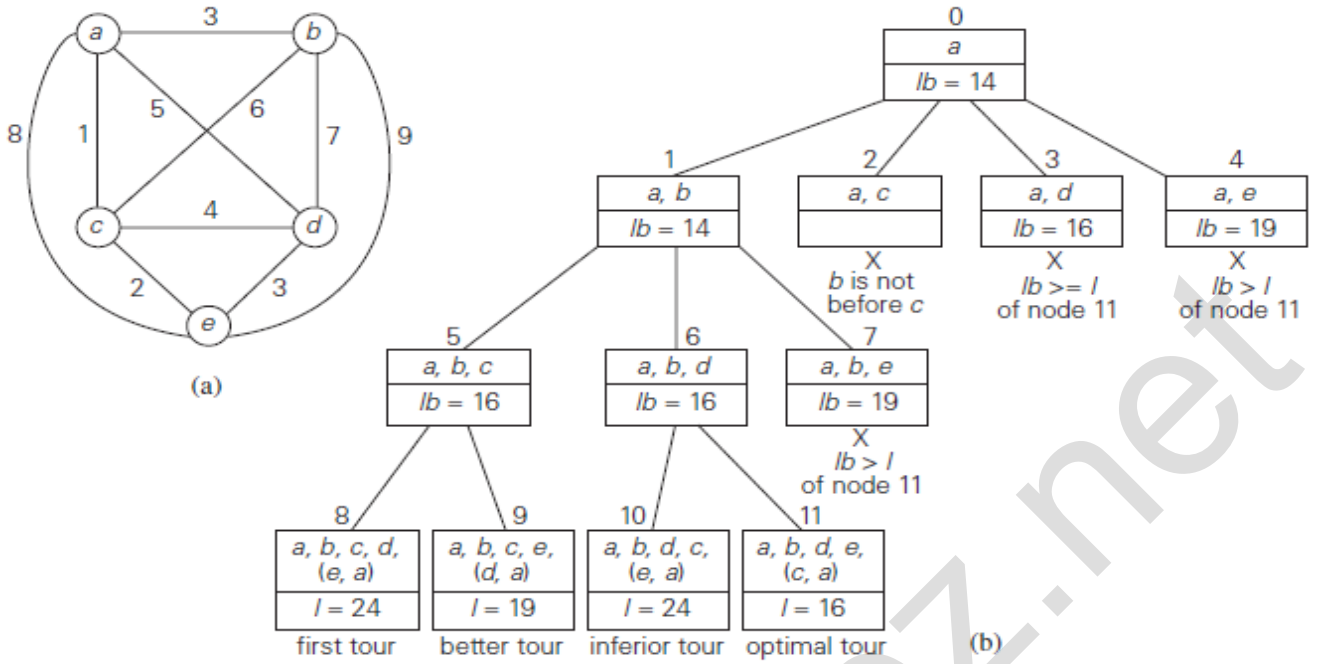


FIGURE 5.19 (a)Weighted graph. (b) State-space tree of the branch-and-bound algorithm to find a shortest Hamiltonian circuit in this graph. The list of vertices in a node specifies a beginning part of the Hamiltonian circuits represented by the node.

For example, for the instance in Figure and above formula yields
$lb = \lceil[(1 + 3) + (3 + 6) + (1 + 2) + (3 + 4) + (2 + 3)]/2\rceil = 14.$

Moreover, for any subset of tours that must include particular edges of a given graph, we can modify lower bound accordingly. For example, for all the Hamiltonian circuits of the graph in Figure that must include edge *(a, d)*, we get the following lower bound by summing up the lengths of the two shortest edges incident with each of the vertices, with the required inclusion of edges *(a, d)* and *(d, a)*:

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

We now apply the branch-and-bound algorithm, with the bounding function given by formula, to find the shortest Hamiltonian circuit for the graph in Figure 5.19a. To reduce the amount of potential work. First, without loss of generality, we can consider only tours that 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. The state-space tree tracing the algorithm's application is given in Figure 5.19b.

5.14 APPROXIMATION ALGORITHMS FOR NP HARD PROBLEMS

Now we are going to discuss a different approach to handling difficult problems of combinatorial optimization, such as the **travelling salesman problem and the knapsack problem.** The decision versions of these problems are *NP*-complete. Their optimization versions 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, and there are serious theoretical reasons to believe that such algorithms do not exist.

Approximation algorithms run a gamut in level of sophistication; most of them are based on some problem-specific heuristic. A ***heuristic*** is a common-sense rule drawn from experience rather than from a mathematically proved assertion. For example, going to the nearest unvisited city in the travelling salesman problem is a good illustration of this notion.

Of course, if we use an algorithm whose output is just an approximation of the actual optimal solution, we would like to know how accurate this approximation is. We can quantify the accuracy of an approximate solution s_a to a problem of ***minimizing*** some function f by the size of the relative error (***re***) of this approximation,

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

where s^* is an exact solution to the problem. Alternatively, $re(sa) = f(sa)/f(s^*) - 1$, 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 usually 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. Obviously, the closer $r(s_a)$ is to 1, the better the approximate solution is. For most instances, however, we cannot compute the accuracy ratio, because we typically do not know $f(s^*)$, the true optimal value of the objective function. Therefore, our hope should lie in obtaining a good upper bound on the values of $r(s_a)$. This leads to the following definitions.

A polynomial-time approximation algorithm is said to be a ***c approximation algorithm***, where $c \geq 1$, if the accuracy ratio of the approximation it produces does not exceed c for any instance of the problem in question: $r(s_a) \leq c$.

The best (i.e., the smallest) value of c for which inequality holds for all instances of the problem is called the ***performance ratio*** of the algorithm and denoted R_A.

The performance ratio serves as the principal metric indicating the quality of the approximation algorithm. We would like to have approximation algorithms with RA as close to 1 as possible. Unfortunately, as we shall see, some approximation algorithms have infinitely large performance ratios (RA = ∞). This does not necessarily rule out using such algorithms, but it does call for a cautious treatment of their outputs.

Approximation Algorithms for NP Hard Problems are:

- Traveling salesman problem (tsp)
- Knapsack problem

5.15 TRAVELING SALESMAN PROBLEM (APPROXIMATION ALGORITHM)

Greedy Algorithms for the TSP The simplest approximation algorithms for the traveling salesman problem are based on the greedy technique. We will discuss here two such algorithms.

1. Nearest-neighbor algorithm
2. Minimum-Spanning-Tree–Based Algorithms

NEAREST-NEIGHBOR ALGORITHM

The following well-known greedy algorithm is based on the ***nearest-neighbor*** heuristic: always go next to the nearest unvisited city.

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 1 For the instance represented by the graph in Figure 5.20, 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.

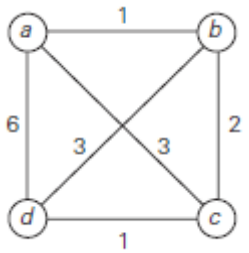


FIGURE 5.20 Instance of the traveling salesman problem.

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 s_a is 25% longer than the optimal tour s^*).

Multifragment-heuristic algorithm

Another natural greedy algorithm for the traveling salesman problem considers it as the problem of finding a minimum-weight collection of edges in a given complete weighted graph so that all the vertices have **degree 2.**

Step 1 Sort the edges in increasing order of their weights. (Ties can be broken arbitrarily.) Initialize the set of tour edges to be constructed to the empty set.

Step 2 Repeat this step n times, where n is the number of cities in the instance being solved: add the next edge on the sorted edge list to the set of tour edges, provided this addition does not create a vertex of degree 3 or a cycle of length less than n; otherwise, skip the edge.

Step 3 Return the set of tour edges.

As an example, applying the algorithm to the graph in Figure 5.20 yields {*(a, b), (c, d), (b, c), (a, d)*}. This set of edges forms the same tour as the one produced by the nearest-neighbor algorithm. In general, the multifragment-heuristic algorithm tends to produce significantly better

tours than the nearest-neighbor algorithm, as we are going to see from the experimental data quoted at the end of this section. But the performance ratio of the multifragment-heuristic algorithm is also unbounded, of course.

There is, however, a very important subset of instances, called ***Euclidean***, for which we can make a nontrivial assertion about the accuracy of both the nearestneighbor and multifragment-heuristic algorithms. These are the instances in which intercity distances satisfy the following natural conditions:

- ***triangle inequality*** $d[i, j] \leq d[i, k] + d[k, j]$ for any triple of cities *i, j,* and *k* (the distance between cities *i* and *j* cannot exceed the length of a two-leg path from *i* to some intermediate city *k* to *j*)
- ***symmetry*** $d[i, j] = d[j, i]$ for any pair of cities *i* and *j* (the distance from *I* to *j* is the same as the distance from *j* to *i*)

MINIMUM-SPANNING-TREE–BASED ALGORITHMS

There are approximation algorithms for the travelling salesman problem that exploit a connection between Hamiltonian circuits and spanning trees of the same graph. Since removing an edge from a Hamiltonian circuit yields a spanning tree, we can expect that the structure of a minimum spanning tree provides a good basis for constructing a shortest tour approximation. Here is an algorithm that implements this idea in a rather straightforward fashion.

Twice-around-the-tree algorithm

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. (This can be done by a DFS traversal.)

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. (This step is equivalent to making shortcuts in the walk.) The vertices remaining on the list will form a Hamiltonian circuit, which is the output of the algorithm.

EXAMPLE 2 Let us apply this algorithm to the graph in Figure 5.21a. The minimum spanning tree of this graph is made up of edges *(a, b), (b, c)*, *(b, d)*, and *(d, e)* (Figure 5.21b). 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*) yields the Hamiltonian circuit *a, b, c, d, e, a* of length 39.

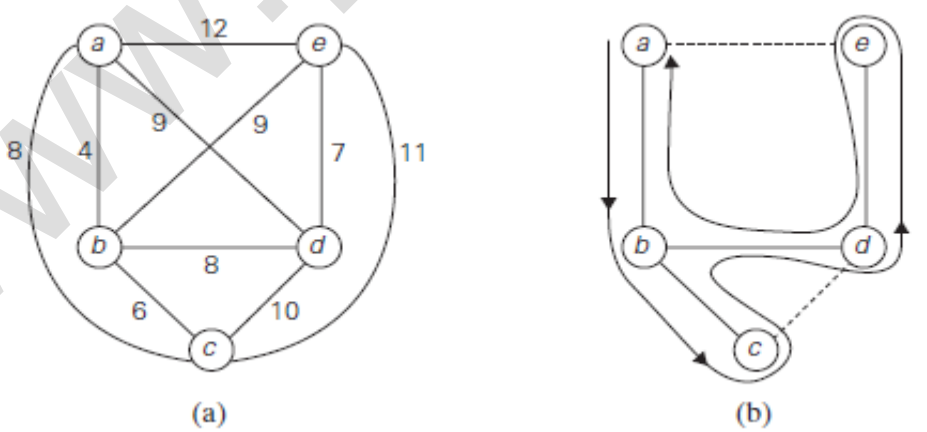


FIGURE 5.21 Illustration of the twice-around-the-tree algorithm. (a) Graph. (b) Walk around the minimum spanning tree with the shortcuts.

5.16 KNAPSACK PROBLEM (APPROXIMATION ALGORITHM)

The knapsack problem is one well-known *NP*-hard problem. Given n items of known weights $w_1, \ldots, w_n$ and values $v_1, \ldots, v_n$ and a knapsack of weight capacity W, find the most valuable subset of the items that fits into the knapsack.

GREEDY ALGORITHMS FOR THE KNAPSACK PROBLEM

We can think of several greedy approaches to this problem. One is to 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 be used efficiently. we find a greedy strategy that takes into account both the weights and values by computing the value-to-weight ratios v_i/w_i, $i = 1, 2, \ldots, n$, and selecting the items in decreasing order of these ratios. Here is the algorithm based on this greedy heuristic.

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 Step 1.(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 and proceed to the next item; otherwise, just proceed to the next item.

EXAMPLE 1 Let us consider the instance of the knapsack problem with the knapsack capacity 10 and the item information as follows:

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

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

Item	weight	value	value / weight	capacity
1	4	$40	10	$W = 10$
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. So the total items value in knapsack is **$65**.

GREEDY ALGORITHM FOR THE CONTINUOUS KNAPSACK PROBLEM

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

Step 2 Sort the items in nonincreasing order of the ratios computed in Step 1. (Ties can be broken arbitrarily.)

Step 3 Repeat the following operation until the knapsack is filled to its full capacity or no item is left in the sorted list: if the current item on the list fits into the knapsack in its

entirety, take it and proceed to the next item; otherwise, take its largest fraction to fill the knapsack to its full capacity and stop.

EXAMPLE 2 A small example of an approximation scheme with $k = 2$ is provided. The algorithm yields {1, 3, 4}, which is the optimal solution for this instance.

Item	weight	value	value / weight	capacity
1	4	$40	10	$W = 10$
2	7	$42	6	
3	5	$25	5	
4	1	$4	4	

subset	Added items	value
{}	1, 3, 4	$69
{1}	3, 4	$69
{2}	4	$46
{3}	1, 4	$69
{4}	1, 3	$69
{1, 2}	Not feasible	
{1, 4}	4	$69
{1, 4}	3	$69
{2, 3}	Not feasible	
{2, 4}	-	$46
{3, 4}	1	$69

For each of those subsets, it needs $O(n)$ time to determine the subset's possible extension. Thus, the algorithm's efficiency is in $O(kn^{k+1})$. Note that although it is polynomial in n, the time efficiency of Sahni's scheme is exponential in k. More sophisticated approximation schemes, called ***fully polynomial schemes***, do not have this shortcoming.