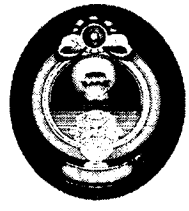
# COMPUTER ALGORITHMS

**MCA**

**Second Year**

**Paper: IV**

**DMCA: 204**



**ACHARYA NAGARJUNA UNIVERSITY**

Centre for Distance Education

VIKAS® PUBLISHING HOUSE PVT LTD
A-22, Sector-4, Noida - 201301 (UP)
Phone: 0120-4078900 • Fax: 0120-4078999
Regd. Office: 576, Masjid Road, Jangpura, New Delhi 110 014
• Website: www.vikaspublishing.com • Email: helpline@vikaspublishing.com

# SYLLABI-BOOK MAPPING TABLE

## Computer Algorithms

| Syllabi | Mapping in Book |
|---|---|
| **UNIT-I**<br>Introduction, Elementary Data Structures | **Unit 1:** Algorithm Complexity and Recurrences<br>**(Pages 3-25)** |
| **UNIT-II**<br>Divide & Conquer, the Greedy Method | **Unit 2:** Divide and Conquer Strategy<br>**(Pages 27-42);**<br>**Unit 3:** Introduction to Greedy Method<br>**(Pages 43-56)** |
| **UNIT-III**<br>Dynamic Programming | **Unit 4:** Dynamic Programming<br>**(Pages 57-72)** |
| **UNIT-IV**<br>Basic Traversal & Search Techniques, Back Tracking | **Unit 5:** Backtracking and Branch-and-Bound Techniques<br>**(Pages 73-94)** |
| **UNIT-V**<br>Branch & Bound | **Unit 6:** Lower Bound Theory<br>**(Pages 95-112)** |

# CONTENTS

## UNIT 6 LOWER BOUND THEORY 95-112

# INTRODUCTION

NOTES

You know that an algorithm as a general term is defined as the finite set of computational steps that helps in accomplishing the desired result. The steps included in the algorithm are used to transform the inputs of the algorithm to the required output. Euclid's algorithm is one such famous algorithm that was used to compute the common divisor of two given integer values. However, computer algorithms deal with the analysis and design of those problems that are related with computer science. 'Algorithmic' are computer algorithms that are handled by computer science.

Analysis and design are the two major features of computer algorithms. The term algorithm analysis, also known as complexity analysis or efficiency analysis, is used to evaluate the performance of algorithms. The complexity analysis is used to determine the amount of resources needed by the algorithm for its successful execution. The performance of the algorithms can be measured in terms of their time and space complexity. The time complexity of an algorithm is generally denoted by a function of time, and that depends upon the input parameters given to the algorithm. Time complexity also depends upon the number of steps in the algorithm. It is denoted by different notations such as Big Oh ($O$), Big Theta ($\theta$) and Big Omega ($\Omega$). Therefore, efficient time complexity of an algorithm helps in executing the algorithm in a small period of time. On the other hand, space complexity of an algorithm deals with the amount of memory space needed by the algorithm for its execution. However, the time complexity of the algorithm can vary with the size of the problem that has to be solved.

The term algorithm design deals with the development of algorithms in terms of the pseudo language. The different paradigms that can be used in designing algorithms are divide and conquer, dynamic programming, greedy method, and backtracking.

This book, *Computer Algorithms*, covers the fundamentals of algorithm analysis and designing as well as some advanced concepts, such as the lower bound theory, that help in identifying the most efficient algorithm. This book has been written in a simple, concise and self-learning style. It is hoped that you will gain sufficient knowledge on the subject.

This book is divided into six units. The first unit discusses the basics of ADA which are the space and time complexity, and the remaining five units deal with the development of algorithms for different problems using different design paradigms. In each unit, we begin with the introduction of the topic; then, we outline the learning objectives; and then we present the details of the contents. At the end of each unit, we have provided a summary for quick recollection. Finally, you will find the 'Question and Exercises' and 'Check Your Progress' sections which will help you to understand the topics better.

NOTES

# UNIT 1 ALGORITHM COMPLEXITY AND RECURRENCES

## Structure



## 1.0 INTRODUCTION

In this unit, you will learn about the basic concepts of algorithms which act as tools for constructing logic for a given computation problem. There are various design paradigms that help a programmer to develop efficient algorithms. These design paradigms are as follows:

- Divide and conquer
- Dynamic programming
- Greedy method
- Backtracking

The efficiency and performance of an algorithm can be measured by analysing the amount of memory and computing time required for its execution. In this unit, you will also learn about various asymptotic notations that are used to describe the running time of an algorithm. These asymptotic notations are as follows:

- Θ-notation
- O-notation
- Ω-notation
- o-notation
- ω-notation

You will also learn about recurrence relations which are generalized equations for describing the running time of an algorithm. A programmer can use either of three methods to solve these equations: recursion tree method, master method, and iteration method.

NOTES

## 1.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Explain the basic concepts and properties of algorithms
- Analyse time and space complexities of an algorithm
- Describe the various asymptotic notations for describing the running time of an algorithm
- Solve the various recurrence relations by using one of the following methods:
    - o Recursion tree method
    - o Master method
    - o Iteration

## 1.2 INTRODUCTION TO ALGORITHMS

An algorithm is a sequence of steps used to perform a particular task successfully. When this sequence of steps is performed on sample data, the result thus obtained is known as output of the algorithm. An algorithm can also be defined as a tool that provides a step by step procedure for solving the given problem. The algorithms are generally expressed in natural language and pseudocode form. They can also be implemented as computer programs in the form of neural network, as an electrical circuit or in a mechanical device. The sequence of steps in an algorithm can also be represented in a graphical form, which is known as a flowchart.

An algorithm must have the following properties:

- **Input**: The algorithm must be able to accept a set of values as an input. The input provides the basic data for computation.
- **Output**: The algorithm must be able to generate a set of values as an output. The algorithm generates the output by performing the specific operations on the input.
- **Finiteness**: The algorithm must terminate after a finite number of steps. The algorithm must contain a certain steps that result to the termination of the algorithm execution.
- **Definiteness**: The steps in the algorithm must be properly defined and easily understandable. Execution of each step must provide a pre-defined result.

An algorithm is considered to be correct, if it generates the correct output for every input. On the other hand, an algorithm is considered to be incorrect if it terminates by generating an incorrect output for any input or does not terminate at all.

Algorithms help a programmer to determine whether a given solution will provide correct result or not. It is possible that two or more algorithms are written for solving a given problem. These algorithms will provide the same output, but may involve different number and sequence of steps. An algorithm, which includes few simple steps, will be executed more efficiently than an algorithm

**NOTES**

that involves large number of complex steps. Efficient algorithms save both computer memory and computing time.

Consider the problem to find the largest number among all the given numbers. The problem can be defined as follows:

```
Algorithm: Largest Number
      Input: A non-empty list of L numbers, N0, N1, N2, ... NL
      Output: The largest number, NLAR, where NLAR > Ni for
0<i<L
```

The following code shows the algorithm to find the largest number:

```
1. Start Algorithm
2. Accept a list of L numbers, N0, N1, N2, ... NL
3. Set NLAR ← N0, i ← 1
4. While i ≤ L
5.    do if Ni > NLAR, then
6.       NLAR ← Ni
7.       Set i ← i + 1
8. return NLAR
9. End Algorithm
```

The above algorithm accepts L numbers and initializes $N_{LAR}$ with the first number from the list. The steps from 4 to 7 process the list and store the largest number in $N_{LAR}$. The algorithm terminates at step 8 and the returns the result stored in $N_{LAR}$. Therefore, the output of the algorithm is the value stored in $N_{LAR}$.

An algorithm can be classified on the basis of some design paradigms, which are general approaches for constructing efficient algorithms. The commonly used design paradigms for algorithms are as follows:

- **Divide and conquer**: It splits an instance of the given problem into various sub-instances of the same problem. Each sub-instance is solved individually and the solutions of all the sub-instances are combined to generate the solution for the problem. The examples of divide and conquer algorithms includes algorithms for merge sort, quick sort and binary search.
- **Dynamic programming**: It breaks the given problem into various sub-problems and the solution of a sub-problem is used to find the solution of the entire problem. Also, a single sub-problem can be used to solve many different problems. The dynamic programming approach avoids the computation of the solution that has already been computed. Unlike divide and conquer approach, the sub-problems in dynamic programming may overlap each other. The examples of dynamic programming algorithms include algorithms for Fibonacci series, the shortest path problem and checkerboard problem.
- **Greedy method**: It is similar to dynamic programming and divides the given problem into a number of sub-problems. At each step, the best available solution is selected, assuming that it will provide the best solution for the entire problem. The greedy method is not suitable for all

NOTES

types of problems, but it provides the quickest solutions for the problems, such as minimal spanning tree, travelling salesman and graph colouring problem.

- **Backtracking**: It provides solution to the problems having a tree structure. It examines all the available options to find the solution to the given problem. For each option, the tree is traversed in the depth-first pattern. If at any stage of the traversal the alternative does not provide a convincing result, then the algorithm backtracks to the point from where a suitable different alternative can be selected. The examples of backtracking algorithms include algorithms for game playing problem, the shortest path problem and theorem proving systems.

## 1.3 COMPLEXITY OF ALGORITHMS

The efficiency of the algorithms depends on two essential elements, time and space. The complexity of an algorithm is the function that provides the running time and space for data, depending on the size provided by you. Sometimes you may need a time-space trade-off that increases the amount of space to store the data.

Time-space trade-off is a situation in which you can reduce the computation time of a process by increasing the memory. Time-space trade-off is mostly used in algorithms to run the algorithms in a fast manner. Time-space trade-off is generally applied on the data storage in the system. For example, the data stored in uncompressed form takes more space as compared to the compressed data. However, the uncompressed data takes less time to be accessed as compared to the compressed data. Using time-space trade-off, you may be able to reduce the time required for processing the data or increase the time required for processing the data.

For example, P is an algorithm and n is the size of the input data. Then, the time and space that P uses will be the two main measures on which the efficiency of P depends. You can measure the time by counting the number of key operations in searching and sorting algorithms using which you can search and sort the elements of the data.

Thus, the two most important criteria for judging the performance of an algorithm are as follows:

- Space complexity
- Time complexity

### 1.3.1 Space Complexity

Space complexity of an algorithm can be defined as the amount of memory needed by an algorithm for its execution and generating the final output. For any algorithm, the space needed can be calculated as the sum of the following two components:

- **Fixed part**: It is independent of the number and size of the input and the output accepted and generated by the algorithm respectively. It mainly includes space for the code, simple variables and constants. It also includes the space required for component variables, which are of fixed size.

NOTES

- **Variable part**: It includes the space needed by the component variables whose size may vary depending upon the problem instance to be solved. It also includes the space required by the referenced variables and recursion stack. The space requirement for reference variables and recursion stack depends on instance characteristics. It is the number and size of the inputs and outputs of the algorithm.

Thus, the space requirements of an algorithm, A, can be defined as follows:

$S(A) = S_{FIXED} + S_{VAR}$

Where, $S_{FIXED}$ is a constant and $S_{VAR}$ is a variable quantity that represents the space requirement of the algorithm, which is dependent on instance characteristics.

Consider the following algorithm to calculate the sum of three numbers:

```
1. Start Algorithm SUM(A, B, C)
2. Set A ← 10, B ← 4, C ← 6
3. S ← A + B + C
4. Return S
5. End Algorithm
```

In the above algorithm, the problem instance is characterized by pre-defined values of A, B and C. Assuming that the space needed by each of the variables is one word, the total space required by A, B, C and S is four words. Since, the algorithm, SUM, is independent of instance characteristics; therefore the value of $S_{VAR}$ is zero. Hence, the space complexity of SUM is calculated as:

$S(SUM) = S_{FIXED} + S_{VAR} = 4 + 0 = 4$

Consider another algorithm, which takes *N* numbers as input and generates their sum as output. The algorithm to calculate the sum of *N* numbers is as follows:

```
1. Start Algorithm SUM_N(A, N)
2. Accept a list of N numbers, A1, A2, A3, … AN
3. Set S ← 0
4. For I ← 1 to N
5.     Do S ← S + AI
6. Return S
7. Stop Algorithm
```

In the above algorithm, the problem instance is characterized by N, which represents the number of elements to be summed. The value of N will decide the size of the array, A, which stores all the numbers to be summed. Therefore, the space needed by A will be at least N words. It means $S_{VAR} \geq N$. Also, the variables S, I and N will require one word each. The space complexity of SUM_N is calculated as:

$S(SUM\_N) \geq 3 + N$

## 1.3.2 Time Complexity

Time complexity of an algorithm can be defined as the amount of computer time needed an algorithm for its execution. The total time required by an algorithm for its execution includes both compile time and run time. The compile time is independent of instance characteristics; therefore the main task is to estimate the

NOTES

run time of an algorithm. The run time of an algorithm is denoted by $T_A(n)$, where A is the name of the algorithm and n represents input characteristic.

In order to estimate the run time of an algorithm, it is essential to determine the number of various operations such as addition, subtraction, division, multiplication, load and store that would be executed by the algorithm. The following is a generalized expression for calculating $T_A(n)$:

$$T_A(n) = c_A ADD(n) + c_S SUB(n) + c_D DIV(n) + c_M MUL(n) + c_{ST} STORE(n) + \ldots$$

Where, $c_A$, $c_S$, $c_D$, $c_M$, $c_{ST}$ represent the time required for addition, subtraction, division, multiplication and store operations. The number of addition, subtraction, division, multiplication and store operations performed by the algorithm are represented by ADD, SUB, DIV, MUL and STORE.

Consider the following algorithm for insertion sort:

```
1. Start Algorithm INSERTION_SORT(A, N)
2. For I ← 2 to N
3.     Do ITEM ← A_I
4.        J ← I - 1
5.        While J > 0 and A_J > ITEM
6.           Do A_J+1 ← A_J
7.              J ← J - 1
8.        A[J + 1] ← ITEM
9. End Algorithm
```

The run time of the algorithm is the total time taken by all the statements in the algorithm for execution. Table 1.1 lists the cost and run time associated with each step in the INSERTION_ SORT algorithm.

***Table 1.1** Cost and Run Times Estimates of INSERTION_SORT Algorithm*

| Step Number | Cost | Estimated Run Time |
|---|---|---|
| Step 2 | C1 | $N$ |
| Step 3 | C2 | $N-1$ |
| Step 4 | C3 | $N-1$ |
| Step 5 | C4 | $\sum_{I=2}^{N} T_I$ |
| Step 6 | C5 | $\sum_{I=2}^{N} (T_I - 1)$ |
| Step 7 | C6 | $\sum_{I=2}^{N} (T_I - 1)$ |
| Step 8 | C7 | $N-1$ |

In Table 1.1, $T_I$ represents the number of time the condition in the while loop in step 5 will execute for a particular value of I.

The estimated time required for the execution of INSERTION_SORT algorithm is as follows:

$$T(N) = C_1(N) + C_2(N-1) + C_3(N-1) + C_4(\sum_{I=2}^{N} T_I) + C_5(\sum_{I=2}^{N} (T_I - 1)) + C_6(\sum_{I=2}^{N} (T_I - 1)) + C_7(N-1)$$

For a particular input size, the running time of an algorithm may vary depending upon the type of input instance. For example, if an input instance to the INSERTION_SORT algorithm consists of numbers in already sorted order then, the best case running time will be achieved. In the above algorithm, the step number 5 will execute only $N - 1$ times and step number 6 and 7 will never execute. Therefore, the best case run time of INSERTION_SORT algorithm is given as:

$$T(N) = C_1(N) + C_2(N-1) + C_3(N-1) + C_4(N-1) + C_7(N-1)$$

$$\Rightarrow T(N) = (C_1 + C_2 + C_3 + C_4 + C_7)N - (C_2 + C_3 + C_4 + C_7)$$

$$\Rightarrow T(N) = AN + B$$

Where, $A = (C_1 + C_2 + C_3 + C_4 + C_7)$ and $B = (C_2 + C_3 + C_4 + C_7)$. Since, both A and B are constants, therefore, T(N) can be considered as a linear function of N.

The worst case occurs when the numbers in the input instance is sorted in the reverse order. In this case, each element $A_I$ is compared with all the elements in the sorted sub-array $A_1, A_2, \ldots A_{I-1}$. Therefore, $T_I = I$ for $I = 2, 3, \ldots, N$. Also,

$$\sum_{I=2}^{N} I = \frac{N(N+1)}{2} - 1$$

$$\sum_{I=2}^{N} I - 1 = \frac{N(N-1)}{2}$$

Therefore, the worst case running time of the INSERTION_SORT algorithm is as follows:

$$T(N) = C_1(N) + C_2(N-1) + C_3(N-1) + C_4(\frac{N(N+1)}{2} - 1) + C_5(\frac{N(N-1)}{2}) + C_6(\frac{N(N-1)}{2}) + C_7(N-1)$$

$$\Rightarrow T(N) = (\frac{C_4}{2} + \frac{C_5}{2} + \frac{C_6}{2})N^2 + (C_1 + C_2 + C_3 + \frac{C_4}{2} - \frac{C_5}{2} - \frac{C_6}{2} + C_7)N - (C_2 + C_3 + C_4 + C_7)$$

For a particular input size N, the worst case running time, gives an upper bound on run time. This means that the algorithm will not take more than the worst case run time for its execution.

**NOTES**

**CHECK YOUR PROGRESS**

1. What is an algorithm?
2. What is meant by finiteness of an algorithm?
3. What is space complexity of an algorithm?
4. Give the generalized expression for the calculation of the time complexity of an algorithm.

NOTES

## 1.4 ASYMPTOTIC NOTATIONS

Asymptotic notations are used to represent asymptotic running time of algorithms. These notations are defined in terms of function T(n) related to a set of natural numbers, 1, 2, 3, 4.....n. The asymptotic notations are mainly used in representing the worst-case running time required for the execution of an algorithm. The basic asymptotic notations used for defining the complexity of algorithms are as follows:

- Θ-notation
- O-notation
- Ω-notation
- o-notation
- ω-notation

### 1.4.1 Θ-Notation

The Θ-notation is used to represent the worst case running time of an algorithm. To represent a Θ-notation for a given function g(n), Θ(g(n)) is used. This expression is used to represent a set of functions. Therefore, Θ-notation for a particular g(n) is defined as follows:

Θ(g(n)) = {f(n): there exits non-negative constants $a_1$, $a_2$ and $n_0$ such that f(n) lies between a1g(n) and a2g(n), i.e. $0 \leq a_1 g(n) \leq f(n) \leq a_2 g(n)$ for all $n \geq n_0$}

Since, Θ(g(n)) is a set, therefore you can write f(n) ∈ Θ(g(n)), which specifies that f(n) is a member of Θ(g(n)). Figure 1.1 shows the graphical representation of Θ-notation.

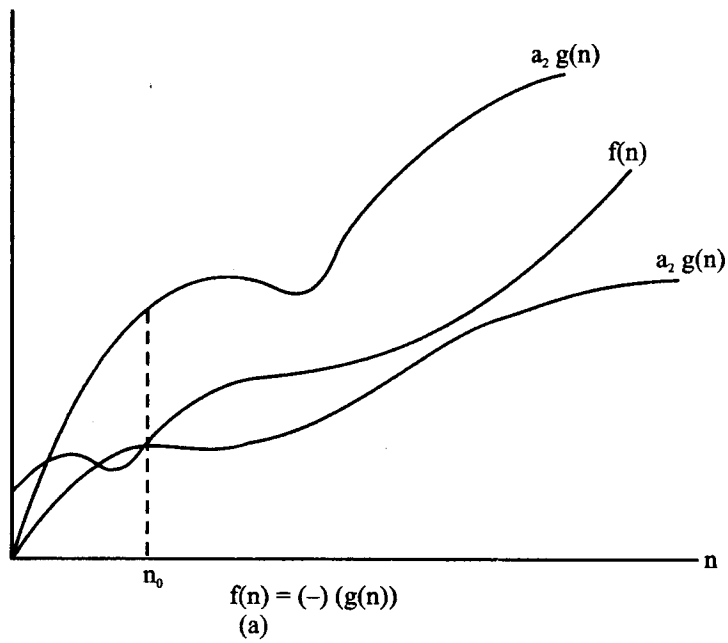



***Figure 1.1*** *Graphical Representation of Θ-Notation*

In Figure 1.1, function f(n), whose Θ-notation is given as Θ(g(n)), lies between a1g(n) and a2g(n). The values of n are represented at the right of $n_0$ and the value

NOTES

of f(n) lies at or above a1g(n) and at or below a2g(n). This means for all values of n, which is greater than $n_0$, the function f(n) is equal to g(n) within a constant factor. Thus, you can say that g(n) is asymptotically tight bound for the function f(n).

An essential condition for the definition of Θ(g(n)) is that both the function f(n) and g(n) must be asymptotically positive, which means that the functions must be positive for all n that are sufficiently large.

**Example 1**: Determine Θ(g(n)) for the function f(n) = 5n + 3.

**Solution**: According to the definition of Θ-notation:

$f(n) = \Theta(g(n))$ if $0 \leq a_1 g(n) \leq f(n) \leq a_2 g(n)$ for all $n \geq n_0$

Here, $5n + 3 \geq 5n$, for $n \geq 3$

And, $5n + 3 \leq 6n$, for $n \geq 3$

$\Rightarrow 5n \leq 5n + 3 \leq 6n$, for $n \geq 3$

And, $a_1 = 5$, $a_2 = 6$, $n_0 = 3$ and $g(n) = n$

Therefore, $f(n) = \Theta(g(n))$

$= \Theta(n)$

**Example 2**: Determine the Θ notation for the following functions:

1. $f(n) = 10n + 4$
2. $f(n) = 16n^2 + 7n + 3$
3. $f(n) = 5 * 3^n + n^2$
4. $f(n) = 2 \log n + 10$
5. $f(n) = n^3 + \log n + 10$

**Solution**:

1. Consider the function $f(n) = 10n + 4$

Here, $g(n) = n$

Therefore, $f(n) = \Theta(g(n)) = \Theta(n)$

2. Consider the function $f(n) = 16n^2 + 7n + 3$

Here, $g(n) = n^2$

Therefore, $f(n) = \Theta(n^2)$

3. Consider the function $f(n) = 5 * 3^n + n^2$

Here, $g(n) = 3^n$

Therefore, $f(n) = \Theta(3^n)$

4. Consider the function $f(n) = 2 \log n + 10$

Here, $g(n) = \log n$

Therefore, $f(n) = \Theta(\log n)$

5. Consider the function $f(n) = n^3 + \log n + 10$

NOTES

Here, $g(n) = n^3$

Therefore, $f(n) = \Theta(n^3)$

## 1.4.2 O-Notation

O-notation provides upper boundary constraints over a function. For a function h(n), the O-notation, which is denoted by O(h(n)), is defined as:

$O(h(n)) = f(n)$

Where f(n) is a function, which is defined as:

$0 \leq f(n) \leq c\ h(n)$, for all $n \geq n_0$

and c and $n_0$ are positive integers.

Figure 1.2 shows the O-notation of a function.

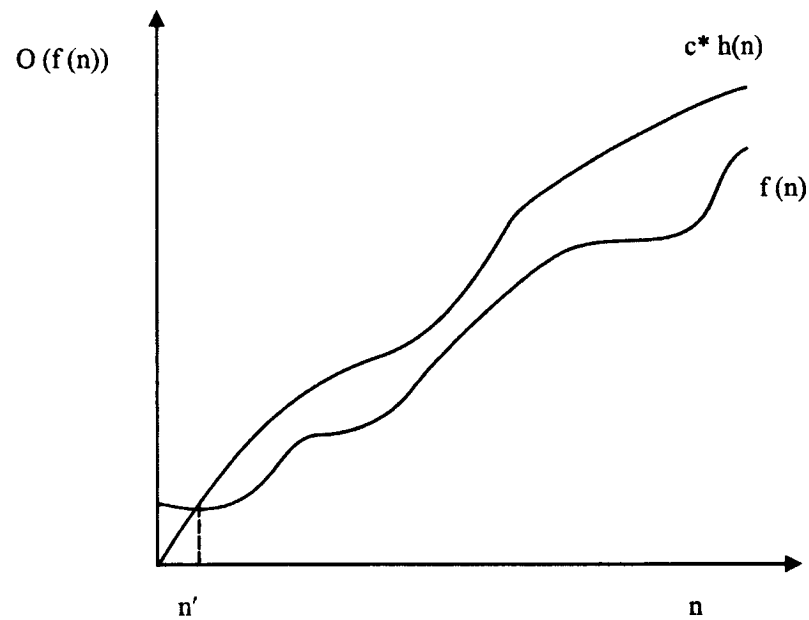



***Figure 1.2** Graphical Representation of O Notation of a Function*

Figure 1.2 shows the O notation of the h(n) function, which ensures that the f(n) function never exceeds beyond c h(n), for all the values of n, which are greater than or equal to $n_0$.

**Example 3**: Determine the O notation for the function $f(n) = 4n + 3$.

**Solution**: According to the definition of O(h(n)),

$f(n) = O(g(n))$ if $f(n) \leq c\ h(n)$ for all $n \geq n_0$

Here, $4n + 3 \leq 5n$, for $n \geq 3$

$\Rightarrow a = 5$, $n_0 = 3$ and $h(n) = n$

Therefore, $f(n) = O(h(n))$

$= \Theta(n)$

**Example 4**: Determine the O notation for the following functions:

1. $f(n) = 110n + 20$

2. $f(n) = 21n^2 + 9n$
3. $f(n) = 6 * 3^n + n^2$
4. $f(n) = 3n + n$

**Solution:**

NOTES

1. Consider the function $f(n) = 110n + 20$

Here, $110n + 2 \leq 111n$, for $n \geq 2$

$\Rightarrow a = 111$, $n_0 = 3$ and $h(n) = n$

Therefore, $f(n) = O(h(n)) = O(n)$

2. Consider the function $f(n) = 21n^2 + 9n$

Here, $21n^2 + 5n \leq 22n^2$, for $n \geq 5$

$\Rightarrow a = 22$, $n_0 = 5$ and $h(n) = n^2$

Therefore, $f(n) = O(n^2)$

3. Consider the function $f(n) = 6 * 3^n + n^2$

Here, $6 * 3^n + n^2 \leq 7 * 3^n$, for $n \geq 1$

$\Rightarrow a = 7$, $n_0 = 1$ and $h(n) = 3^n$

Therefore, $f(n) = O(3^n)$

4. Consider the function $f(n) = 3n + 3$

Here, $3n + 3 \leq 4n$, for $n \geq 3$

$\Rightarrow a = 4$, $n_0 = 3$ and $h(n) = n$

Therefore, $f(n) = O(n)$

**Example 5**: Derive the O notation for the following function:

$f(n) = a_0 + a_1n + a_2n^2 + \ldots + a_{m-1}n^{m-1} + a_mn^m$

**Solution**: You can write the given function in the summation form as follows:

$$f(n) \leq \sum_{i=0}^{m} |a_i| n^i$$

$$\leq n^m \sum_{i=0}^{m} |a_i| n^{i-m}$$

$$\leq n^m \sum_{i=0}^{m} |a_i|, \text{ for all } n \geq 1$$

Here, $a = \sum_{i=0}^{m} |a_i|$, $n_0 = 1$, and $h(n) = n^m$.

Therefore, $f(n) = O(h(n)) = O(n^m)$.

## 1.4.3 Ω-notation

The Ω-notation is used to provide an asymptotic lower bound on a function. The Ω-notation of a function g(n) is defined as follows:

NOTES

$\Omega(g(n))$ = {f(n): there exist non-negative constants, a and $n_0$, such that f(n) lies above a * g(n), i.e. $0 \leq a * g(n) \leq f(n)$, for all $n \geq n_0$}.

In the above expression, a is a constant value and $\Omega(g(n))$ is pronounced as big omega of g of n.

Figure 1.3 shows the graphical representation of Ω-notation of a function f(n).

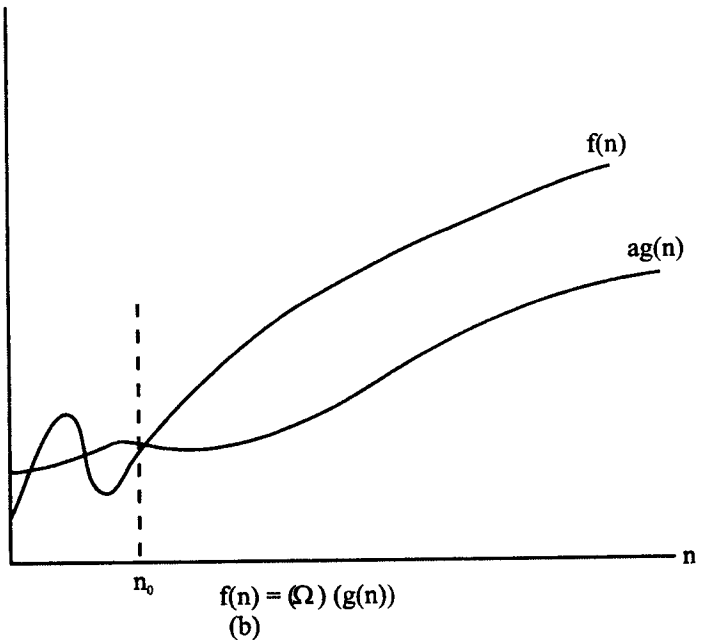



***Figure 1.3** Graphical Representation of Ω-Notation of f(n)*

**Example 6**: Determine the Ω-notation for the function f(n) = 9n +11

**Solution**: According to the definition of the Ω-notation:

$f(n) = \Omega(g(n))$ if $f(n) \geq a * g(n)$ for all $n \geq n_0$

Here, $9n + 11 \geq 9n$, for $n \geq 1$

$\Rightarrow a = 9$, $n_0 = 1$ and $h(n) = n$

Therefore, $f(n) = \Omega(h(n))$

$= \Omega(n)$

$9n + 11 \geq$ any constant for any positive value of n, therefore $9n + 11 = \Omega(1)$ is also correct.

**Example 7**: Determine the Ω notation for the following functions:

5. $f(n) = 7n^2 + 6$
6. $f(n) = 21n^2 + 9n$
7. $f(n) = 10n^3 + 5n^2 + 17$
8. $f(n) = 2 * 3^n + n^2$

**Solution**:

1. Consider the function $f(n) = 7n^2 + 6$

NOTES

Here, $7n^2 + 6 \geq 7n^2$, for $n \geq 1$

=> $a = 7$, $n_0 = 1$ and $h(n) = n^2$

Therefore, $f(n) = \Omega(h(n)) = \Omega(n^2)$.

2. Consider the function $f(n) = 21n^2 + 9n$

Here, $21n^2 + 9n \geq n^2$, for $n \geq 1$

=> $a = 1$, $n_0 = 1$ and $h(n) = n^2$

Therefore, $f(n) = \Omega(n^2)$.

3. Consider the function $f(n) = 10n^3 + 5n^2 + 17$

Here, $10n^3 + 5n^2 + 17 \geq 10n^3$, for $n \geq 1$

=> $a = 10$, $n_0 = 1$ and $h(n) = n^3$

Therefore, $f(n) = \Omega(n^3)$.

4. Consider the function $f(n) = 2 * 3^n + n^2$

Here, $2 * 3^n + n^2 \geq 2 * 3^n$, for $n \geq 1$

=> $a = 2$, $n_0 = 1$ and $h(n) = 3^n$

Therefore, $f(n) = \Omega(3^n)$.

### 1.4.4 o-notation

The o-notation is used to denote asymptotic loose upper bound. This notation is known as little-oh notation. Consider the following function:

$f(n)=o(g(n))$

This means f(n) is little-oh of g(n) for n>0, if and only if f(n) is O(g(n)) but f(n) is not $\Theta(g(n))$. In the function f(n)=o(g(n)), o-notation represents asymptotic loose upper bound because g(n) is an asymptotic upper bound not an asymptotic lower bound due to f(n)=O(g(n)),also $f(n) \neq \Theta(g(n))$, which means $f(n) \neq \Omega(g(n))$.

You can also define o(g(n)), which is known as little-oh of g of n as the set, in the following manner:

o(g(n))={f(n): for any constant a>0, there exists a constant $n_0$>0 such that $0 \leq f(n) \leq a * g(n)$ for all $n >= n_0$}.

The O-notation and o-notation are the same except for one difference. The difference is that in f(n)=O(g(n)), the bound 0<=f(n)<=ag(n) is true for some constant a>0. Whereas in f(n)=o(g(n)), the bound 0<=f(n)<ag(n) is true for all constants a>0. When n approaches infinity, then the function f(n) becomes insignificant relative to g(n) in the o-notation, which can be represented as:

$$\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

**Example 8**: Determine the o-notation for the function f(n) = 4n + 3.

**Solution**: According to the definition of the o-notation:

$f(n) = o(g(n))$ if $\lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$.

NOTES

Let $g(n) = n^2$, then,

$4n + 3 \leq n^2$, for $n \geq 5$

Also, $\lim_{n\to\infty} \frac{4n+3}{n^2} = 0$

Therefore, $f(n) = 4n + 3 = o(g(n)) = o(n^2)$.

**Example 9**: Determine the o-notation for the following functions:

1. $f(n) = \log n$
2. $f(n) = 2n^3 + 9n^2 + 3$

**Solution**:

1. Consider the function $f(n) = \log n$

Let $g(n) = n$

Then, $\lim_{n\to\infty} \frac{\log n}{n} = 0$

Therefore, $f(n) = \log n = o(n)$.

2. Consider the function $f(n) = 2n^3 + 9n^2 + 3$

Let $g(n) = n^4$

Then, $\lim_{n\to\infty} \frac{2n^3 + 9n^2 + 3}{n^4} = 0$

Therefore, $f(n) = 2n^3 + 9n^2 + 3 = o(n^4)$.

### 1.4.5 ω-notation

The ω-notation is used to denote asymptotic loose lower bound. This notation is known as little-omega notation. Consider the following function:

$f(n)=\omega(g(n))$

In this function, for any positive constant $a>0$, there exists a constant $n_0$ such that $0 \leq a\, g(n) < f(n)$ for all $n \geq n_0$. Also, the value of $n_0$ should not depend on n, but can depend on a.

As o-notation is similar to O-notation, ω-notation is similar to Ω-notation. You can also define ω(g(n)), which is known as little-omega of g of n as the following set:

$\omega(g(n))=\{f(n)$: for any positive constant $a>0$, then there exists a constant $n_0$ such that $0 \leq ag(n)<f(n)$ for all $n \geq n_0\}$.

When n approaches infinity, then the function f(n) becomes arbitrarily large relative to g(n) in the ω-notation, which can be represented as:

$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$, if the limit exists.

**Example 10**: Determine the ω-notation for the function $f(n) = 3n^2 + 1$

**Solution**: According to the definition of ω-notation

$f(n) = \omega(g(n))$ if $0 \leq a\, g(n) < f(n)$ and $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$

Let g(n) = n, then,

$3n^2 + 1 > 3n \geq 0$, for $n \geq 1$

=> a = 3 and $n_0 = 1$

Also, $\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \frac{3n^2+1}{n} = \infty$

Therefore, $f(n) = 3n^3 + 1 = \omega(n)$.

NOTES

## 1.5 RECURRENCE RELATIONS

Recurrence relations can be defined as the equation that expresses the running time of an algorithm on a particular problem of size N. This equation is expressed in terms of the running time of the algorithm on smaller inputs. For example, the following recurrence relation gives the worst case running time of a sorting algorithm:

$$T(N) = \begin{cases} \theta(1) & \text{if } N = 1 \\ 2T(N/2) + \theta(N) & \text{if } N > 1 \end{cases}$$

There are three methods to solve a recurrence relation, which are as follows:

- Recursion tree method
- Master method
- Iteration method

### 1.5.1 Recursion Tree Method

In a recursive method, a recursive tree is to be drawn for solving the recurrence relation of a particular algorithm. Each node in the recursion tree represents the cost of the sub-problem, which may have been invoked recursively during the execution of the algorithm. The costs associated with each node at a single level of the recursion tree are summed to obtain the per-level cost. Sum of all the per-level cost is used to determine the total cost associated with the recursion. The recursion tree method is useful to describe the running time of those algorithms that follow the divide and conquer approach.

Consider the following recurrence relation:

$$T(N) = 3T\left(\frac{N}{4}\right) + CN^2, \text{ where } C > 0$$

Here, you can assume that N is a multiple of 4. Figure 1.4 shows the expansion of T(N).

NOTES

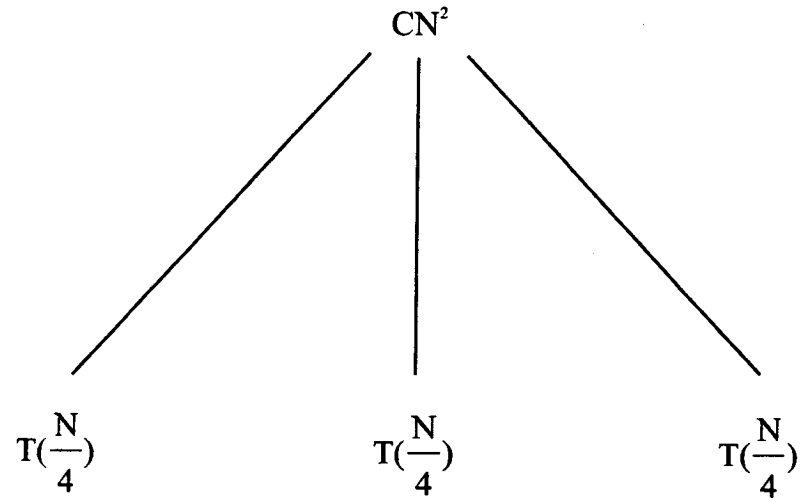



***Figure 1.4*** *Expansion of T(N)*

The recurrence T(N/4) is further expanded to form the next level of the recursion tree. Figure 1.5 shows the expansion of recurrence T(N/4).

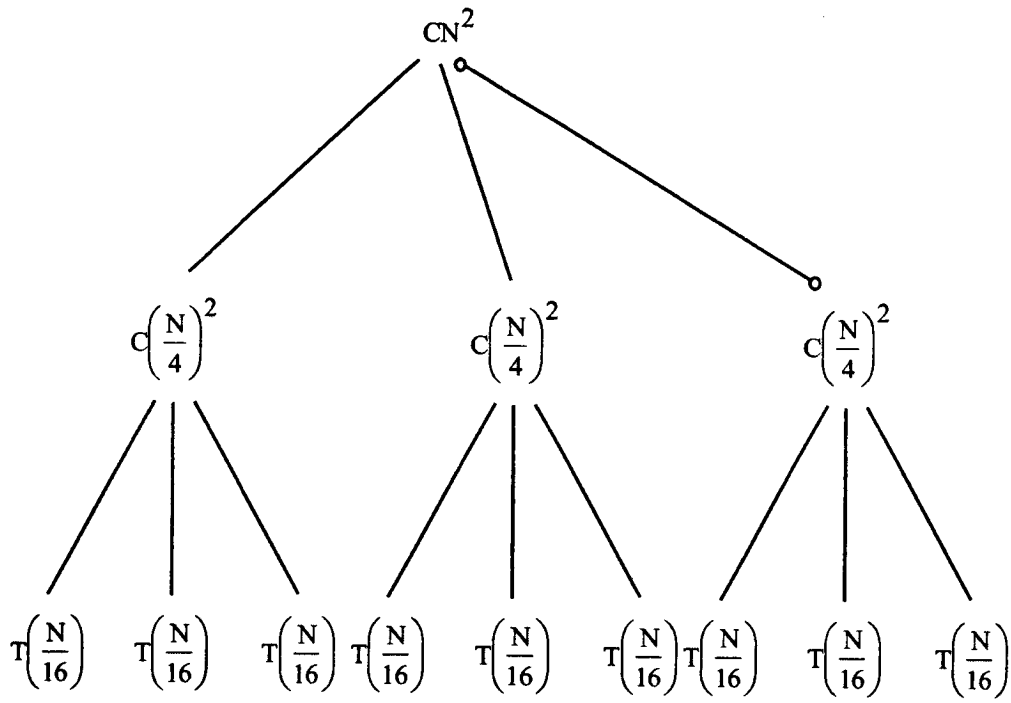



***Figure 1.5*** *Expansion of T(N/4)*

The recurrence T(N/16) is further expanded to form the next level of the recursion tree. Similarly, all the recurrences are expanded progressively till T(1) is achieved. Figure 1.6 shows the recursive tree, which represents the expansion of T(N).

NOTES

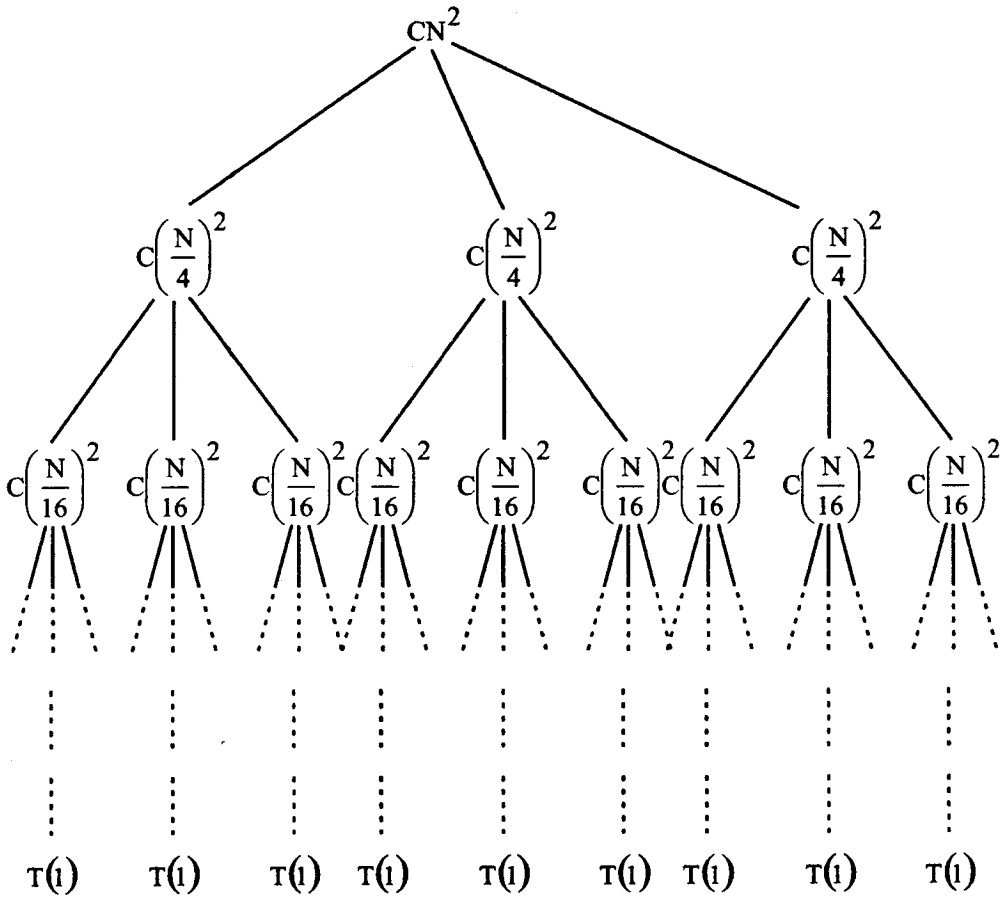



***Figure 1.6** Recursive Tree for T(N)*

It is clear that the size of sub-problem decreases as you progress down the tree and the size of each sub-problem at depth $h$ is $N/4^h$. Since, value of N is power of 4, therefore the size of sub-problem becomes one when the value of $N/4^h$ is one.

That is, $\frac{N}{4^h} = 1$

$\Rightarrow h = \log_4 N$

It means that the recursive tree has $\log_4 N + 1$ levels, which are 0, 1, 2, ... , $\log_4 N$.

In the recursion tree constructed above, the number of nodes at each level is three times more than the previous level. Therefore, the number of nodes at depth h is $3^h$. The cost associated with each node is $C(N/4^h)^2$. Therefore, the total cost of all nodes at depth h = 0, 1, 2, ... , $\log_4 N - 1$, is $3^h(C(N/4^h)^2)$, which is equal to $(3/16)^h(CN^2)$.

At the last level, the number of nodes is $3^{\log_4 N}$, each of which has cost equivalent to T(1).

Since, $3^{\log_4 N} = N^{\log_4 3}$

Therefore, the total cost of all nodes at the last level = $N^{\log_4 3}\, T(1) = \Theta(N^{\log_4 3})$

The cost of the entire recursive tree, which is represented by T(N), is calculated by adding the total cost of each level.

NOTES

$$\Rightarrow T(N) = CN^2 + \frac{3}{16}CN^2 + \left(\frac{3}{16}\right)^2 CN^2 + \ldots + \left(\frac{3}{16}\right)^{\log_4 N - 1} CN^2 + \Theta(N^{\log_4 3})$$

$$\Rightarrow T(N) = \sum_{i=0}^{\log_4 N - 1} \left(\frac{3}{16}\right)^i CN^2 + \Theta(N^{\log_4 3})$$

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i CN^2 + \Theta(N^{\log_4 3})$$

$$= \frac{1}{1 - \left(\frac{3}{16}\right)} CN^2 + \theta(N^{\log_4 3})$$

$$= \frac{16}{13}CN^2 + \Theta(N^{\log_4 3})$$

$$= O(N^2)$$

## 1.5.2 Master Method

Master method is another commonly used method to solve the recurrence relations. The method is mostly used to find the solution for recurrence relations of those algorithms that follow the divide and conquer approach. In general, the master method can solve all the recurrence relations of the following form:

$$T(N) = aT\left(\frac{N}{b}\right) + f(N), \quad a \geq 1,\ b > 1$$

Where, both a and b are constants and f(N) represents an asymptotically positive function. In the above recurrence relation, N represents the size of a particular problem. The constant a represents the number of sub-problems and the ration N/b represents the size of each sub-problem. f(N) represents the cost involved in breaking the problem into several sub-problems and combing the solutions of all the sub-problems. Each of the sub-problems is solved recursively and requires T(N/b) running time.

The master method is based on the Master theorem, which is stated as follows:

Let T(N) be defined on the positive integers by the following recurrence relation:

$$T(N) = aT\left(\frac{N}{b}\right) + f(N)$$

Where, a and b are positive constants, such that $a \geq 1$ and $b > 1$, and f(N) is a asymptotically positive function on N. Also, N/b can be interpreted either as $\lfloor N/b \rfloor$ or as $\lceil N/b \rceil$. Then, T(N) can be asymptotically bounded as follows:

- **Case 1**: If $f(N) = O\left(N^{\log_b a - \epsilon}\right)$ for some constant $\varepsilon > 0$, then

  $T(N) = \Theta\left(N^{\log_b a}\right)$.

- **Case 2**: If $f(N) = \Theta\left(N^{\log_b a}\right)$, then

$$T(N) = \Theta\left(N^{\log_b a} \lg N\right).$$

- **Case 3**: If $f(N) = \Omega\left(N^{\log_b a+\epsilon}\right)$ for some constant ε > 0, and if $af\left(\frac{N}{b}\right) \leq cf(N)$ for some constant c < 1 and for all N having sufficiently large value, then
$T(N) = \Theta(f(N))$.

NOTES

**Example 11**: Solve the following recurrence relation:

$$T(N) = 8T\left(\frac{N}{2}\right) + N^2$$

**Solution**: Comparing the given recurrence relation with the following relation:

$T(N) = aT\left(\frac{N}{b}\right) + f(N)$, you will get:

a = 8, b = 2 and f(N) = $N^2$.

=> $N^{\log_b a} = N^{\log_2 8} = O(N^3)$

Hence, $N^{\log_b a}$ is greater than f(N), therefore according to case 1 of master theorem f(N) is represented as follows:

$f(N) = O\left(N^{\log_2 8-\epsilon}\right)$, with ε = 1

Now, by applying the case 1 of Master theorem, the solution of given recurrence relation is T(N) = Θ($N^3$).

**Example 12**: Solve the given recurrence relation:

$$T(N) = T\left(\frac{3N}{5}\right) + 1$$

**Solution**: Here,

a = 1

b = 5/3

f(N) = 1

Therefore,

$$N^{\log_b a} = N^{\log_{5/3} 1}$$
$$= N^0 = 1$$
$$= \Theta(1)$$

Here, $N^{\log_b a}$ is equal to f(N), therefore according to case 2 of Master theorem f(N) can be represented as follows:

NOTES

$$f(N) = \Theta\left(N^{\log_{5/3} 1}\right)$$

$$= \Theta(1)$$

Therefore, the solution to the given recurrence relation is:

$$T(N) = \Theta\left(N^{\log_b a} \lg N\right)$$

$$= \Theta(\lg N)$$

**Example 13**: Sole the recurrence relation:

$$T(N) = 3T\left(\frac{N}{4}\right) + N^3$$

**Solution**: Here,

a = 3

b = 4

$f(N) = N^3$

Therefore,

$$N^{\log_b a} = N^{\log_4 3}$$

$$= O(N^{0.793})$$

Here, $N^{\log_b a}$ is less than f(N), therefore according to case 3 of master theorem f(N) can be represented as follows:

$$f(N) = \Omega\left(N^{\log_4 3+\epsilon}\right), \text{ with } \varepsilon \approx 0.2$$

Also, $af\left(\frac{N}{b}\right) \leq cf(N)$ should be true.

$$3\left(\frac{N}{4}\right)^3 \leq \frac{3}{4}\left(N^3\right), \text{ with } c = ¾$$

=> 1/16 < 1

Therefore, the solution to the given recurrence relation is:

$$T(N) = \Theta(f(N))$$

$$= \Theta(N^3)$$

**CHECK YOUR PROGRESS**

5. Define Θ-notation.
6. What is a recurrence relation?
7. Name all the methods used in solving recurrence relations.
8. What is the utility of the iteration method?

## 1.5.3 Iteration Method

Iteration method is also used to solve recurrence relations. In this method, the right hand side of the recurrence relation is expanded repeatedly. A common formula is then constructed using the given recurrence relation. For example, consider the following recurrence relation:

$$T(N) = \begin{cases} 2 & \text{if } N = 0 \\ 2 + T(N-1) & \text{if } N > 0 \end{cases}$$

The solution to the above problem by using iterative method is as follows:

NOTES

$T(N) = 2 + T(N-1)$

$= 2 + 2 + T(N-2)$

$= 2(2) + T(N-2)$

. . .

. . .

. . .

$= (N-1)2 + T(1)$

$= 2N + T(0)$

$= 2N + 2$ [because, $T(0) = 2$]

Therefore, $T(N) = 2N + 2 = O(N)$.

## 1.6 SUMMARY

In this unit, you have learnt about the basics of algorithms and the different design paradigms such as divide and conquer, dynamic programming and greedy method for the construction of algorithms. The efficiency of an algorithm can be determined by analysing the space and time complexity of the algorithm. Space complexity is the measure of computer memory required by an algorithm and time complexity is the measure of running time required for the execution of the algorithm.

The running time of algorithms are expressed in terms of various asymptotic notations such as Θ-notation, O-notation, Ω-notation, o-notation and ω-notation. You have also studied about the running time of the algorithms which is expressed using generalized equations that are known as recurrence relations. These recurrence relations are solved using recursion tree method, master method and iteration method.

## 1.7 KEY TERMS

- **Algorithm:** It is a sequence of steps used to perform a particular task successfully. The result of this sequence of steps performed on sample data is known as output of the algorithm.
- **Space complexity:** The amount of memory needed by an algorithm for its execution and generating the final output is called space complexity.
- **Time complexity:** Time complexity of an algorithm is the amount of time needed by an algorithm for its execution.
- **Asymptotic notations:** These notations are used to represent asymptotic running time of algorithms.

NOTES

- **Recurrence relations:** Recurrence relations can be defined as the equation that expresses the running time of an algorithm on a particular problem of size N.

## 1.8 ANSWERS TO 'CHECK YOUR PROGRESS'

1. An algorithm can be defined as a tool that provides a step-by-step procedure for solving a given problem.
2. Finiteness of an algorithm means that the algorithm must terminate after a finite number of steps.
3. Space complexity of an algorithm can be defined as the amount of memory needed by the algorithm for its execution and generating the final output.
4. The following is a generalized expression for calculating $T_A(n)$:

   $$T_A(n) = c_A ADD(n) + c_S SUB(n) + c_D DIV(n) + c_M MUL(n) + c_{ST} STORE(n) + \ldots$$

   Where, $c_A$, $c_S$, $c_D$, $c_M$, $c_{ST}$ represents the time required for the addition, subtraction, division, multiplication and store operations. The number of addition, subtraction, division, multiplication and store operations performed by the algorithm are represented by ADD, SUB, DIV, MUL and STORE.
5. Θ-notation for a particular g(n) is defined as follows:

   $\Theta(g(n))$ = {f(n): there exits non-negative constants $a_1$, $a_2$ and $n_0$ such that f(n) lies between a1g(n) and a2g(n) i.e. $0 \leq a_1 g(n) \leq f(n) \leq a_2 g(n)$ for all $n \geq n_0$}.
6. Recurrence relations can be defined as an equation that expresses the running time of an algorithm on a particular problem of size N.
7. The methods that are used to solve the recurrence relations are stated below:
   A. Recursion tree method
   B. Master method
   C. Iteration method
8. Iteration method is used to solve the recurrence relation. In this method, the right hand side of the recurrence relation is expanded repeatedly using the given recurrence relation and then a common formula is constructed.

## 1.9 QUESTIONS AND EXERCISES

### Short-Answer Questions

1. Write a brief note on algorithms.
2. What do you mean by asymptotic notations? Name them.
3. Distinguish between Θ-notation and O-notation.

4. Write a short note on space complexity of an algorithm.
5. Explain the recurrence relation.

## Long-Answer Questions

1. Describe the different design paradigms for algorithm designing.
2. Discuss all the asymptotic notations.
3. Analyse space and time complexity of the divide and conquer algorithm.
4. Describe the recursion tree method with the help of an example.
5. Solve the following recurrence relations:

   A. $T(N) = 4T\left(\frac{N}{2}\right) + N$

   B. $T(N) = 4T\left(\frac{N}{2}\right) + N^2$

   C. $T(N) = 8T\left(\frac{N}{4}\right) + 1$

NOTES

## 1.10 FURTHER READING

Cormen, Thomas H. 2005. *Introduction to Algorithms*. New Delhi: Prentice-Hall of India.

# UNIT 2 DIVIDE AND CONQUER STRATEGY

NOTES

## Structure



## 2.0 INTRODUCTION

You are already familiar with basic algorithm complexities and recurrences. In this unit, you will learn about searching and sorting techniques. Searching and sorting are the basic operations that help to search and sort data elements in a program. Searching is the process of locating data from a set of given data items. The different searching techniques are linear search and binary search.

Sorting allows you to arrange data in a specific order, such as ascending or descending order. It can arrange both numeric and alphabetical data in ascending or descending order. Sorting techniques are applied on data so that it can be analysed and accurate decisions can be taken. This becomes possible because it arranges the data in a predetermined order. For example, proper sequence of allotting house numbers in a sector makes it easy to locate the house. The different sorting techniques are:

- Selection sort
- Merge sort
- Quick sort

In this unit, you will also learn the complexity of various sorting algorithms. You can also measure the complexity of an algorithm by using the complexity theory, which depends on the worst case and the average case.

NOTES

## 2.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Explain the concept of searching
- Use the binary search technique
- Describe the preliminaries of sorting
- Use different sorting techniques, which are as follows:
  - o Bubble
  - o Insertion
  - o Selection
- Compare the complexity of various sorting techniques
- Use Strassen's Matrix Multiplication method

## 2.2 OVERVIEW OF DIVIDE AND CONQUER STRATEGY

Divide and conquer strategy follows top-down approach to design algorithms for solving large problems. This strategy consists of dividing the problem into small size problems that are called sub problems. The composition of solution of all sub problems provides the solution of the original problem. In other words, the following steps are involved in the divide and conquer design strategy:

- **Divide**: Break the problem into many small size sub-problems that are the same as original problem.
- **Conquer**: Solve all the sub problems independently and recursively.
- **Combine**: Compose the solutions of all the sub problems to build a solution for the original problem.

The general form of divide and conquer algorithm is as follows:

```
1.  Algorithm D-and-C (p : input size)
2.  {
3.  begin
4.  if p < = p0 then
5.  Solve problem without further sub division of the
6.  problem;
7.  else
8.  Split into x sub problem
9.  each of size p/k;
10. for each of the x sub problem do
11. D-and-C (p/k);
12. Combine the x resulting sub solutions to produce the
13. solution to the original problem;
14. end if;
15. end D-and-C;
```

NOTES

The computing time to solve the problem $P$ can be calculated using the following function:

$$T(n)=\begin{cases} g(n) & \text{if } n \text{ is small else} \\ T(n_1)+T(n_2)+T(n_3)...+T(n_k)+f(n) \end{cases}$$

(2.1)

Where, $T(n)$ is the time required for computing the problem $P$ with $n$ inputs, $g(n)$ is the time required for computing the small size problem and $f(n)$ is the time required for dividing the problem P in sub problems.

The complexity of the divide and conquer algorithm can be represented in the following form:

$$T(n)=\begin{cases} T(1) & n=1 \\ aT(n/b)+f(n) & n>1 \end{cases}$$

(2.2)

Where, $a$ and $b$ are the known constants.

## 2.3 SEARCHING TECHNIQUES

Searching techniques are used to retrieve a particular record from a list of records in an efficient manner so that the least possible time is consumed. The list of records can have a key field, which can be used to identify a record. Therefore, the given value must be matched with the key value in order to find a particular record. If it identifies a particular record, then the search operation is said to be successful, otherwise, it is said to be unsuccessful. Generally, there are two types of searching techniques, which can be used under divide and conquer strategy to locate a particular record in the given list of records. These techniques are linear search and binary search.

### 2.3.1 Binary Search

Binary search is a method, which involves the comparison of the search data with the data at the centre of the list. It also involves the comparison of elements in the first half or second half of the list. The comparison with the first or second half of the list depends on whether the search data is less than or greater than the data item at the middle. If the search data is greater than the data item at the middle, then the comparison is done with the elements in the second half of the array.

The algorithm, which helps to perform binary search, uses an index, top, to indicate the top of the list and an index, bottom, to indicate the bottom of the list. Each time the loop is executed, the size of the list enclosed by the top and the bottom indices is reduced to half. The loop is executed until there are data items in the list or till a successful search occurs.

The recursive binary search algorithm is as follows:

```
1.  Algorithm BinSrch(a,i ,l,x)
2.  {
3.  if (l=i) then //if problem P is small
4.  {
```

NOTES

```
5.   if (x=a[i]) then return i;
6.   else return 0;
7.   }
8.   else
9.   {
10.  //divide problem P into smaller sub problem
11.  mid:=[(i+l)/2];
12.  if (x<a[mid])
13.  then return mid;
14.  else if (x<a[mid])
15.  then
16.  return BinSrch(a,i,mid-1,x);
17.  else return BinSrch(a,mid+1,l,x);
18.  }
19.  }
```

In the above algorithm, $a[i:l]$ represents an array of elements in non-decreasing order. $1 \leq i \leq l$ determines whether $x$ is present in the range of i or not. If $x$ is present, then, return i returns the value of a[i]; else it returns 0.

The Iterative binary search algorithm is as follows:

```
1.   Algorithm BinSrch(a,n,x)
2.   {
3.   low:=1; high:=n;
4.   {
5.   while (low ≤ high) do
6.   {
7.   mid:= [(low+high)/2];
8.   if(x<a[mid]) then high:=mid-1;
9.   else if(x>a[mid]) then low:=mid+1;
10.  else return mid;
11.  }
12.  else return mid;
13.  }
14.  return 0;
```

In the above algorithm, $a[1:n]$ represents an array of elements in non-decreasing order. $n \geq 0$ determines whether $x$ is present or not. If present, return mid *returns a[j]*; else it returns 0.

**CHECK YOUR PROGRESS**

1. What do you mean by searching?
2. List the different searching techniques.
3. Define the binary search technique.

## 2.4 SORTING TECHNIQUES

NOTES

Another technique of divide and conquer strategy is sorting. Sorting techniques can be applied on a list of data, so that the list can be analysed easily for making accurate decisions. This becomes possible because sorting arranges the data in a predetermined order. For example, proper sequence of allotting the house numbers in a sector makes you easily identify the house because the house numbers are arranged in an increasing order. You can arrange the data using various sorting techniques, which are as follows:

- Selection
- Merge
- Quick

### 2.4.1 Selection Sort

Selection sort is a technique in which the smallest element of an array is searched and swapped with the first element of an array. Then, the second element is swapped with the next smallest element and this process continues till the array is sorted. The first pass compares $n - 1$ comparisons, the second pass compares $n - 2$ comparisons and the third pass compares $n - 3$ comparisons. The total number of comparisons of the selection sort technique is calculated using the following formula:

$$(n-1)+(n-2)+(n-3)+.....1=n*n(n-1)/2$$

The complexity of the selection sort is $O(n^2)$, where $n$ is the number of data items stored in an array.

The algorithm of selection sort for finding the smallest element is as follows:

```
1.  Algorithm Selection(a,n,x)
2.  {
3.  low:=1; high:=n+1;
4.  a[n+1]:=∞ //a[n+1] is a set to infinity
5.  repeat
6.  {
7.  //Each time the loop is entered, 1≤ low ≤ x ≤ high ≤
    n+1
8.  j := Partition(a, low, high);
9.  //j is such that a[J] is the jth-smallest value in a[]
10. if(x=j) then return;
11. else if(x<j) then high:=j// j is the new upper limit.
12. else low :=j+1;//j+1 is the new lower limit.
13. }
14. until (false);
15. }
```

NOTES

In the above algorithm, *a[i:l]* represents an array of elements in which you need to select the *x*th smallest element at *x*th position of array *a*[]. The remaining elements are represented such that $a[m] \leq a[x]$ for $1 \leq m < x$ and $a[m] \geq a[x]$ for $k < m \leq n$.

**Example 1**: An array, num, has five data items and you have to sort these data items by using the selection sort technique. The data items stored in an array, num are stated below:

```
num = {55, 67, 21, 89,16}
```

**Solution**: In the num array, the smallest data item 16 replaces the first data item 55, then the second smallest data item 21 replaces the second data item 67 and this continues until all the data items are sorted. The steps involved in sorting the num array are as follows:

1. (55), 67, 21, 89, (16)
2. 16, (67), (21), 89, 55
3. 16, 21, (67), 89, (55)
4. 16, 21, 55, (89), (67)

Finally, the sorted array is 16, 21, 55, 67, 89.

The algorithm to select *x*th smallest element uses the median of medians rule to determine a partition element. For selecting the element, you can use the recursive application of the selection algorithm. The high-level version of the selection algorithm is Select2.

The algorithm of selection sort using median of medians rule is as follows:

```
1.  Algorithm Selec2(a,x,low,high)
2.  {
3.  n:=high-low+1;
4.  if(n ≤r) then sort a[low:high] and return the x-th
    element;
5.  Divide a[low:high] into n/r be the set of medians of
    the above n/r subsets of size r each;
6.  Ignore excess elements;
7.  Let m[i], 1 ≤ i ≤ (n/r) be the set of medians of the
    above n/r subsets.
8.  v:=Select2(m,[n/r]/2);1,n/r);
9.  Partition a[low:high] using v as the partition
    element;
10. Assume that v is at position j;
11. if(x=(j-low+1)) then return v;
12. elseif(k<(j-low+1)) then
13. return Select2(a,x,low,j-1);
14. else return Select2(a,k-(j-low+1),j+1,high);
```

NOTES

## 2.4.2 Merge Sort

The merge sort technique combines two sorted arrays into one larger sorted array. For example, the sorted array, $A$, contains $p$ elements and the sorted array, $B$, contains $q$ elements. The merge sort technique combines the elements of $A$ and $B$ into a single sorted array, $C$, with $p + q$ elements.

The total number of comparisons in the merge sort technique to sort $n$ data-items of an array is $\log n$. The merge sort technique requires almost $\log n$ passes, so the complexity of the merge sort is $O(n \log n)$.

Algorithm for merge sort is as follows:

```
1.   Algorithm MergeSort(low,high)
2.  {
3.  if(low < high) then
4.  {
5.  //Divide P into sub problems
6.  //find where to split the set
7.  mid:=[(low+high)/2];
8.  //solve the sub problems
9.  MergeSort(low,high);
10.   MergeSort(mid+1,high);
11.   //Combine the solutions.
12.   Merge(low,mid,high)
```

**Example 2**: Merge the content of two arrays $A$ and $B$. The data items stored in the array, $A$, are as follows:

```
A={56,78}
```

The data items stored in the array, $B$, are as follows:

```
B={25,67,89}
```

**Solution**:

The data items of array, $C$, after merging the two sorted arrays, $A$ and $B$, are as follows:

```
C={25, 56, 67, 78, 89}
```

The first data item of the $A$ array is compared with the first data item of the $B$ array. If the first data item of $A$ is smaller than the first data item of $B$, then that data item from $A$ is moved to the new array, $C$. If the data item of $B$ is smaller than the data item of $A$, then $B$, is moved to the array, $C$. This comparing of data items continues, until one of the arrays ends.

Merge sort is of the following types:

- Simple merge sort
- Recursive merge sort
- Iterative merge sort

NOTES

### 2.4.2.1 Simple merge sort

Consider the sorted array, $A$, which contains $p$ elements and the sorted array, $B$, containing $q$ elements. The simple merge sort technique combines the elements of $A$ and $B$ into a single sorted array, $C$, with $p + q$ elements.

The total number of comparisons in the merge sort technique to sort $n$ data-items of an array is log $n$. It requires at the most log $n$ passes, so that the complexity of the merge sort is $O(n \log n)$.

Algorithm to merge two sorted sub arrays using auxiliary storage is as follows:

```
1.  Algorithm MergeSort(low,mid,high)
2.  {
3.  h:=low; I:=low; j:=mid+1;
4.  while ((h≤mid) and (j≤high))do
5.  {
6.  if (a[h] ≤ a[j]) then
7.  {
8.  b[i]:= a[h]; h:=h+1;
9.  }
10. else
11. {
12. b[i]:= a[j]; j:=j+1;
13. }
14. i:=i+1;
15. }
16. if (h>mid)then
17. for k:=j to high do
18. {
19. b[i]:=a[k]; i:=i+1;
20. }
21. for k:=low to high do a[k]:=b[k];
```

**Example 3**: Merge two sorted arrays into a single sorted array by using the merge sort technique. The data items stored in an array, $A$, are as follows:

```
A={56,78}
```

The data items stored in an array, $B$, are as follows:

```
B={45,67,89}
```

**Solution**:

The data items of the $C$ array after merging the two sorted arrays, $A$ and $B$, are as follows:

```
C={45, 56, 67, 78, 89}
```

NOTES

The first data item of the *A* array is compared with the first data item of the *B* array. If the first data item of *A* is smaller than the first data item of *B*, then that data item from *A* is moved to the new array, *C*. If the data item of *B* is smaller than the data item of *A*, then *B*, is moved to the array, *C*. This comparing of data items continues until one of the arrays ends.

### 2.4.2.2 Recursive merge sort

Recursive merge sort divides the given list in two halves, sorts each half independently and merges the sorted parts. Consider an array *A*, which is composed of the following elements:

(25, 3, 44, 15, 14, 34)

To sort the above array in ascending order using the recursive merge sort. The steps are as follows:

1. **Step 1:** Divide the array into two equal halves by calculating mid using the following formula:

```
Mid = (lower + upper)/2
```

2. **Step 2:** The first array is A(low:mid) and the second array becomes A(mid+1:high) as shown below:

```
(25, 3, 44)        (15, 14, 34)
```

3. **Step 3**: Again divide the array (24, 3, 44) into two lists: (24, 3) and (44). Consider the first list and sort it. It becomes (3, 24), which is merged with (44) so that the list becomes (3, 24, 44).
4. **Step 4**: Similarly the list (15, 14, 34) is sorted and becomes (14, 15, 34).
5. **Step 5**: The list obtained in the last two steps is merged to obtain a final list as (3, 14, 15, 25, 34, 44).

### 2.4.2.3 Iterative merge sort

In this sort, the elements are sorted in a non-recursive manner. In the beginning of the iterative merge sort, an array of *n* elements is divided into *n* groups with each of its length as 1. In the first iteration, two groups are merged into a group of length 2. Similarly, in the second iteration, two groups each of length 2 are merged to form a group of length 4. At each iteration, the elements are arranged in sorted order.

## 2.4.3 Quick Sort

The quick sort technique sorts the elements of an array faster than any other sorting technique. The various steps to perform quick sort technique on an array are listed below:

1. Return the array, if the array contains one element.
2. Select an element in the array, termed as pivot.
3. Split the array into two arrays, one with data element values larger than the pivot and the other with data element values smaller than the pivot.

NOTES

4. Repeat quick sorting recursively on these two parts of the linear array, until the linear array is sorted.

The algorithm, which is used for quick sort is as follows:

```
1.   Algorithm QuickSort(p,q)
2.  {
3.  if(p<q) then
4.  {
5.  //Divide P into sub problems
6.  j:=Partition (a,p,q+1);
7.  // j is the position of the partitioning element.
8.  //Solve the sub problems
9.  QuickSort(p,j-1);
10.   QuickSort(j+1,q);
11.   //There is no need to combine the solutions.
12.   }
13.   }
```

**Example 4**: To sort an array, A, in the increasing orders, using the quick sort technique. The various elements of the A array are as follows:

```
{21, 75, 48, 12, 45}
```

**Solution**: To sort the *A* array using the quick sort technique:

1. Select 48 as the pivot for the *A* array and split the *A* array into two arrays, *B* and *C*.

```
Pivot for A array = 48
B = {21, 12, 45}
C = {75}
```

2. Select 21 as the pivot for the *B* array and split the *B* array into two arrays, *D* and *E*.

```
Pivot for B array = 21
D = {12}
E = {45}
```

3. Return *C*, *D* and *E*, as the sorted arrays because these arrays contain single data element.

```
C = Sorted C
D = Sorted D
E = Sorted E
```

4. Determine the sorted *B* as:

Sorted *D*, Pivot of *B*, Sorted *E*

Sorted *B* = {12, 21, 45}

5. Arrange the sorted elements of array *A* as:

Sorted *B*, Pivot of *A*, Sorted *C*

Sorted *A* = {12, 21, 45, 48, 75}

**Example 5**: Consider an array that includes the following items:


**Solution:**

Partition the array so that all the items smaller than pivot item are to the left side of it and all the items larger than it are on the right side. Figure 2.1 shows the partitioned form of the array.

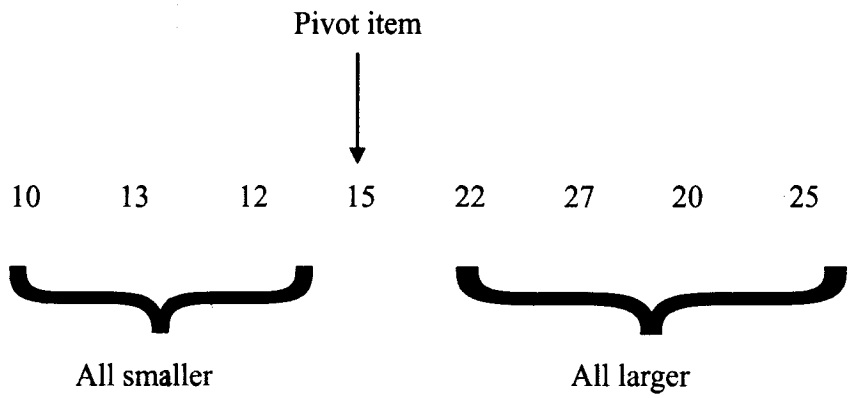



***Figure 2.1** Partitioning the Array around the Pivot Item*

Repeat the same process to sort the array. Figure 2.2 shows that all the sub arrays are enclosed and in rectangles, whereas the pivot points are free.

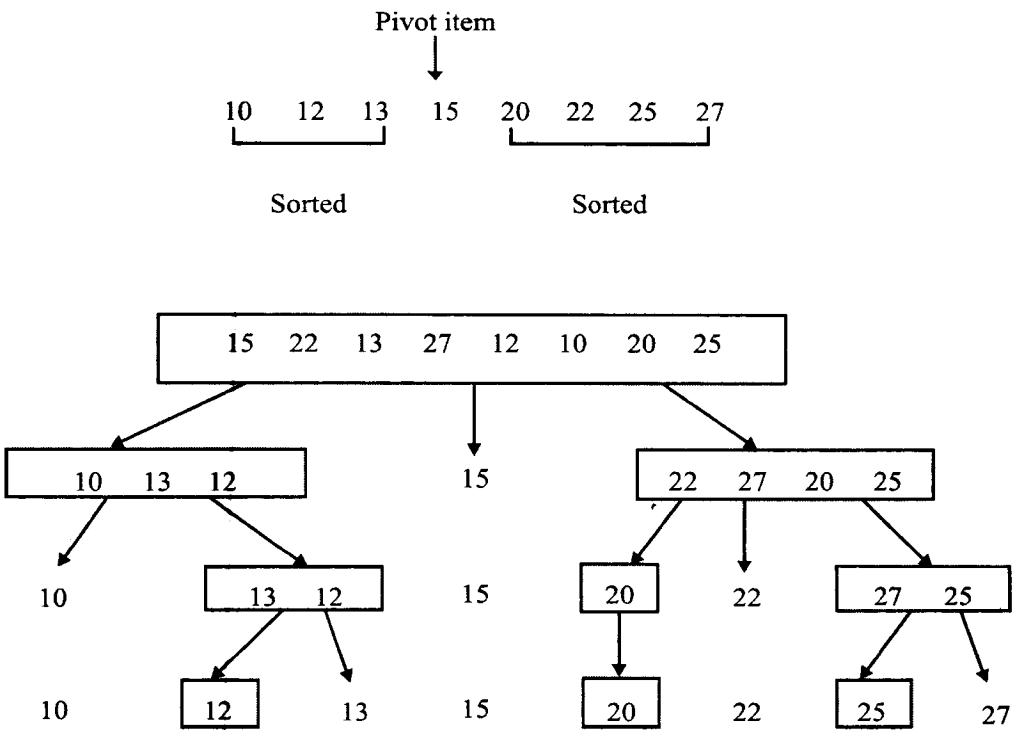



***Figure 2.2** Shorted Form of the Array*

**NOTES**

**CHECK YOUR PROGRESS**

4. What do you mean by sorting?
5. Enlist different sorting techniques.
6. What do you understand by selection sort?

NOTES

## 2.5 ALGORITHM COMPLEXITY

The efficiency of the algorithms depends on two essential elements, time and space. The complexity of an algorithm is a function that provides the running time and space for data depending on the size provided by you. Sometimes, you may need a time-space tradeoff that increases the amount of space to store the data. Using time-space tradeoff, you may be able to reduce the time required for processing the data or you may increase the time required for processing the data.

For example, *P* is an algorithm and *n* is the size of the input data. Then, the time and space that *P* uses will be the two main measures on which the efficiency of *P* depends. You can measure the time by counting the number of key operations in the searching and sorting algorithms.

The complexity of the algorithm, *P* is a function, *g(n)*, that provides you the running time and space of operations performed by an algorithm when the input size is *n*. The storage space for an algorithm is a multiple of its data size, *n*. You need to use the following two cases to investigate the complexity theory:

- The worst case complexity
- The average case complexity

### 2.5.1 The Worst Case Complexity

The worst case occurs when the items in the last element in the array, data, is not available. In this situation, you have:

$C(n) = n$

### 2.5.2 The Average Case Complexity

In this case, the item is present in the array, data, and can occur at any position in the array. Then,

$C(n) = 1.1\backslash n + 2.1\backslash n + \ldots\ldots\ldots + n.1\backslash n$

$= (1 + 2 + \ldots\ldots\ldots + n).1 \backslash n$

$= n(n + 1)\backslash 2.1\backslash n = n + 1 \backslash 2$

### 2.5.3 Complexity and Efficiency of Sorting Techniques

Complexity is the efficiency measure of an algorithm or a technique. For example, you can add first 20 natural numbers using two methods. In the first method, you need to add the 20 numbers using general addition as:

Sum = 1 + 2 + ..... + 20

Sum = 210

Using the second method, you need to add the 20 numbers using the following formula:

Sum = (*n* * (*n* + 1))/2

Here, *n* = 20, therefore the sum of first 20 natural numbers is calculated as:

Sum = (20 * (20 + 1)/2)

Sum = 210

NOTES

You can see that the second method to calculate the sum of first 20 natural numbers is less complex and requires less time to calculate in comparison to the first method.

The complexity function is a notation of different data comparisons done while sorting the data items of an array. The complexity of sorting is represented using Big $O$ notation. The Big $O$ notation is defined as $O((f(n))$, where $f(n)$ is the function of the number of data items stored in an array. The complexity enables you to compare the memory space occupied and the run time required for each sorting technique.

Search operations are faster if the data array is sorted. It's because sorting eliminates the unnecessary comparisons. The complexities of various sorting techniques are as follows:

- **Selection sort**: Quadratic time complexity or $O(n^2)$.
- **Merge sort**: Optimal complexity or $O(n\log(n))$.
- **Quick sort**: Quadratic time complexity or $O(n^2)$.

### 2.5.4 Efficiency Parameters of Sorting

The various efficiency parameters of the sorting techniques are as follows:

- **Number of comparisons**: It specifies the number of times, the search data is compared with the data elements of the data structure.
- **Number of swaps**: It specifies the number of moves of the data elements during the sorting process.

You can use the $O$ notation to measure the complexity of the sorting technique. The lesser the complexity of the sorting technique the greater will be the efficiency. Consider that $N$ is the number of data elements, which are to be sorted. Then, $O$ notation represents the number of operations performed on the $N$ data elements. These operations include the comparing and swapping of data elements. The complexity measures of the sorting techniques using the efficiency parameters are as follows:

- $O(\log2N) = O(\log2\ 8) = 3+$
- $O(N) = O(8) = 8$
- $O(N\log2N) = O(8\log28) = 24$
- $O(N2) = O(82) = 64$

## 2.6 STRASSEN'S MATRIX MULTIPLICATION

In linear algebra, the Strassen's algorithm, which is named after the German mathematician Volker Strassen, is used for matrix multiplication. Volker Strassen published this algorithm in 1969. It is faster than the standard matrix multiplication algorithms and is useful for solving large matrices.

Let $X$ and $Y$ be two square matrices over a ring $R$. The matrix product of these two matrices is:

$Z=XY$

Where,

NOTES

$X, Y, Z \in R^{2^n * 2^n}$

***Note:*** *If the matrices X and Y are not of type $2^n * 2^n$, you can fill the missing elements of row and column with zero.*

Consider the following matrices:

$$X = \begin{pmatrix} x_{11} & x_{12} \\ x_{21} & x_{22} \end{pmatrix} a$$

$$Y = \begin{pmatrix} y_{11} & y_{12} \\ y_{21} & y_{22} \end{pmatrix}$$

$$Z = \begin{pmatrix} z_{11} & z_{12} \\ z_{21} & z_{22} \end{pmatrix}$$

With,

$X_{ij}, Y_{ij}, Z_{ij} \in R^{2^{n-1} * 2^{n-1}}$

Then,

$Z_{11} = X_{11}Y_{11} + X_{12}Y_{21}$

$Z_{12} = X_{11}Y_{12} + X_{12}Y_{22}$

$Z_{21} = X_{21}Y_{11} + X_{22}Y_{21}$

$Z_{22} = X_{21}Y_{12} + X_{22}Y_{22}$

Now, define new matrices using the above products

$P_1 = (X_{11} + Y_{22})(X_{11} + Y_{22})$

$P_2 = (X_{21} + Y_{22})Y_{11}$

$P_3 = X_{11}(Y_{12} - Y_{22})$

$P_4 = X_{22}(Y_{21} - Y_{11})$

$P_5 = (X_{11} + X_{12})Y_{22}$

$P_6 = (X_{21} - X_{11})(Y_{11} + Y_{12})$

$P_7 = (X_{12} + X_{22})(Y_{21} + Y_{22})$

Therefore,

$Z_{11} = P_1 + P_4 - P_5 + P_7$

$Z_{12} = P_3 + P_5$

$Z_{21} = P_2 + P_4$

$Z_{22} = P_1 - P_2 + P_3 + P_6$

You can iterate this division process n-time until the sub matrices degenerate into numbers.

## 2.7 SUMMARY

NOTES

In this unit, you have learnt about searching and sorting techniques under the divide and conquer strategy. Searching techniques are used to search a particular record from a list of records based on the key value. A key value denotes a field which can be used to retrieve a particular record from the list. The searching technique that needs to be applied depends on whether the given list is in sorted or unsorted order. If the given data is in unsorted order, then you can apply the linear search technique. If the given data is in sorted order, then it is efficient to apply the binary search technique.

In this unit, you have also learnt about the different sorting techniques. These techniques are used to arrange the given data in a sorted order. The different sorting techniques that you can apply on the given unsorted data are as follows:

- Selection
- Merge
- Quick

You can also measure the complexity of an algorithm by using the complexity theory, which depends on the worst case and the average case.

## 2.8 KEY TERMS

- **Divide and conquer strategy:** This strategy follows top-down approach to design algorithms for solving large problems. It consists of dividing the problem into small size problems that are called sub problems.
- **Searching techniques:** These techniques are used to retrieve a particular record from a list of records in an efficient manner so that the least possible time is consumed.
- **Sorting techniques:** Another technique of divide and conquer strategy is sorting. It can be applied on a list of data, so that the list can be analysed easily for making accurate decisions.
- **Algorithm complexity:** The complexity of an algorithm is a function that provides the running time and space for data depending on the size provided by you.
- **Strassen's matrix multiplication:** In linear algebra, the Strassen's algorithm is used for matrix multiplication. It is faster than the standard matrix multiplication algorithms and is useful for solving large matrices.

## 2.9 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Searching techniques are used to retrieve a particular record from a list of records in an efficient manner so that it consumes the least possible time.
2. The different searching techniques are as follows:

NOTES

A. Linear search
B. Binary search

3. Binary search is a method, which involves the comparison of the search data with the data at the centre of the list. It also involves a comparison of elements in the first half or second half of the list.
4. Sorting techniques can be applied on a list of data. The list can be analyzed easily for making accurate decisions.
5. The different sorting techniques are as follows:
   a. Selection
   b. Merge
   c. Quick
6. Selection sort is a technique in which the smallest element of an array is searched and swapped with the first element of an array.

## 2.10 QUESTIONS AND EXERCISES

### Short-Answer Questions

1. Define sorting.
2. Write the algorithm for a binary search.
3. How can you implement merge sort? Explain with an example.
4. Define the searching technique.

### Long-Answer Questions

1. Explain the different sorting methods.
2. What is quick sort? Explain with the help of an example.
3. Explain selection sorting along with an example.
4. Compare the timing complexities of various sorting algorithms.

## 2.11 FURTHER READING

Horowitz, Ellis, Sartaj Sahni and Sanguthevar Rajasekaran. 2006. *Fundamentals of Computer Algorithm.* New Delhi: Galgotia Publication Pvt. Ltd.

# UNIT 3 INTRODUCTION TO GREEDY METHOD

NOTES

## Structure



## 3.0 INTRODUCTION

You are already familiar with searching and sorting techniques. In this unit, you will learn about the greedy method. It is an important design technique that can be applied to calculate the optimal solution for a number of problems. The input that seems best at the current moment is selected iteratively by following the greedy choice property at each step until an optimum solution is obtained. The knapsack and job sequencing problems can be easily resolved by following the corresponding greedy method algorithms.

You will also learn about Prim's algorithm which is based on the greedy method approach. A minimum-cost spanning tree can be determined by following Prim's algorithm. At each step a suitable edge is added to the tree so that the sum of all the selected edges remains the lowest. Kruskal's algorithm can also be used for determining the minimum-cost spanning tree. While following Kruskal's algorithm, the edge with the lowest cost is selected.

## 3.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Understand the greedy method design technique
- Determine the optimal solution for a knapsack problem by following the greedy method approach
- Determine the optimal solution for job sequencing with deadlines
- Explain Prim's and Kruskal's algorithms

NOTES

## 3.2 OVERVIEW OF THE GREEDY METHOD

The greedy methods are simple and straightforward algorithms that are used to find optimal solutions for a wide variety of problems. These algorithms follow an approach where the most optimum option is selected at each and every step. This selection procedure is based on the information available at the current step. The problems generally consist of n inputs, which can be solved by finding a subset that satisfies some constraints. This subset of inputs that leads to an optimal solution is known as the feasible solution. Initially, the solution set is empty. At each step the selection function is used to choose the best option and the selected item is added to the solution set. The item under consideration is rejected if the solution set is no longer accessible and is never considered again. Otherwise if it is feasible, then the item is merged to obtain an updated solution set.

At each step, the greedy choice property is followed and the input that seems best at that moment is selected iteratively, until the optimum solution is met. With each greedy choice the given problem is reduced into smaller sub problems, each of which can be further solved. Thus, in order to find the optimal solution for a problem, the greedy technique will generate sub-optimal solutions. This version of greedy method algorithm is known as subset paradigm. The following algorithm helps use the subset paradigm in the greedy method:

```
1. Algorithm Greedy(a,b)
2. //a[1:b] contains the n inputs
3. {
4. solution = ∅; // solution initialized
5. for n = 1 to b
6. do
7. {
8.     s = Select(a);
9.     if Feasible(solution,s)then
10.    solution = Merge(solution,s)
11. }
12. return solution;
13. }
```

In the above algorithm, three functions are defined namely, Select, Feasible and Merge. The set *a*[] refers to the inputs available for the given problem. The Chose function selects an input from set *a*[] and assigns its value to *s*. Meanwhile, the selected input value is removed from *a*[]. The feasible function represents a Boolean function that helps to decide whether value of s will provide a feasible solution or not. If *s* can be included into the solution vector, then Merge function is further used to combine the value of s with the solution. Thus, for a given problem, the Greedy function explains the greedy algorithm by using Select, Feasible and Merge functions.

Sometimes, the greedy algorithm is unable to find the optimal solution for the problems. This happens because of certain choices made during the early phase of the algorithm, which prevent from reaching the optimal solution. However,

greedy approach is still a preferred method for solving problems because it is simple to implement and time efficient. Some of the commonly used algorithms such as Prim's algorithm and Kruskal's algorithm are based on greedy method design technique.

NOTES

## 3.3 FRACTIONAL KNAPSACK

The knapsack problem involves in selecting the best available objects in a knapsack having fixed capacity. Let us consider a given set of m objects and the knapsack with the capacity c. Each object $j$ has a weight of $w_j$ and a profit value of $p_j$. The aim is to fill the knapsack with the selected objects so that it maximizes the total profit. Along with this, the total weight of all the selected objects should be less than or equal to the capacity of the knapsack. By using the greedy method, the objects can be broken down into pieces and can be used to fill the knapsack. This version of the solution is termed as fractional knapsack. Consider a fraction $f_j$ of object j such that $0 \le f_j \le 1$. By adding this fraction to knapsack, the profit earned will be $f_j\, p_j$ and the weight will be $f_j\, w_j$. Mathematically, the knapsack problem can be formulated as,

maximize $\sum_{1 \le j \le m} p_j f_j$

subject to $\sum_{1 \le j \le m} w_j f_j \le c$

where, $0 \le f_j \le 1$ and $1 \le j \le m$

The optimal solution will be any set of objects $f_1, f_2, \ldots, f_m$, which maximizes the first equation (knapsack profit) and satisfies the second equation (knapsack capacity).

Since the knapsack involves in finding the subset of the given objects, it can be solved by following subset paradigm. By choosing the object, knapsack also involves in selecting $f_j$ fraction of each object. The following algorithm is termed as Greedy Knapsack algorithm. It helps to find the optimal solution for the knapsack problem:

```
1.   for j = 1 to m do
2.     f[j] = 0
3.   weight = c
3.   for j = 1 to m do
4.   {
6.         if (w[j]>weight)then break;
7.         x[j] = 1
8.         weight = weight-w[j]
9.   }
10.  if(j<=m)then f[j]= weight/w[j]
```

NOTES

In the above algorithm, w[1:m] and p[1:m] represent the weight and profit of n objects, such that $\frac{p[j]}{w[j]} \geq \frac{p[j+1]}{w[j+1]}$. Also, f[1:m] represents the solution vector. While following the above algorithm, the object having maximum profit per unit capacity is selected at each step. Thus, you just need to arrange the objects in order of the ratio $p_j/w_j$. The theorem 3.1 shows that the above algorithm always leads to an optimal solution.

**Theorem 3.1**: If $p_1/w_1 \geq p_2/w_2 \geq ... \geq p_m/w_m$, then GreedyKnapsack algorithm will generate an optimal solution to the given instance of the knapsack problem.

**Proof**: Consider $f=(f_1, f_2, ..., f_m)$ be the solution generated by the following GreedyKnapsack algorithm. The solution is referred as the optimal solution for all $f_j$ that are equal to one. Also, consider k as the smallest index such that $f_k \neq 1$. Let $g=(g_1, g_2, ..., g_j)$ be the optimal solution. While following the algorithm, it can be determined that,

$f_j=1$ for $1 \leq j < k$

$f_j=0$ for $k < j \leq m$ and $0 \leq f_k < 1$

Since an optimal solution will fill the knapsack exactly, we can assume that $\sum w_j g_j = c$. Now consider a as the least index where, $f_a \neq g_a$. It follows that $f_a < g_a$. It implies that either a>k, a=k, or a<k. Let us analyze these three options.

- If a>k, then $\sum w_j g_j > c$, which is not possible.
- If a=k, then either $f_a < g_a$ or $\sum w_j g_j > c$. This happens because for $1 \leq j < 1$ either $\sum w_j g_j = c$ or $g_j = f_j$.
- If a<k, then $f_a = 1$. And since, $f_a \neq g_a$, therefore, $g_a < f_a$.

If $g_a$ is increased to $f_a$ and $(g_{a+1}, ..., g_m)$ is decreased such that the total capacity used remains c, then a new solution, $s=(s_1, s_2, ..., s_m)$ will be generated where, $s_j = x_j$ and $\sum_{a<j\leq m} w_j(g_j - s_j) = w_a(s_a - g_a)$. Therefore, it can be determined that for s, we have,

$$\sum_{1\leq j\leq m} p_j s_j = \sum_{1\leq j\leq m} p_j g_j + (s_a - g_a) w_a \frac{p_a}{w_a} - \sum_{a<j\leq m} (g_j - s_j) w_j \frac{p_j}{w_j}$$

$$\geq \sum_{1\leq j\leq m} p_j g_j + \left[ (s_a - g_a) w_a - \sum_{a<j\leq m} (g_j - s_j) w_j \right] \frac{p_a}{w_a}$$

$$= \sum_{1\leq j\leq m} p_j g_j$$

Since $\sum p_j s_j = \sum p_j g_j$, it implies that either s=f or $s \neq f$. In case, s and f are equal, then f is the optimal solution. However, if $s \neq f$, then following the above steps repeatedly will show that either g is not an optimal solution, or f will be transformed into g showing that f is also an optimal solution. Hence, the theorem is proved.

NOTES

## 3.4 JOB SEQUENCING WITH DEADLINES

Greedy method is also used to find the optimal solution for a problem that which involves in sequencing the jobs, so that each of them is completed by its deadline. Consider a set of m jobs, where only one job is available for processing the jobs. If *j* is a job, then $d_j \geq 0$ and $p_j \geq 0$ represent the deadline and profit associated with the job. If the job j is completed by its deadline, then profit $p_j$ is obtained. The feasible solution S is a subset of jobs, so that the jobs in this subset are finished on time. The sum of the profits of jobs in subset S is given as $\sum j \in S p_j$. The optimal solution will maximize this profit value. The following algorithm presents the high-level description of job sequencing:

```
1.   S = {1}
2.   for j=2 to m do
4.   {
5.         if (all jobs in S∪{j} can be completed by their
6.         deadline)
7.         then S=S∪{j}
8.   }
```

In the above algorithm 3.3, S refers to the subset of jobs that can be completed by the given deadline. At each step, a new job is selected and added to S if that job can be accomplished on time. Thus, set S is a feasible solution for the given problem. Theorem 3.2 helps to determine the order in which selected jobs can be processed.

**Theorem 3.2**: Consider S as the set of m jobs and $\sigma = j_1, j_2, \ldots, j_m$ as a permutation of jobs in S so that $d_{j1} \leq d_{j2} \leq \ldots d_{jk}$. If by following the sequence order $\sigma$, the jobs in S can be completed by their deadline, then S is a feasible solution.

**Proof**: To prove the solution, it is required to show that if S is feasible, then $\sigma$ refers to a possible order in which jobs need to be processed. There also exist $\sigma' = k_1, k_2, \ldots, k_m$ such that $d_{kp} \geq p$ and $1 \leq p \leq m$. If we assume that $\sigma' \neq \sigma$, then x is the smallest index so that $k_a \neq j_a$. Also, let $k_b = j_a$ then b>a. Further $k_a$ and $k_b$ can be interchanged in $\sigma'$. Now, since $d_{ka} \geq d_{kb}$, the new permutation $\sigma'' = l_1, l_2, l_3, \ldots, l_m$ depicts the order in which jobs can be completed with by given deadlines. Thus, repeating the above steps $\sigma'$ can be transformed into $\sigma$. Hence, the theorem is proved.

**CHECK YOUR PROGRESS**

1. What is the greedy method design technique?
2. What is the main advantage of the greedy method?
3. What is a fractional knapsack?

NOTES

Let us consider another Greedy algorithm for sequencing the given jobs on the basis of deadline and profit value. The sorting of jobs as described in Theorem 3.2 can be avoided by keeping the jobs in order of their deadlines. The following algorithm describes this version of Greedy method:

```
1.   d[0]= S[0]=0
2.   S[1]=1
3.   m=1
4.   for j=2 to x
5.   do
6.   {
7.         k=m;
8.         while ((d[S[k]]>d[j])and (d[S[k]]≠k)) do k=k-1
9.         if((d[S[k]]≤d[j]) and (d[j]>k)) then
10.        {
11.        for t=m to (k+1)
12.        step-1 do S[t+1]=S[q]
13.        S[k+1]=j
14.        M=m+1
15.        }
16. }
return m
```

In the above algorithm, the d[j] refers to the deadline of a job j. These jobs are ordered such that the profit value is p[1]≥ p[2] ≥p[3] ≥....≥p[x]. An array d[1:m] can be used to store the deadlines of the jobs in accordance with their profit values. The solution set S can be depicted as one-dimensional array S[1:k] such that S[k], 1≤k≤m are the jobs in set S and d[S[1]]≤ d[S[2]]≤ d[S[3]]≤... ≤d[S[m]]. By inserting j into S and then verifying that d[S[k]]≤k, you can easily identify whether S∪{j}is feasible or not. As the algorithm shows, the positions of jobs S[t], S[t+1] , S[t+2], ..., S[m] are changed, when job j is inserted at position t.

## 3.5 PRIM'S ALGORITHM

Prim's algorithm is based on greedy method approach which is used to find a minimum-cost spanning tree. A spanning tree of a graph is represented as a sub-graph containing all the vertices and forming a tree structure. The graph may have some numerical costs attached with the vertices. A minimum-cost spanning tree refers to a tree containing all the edges of which the sum of the costs associated with the edges is minimum. In the greedy method, at each step an edge is selected in such as way that the sum of costs of edges remains minimum. Consider the following Prim's algorithm for finding a minimum-cost spanning tree where G as the set of edges:

```
1.   Let (a, b) as an edge of minimum cost in G
2.   mncost=cost[c, b]
3.   r[1,1]=a
```

NOTES

```
4.   r[1,2]=b
5.   for c=1 to m  // initialize min[]
6.   do
7.         if (cost[c, b]<cost[c, a]then min[c]=b
8.         else min[c]=a
9.   min[a]=min[b]=0
10.  for c=2 to m-1 do
11.  {  // find -2 additional edges for r
12.        Let d be an index so that min[d]≠ 0 and
13.        cost[d, min[d]] us minimum
14.        s[c,1]=d
15.        s[c,2]=min[d]
16.        mncost=mncost + cost[d, min[d]]
17.        min[d]=0
18.        for a=1 to m do  // update min[]
19.        if((min[a]≠0) and (cost[d, min[a]]>cost[a, d]))
20.        then min[a]=d
21.  }
22.  return mncost
```

In the above algorithm, cost[1:m, 1:m] is a matrix of m vertices in a graph. The process starts with a tree having only one minimum-cost edge of G. The remaining edges are added one by one. The edge (c, d) is the next edge to be added in the tree. Here, the vertex c is already included into the tree and d is yet to be added. Also, cost(c, d) represents the weight attached with the edge (c, d), which is minimum among all the available edges. In order to determine such a minimum-cost edge, a value min[d] is attached with each vertex that is not yet added to the tree. Among all the choices, the value min[d] is a vertex in the tree such that cost[d, min[d]] is minimum. To calculate the minimum-spanning tree Prim's algorithm will require $O(m^2)$ time, where m is the number of vertices in graph G.

**Example 1**: Consider the graph shown in Figure 3.1.

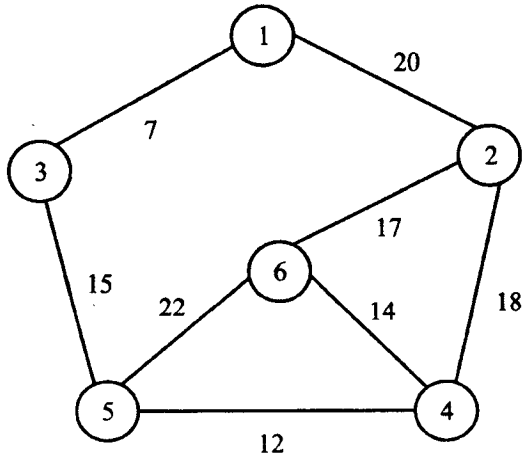



***Figure 3.1** Displaying a Graph*

NOTES

In order to generate the minimum-cost spanning tree, we follow Prim's algorithm. Since Prim's algorithm is based on greedy method, the process starts with selection of an edge having the minimum cost. Figure 3.2(a) to (e) shows the various steps involved in finding the minimum-cost tree for the given graph.

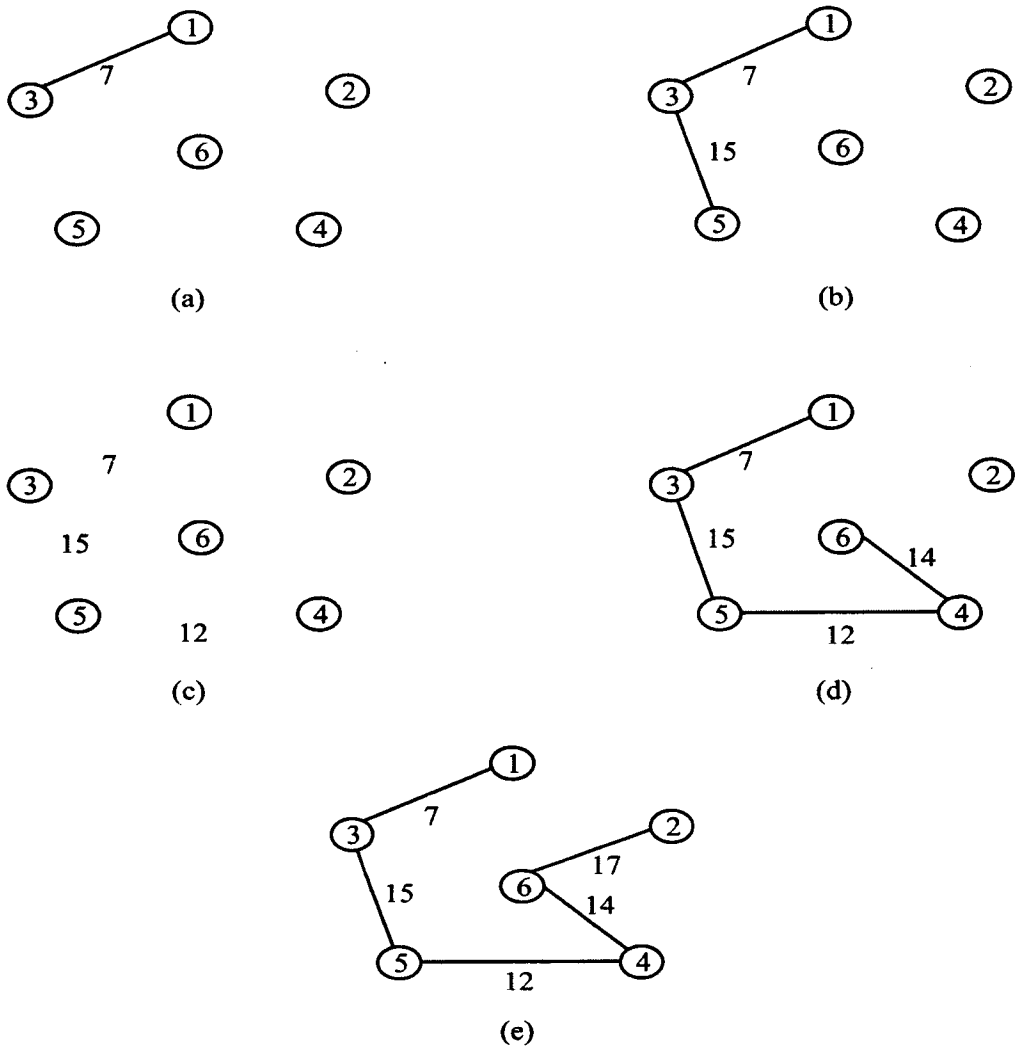



***Figure 3.2** Steps of Prim's algorithm*

Since the edge (1,3) has the lowest cost, therefore, it is selected as the first edge of the minimum-cost spanning tree. The resulting tree is shown in Figure 3.2(a). In the next step, the edge (3,5) is selected and included into the tree as depicted in Figure 3.2(b). At vertex 5 there can be two edges which include (5,6) and (5,4). The cost of edge (5,4) is lower than that of (5,6). Therefore, in Figure 3.2(c) edge (5,6) is discarded and edge (5,4) is selected as the next edge of the tree. Again, at vertex 6 there can be two edges. Here, cost of edge (4,6) is less than cost of edge (4, 2). Therefore, edge (4,6) is included in the tree, as shown in Figure 3.2(d). Finally, the edge (6,2) is added to form the minimum-cost spanning tree. Figure 3.2(e) shows the final minimum-cost of the spanning tree with cost of sixty-five.

### 3.5.1 Modified Prim's Algorithm

Prim's algorithm can be further modified, so that we can determine a minimum-spanning tree faster. Let r is the minimum-cost spanning tree of graph P=(V,E),

where a is a vertex in r. Suppose (a, b) is the edge having the minimum cost among all the edges touching the vertex a. For all the edges (a, c)$\in$E(r), assume that (a, b)$\notin$E(r) and cost[a, b] < cost[a, c]. A cycle is created with inclusion of (a, c) in the tree r. An edge (a, b) is also included in this cycle such that a$\neq$b. Without disconnecting the graph (V,E(r)$\cup${(a,b)}), this cycle can be broken by removing the edge (a,c) from E(r) $\cup${(a,b)}. As a result, (V,E(r)$\cup${(a,b)} – {(a,c)} is also a spanning tree. The new spanning tree has lower cost than r because cost[a,b]<cost[a,c]. Thus, by following the above observation, the algorithm can be modified. The process starts with a tree having an arbitrary vertex and no edge, in which the other edges are included one by one. The updated Prim's algorithm is given as follows:

```
1.   Let (a, b) as an edge of minimum cost in G
2.   mncost=0
3.   For c=2 to m do
4.         min[c]=1  // vertex 1 is initially in r
5.   min[1]=0
6.   for c=1 to m-1 do
7.   {  // find m-1 edges for r
```

```
8.         Let d be an index so that min[d]≠ 0 and
9.         cost[d, min[d]] us minimum
10.        s[c,1]=d
11.        s[c,2]=min[d]
12.        mncost=mncost + cost[d, min[d]]
13.        min[d]=0
14.        for a=1 to m do
15.        if((min[a]≠0) and (cost[d, min[a]]>cost[a, d]))
16.        then min[a]=d
17.   }
18.  return mncost
```

## 3.6 KRUSKAL'S ALGORITHM

A minimum-cost spanning tree can also be determined using Kruskal's algorithm where edges are arranged in non-decreasing order of cost. Kruskal's algorithm is also based on another greedy method approach. Here, r represents the set of edges selected so far to make the spanning tree. The set r may not be a tree at every stage in the algorithm. Unlike Prim's algorithm (where a tree should be formed at each step) in Kruskal's algorithm, it is not necessary to form a tree structure at each step. Two distinct edges can be selected in consecutive steps. An edge is discarded if its inclusion leads to formation of a cycle. Consider the following algorithm that gives a high-level description of Kruskal's algorithm:

NOTES

```
1.  r=∅
2.  while((r has less than m-1 edges) and (E≠∅)) do
3.  {
4.  Choose edge (a, b) from E of lowest cost
5.  Delete (a, b) from E;
6.  if (a, b)does not create a cycle in r
7.     then add (a, b) to r
8.  else
9.      discard (a, b)
10.}
```

In the above algorithm, initially the set r is empty. At each step, an edge (a, b) is selected from the set of available edges E. In case, the inclusion of edge (a, b) in r does not create any cycle, then the selected edge is added into the tree r. Otherwise, edge (a, b) is not added in the tree r. In order to determine the minimum-cost edge at each step efficiently, the edges are sorted and arranged in a heap. As a result of heap structure, the next edge can be obtained in O(log|E|) time. The following updated Kruskal's algorithm gives the detailed steps for calculating the minimum-coat spanning tree:

```
1.  Construct a heap out of the edge costs
2.  for j=1 to m do
3.    parent[j]= -1
4.  j=0
5.  mncost=0
6.  while((j<m-1) and (heap not empty)) do
7.  {
8.        Delete a minimum cost edge (a, b) from the heap
9.        k=Find(a)
10.       l=Find(b)
11.       if (k≠l) then
12.       {
13.         j=j+1
14.         r[j, 1]=a
15.         r[j, 2]=b
16.         mncost=mncost+cost[a,b]
17.         Union(k,l)
18.       }
19. }
20. if (j≠m-1) then write ("No spanning tree")
21. else
22. return mncost
```

In the above algorithm, we consider a graph H having m vertices. The cost[a, b] is associated with each edge (a, b). Also, let r be the set of edges that forms the minimum-cost spanning tree, having j number of edges. Initially all the edges are stored in a heap and each vertex is assigned to a distinct set. With the assignments r[j, 1]=a and r[j,2]=b, the edge (a, b) is included into the tree set, r. Meanwhile, the edges are removed from the heap in non-decreasing order of cost. The sets containing a and b vertices are also determined. If k≠l then both the vertices a

and b are in different sets. The sets containing a and b are combined and the edge (a, b) is added to the tree r. The edge (a, b) is discarded, if a = b because its inclusion will create a cycle. Following the above algorithm for computing the time will be O(|E| log |E|), where E represents the set of edges in the given graph.

**Example 2**: Consider the following graph shown in Figure 3.3.

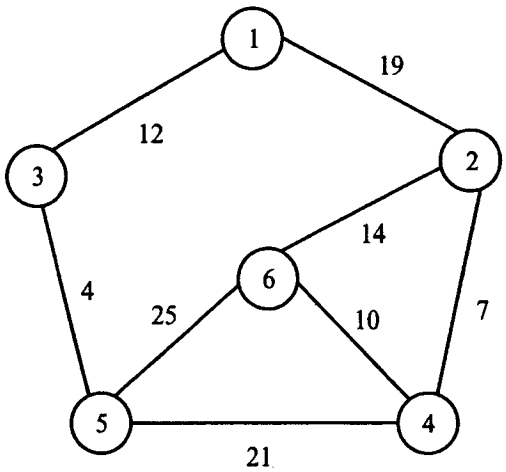



***Figure 3.3** Graph*

In order to determine the minimum-cost of a spanning tree, we follow Kruskal's algorithm. Figure 3.4 (a) to (f) shows the various steps involved in finding the minimum-cost tree for the given graph.

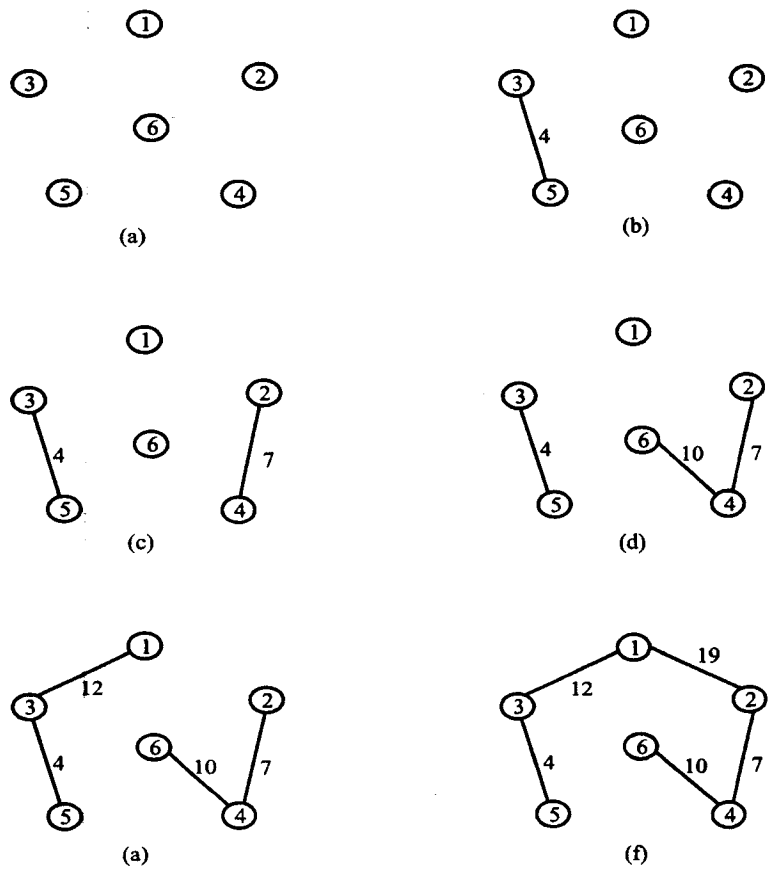



***Figure 3.4** Steps of Kruskal's algorithm*

**NOTES**

**CHECK YOUR PROGRESS**

4. What is the feasible solution for a job sequencing problem?
5. What is a spanning tree?
6. What is the main difference between Prim's algorithm and Kruskal's algorithm?

NOTES

Initially, no edges are selected as shown in Figure 3.4(a). Since the edge (3,5) has the lowest cost, therefore (3,5) is selected as the first edge of the minimum-cost spanning tree. Figure 3.4(b) shows the resulting graph. In the next step, the edge (2,4) is selected and included into the tree as shown in Figure 3.4(c). Since the inclusion of edges (4,6) and (1,3) does not create any cycle, they are also included into the tree, as shown in Figure 3.4(d) and Figure 3.4(e) respectively. Now in the remaining edges, the edge (2,6) has the least cost. If edge (2,6) is included in the tree, then a cycle is formed. Therefore, edge (2,6) is discarded. The next edge with lowest cost is (1,2). Since the inclusion of edge (1,2) does not create any cycle, therefore it is added to the tree. Figure 3.4(f) shows the final minimum-cost tree having cost of fifty-two.

**Theorem 3.3**: Kruskal's algorithm will generate a minimum-cost spanning tree for every connected undirected graph H.

**Proof**: Assume H as the undirected connected graph and r is the spanning tree for H determined by using Kruskal's algorithm. Also, let s be the minimum-cost spanning tree for graph H. To prove the theorem, we need to show that r and s have the same cost.

Consider E(r) and E(s) be the edges in tree r and s, such that either E(r)=E(s) or $E(r) \neq E(s)$. In the first case, the s will have minimum cost. For the latter case, let k as a minimum-cost edge so that $k \in E(r)$ and $k \notin E(s)$. A unique cycle will be created when k is included into s. Let k, $d_1$, $d_2$, $d_3$, ... , $d_x$ be this unique cycle. Let $d_y$ be an edge in this cycle so that $d_y \notin E(r)$. If $cost(d_y) \geq cost(k)$, i.e. $d_y$ has lower cost than k, then $d_y$ will be added to the tree r. Now consider the graph having edge set $E(s) \cup \{k\}$. If an edge is removed from the cycle k, $d_1$, $d_2$, $d_3$, ..., $d_x$, then a new tree t will be formed. Specifically, the tree formed by removing the edge $d_y$ will have less cost than the cost of tree s. Since $cost(d_y) \geq cost(d)$, the new tree t is also a minimum-cost spanning tree. With repeated transformations, the tree s can be converted to a spanning tree without any increase in the cost. Therefore, tree r is a minimum-cost spanning tree.

## 3.7 SUMMARY

In this unit, you have learnt about the computation of an optimal solution for several problems. The optimal solution can be obtained with the help of the greedy method. This method follows a straightforward design in which the best option at the current stage is selected and the problem is divided into smaller sub-problems. The sub-optimal solutions are found following a subset paradigm of the greedy method. The optimal solution for knapsack and job sequencing problems can be determined by following the subset paradigm.

You have also learnt about minimum-cost spanning tree in which the sum of all costs associated with each edge is minimum. Either Prim's algorithm or Kruskal's algorithm can be used to generate a minimum-cost spanning tree. While following Prim's algorithm, at each step, a tree is formed and the sum of the cost of the edges of the tree remains minimum. However, in Kruskal's algorithm, it is not necessary to form a tree structure at each step. The edge with the lowest cost is selected at each step so that a minimum-cost tree is generated finally.

NOTES

## 3.8 KEY TERMS

- **Greedy method:** The greedy methods are simple and straightforward algorithms that are used to find optimal solutions for a wide variety of problems. These algorithms follow an approach where the most optimal option is selected at each and every step.
- **Fractional knapsack:** The knapsack problem involves in selecting the best available objects in a knapsack having fixed capacity.
- **Prim's algorithm:** Prim's algorithm is based on greedy method approach that is used to find a minimum-cost spanning tree. A spanning tree of a graph is represented as a sub-graph containing all the vertices and forming a tree structure.
- **Kruskal's algorithm:** Kruskal's algorithm is also based on another greedy method approach. In this not necessary to form a tree structure at each step in this method.

## 3.9 ANSWERS TO 'CHECK YOUR PROGRESS'

1. In the greedy method design technique, the optimum option is selected at each step. This selection procedure is based on the information available at the current step.
2. The main advantage is that it is simple to implement and requires less time for finding the optimal solution.
3. Fractional knapsack is based on the greedy method in which the objects are broken down into smaller pieces and then suitable pieces are put into the knapsack to attain the maximum profit.
4. The feasible solution for job sequencing problem is a subset S of jobs, where jobs are arranged in such a way that all the jobs in this subset are finished on time.
5. A spanning tree of a graph is represented as a sub-graph containing all the vertices and forming a tree structure. A spanning tree may have some numerical costs attached with the vertices.
6. In Prim's algorithm, a tree structure should be formed at each step. However, in case of Kruskal's algorithm, it is not necessary to form a tree structure at each step.

## 3.10 QUESTIONS AND EXERCISES

### Short-Answer Questions

1. What are the roles of the select, feasible and merge functions involved in the greedy method algorithm?
2. What is the disadvantage of the greedy method?
3. Explain the knapsack problem.
4. Write the greedy method algorithm giving a high-level description of job sequencing.

NOTES

5. Describe the basic concept of Prim's algorithm.
6. Explain the modified Prim's algorithm.
7. Explain Kruskal's algorithm.

### Long-Answer Questions

1. Write the algorithm for a subset paradigm.
2. Explain the Greedy Knapsack algorithm.
3. Give a detailed description of job sequencing with deadlines.
4. How will you determine a minimum-cost spanning tree by following Prim's algorithm? Also write Prim's algorithm.
5. Explain Kruskal's algorithm.

## 3.11 FURTHER READING

Horowitz, Ellis, Sartaj Sahni and Sanguthevar Rajasekaran. 2006. *Fundamentals of Computer Algorithms.* New Delhi: Galgotia Publication Pvt. Ltd.

# UNIT 4 DYNAMIC PROGRAMMING

NOTES

## Structure



## 4.0 INTRODUCTION

You are already familiar with the greedy method and job sequencing problems. In this unit, you will learn about the concept of dynamic programming. In dynamic programming, the problem is divided into sub-problems and the solution of a sub-problem can be used to find the solution for the entire problem. The main characteristic of dynamic programming is that it reduces the re-computation effort by storing the solution of various sub-problems. Dynamic programming is mainly useful in solving the following types of problems:

- Multistage graph problem
- Shortest path problem
- 0/1 Knapsack problem
- Travelling salesperson problem
- Longest common subsequence
- Matrix multiplication problem

## 4.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Explain the concept of dynamic programming
- Explain the characteristics of dynamic programming

NOTES

- Define the steps involved in dynamic programming
- Explain how the multistage graph problem is solved
- Describe how dynamic programming can be implemented in the shortest path problem
- Implement dynamic programming to solve 0/1 knapsack problem
- Understand the travelling salesperson problem
- Explain the use of dynamic programming to provide solutions for the longest common subsequence problem
- Explain matrix chain multiplication problem

## 4.2 OVERVIEW OF DYNAMIC PROGRAMMING

In the analysis and design of algorithms, there are various designing strategies such as recursion, divide-and-conquer, dynamic programming, greedy method, backtracking and branch-and-bound. These techniques help to develop an algorithm and solve the given problem. Dynamic programming is a design methodology that helps in building up the solution of a problem by combining the solutions of several sub problems. In general, it is applicable in a situation where the given problem is to be optimized to an objective function and where the combination of all the optimal solutions to the subproblems provides the optimal solution to the given problem. You can use dynamic programming to design the strategy for solving different sorts of problems. Some of them include:

- Multistage graph problem
- Shortest path problem
- 0/1 Knapsack problem
- Travelling salesperson problem
- Longest common subsequence
- Matrix multiplication problem

### 4.2.1 Characteristics of Dynamic Programming

Dynamic programming is an algorithm design technique, which has the following characteristics:

- It solves the problems by combining the solutions of all sub problems.
- In dynamic programming, all the sub problems are dependent but the solutions of sub problems may not affect the solutions to other sub problems of the same problem.
- It reduces the computation by:
    - Solving the sub problems in bottom-up fashion.
    - Storing the solution of a sub problem for the first time it is solved.
    - Looking up the solution when a sub problem is encountered again.
- It depends on the principle of optimality, which states that in an optimal sequence of solutions, each subsequence must also be optimal.

### 4.2.2 Steps in Dynamic Programming

The dynamic programming process involves the following steps to find the solution of given problem.

1. Characterize the structure of an optimal solution.
2. Define the value of optimal solution recursively.
3. Compute the optimal solution values either by top-down or with caching the bottom-up in a table.
4. Construct an optimal solution from the computed values.

NOTES

## 4.3 MULTISTAGE GRAPHS

A multistage graph is a directed graph $G$ that has a pair $(V, E)$ where $V$ represents set of vertices. It is a finite set and $E$ represents set of edges that has a binary relation on $V$. Pairs of vertices represent the edges such that the pair $(V_1, V_2)$ represents the edge from the vertex $V_1$ to vertex $V_2$. In other words, a multistage graph is a graph in which:

- $G = (V, E)$, where, vertices $V$ are partitioned into $k \geq 2$ disjoint subsets such that if $(a, b)$ are the elements of the set of edges $E$, then $a$ is in $V_i$ and $b$ is in $V_{i+1}$ for some subsets in the partition. Where, $i$ is less than $k$ and greater than equal to 1, i.e. $1 \leq i < k$.
- In graph $G = (V, E)$, the sets $V_1$ and $V_k$ are such that $| V_1 | = | V_K | = 1$.

Let us consider that $s$ and $t$ are vertices in $V_1$ and $V_k$, where $s$ and $t$ are the source and sink respectively. Now, suppose $c(i, j)$ is the cost of the edge $<i, j>$. The cost of the path from $s$ to $t$ is the sum of the costs of all the edges on the path. In the multistage graph problem, we need to find a minimum-cost graph from $s$ to $t$.

In the graph $G$, each set $V_i$ defines a stage and every path from $s$ to $t$ starts from stage 1. The path then goes to stage 2, then stage 3 and finally terminates in some stage $k$. Figure 4.1 shows a five-stage graph where minimum-cost path from $s$ to $t$ is indicated by the darkened lines.

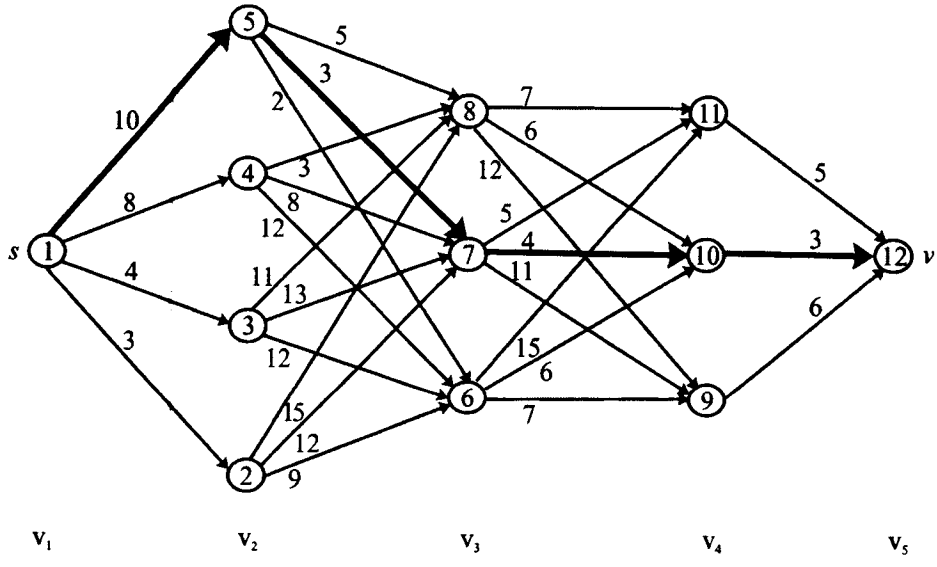



*Figure 4.1 Five-stage Graph to Determine Minimum-cost Path*

NOTES

From Figure 4.1, it is easy to observe that traversing through the path that which includes nodes 1, 5, 7, 10 and 12, costs minimum.

You can use dynamic programming to formulate a k-stage graph problem, where every path from $s$ to $t$ is a result of a sequence of $k-2$ decisions. Here, the $i^{th}$ decision helps in determining the vertex in $V_{i+1}$, $1 \le i \le k$, on the path. Suppose, $p(i, j)$ is the minimum-cost path from a vertex $j$ in $V_i$ to vertex $t$. If cost$(i, j)$ is the cost of this path, then using the forward approach, we can obtain

$$\text{cost}(i,j) = \min_{\substack{l \in V_{i+1} \\ \langle j,l \rangle \in E}} \{c(j, l) + \text{cost}(i+1, l)\} \quad (4.1)$$

Now, cost$(k-1, j) = c(j, t)$ if $<j, t> \in E$ and cost$(k-1, j) = \infty$ if $<j, t> \notin E$. Therefore, equation 4.1 can be solved for cost$(1, s)$ by computing cost$(k-2, j)$ first for all $j \in V_{k-2}$, then cost$(k-3, j)$ for all $j \in V_{k-3}$, and so on, and finally cost$(1, s)$.

To write an algorithm to solve the equation 4.1 for a general k-stage graph, let us impose an ordering on the vertices in $V$, which makes the algorithm writing easier. It is needed that $n$ vertices in $V$ are indexed through 1 to $n$ and these indices are assigned in order of the stages. The first index 1 is assigned to $s$, then vertices in $V_2$ are assigned followed by vertices on $V_3$ and so on. Finally, n is assigned to vertex $t$. From this, it is cleared that indices assigned to vertices in $V_{i+1}$ are greater than those assigned to the vertices in $V_i$. This indexing scheme results in computation of cost and $d$ in the order of $n-1, n-2, \ldots, 1$, where $d$ is the value of a node $l$ that minimizes $c(j, l) + \text{cost}(i+1, l)$. This approach is known as forward approach and the resulting algorithm, in pseudocode is called FGraph. Since, the first subscript in cost, $p$ and $d$ only identifies the stage number; therefore, it is omitted in FGraph algorithm.

The algorithm for multistage graph pseudocode corresponding to the forward approach is as follows:

```
//Algorithm FGraph(G, k, n, p)
/* The input is a k-stage graph G = (V,E) with n vertices
indexed in order to stages. E is a set of edges and
c(i,j)is the cost of <i,j>. p[1:k]is a minimum-cost path.*/
1. {
2. cost[n]:=0.0;
3. for j:=n-1 to 1 step -1 do
4. {//Compute cost[j].
5. Let r be a vertex such that <j, r> is an edge of G and
6. c[j,r]+cost[r] is minimum;
7. cost[j]:= c[j,r]+ cost[r];
8. d[j]:=r;
9. }
10.    //Find a minimum-cost path.
11.    p[1]:=1;p[k]:=n;
12.    for j:=2 to k-1 do p[j]:=d[p[j-1]];
}
```

NOTES

You can also solve the multistage graph problem with the help of backward approach. Let us consider that $bp(i, j)$ is a minimum cost path from the vertex $s$ to the vertex $t$. Let $b$cost$(i, j)$ to be the cost of $bp(i, j)$. Using backward approach you can find that:

$$b\text{cost}(i, j) = \min_{\substack{l \in v_{i-1} \\ \langle l, j \rangle \in E}} \{b\text{cost}(i-1, l) + c(l, j)\} \quad (4.2)$$

Since $b$cost$(2, j) = c(1, j)$ if $\langle 1, j \rangle \in E$ and $b$cost$(2, j) = \infty$ if $\langle 1, j \rangle \notin E$, $b$cost$(i, j)$ can be calculated using equation 4.2 for $i$=3 and so on. BGraph is a backward approach algorithm in pseudo code to obtain the minimum-cost path from $s$ to $t$. In this algorithm, like Fgraph, the first subscript on $b$cost, $p$ and $d$ are omitted.

Algorithm for multistage graph in pseudo code corresponding to the backward approach is as follows:

```
//Algorithm BGraph(G, k, n, p)
/* Same function as FGraph*/
1. {
2. bcost[1]:=0.0;
3. for j:=2 to n do
4. {//Compute bcost[j].
5. Let r be a vertex such that <r,j> is an edge of G and
6. bcost[r]+c[r,j] is minimum;
7. bcost[j]:= bcost [r]+c[r,j];
8. d[j]:=r;
9. }
10.    //Find a minimum-cost path.
11.    p[1]:=1;p[k]:=n;
12.    for j:=k-1  to 2 do p[j]:=d[p[j+1]];
}
```

## 4.4 SHORTEST PATH

Shortest path technique helps in finding a path between two vertices such that the sum of the cost (weight) of its ingredient edges is minimized. For example, if you want to travel from one location to another location within the city, you will prefer the path that must be the quickest way to get from one location to another on a road map. In such a situation; the vertices $V$ represent the origin and destination locations. The edges E represent segments of road that are weighted by the time needed to travel that road segment. The following are the two types of shortest path approaches:

- Single-source shortest path
- All-pairs shortest path

NOTES

## 4.4.1 Single-Source Shortest Path

The single-source shortest path is an approach to solve the shortest path problem in which you find paths from a source vertex v to all other vertices in a directed graph. Let $d^i$ [$s$] to be the length of a shortest path from the source vertex $s$ to vertex $t$ under the constraints that the shortest path includes E edges. Then, $d^i$[s] – cost [$s$, $t$] in which u is less than or equal to n and greater than or equal to 1, i.e.

$1 \leq u \leq n$.

Here we calculate $d^{n-1}[s]$ for all $s$ and this can be done with the help of dynamic programming by using the following observation.

1. The distance will be $d^{k-1}[s] = d^k[s]$, if the shortest path from $t$ to s with at the most $k$, $k$>1 and edges no more than $k$-1 edges.
2. The distance will be $d^{k-1}[i]$ + cost$[i,s]$, if the shortest path from $s$ to $t$ with at the most $k$, $k$>1, edges exactly $k$ edges, than it shows the shortest path from s to some vertex j followed by the edge $\langle j,s \rangle$. The path from $t$ to $j$ has $k$-$1$edges that has length of $d^{k-1}[j]$.

Final result of observations is:

$$d^k[s] = \min \{ d^{k-1}[s], \min_i \{ d^{k-1}[s] + \text{cost}[i,s] \} \}$$

Bellman and Ford algorithm is based on dynamic programming that helps to compute the single-source shortest paths problem in which directed graph contains some edges with negative weight. Following is the algorithm:

```
//Algorithm BellmanFord(s,cost,d,n)
/* Single-source shortest paths with negative edge costs*/
1. {
2. for i:= 1 to n do
3. d[i]:=cost[t,i];
4. for k:=2 to n-1 do
5. for each s such that s ≠ t and s has at least one
6. incoming edge do
7. for each ⟨ι,s⟩ in the graph do
8. if each[t]>d[i] + cost[i,s];
9. }
```

**Note:** *Dijkstra's algorithm helps to compute the same problem with non-negative edge weights. Thus, Bellman–Ford is used only when there are negative edge weights in a directed graph.*

To understand the single-source shortest path problem, consider the following example.

**Example 1**: Consider the graph with seven-vertex that has array of distance $d^k$, k = 1, 2, ….,6. Therefore, $d^k$ [1] = 0 for all $k$ and 1 is the source node. Figure 4.2 shows all the nodes and distances between nodes.

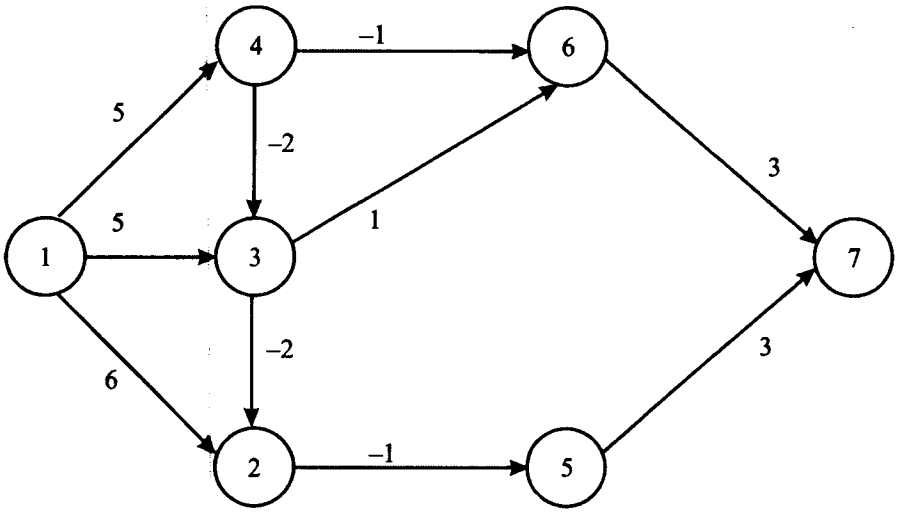



***Figure 4.2** The Directed Graph*

**Solution**: To find out the solution of single-source shortest path problem, consider the following equation.

$d^k[s] = \min \{ d^{k-1}[s], \min_i \{ d^{k-1}[s] + \text{cost}[i,s]\}\}$

Where,

$\Rightarrow d^2[2] = \min \{ d^{k-1}[2], \min_i \{ d^{2-1}[i] + \text{cost}[i,2]\}\}$

$\Rightarrow d^2[2] = \min \{ d^{k-1}[2], \min_i \{ d^1[i] + \text{cost}[i,2]\}\}$

$\Rightarrow d^2[2] = \min \{ 6,0 + 6, 5\text{-}2, 5 + \infty, \infty + \infty, \infty + \infty, \infty + \infty\} = 3$

Resultant matrix is:

| $k$ | $d^k$ [1....7] | | | | | | |
|---|---|---|---|---|---|---|---|
| | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **1** | 0 | 6 | 5 | 5 | ∞ | ∞ | ∞ |
| **2** | 0 | 3 | 3 | 5 | 5 | 4 | ∞ |
| **3** | 0 | 1 | 3 | 5 | 2 | 4 | 7 |
| **4** | 0 | 1 | 3 | 5 | 0 | 4 | 5 |
| **5** | 0 | 1 | 3 | 5 | 0 | 4 | 3 |
| **6** | 0 | 1 | 3 | 5 | 0 | 4 | 3 |

**NOTES**

**CHECK YOUR PROGRESS**

1. What type of technique is dynamic programming?
2. Name four problems that can be solved using dynamic programming.
3. How does dynamic programming help in reducing computation?
4. What is a multistage graph?
5. What are the two approaches to solving a multistage graph problem?

NOTES

## 4.4.2 All-Pairs Shortest Path

The all-pairs shortest path is an approach to solve the shortest path problem in which you find the shortest path between all pairs of vertices. Let $G = (V, E)$ be a directed graph with edge weights. If $(s, t)$ E, is an edge of directed graph $G$, then the weight of this edge is denoted by $w(s, t)$. In this graph, the cost of a path from one node to another node is the sum of the weights of the edges along the path. In this process the graph $G$ can have negative cost edges. But graph $G$ cannot have any negative cost cycle.

This problem needs to determine a matrix $A$ where $A(i, j)$ represents the shortest path from $i$ to $j$. This matrix can be obtained by solving n single-source problems using the shortest path algorithm 4.3. This procedure requires $O(n^2)$ times of each of the application, thereby the matrix $A$ can be obtained in $O(n^3)$ times. You can also use the principle of optimality to get an alternate $O(n^3)$ solution to this problem. Here, it is required that $G$ has no cycles with negative length, rather than requiring a cost$(i, j) \geq 0$ for every edge $<i, j>$. If you allow $G$ to contain a cycle having negative value, then the shortest path between any two vertices on this cycle will have the length equal to $-\infty$.

Consider the shortest path from $i$ to $j$ in $G$, $i \neq j$, which originates at $i$ and moves through some intermediate vertices to terminate at $j$. Let us assume that this path does not contain any cycle and let one intermediate vertex be $k$. Now, the sub-paths from $i$ to $k$ and $k$ to $j$ must also be the shortest paths from $i$ to $k$ and $k$ to $j$ respectively. This holds the theory of optimality and also provides us the prospect to implement dynamic programming in all-pairs shortest paths.

If $k$ is the intermediate vertex having highest index, then the path from $i$ to $k$ will be the shortest path in $G$. This goes through no vertex having index greater than $k - 1$. Similarly, the path from $k$ to $j$ will also be the shortest path in $G$ having no intermediate vertex greater than $k - 1$. So, to construct the shortest path from $i$ to $j$, the first requirement is the decision that determines the highest indexed intermediate vertex between $i$ and $j$. After making the decision, the next requirement is to find the two shortest paths, from $i$ to $k$ and from $k$ to $j$. It is known that neither of these two paths can go through a vertex having index greater than $k - 1$. Now, if we use $A^k(i, j)$ to represent the shortest path length from $i$ to $j$ that do not go through a vertex having index greater than $k$, then

$$A(i, j) = \min\left\{\min_{1 \leq k \leq n}\left\{A^{k-1}(i,k) + A^{k-1}(k, j), \cos t(i, j)\right\}\right\} \quad (4.3)$$

From the above equation, it is clear that $A^0(i, j) = \text{cost}(i, j)$, $1 \leq i \leq n,$, $1 \leq j \leq n$. Now, using similar arguments, as used before, we can derive a recurrence for $A^k(i, j)$. It is evident that a path from $i$ to $j$, which has no intermediate vertex higher than $k$, either goes through $k$ or not. If k is the intermediate vertex, then $A^k(i, j) = A^{k-1}(i, k) + A^{k-1}(k, j)$. Otherwise, $A^k(i, j) = A^{k-1}(i, j)$. Combining these two equations, we get the following resulting equation

$$A^k(i, j) = \min\left\{A^{k-1}(i, j),\ A^{k-1}(i,k) + A^{k-1}(k, j)\right\},\ k \geq 1 \quad (4.4)$$

The above equation can be solved for $A^n$ by computing $A^1$ first, then followed by computations $A^2$, $A^3$ and so on. There is no vertex in $G$ that is greater than $n$, therefore, $A(i, j) = A^n(i, j)$. We need to use the algorithm All Paths to compute $A^n(i, j)$.

Algorithm to compute the length of shortest path is as follows:

```
Algorithm All Paths (cost, A, n)
/*cost[1:n, 1:n] is the cost adjacency matrix of a graph
with n vertices; A[i, j] is the cost of a shortest path
from vertex i to j. cost[i, j] = 0.0, for 1 ≤ i ≤ n*/
1. {
2. for i := 1 to n do
3. for j := 1 to n do
4. A[i, j] := cost[i, j]; //Copy cost into A
5. for k := 1 to n do
6. for i := 1 to d do
7. for j := 1 to n do
8. A[i, j] := min(A[i, j], A[i, k] + A[k, j];
9. }
```

## 4.5 0/1 KNAPSACK PROBLEM

The 0/1 knapsack problem is related to the change of counting in the problem, where from a given set of n items we have to select some number of items to be carried in a knapsack. Here, each item has a weight and a profit. The main objective is to select the set of items for the knapsack, which can be maximized to the profit. The 0/1 knapsack problem is the classic integer linear programming problem having a single constraint. The problem can be formulated as follows:

$\max c_1x_1 + c_2x_2 + ...c_nx_n$

where,

$a_1x_1 + a_2x_2 + ...a_nx_n \leq b$

and $x_i = 0,1$ for $i = 1,2,...,n$

Here $c_i$ represents the value of an item $i$, which is selected to include in the knapsack. $a_i$ is the weight of the item $i$, while $b$ represents the maximum weight that the knapsack can hold.

To solve the problem, we need to look for a solution that can be represented in terms of sub problems, which can be implemented by expressing the solution recursively. Therefore, the classic approach to solve the 0/1 knapsack problem is to solve the problem for one item at a time. Let us consider that the problem is solved for every weight from 0 to $b$ for each of the item. When there is no item, the maximum value becomes 0 and it does not depend on any of the available weight. This case is considered as the base case for the recursive formulation of 0/1 knapsack problem. We cannot place the first item in the knapsack until it reaches a weight $a_i$ and where the optimal value is $c_1$. Now, at each weight $y$ for the item k, it must be determined between the values of the solution, when the item is used and when it is not used. When the item is not used the value becomes $Knap(k-1, y)$, where as when the item is used, it becomes $Knap(k-1, y-a_k) + c_k$. It is clear that $y - a_k \geq 0$ for $k$ items to fit into the knapsack. In case of selecting $k$ items for the knapsack, the value of $Knap(k-1, y-a_k)$ is examined by choosing

NOTES

$k-1$ as the index of the sub problem. This in turn guarantees that the item $k$ is selected at most once for getting the solution.

The complete recursive solution of the 0/1 knapsack problem can be as follows:

$$\text{Knap}(k,y) = \text{Knap}(k-1,y) \quad \text{if } y < a_k$$

$$\text{Knap}(k,y) = \max\{Knap(k-1,y), \text{Knap}(k-1,y-a_k)+c_k\} \quad \text{if } y > a_k$$

$$\text{Knap}(k,y) = \max\{Knap(k-1,y), +c_k\} \quad \text{if } y = a_k$$

$$\text{Knap}(0,y) = 0$$

## 4.6 THE TRAVELLING SALESPERSON PROBLEM

You can apply the dynamic programming to solve the travelling salesperson problem, which is a permutation problem. The travelling salesperson problem is the best possible means of finding the lowest possible cost for visiting the various cities and returning to the starting point. This method logically occurs as a sub problem in many transportation and logistic applications. In other words it is the cheapest round trip route from one city to the other. This problem explains that if a salesman has to visit number of cities, then how should he travel so as to ensure that the distance is minimized.

Usually, permutation problems are much harder to solve in comparison to the subset problems. It is because there are $n!$ different permutations of $n$ objects, which is greater than $2^n$ different subsets of n objects. Suppose, $G = (V, E)$ is a directed graph having edge cost $c_{ij}$. Here, $c_{ij} > 0$ for all $i$ and $j$ and $c_{ij} = \infty$ when $<i, j> \notin E$. Let $|V| = n$ and assume that $n$ is greater than 1. A tour of $G$ is a directed simple cycle, which traverses every vertex in $V$. You can calculate the cost of the tour by summing up all the costs of the edges on the tour. Now the travelling salesperson problem is to find out the tour having the minimum cost.

You can find the travelling salesperson problem in a variety of applications. Suppose there are two routes for a postal van to pick up mails from the mailboxes, which are located in n different sites. This situation can be represented using an $n + 1$ vertex graph, where one vertex represents the post office from where the postal van starts for collecting the mails. Edge $<i, j>$ represents the distance covered by the van from a site $i$ to another site $j$. Here, the complete route taken by the postal van can be considered as the tour and the travelling salesperson problem will arise in finding the tour of the van that has minimum length.

Another example of travelling salesperson is a robot arm, which can tighten the nuts on some machinery on an assembly line. The robot arm starts over the first nut to be tightened, which is its initial position and then moves successively to tighten the other nuts on the machinery, and ultimately returns to its initial path. The path covered by the arm starting from its initial position till it returns back to the same position can be regarded as a tour of the graph. Here, the nuts on the machinery represent the vertices on the graph. Therefore, a minimum-cost tour can minimize the time needed by the arm to complete its task of nut tightening. The travelling salesperson problem can be experienced here in finding the minimum-cost tour for the robot arm.

NOTES

Now let us consider a simple tour, where the path starts and ends at the vertex 1, and every tour consists of an edge <1, $k$> for some $k \in V - \{1\}$ along with a path from vertex $k$ to vertex 1. The path from $k$ to 1 traverses each of the vertices in $V - \{1, k\}$ exactly once. Now, for optimal tour, the path from $k$ to 1 must be the shortest path, which goes through all the vertices in $V - \{1, k\}$. Hence, it signifies the principle of optimality. Now, consider $g(i, S)$ to be the length of the shortest path, which starts at the vertex $i$ and goes through all the vertices in $S$ to terminate at vertex 1. Therefore, the length of an optimal salesperson tour will be $g(1, V - \{1, k\})$. Thus, from the optimality principle,

$$g(1, V - \{1\}) = \min_{1 \le k \le n} \{c_{1k} + g(k, V - \{1, K\})\} \quad (4.5)$$

This equation can be generalized for $i \in S$

$$G(i, S) = \min_{j \in S} \{c_{ij} + g(j, S - \{j\})\} \quad (4.6)$$

Now, you can solve the equation 4.5 for $g(1, V - \{1, k\})$, if $g(k, V - \{1, k\})$ is known for all the choices of $k$. It is clearly seen that $g(i, \phi) = C_{i,j}$ for $1 \le i \le n$. Therefore, we can use equation 4.6 for obtaining $g(i, S)$ for all $S$ having size 1. Subsequently, we can obtain $g(i, S)$ for all $S$ of size 2, and so on. For $|S| < n - 1$, the $i$ and $S$ are such that $i \neq 1$, $1 \notin S$ and $i \notin S$.

Let $N$ be the number of $g(i, S)$'s, which are to be computed to use equation 4.5 for computing $g(1, V - \{1\})$. Here, for every value of $|S|$, there are $n - 1$ choices for $i$ and the number of distinct sets $S$ having size $k$, which do not include 1 and $i$ is $\binom{n-2}{k}$. Therefore,

$$N = \sum_{k=0}^{n-2} (n-1)\binom{n-2}{k} = (n-1)2^{n-2}$$

## 4.7 LONGEST COMMON SUBSEQUENCE

The Longest Common Subsequence (LCS) problem helps in determining the longest subsequence which is common to all the sequences in a set of sequences of two or more. A subsequence is a new sequence that is formed from the original sequence by deleting some of its elements in such a way that the deletion of the elements does not disturb the relative positions of the remaining elements in the sequence. The LCS problem is a good example of dynamic programming implementation, which can be solved using the following steps:

1. Computing the length of the LCS
2. Reading out an LCS
3. Reading out all LCS
4. Printing the difference

### 4.7.1 Computing the Length of the LCS

Consider two sequences $X[1...i]$ and $Y[1...j]$ for $i, j \ge 1$. Now the following function, length_LCS, takes two input sequences $X[1...m]$ and $Y[1...n]$ to

NOTES

calculate the LCS between the first two sequences. Here $1 \leq i \leq m$ and $1 \leq j \leq n$. The function stores the input sequences in the $C[i, j]$ table.

```
1. function  length_LCS(X[1..m], Y[1..n])
2. C = array(0..m, 0..n)
3. for i := 0..m
4. C[i,0] = 0
5. for j := 0..n
6. C[0,j] = 0
7. for i := 1..m
8. for j := 1..n
9. if X[i] = Y[j]
10.    C[i,j] := C[i-1,j-1] + 1
11.    else:
12.    C[i,j] := max(C[i,j-1], C[i-1,j])
13.    return C[m,n]
```

## 4.7.2 Reading out an LCS

You can use the following function, backTrack to retrace the choices, which are considered while computing the C table. If the last characters in the prefixes of the choices are equal, then LCS must exist. Otherwise, you need to check the choice, which gives the largest LCS for keep $x_i$ and $y_j$ and select that choice. However, if there are choices of equal lengths, then you need to select any one from them and call the function with $i = m$ and $j = n$.

```
1. function backTrack(C[0..m,0..n], X[1..m], Y[1..n], i, j)
2. if i = 0 or j = 0
3. return ""
4. else if  X[i] = Y[j]
5. return backTrack(C, X, Y, i-1, j-1) + X[i]
6. else
7. if C[i,j-1] > C[i-1,j]
8. return backTrack(C, X, Y, i, j-1)
9. else
10.    return backTrack(C, X, Y, i-1, j)
```

## 4.7.3 Reading out all LCS

If the selection of $x_i$ and $y_j$ produces an equally long result, then you need to read out both the resulting subsequences. This is returned as a set by the following backTrack_All function.

```
1. function backTrack_All(C[0..m,0..n], X[1..m], Y[1..n],
   i, j)
2. if i = 0 or j = 0
3. return {}
```

NOTES

```
4. else if X[i] = Y[j]
5. return {Z + X[i] for all Z in backTrackAll(C, X, Y, i-1,
   j-1)}
6. else
7. R := {}
8. if C[i,j-1] ≥ C[i-1,j]
9. R := backTrack_All(C, X, Y, i, j-1)
10.    if C[i-1,j] ≥ C[i,j-1]
11.    R := R ∪ backTrack_All(C, X, Y, i-1, j)
12.    return R
```

### 4.7.4 Printing the Difference

You can use the following function, print_Diff to retrace the C matrix and print the difference between the two input sequences.

```
1. function print_Diff(C[0..m,0..n], X[1..m], Y[1..n], i,
   j)
2. if i > 0 and j > 0 and X[i] = Y[j]
3. print_Diff(C, X, Y, i-1, j-1)
4. print "  " + X[i]
5. else
6. if j > 0 and (i = 0 or C[i,j-1] ≥ C[i-1,j])
7. print_Diff(C, X, Y, i, j-1)
8. print "+ " + Y[j]
9. else if i > 0 and (j = 0 or C[i,j-1] < C[i-1,j])
10.    print_Diff(C, X, Y, i-1, j)
11.    print "- " + X[i]
```

## 4.8 MATRIX CHAIN MULTIPLICATION

The matrix chain multiplication is a classic example of dynamic programming. Here, in a given sequence of matrices, you need to determine the most efficient way to multiply those matrices together. In other words, the problem is not actually to multiply the matrices, rather identifying the order, in which the matrices should be multiplied efficiently.

To solve the matrix chain multiplication, let us assume that the only requirement for this is to know the minimum cost or minimum number of operations involved in multiplying the matrices. Now, if there are two matrices, then to multiply the matrices, there is only one way, which is nothing but the minimum cost to do so. Considering this scenario, we can follow recursively to find the minimum cost as follows:

1. Take the total sequence of matrices and divide them into two separate subsequences.

NOTES

2. Calculate the minimum cost to multiply the matrices within each of the subsequence.
3. Add the above costs together and add it with the cost of multiplying the two resultant matrices.
4. Follow this approach for each of the possible position at which you can split the sequence of matrices, and take the minimum over all of them.

For example, if there are four matrices *W, X, Y* and *Z*, then we make recursive calls to compute the cost required to find each of *(W)(XYZ)*, *(WX)(YZ)* and *(WXY)(Z)*. This helps in finding the minimum cost for computing WXY, WX, YZ and *XYZ*. This approach not only gives the minimum cost but also describes the best way in which the matrix multiplication can be done.

The following algorithm shows the algorithm for matrix chain multiplication:

```
1. Matrix-Chain-Order(int p[])
2. {
3. n = p.length - 1;
4. for (i = 1; i <= n; i++)
5. m[i,i] = 0;
6. for (l=2; l<=n; l++) { // l is chain length
7. for (i=1; i<=n-l+1; i++) {
8. j = i+l-1;
9. m[i,j] = MAXINT;
10.    for (k=i; k<=j-1; k++) {
11.    q = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j];
12.    //Matrix Ai has the dimension p[i-1] x p[i].
13.    if (q < m[i,j]) {
14.    m[i,j] = q;
15.    s[i,j] = k;
16.    }
17.    }
18.    }
19.    }
20.    }
```

**CHECK YOUR PROGRESS**

6. What are the types of shortest path approaches?
7. When is the Bellman–Ford algorithm used in a shortest path problem?
8. What is a subsequence?
9. What is the aim of matrix chain multiplication problem?

## 4.9 SUMMARY

In this unit, you have learnt about the basic concepts and characteristics of dynamic programming. It is a technique that helps in designing efficient algorithms for problems that can be easily divided into sub-problems. The solution of a given problem is obtained by combining the solutions of all the sub-problems. Dynamic programming is helpful in constructing algorithms in situations where the problem focuses on optimization of the objective function. In

this unit, you have learnt the solution provided by dynamic programming for the following problems:

- Multistage graph problem
- Shortest path problem
- 0/1 Knapsack problem
- Travelling salesperson problem
- Longest Common Subsequence
- Matrix multiplication problem

NOTES

## 4.10 KEY TERMS

- **Dynamic programming:** Dynamic programming is an algorithm design methodology that helps in building up the solution of a problem by combining the solutions of several sub problems.
- **Shortest path:** The shortest path technique helps in finding a path between two vertices such that the sum of the cost (weight) of its ingredient edges is minimized.
- **The travelling salesperson problem:** The travelling salesperson is the best possible means of finding the lowest possible cost for visiting the various cities and returning to the starting point.
- **The longest common subsequence (LCS) problem:** This problem helps in determining the longest subsequence that is common to all the sequences in a set of sequences of two or more.
- **Matrix chain multiplication:** The matrix chain multiplication is a classic example of dynamic programming. In a given sequence of matrices, you need to determine the most efficient way to multiply those matrices together.

## 4.11 ANSWERS TO 'CHECK YOUR PROGRESS'

1. Dynamic programming is an algorithm design technique.
2. The four problems that can be solved using dynamic programming are:
   A. Shortest path problem
   B. 0/1 knapsack problem
   C. Travelling salesperson problem
   D. Matrix chain multiplication problem
3. Dynamic programming helps in reducing computation by:
   - Solving sub-problems in a bottom-up fashion
   - Storing the solutions of sub-problems so that they can be solved in the first attempt
   - Looking up the solution when a sub-problem is encountered again
4. A multistage graph is a directed graph $G$ that has a pair $(V, E)$ where $V$ represents set of vertices, which is a finite set and E represents set of edges that has a binary relation on $V$.

NOTES

5. The two approaches to solving a multistage graph problem are:
   - Forward approach
   - Backward approach
6. The two types of shortest path approaches are:
   a. Single-source shortest path
   b. All-pairs shortest path
7. The Bellman–Ford algorithm is used only when there are negative edge weights in a directed graph.
8. A subsequence is a new sequence that is formed from the original sequence by deleting some of its elements in such a way that the deletion of the elements does not disturb the relative positions of the remaining elements in the sequence.
9. The aim of the matrix chain multiplication problem is to determine the most efficient way to multiply a sequence of matrices together.

## 4.12 QUESTIONS AND EXERCISES

### Short-Answer Questions

1. Write a short note on dynamic programming.
2. Briefly describe the characteristics of dynamic programming.
3. Discuss the multistage graph problem.
4. Briefly explain the two types of shortest path approaches.
5. Define the 0/1 knapsack problem.

### Long-Answer Questions

1. How will you solve the multistage graph problem? Explain with an example.
2. Explain the Bellman-Ford algorithm for the shortest path problem.
3. Explain the dynamic programming solution for the shortest path problem.
4. Explain the travelling salesman problem with an example.
5. Describe all the steps used for solving the LCS problem.
6. Explain the algorithm for matrix chain multiplication.

## 4.13 FURTHER READING

Horowitz, Ellis, Sartaj Sahni and Sanguthevar Rajasekaran. 2006. *Fundamentals of Computer Algorithms*. New Delhi: Galgotia Publication Pvt. Ltd.

# UNIT 5 BACKTRACKING AND BRANCH-AND-BOUND TECHNIQUES

NOTES

## Structure



## 5.0 INTRODUCTION

You are already familiar with the concept of dynamic programming. In this unit, you will learn about the backtracking technique. It is one of the design paradigms of an algorithm. This technique helps in determining the most efficient solution for a given problem without examining all the possible solutions. The different problems that are solved by using the backtracking technique in this unit are as follows:

- 8-Queen problem
- Sum of subsets
- Graph colouring
- Knapsack problem

In this unit, you will also learn about the branch-and-bound method. It is a state space search method that is used to find the optimal solution of discrete and combinatorial problems. The different problems solved by the branch-and-bound method are as follows:

- 0/1 knapsack problem
- Travelling salesman problem

NOTES

## 5.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Explain the backtracking paradigm in designing algorithms
- Solve problems using the backtracking technique, which are:
  - The 8-Queen problem
  - Sum of subsets
  - Graph colouring
  - Knapsack problem
- Discuss the branch-and-bound state space method
- Solve problems using the branch-and-bound method, which are:
  - 0/1 knapsack problem
  - Travelling salesman problem

## 5.2 BACKTRACKING—THE GENERAL METHOD

Backtracking represents the general technique of an algorithm design. Many problems that deal with searching a solution or that look for an optimal solution depending on some constraints can be solved by using backtracking formulation. Several applications of backtracking requires a solution, which is expressible as an n-tuple $(x_1, x_2,...x_n)$, where $x_i$ is taken from some finite set $S_i$. Usually, the problem to be solved look out to find one vector that maximizes or minimizes a particular function F $(x_1, x_2,...x_n)$. For example, sorting an array of integer k[1 : n] is expressible by an n-tuple, where $x_i$ is the index in k of the ith smallest element. The criterion function F is the inequality $k[x_i] \leq k[x_i + 1]$ for $1 \leq i < n$. $S_i$ is the finite set, which includes integer 1 to n.

Suppose $p_i$ is the size of set $S_i$, then $m = m_1 m_2...m_n$ n-tuples are the possible candidates for satisfying the function F. Then brute force approach evaluates each one of these n-tuples with the function F and saves those tuples, which provide optimal solution. The back tracking algorithm is capable of providing the same with less number of trials. It is based on the idea of building the solution vector of one component at a time and to use modified criterion functions F $(x_1, x_2,...x_n)$. This is referred to as bounding functions, which test the chance of success formed from one vector. One of the most important advantage of this method is that if it is found that the partial vector $(x_1, x_2,...x_n)$ does not give any desired result, then $m_{i+1}...m_n$ possible test vectors are skipped.

Problems being solved using backtracking satisfy a complex constraint. These constraints are divided into the following two parts:

- **Explicit constraints**: It depends on the particular instance I of the problem being solved. A possible solution space for I is being defined by all the tuples that satisfy the explicit constraints. Rules that restrict each $x_i$ to take the value from a given set only are known as explicit constraints. Examples of the explicit constraints are as follows:
  - $x_i \geq 0$ or $S_i$ = {all non-negative real numbers}
  - $x_i = 0$ or 1 or $S_i = \{1, 0\}$
  - $k_i \leq x_i \leq u_i$ or $S_i = \{a : k_i \leq a \leq u_i\}$

NOTES

- **Implicit constraints**: Rules that determine which of the tuple in the solution space I satisfy the criterion functions are known as implicit constraints. Therefore, the way in which the $x_i$ relates to each other is described by the implicit constraint.

As backtracking is based on post-order traversal of a tree, so it can be represented as a recursive process. A recursive formulation of backtracking technique is described in the following algorithm:

```
1.   Algorithm RecurBacktrack(p)
2.   {
3.   for (each a[p]∈ S(a[1], …, a[p-1])
4.   do
5.   {
6.         if(Dp (a[1], a[2],…, a[p])≠0)
7.         then
8.         {
9.           if(a[1], a[2],…, a[p] is a path to an answer
             node)
10.          then
11.              write(a[1:p])
12.          if(p<m)
13.              then RecurBacktrack(p+1)
14.        }
15.  }
16.  }
```

In the above algorithm the backtracking process is described using recursion. The solution vector $(a_1,a_2,...,a_m)$ is represented as a global array a[1:m]Initially the first p-1 values, i.e., a[1], a[2], …, a[p], of the solution vector a[1:m] are assigned. The suitable elements for pth position, which satisfy $D_p$ are calculated. These elements are determined one by one and added to the current vector$(a_1,a_2,...,a_{p-1})$. A check is performed whenever $a_p$ is added. This check helps to determine whether a solution is found or not.

Apart from the recursive approach this backtracking process can also be described as an iterative process. The following algorithm shows the iterative backtracking method:

```
1.   Algorithm ItrvBacktrack(m)
2.   {
3.   p=1
4.   while (p≠0)
5.   do
6.   {
```

NOTES

```
7.          if (there remains an untried a[p]∈S(a[1], a[2],
            …,a[p-1]) and D_p(a[1], …, a[p]) is true)
8.          then
9.          {
10.           if(a[1],a[2],…, a[p]is a path to answer node)
11.           then
12.               write (a[1:p])
13.           p=p+1
14.         }
15.         else
16.           p=p-1
17.   }
18.   }
```

In the above algorithm, the solutions are generated in the solution vector a[1:m] and printed side by side. The function S() generates a set of values that can be used as the first component of $a_1$ inside the solution vector. The value of p is incremented repeatedly resulting in the growth of solution vector. This process continues until the solution is determined or no value of $a_p$ remains untried.

## 5.3 THE 8-QUEENS PROBLEM

The 8-queens problem is considered as an n*n chessboard in which no two queens attack each other, i.e., no two of them are in the same row, column or diagonal. Let $(x_1, x_2, \ldots x_n)$ represent a solution in which $x_i$ is the ith row having the $i^{th}$ queen. Different values of $x_i$ depicts that no two queens are in the same column. To check whether the two queens are in the same diagonal or not, assume that the squares of chessboard being numbered as indices of the two-dimensional array a[1 : m, 1:m]. Now, examine that every element on the same diagonal that runs from the upper left to the lower right has the same row-column value. For example consider Figure 5.1.

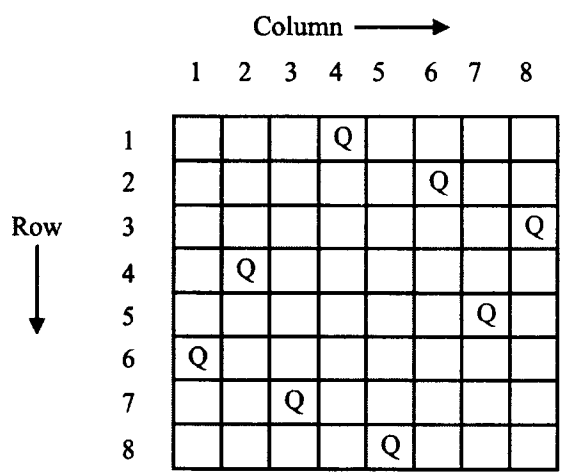



***Figure 5.1*** *Solution to 8-Queens Problem*

NOTES

In Figure 5.1, consider the queue at a[5, 2], the squares diagonal from the upper right to the lower left of the position a[5, 2] are a[3, 1], a[5, 3], a[6, 5], a[7, 5] and a[8, 6]. The row-column value of all these squares is 2. All the elements on the same diagonal that runs from upper right to lower left has the same row + column value. Assume that the two queens are placed at positions (m, r) and (p, q), then they are on the same diagonal if and only if

```
m - r = p - q    or    m + r = p + q
```

The above equations imply the following results:

```
r - q = m - p    or    r - q = p - m
```

The above equations show that the two queens lie on the same diagonal if and only if $|r - q| = |m - p|$.

The algorithm to find whether a queen can be placed at a position (p, m) is as follows:

```
1.  Algorithm Place (p, m)
2.  //Return true if a queen can be placed in the pth row
    and //mth column. Otherwise return false.
3.  //x[] is a global array whose first (p - 1) value have
    //been set.
4.  //Abs (z) returns the absolute value of z.
5.  {
6.  for r = 1 to p - 1 do
7.  // two queens are in the same column or diagonal
8.  if ((x[r] = m) or (Abs (x[r] - m) = Abs (r - p) ))
9.  then return false;
10. else return true;
11. }
```

In the above algorithm, Place (p, m) returns a Boolean value true if the pth queen can be placed in the column m. It tests whether m is different from all the previous values of x[1], x[2],...,x[k – 1] as well as whether there is no other queen on the same diagonal. The computing time of this algorithm is O(p – 1).

The following algorithm is used to find the solution of the N-Queens problems:

```
1.  Algorithm N-Queens (p, n)
2.  //Using backtracking, this procedure depicts all
    possible //placements of n queens on an n*n chessboard
    in such a way //that not a single queen attack the
    other queen.
3.  {
4.  for m = 1 to n do
5.  {
6.  if Place (p, m) then
7.  {
8.  x[p] = m;
```

NOTES

```
9.   if (p = n) then write (x[1 : n]);
10.  else N-Queens (p + 1, n);
11.  }
12.  }
13.  }
```

The N-Queens algorithm is efficient over brute force approach. By brute force approach, in an 8*8 chessboard there are $^{65}C_8$ possible ways to place eight queens, that is, almost 5.5 billion eight-tuples to examine. On the other hand, by allowing placement of queens on different rows and columns, only the examination of 8! or only 50,320 eight-tuples is required.

## 5.4 SUM OF SUBSETS

Suppose that you have n different positive numbers and you are required to find all the possible combinations of these numbers whose sum is m. This is known as sum of subset problem. The sum of subset problem can be formulated either by fixed-sized tuple or variable-sized tuple. The element $e_i$ of the solution vector is either one or zero depending on whether the weight $w_i$ is included or not.

The children of any node of a tree are easily generated. If a node is at level i, then the left child corresponds to $e_i = 1$ and left right child corresponds to $e_i = 0$.

A simple choce for bounding the functions is $B_k(x_1, x_2,\ldots, x_k)$ = true if

$$\sum_{i=1}^{k} w_i e_i + \sum_{i=k+1}^{p} w_i \geq m$$

In the above equation, $x_1$, $x_2,\ldots, x_k$ cannot produce any node if the above condition is not satisfied. The bounding function can be strengthened by assuming that the $w_i$'s are in non-decreasing order. In such case $x_1$, $x_2,\ldots, x_k$ cannot produce child node if

$$\sum_{i=1}^{k} w_i e_i + w_{k+1} > m$$

Therefore, the bounding function will be as shown below:

$$B_k(x_1, x_2,\ldots, x_k) = \text{true iff } \sum_{i=1}^{k} w_i e_i + \sum_{i=k+1}^{p} w_i \geq m \text{ and } \sum_{i=1}^{k} w_i e_i + w_{k+1} \leq m$$

By assuming $x_k = 1$, the above equation can be simplified as follows:

$$\sum_{i=1}^{k} w_i e_i + \sum_{i=k+1}^{n} w_i > s$$

Algorithm of sum of sub is as follows:

```
1.   Algorithm SumOfSub (s, k, r)
2.   //Find all subset of w[1 : p] that sum to s. The value
     of //x[j] is determined. s = Σ_{j=1}^{k-1} w[j] * x[j] and r =
```

NOTES

$\sum_{j=k}^{n} w[j]$. The //w[j]'s are in non-decreasing order. It is assumed that //$w[1] \leq m$ and $\sum_{i=1}^{n} w[i] \geq m$.

```
3.   {
4.   //Generate left child. S + w[k] because B_{k-1} is true.
5.   x[k]= 1
6.   if (s + w[k] = m) then write (x[1 : k]);
7.   elseif (s + w[k] + w[k+1] ≤m)
8.   Then SumOfSub(s+w[k], k+1, r-w[k]);
9.   //Generate right child and evaluate B_k.
10.  if (s+r-w[k] ≥m) and (s + w[k+1] ≤m) then
11.  {
12.  x[k] = 0;
13.  SumOfSub (s, k+1, r-w[k]);
14.  }
15.  }
```

## 5.5 GRAPH COLOURING

Graph colouring is the method of allocating the colours to the vertices of the given graph in a manner such that no two adjacent vertices have the same colour. A graph is said to be m-colourable if it uses m different colours in painting its vertices. The minimum number of colours required to paint a graph is called the chromatic number of G. The chromatic number of the given graph is denoted by $\chi$ (G).

The Welch and Powell provided an algorithm for colouring the graph G. The different steps of this algorithm are as follows:

1. Arrange all the vertices of the graph in decreasing order of their degrees and make sure that no vertex of the graph is initially coloured.
2. Traverse the vertices and assign the colour 1 to the first vertex that is uncoloured. Make sure that its adjacent vertex has not assigned colour 1.
3. Repeat step 2 with colours 2, 3 and so on until no vertex of the graph remains uncoloured.

**Note:** *This algorithm does not necessarily find the chromatic number for the graph.*

Consider the following algorithm for determining n-colourings of a graph:

```
1.   Repeat
2.   {  //All legal assignments for p[a]are generated
3.   NextValue(a)    //A legal colour is assigned to p[a]
4.   if(p[a]=0)      // New colour is not possible
5.   then return
6.   if (a=m)
```

NOTES

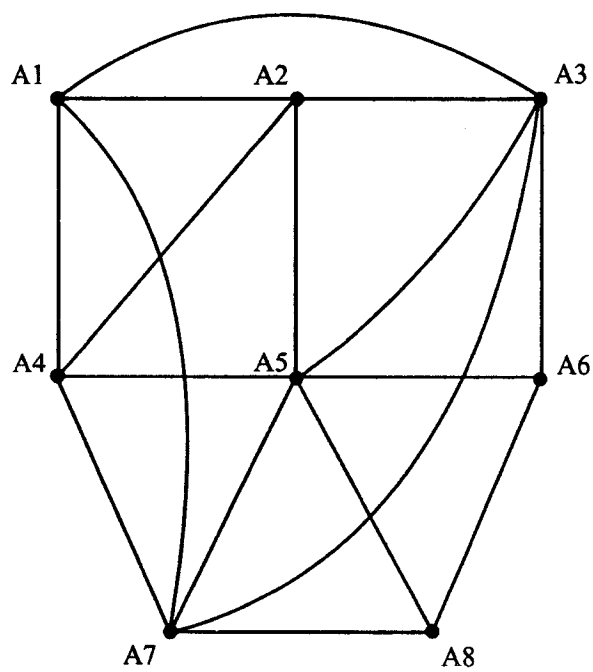
```
7.   then write (p[1:m])
8.   else ncolouring(a+1)
9.   }
10.  until(false)
```

In the above algorithm the graph is represented as a Boolean adjacency matrix H[1:m,1:m]. The colours are represented by integers 1,2,3....n. The assignments are made such that adjacent vertices having distinct integers are printed. The function NextValue(a) is used to generate the other possible colours. An algorithm describing the function NextValue(a) is described as follows:

```
1.   Repeat
2.   {
3.   p[a]= (p[a]+1) mod(n+1)  // Next highest colour
4.   if(p[k]=0)  // All the colours are used
5.   then return
6.   for I=1 to m do
7.   {
8.   if(H[a,i]≠0) and (p[a]=p[i]))
9.   then break
10.  }
11.  if (i=m+1)  // A new colour found
12.  then return
13.  }
14.  until (false)   // Find another colour
```

In the above algorithm p[1], p[2], ..., p[a-1] are assigned to the integer values such that adjacent vertices have different integers. The value for p[a] is determined in the range (0, n) and p[a] is assigned to the next highest numbered colour. This colour needs to be distinctive from the adjacent vertices. In case no such colour exists then p[a] is assigned zero.

**Example 1**: Colour the below graph by using the Wetch-Powell algorithm.



NOTES

**Solution:**

Degree of vertex $A_1$ is 5.

Degree of vertex $A_2$ is 5.

Degree of vertex $A_3$ is 5.

Degree of vertex $A_5$ is 5.

Degree of vertex $A_5$ is 5.

Degree of vertex $A_6$ is 3.

Degree of vertex $A_7$ is 5.

Degree of vertex $A_8$ is 3.

Now arrange the vertices in the decreasing order of their degree, you get the following order:

$A_5, A_3, A_7, A_1, A_2, A_4, A_6, A_8$

Now first colour is assigned to the vertex $A_5$. The same colour can not be assigned to the vertex $A_3$ and $A_7$ because these are adjacent vertices of the vertex $A_5$. This same colour is assigned to the non-adjacent vertex $A_1$. Therefore, vertices $A_5$ and $A_1$ have been assigned the same colour.

The second colour is now assigned to the vertex $A_3$ and the non-adjacent vertices $A_3$ and $A_8$.

In the end, the third colour is enough to paint the remaining vertices of the graph, which are $A_7$, $A_2$, and $A_5$.

Thus the total of 3 colours has been used in colouring the given graph. Therefore, the given graph is said to be three-colour graph.

## 5.5.1 Matching

Matching, also known as edge independent set can be referred as a set of edges, which do not have any common vertices. It can also be defined as an entire graph consisting of number of edges without common vertex. For a graph G = (V, E), matching M is a set of non-adjacent edges. Non-adjacent edges are those edges, which do not have a common vertex. On the other hand, a vertex is said to be matched, if it is incident on the edge, which is present in the matching set M. In graph theory, matching can be classified into following categories:

- **Maximal matching**: Maximal matching can be referred as matching M of a graph G such that if any edge, which is not present in matching set M is added to the set M, then the edges present in the set becomes adjacent. In other words, a matching M of a graph is maximal if it is not a proper subset of any other matching in the graph.
- **Maximum matching**: Maximum matching can be referred as a matching that contains the largest possible number of edges. A graph can have number of maximum matching sets. The size of maximum matching set can be determined by using the matching number of the graph.
- **Perfect matching**: Perfect matching set is a matching set, whose edges cover each and every vertex of the graph.

NOTES

## 5.6 KNAPSACK PROBLEM

Knapsack problem is a type of maximization problem, in which elements having maximum economic value are to be selected and that can fit in a bag or container. The problem is to determine the number of items that should be included in a collection, from a set of items each having a specific cost and value. The items should be selected in such a way that the total value of the selected items is maximum and the net cost involved is less than some specified cost.

Here, you have n positive weights $w_i$, n, positive profits, $p_i$ and a positive number m, which denotes the knapsack capacity. This problem chooses a subset of weights such that

$$\sum_{1 \le i \le n} w_i x_i \le m \text{ and } \sum_{1 \le i \le n} p_i x_i \text{ is maximized.}$$

The $x_i$'s corresponds to a zero-one-valued vector.

You can get a good bounding function for this by using an upper bound on the value of the best feasible solution that can be obtained by expanding the given node and any of its descendants. The node is deleted it the upper bound is not higher than the best solution determined so far..

If you have determined the value of $x_i$, $1 \le i \le k$, at node Z, then you can easily get the upper bound of Z by relaxing the requirement $x_i = 0$ or 1 to $1 \le i \le k$ for $k + 1 \le i \le n$, with the help of greedy algorithm. The function Bound (cp, cw, k) can be used to determine the upper bound on the best solution obtained by expending any node Z at level k + 1 of a state space tree. The object weight is represented by $w_i$ and profit is represented by $p_i$. The algorithm, Bound (cp, cw, k) is shown in the following code:

```
1.  Algorithm Bound (cp, cw, k)
2.  //cp is the total current profit, cw is the total
    current //weight, k is the index of the last removed
    item and m is //the knapsack size.
3.  {
4.  b = cp; c = cw;
5.  for i = k + 1 to n do
6.  {
7.  c = c + w[i];
8.  if (c < m) then b = b + p[i];
9.  else return b + (1 - (c - m) / w[i]) * p[i];
10. }
11. return b;
12. }
```

From the above algorithm, you can make out that the bound for a feasible left child of a node Z is the same as that of Z. Therefore, there is no need of using bound function when the backtracking algorithm moves towards the left child of a node. In this case, you should use Backtracking Knapsack (BKnap) algorithm,

NOTES

which is obtained by using recursive backtracking schema. The following code shows the BKnap algorithm:

```
1.  Algorithm BKnap (cp, cw, k)
2.  //m is the knapsack size; n represents the number of
    //weights and profits. p[] and w[] are the profits and
    //weights. Final weight of knapsack problem is fw and
    fp is //the final maximum profit. x[k] = 0 if w[k] is
    not in the //knapsack; else x[k] = 1.
3.  {
4.  //Generate left child
5.  if (cw + w[k] ≤m) then
6.  {
7.  y[k] = 1;
8.  if (k<n) then BKnap (k + 1, cp + p[k], cw + w[k]);
9.  if ((cp + p[k] > fp) and (k = n)) then
10. {
11. fp = cp + p[k];
12. fw = cw + w[k];
13. for j = 1 to k do x[j] = y [j];
14. }
15. }
16. //Generating right child
17. if (Bound (cp, cw, k) ≥ fp) then
18. {
19. y[k] = 0; if (k < n) then Bknap (k + 1, cp, cw);
20. if ((cp > fp) and (k = n)) then
21. {
22. fp = cp;
23. fw = cw;
24. for j = 1 to k do
25. x[j] = y[j];
26. }
27. }
28. }
```

Initially set $fp = 1$ in the BKnap algorithm, it will be invoked as BKnap (1, 0, 0);

When $fp \neq -1$, x[i], $1 \leq i \leq n$, is such that $\sum_{i=1}^{n} p[i]\, x[i] = fp$.

NOTES

## 5.7 BRANCH-AND-BOUND—THE GENERAL METHOD

The term, branch and bound refers to all the state space search methods that generate all children of the E-node before any other node becomes E-node. There are two graph search strategies Breadth First Search (BFS) and Depth First Search(DFS) in which the exploration of new node is not started unless the current node is fully explored. In reference to branch and bound terminology, the BFS-like state space search is known as First In First Out (FIFO) search because the list of live nodes are in queue. The DFS-like state search is known as Last In First Out (LIFO) search because the list of live nodes are in stack.

### 5.7.1 FIFO Search

FIFO involves in choosing a node from the point from where the traversal begins and accesses all the neighbouring nodes of the start vertex. A field called STATUS is used in association with each vertex, as used with depth-first traversal, to indicate the status of a vertex as ready, waiting or processed. You need to use a queue to store the vertices that are being processed. The algorithm for breadth-first traversal is as follows:

1. All vertices are initialized to ready state.
2. Place the start vertex in the queue and change the status of the start vertex to waiting.
3. Repeat the steps 4 and 5 till the queue has no more elements.
    4. Remove the vertex V at the front. Print the vertex and change the status of the vertex V to processed state.
    5. Add the neighbouring vertices of V that are in ready state and change their status to waiting.
    6. End.

Figure 5.2 shows a graph that you can use for the FIFO search.

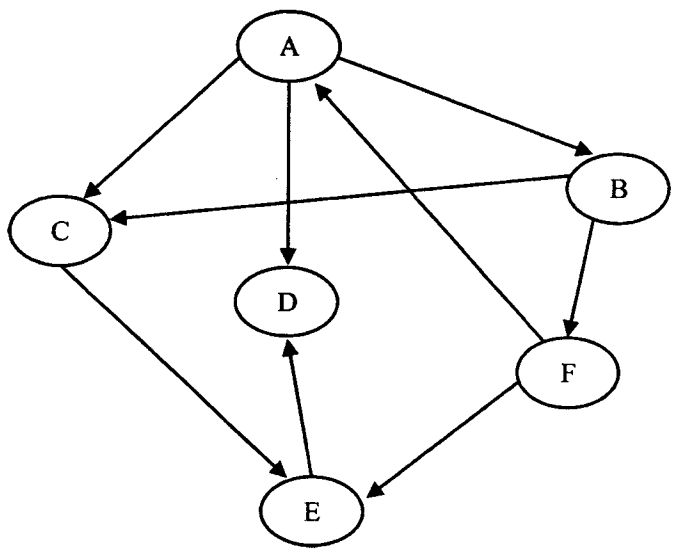



***Figure 5.2** Graph used for FIFO Search*

NOTES

The various steps that will lead to the breadth first traversal of the graph assuming A, as the starting vertex in the figure are:

- Initially, add A to Queue.

  Front: 1, Rear: 1, Queue: A
- Remove the front element A from the Queue and add the neighbours of A to rear.

  Front 2, Rear 4, Queue A C D B
- Remove the front element C from Queue and add the neighbouring vertices of C to rear.

  Front 3, Rear 5, Queue A C D B E
- Remove the front element D from Queue and add the neighbours of D to rear.

  Front 4, Rear 6, Queue A C D B E F
- Remove the front element B from Queue and add the neighbours of B to rear.

  Front 5, Rear 6, Queue A C D B E F
- Remove the front element E from Queue and add the neighbours of E to rear.

  Front 6, Rear 6, Queue A C D B E F
- Remove the front element F from Queue and add the neighbours of F to rear.

  Front 7, Rear 6, Queue A C D B E F

The completion of traversal is indicated by the empty queue. The order in which the vertices that have been traversed is:

A C D B E F

## 5.7.2 LIFO Search

LIFO search involves in choosing a node from where the traversal in a tree needs to start, and then you need to access the neighbour node of the start node. After this you again need to access the neighbour node of the selected node. This process continues, until you encounter a NULL node by the pointer, backtracking is done to reach the other neighbour of the start node. The value of the nodes is stored in a stack. A field called STATUS is used in association with each node to indicate the status of the node as ready, waiting or processed. The ready state is the initial state of the node. The waiting state is when the node is in the stack and is waiting to be processed. The processed state is, when the node has been processed. The algorithm for LIFO search is as follows:

```
1. All nodes are initialized to ready state.
2. Push the start node in the stack.
3. Repeat the steps 4 and 5 till the stack has no more
nodes.
4. Pop the top node, V from the stack and change the status
of the node to processed state.
5. Push all neighbours of the node, V that are in ready
state into the stack and change their status to waiting.
6. End.
```

NOTES

Consider Figure 5.3 that shows a graph, which you can use for the purpose of LIFO search.

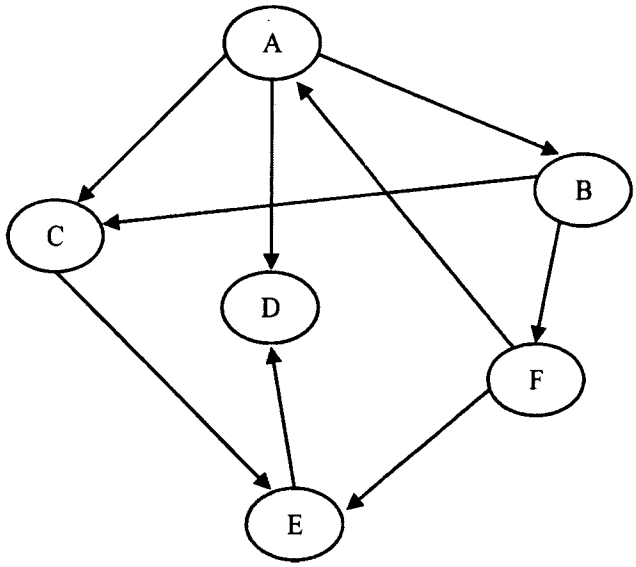



***Figure 5.3** Graph used for LIFO Search*

Following are the steps that will lead to the depth first traversal of the above graph. Assuming, A as the starting node in the above graph the steps are:

- Initially push node A into the stack.

  Stack: A
- Pop and print the top element A and then push the neighbours of A into the stack.

  Stack: C D B
- Pop and print the top element B and then push the neighbours of B into the stack.

  Stack: C D F
- Pop and print the top element F and then push the neighbours of F into the stack.

  Stack: C D E
- Pop and print the top element E and then push the neighbours of E into the stack.

  Stack: C D
- Pop and print the top element D and then push the neighbours of D into the stack.

  Stack: C
- Pop and print the top element C and then push the neighbours of C into the stack.

  Stack: NULL

The stack contains NULL node, which indicates that the traversal of graph is complete now. The vertices that have been processed as a result are:

A B F E D C

NOTES

## 5.7.3 Least Count (LC) search

In both the branch and bound strategies, FIFO and LIFO, the rule is to select the next node is bit complex because it does not give any preference to the node which takes you to the answer node quickly.

Consider the following example to see how a FIFO branch and bound algorithm searches for a state space tree for the five-queens problem. Figure 5.4 shows the five-queens state space tree generated by FIFO branch and bound algorithm.

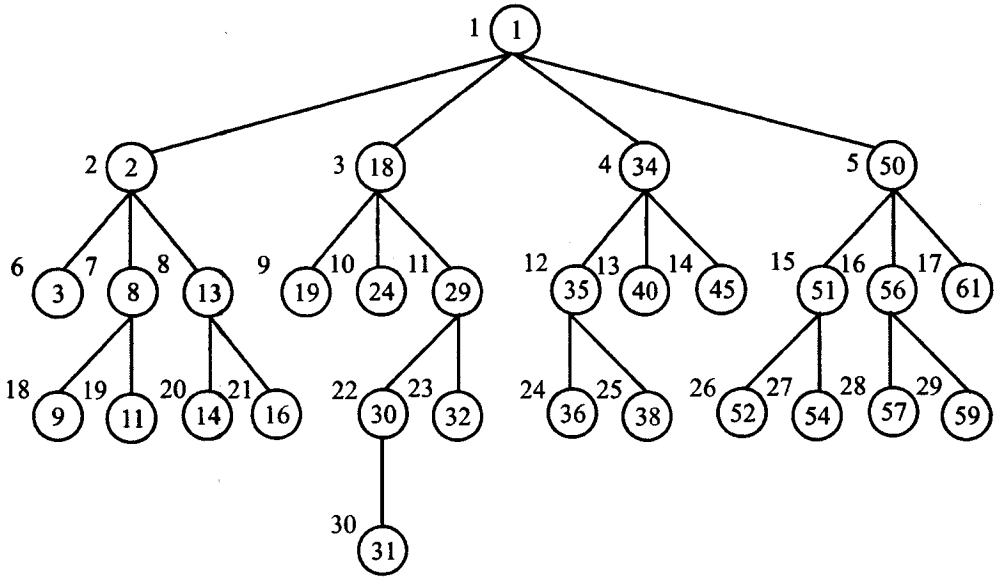



***Figure 5.4** Five-Queens State Space generated FIFO by Branch and Bound Algorithm*

In Figure 5.4, when the node 30 is generated, it should lead to the answer node in one move. But, the rigid FIFO rules state that all the live nodes generated must be expanded before expanding the node 30. An intelligent ranking function, $\hat{c}(.)$ for live nodes speeds up the search process to obtain an answer node. This intelligent ranking function selects the next E-node. If this ranking function assigns a higher rank to the node 30 as compared to all other live nodes, the node 30 becomes the E-node for the following the node 29. Now, the expansion of node 30 generates the answer node.

The ranks to nodes can be assigned on the basis of cost required to reach the answer for the node from the live node. For any node x, this cost would be determined by

(1) The number of nodes present in the subtree x that are required to be generated prior to the generation of the answer node.

(2) The number of levels and the nearest answer node is from x.

Using this cost measure, you can conclude that the cost of the root of the tree in Figure 5.4 is 5 because the answer node 31 is at the distance of 5 levels from the root node 1. The cost of node 18 and 35, 29 and 35, 30 and 38 are 3, 2 and 1 respectively. The cost of other nodes at level 2, 3 and 5 are respectively greater than 3, 2 and 1. On the basis of these costs, the E-nodes being selected are 1, 18, 29 and 30. If the cost measure one is used, then the search would always generate a minimum number of nodes and if cost measure 2 is used then only the nodes between the path from root node to the answer node becomes the E-node. The

NOTES

demerits of using any of these cost function is that the estimation of the cost of a node includes search of the subtree x for an answer node. Therefore, by the time the cost is being determined the subtree has been searched and there is no need to explore x again. So, search algorithms generally ranks nodes on the bases of an estimate $\hat{g}(.)$ of their cost.

Suppose, an estimate of additional effort to reach an answer node for x is given by $\hat{g}(x)$. The rank to node x is assigned using a function $\hat{c}(.)$ such that $\hat{c} = f(h(x)) + \hat{g}(x)$, where h (x) is the cost of reaching x from the root and f (.) is a non decreasing function. Using $f(.) \equiv 0$ generally biases the search algorithms to deeply search the tree. Generally, $\hat{g}(y) \leq \hat{g}(x)$ for y, which is child of x. Therefore, following x, y will become the E-node, then one of the y's children will become the E-node and so on. Nodes other than that of the subtree x are not generated until the subtree x is searched properly.

The search strategy that uses a cost function $\hat{c}(x) = f(h(x)) + \hat{g}(x)$ to select the next node is always close to the next E-node having least $\hat{c}(.)$. Therefore, this search strategy is known as Least Count (LC) search.

## 5.8 0/1 KNAPSACK PROBLEM

To solve the problem with the help of the branch and bound technique, the state space tree is the necessary requirement. Since knapsack problem is a maximization problem therefore, replace the objective function $\sum p_i x_i$ by the function $-\sum p_i x_i$. The $\sum p_i x_i$ is maximized if $-\sum p_i x_i$ is minimized. This modified knapsack problem is as follows:

Minimize $-\sum_{i=1}^{n} p_i x_i$

Subject to $\sum_{i=1}^{n} w_i x_i \leq m$

$x_i = 0$ or 1, $1 \leq i \leq n$

This problem assumes a fixed tuple size formulation for the solution space but easily extendable to the variable tuple size formulation. Every leaf node in the state space tree represents an assignment for which $\sum_{1 \leq i \leq n} w_i x_i \leq m$ is an answer node. A minimum-cost answer node corresponds to an optimal solution if you define $c(x) = -\sum_{1 \leq i \leq n} p_i x_i \leq m$ for every answer node. For infeasible leaf nodes, the cost $c(x) = \infty$. For non leaf nodes, c (x) is recursively defined as min {c (lchild (x)), c (rchild (x))}.

You need two functions $\hat{c}(x)$ and u (x) such that $\hat{c}(x) \leq c(x) \leq u(x)$ for every node x. The cost $\hat{c}(.)$ and u (x) satisfying the above requirement can be obtained by considering the x as a node at level j, $1 \leq i \leq n + 1$. At node x, assignment have already made to $x_i$, $1 \leq i < j$. These assignments have the cost $\sum_{1 \leq i < j} p_i x_i$. The following code shows the upper bound algorithm:

```
1.  Algorithm UBound (cp, cw, k, m)
2.  {
3.  b = cp;
4.  c = cw;
5.  for i = k + 1 to n do
6.  {
7.  if (c +w[i] ≤m ) then
8.  {
9.  c = c +w[i];
10. b = b - p[i];
11. }
12. }
13. return b;
14. }
```

## 5.9 TRAVELLING SALESMAN PROBLEM

A dynamic programming algorithm for travelling salesman problem is already being discussed in the previous unit. Here, the travelling salesman problem is explained with help of branch and bound algorithm. These algorithms have the worst case complexity of O ($n^2 2^n$) but, the branch and bound algorithm can solve this problem in lesser time as compared to the dynamic programming algorithm. To understand how to solve the travelling salesman problem consider the example given below.

**Example 2**: Assume that a salesman wants to travel all his company branch-offices that are placed in different cities around the country. The distance between each city is known. In this problem, you need to determine the order of cites which is to be followed by salesman to minimize the total cost. Consider the cost matrix shown in Figure 5.5.

| i \ j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | ∞ | 3 | 93 | 13 | 33 | 9 | 57 |
| 2 | 4 | ∞ | 77 | 42 | 21 | 16 | 34 |
| 3 | 45 | 17 | ∞ | 36 | 16 | 28 | 25 |
| 4 | 39 | 90 | 80 | ∞ | 56 | 7 | 91 |
| 5 | 28 | 46 | 88 | 33 | ∞ | 25 | 57 |
| 6 | 3 | 88 | 18 | 46 | 92 | ∞ | 7 |
| 7 | 44 | 26 | 33 | 27 | 84 | 39 | ∞ |

***Figure 5.5*** *Cost Matrix*

NOTES

**Solution**: Following steps are employed to find the optimal solution:

1. The cost of the matrix can be updated to obtain a new reduced cost matrix, as shown in Figure 5.6. As shown, the total cost of the matrix is reduced by eighty-four.

| i \ j | 1 | 2 | 3 | 4 | 5 | 6 | 7 | |
|---|---|---|---|---|---|---|---|---|
| 1 | ∞ | 0 | 90 | 10 | 30 | 6 | 54 | (-3) |
| 2 | 0 | ∞ | 73 | 38 | 17 | 12 | 30 | (-4) |
| 3 | 29 | 1 | ∞ | 20 | 0 | 12 | 9 | (-16) |
| 4 | 32 | 83 | 73 | ∞ | 49 | 0 | 84 | (-7) |
| 5 | 3 | 21 | 63 | 8 | ∞ | 0 | 32 | (-25) |
| 6 | 0 | 85 | 15 | 43 | 89 | ∞ | 4 | (-3) |
| 7 | 18 | 0 | 7 | 1 | 58 | 13 | ∞ | (-26) |

***Figure 5.6** Reduced Cost Matrix*

2. The new cost matrix is again reduced. As shown in Figure 5.7 the total reduced cost is 84+7+1+4= 96. Thus, ninety-six becomes the lower bound of the matrix.

| i \ j | 1 | 2 | 3 | 4 | 5 | 6 | 7 |
|---|---|---|---|---|---|---|---|
| 1 | ∞ | 0 | 83 | 9 | 30 | 6 | 50 |
| 2 | 0 | ∞ | 66 | 37 | 17 | 12 | 26 |
| 3 | 29 | 1 | ∞ | 19 | 0 | 12 | 5 |
| 4 | 32 | 83 | 66 | ∞ | 49 | 0 | 80 |
| 5 | 3 | 21 | 56 | 7 | ∞ | 0 | 28 |
| 6 | 0 | 85 | 8 | 42 | 59 | ∞ | 0 |
| 7 | 18 | 0 | 0 | 0 | 58 | 13 | ∞ |
| | | | (-7) | (-1) | | | (-4) |

***Figure 5.7** Second Reduced Matrix*

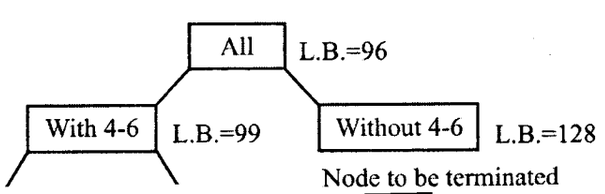
3. Figure 5.8 shows the highest level of decision tree, when ninety-six is the lower bound.





NOTES

6. Thus the total reduced cost is 96+3=99. Now ninety-nine becomes the new lower bound.
7. Figure 5.11 shows the complete branch and bound solution of a traveling salesperson problem.



NOTES

optimal solution depending on some constraints can be solved by using backtracking formulation.

- **The 8-Queens problems:** The 8-queens problem is considered as an n*n chessboard in which no two queens attack each other, i.e., no two of them are in the same row, column or diagonal.
- **Graph colouring:** It is the method of allocating the colours to the vertices of the given graph in a manner such that no two adjacent vertices have the same colour.
- **Matching:** Matching, also known as edge independent set can be referred as a set of edges, which do not have any common vertices. It can also be defined as an entire graph consisting of number of edges without common vertex.
- **FIFO search:** It involves in choosing a node from the point from where the traversal begins and accesses all the neighbouring nodes of the start vertex.
- **LIFO search:** LIFO search involves in choosing a node from where the traversal in tree needs to start, and then you need to access the neighbour node of the selected node.

## 5.12 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The backtracking technique helps in determining the efficient solution by eliminating multiple solutions of the algorithm without examining them based on the different characteristics of the problem.
2. The 8-queens problem is considered as an n*n chessboard in such a way that no two queens attack each other, that is, no two of them are in the same row, column or diagonal.
3. Graph colouring is the method of allocating colours to the vertices of the given graph in the manner such that no two adjacent vertices have the same colour.
4. The Knapsack problem is a type of maximization problem, in which the elements having maximum economic value is to be selected and that can fit in a bag or container.
5. The term branch and bound refers to all the state space search methods that generate all children of the E-node before any other node becomes E-node.

## 5.13 QUESTIONS AND EXERCISES

### Short-Answer Questions

1. What is the backtracking technique?
2. What do you understand by explicit and implicit constraints in the backtracking technique?

NOTES

3. What are the different steps in the Welch and Powel algorithm for colouring a graph?
4. What is the travelling salesman problem?

### Long-Answer Questions

1. Explain the 8-queen problem and its solution using the backtracking method.
2. What is the knapsack problem? Explain.
3. What is the branch-and-bound method?
4. How can the 0/1 knapsack problem be solved using the branch-and-bound method?

## 5.14 FURTHER READING

Horowitz, Ellis, Sartaj Sahni, Sanguthevar Rajasekaran. 2006. *Fundamentals of Computer Algorithms*. New Delhi: Galgotia Publication Pvt. Ltd.

# UNIT 6 LOWER BOUND THEORY

NOTES

## Structure



## 6.0 INTRODUCTION

You are already familiar with the backtracking and branch-and-bound techniques. In this unit, you will learn about the lower bound theory that helps in determining the most efficient algorithm for solving a given problem. Since there may exist a number of different algorithms for solving the problem, this can be done by identifying a function that will be the lower bound on the time complexity of any algorithm and can be used to solve the given problem. The lower bound for any problem can be determined by using the following techniques:

- Comparison trees
- Oracle
- Reduction technique

A comparison tree is the computational model that is used to find out the lower bound of specific kinds of algorithms such as searching, sorting and selection.

The oracle technique can also be used for determining the lower bound. The concept of oracle is used to determine the comparison output of any value. Generally, merging the largest and second largest problems can be solved by using this technique. The other methods that make the use of oracle in finding the lower bound is the state space method.

NOTES

The lower bound through the reduction is another easy and efficient technique for finding the lower bound of a given problem. In this technique, the given problem is reduced into some other problem whose lower bound is known to us. The various kinds of problems that can be solved by this technique are the convex hull problem, disjoint sets problem, online median problem and multiplication of triangular matrices.

## 6.1 UNIT OBJECTIVES

After going through this unit, you will be able to:

- Explain the concept of lower bound theory in designing algorithms
- Discuss the concept of comparison tree for finding the lower bound
- Illustrate the use of oracles in solving the different problems such as merging, largest and the second largest
- Use the reduction technique in finding the lower bound for various problems, which are:
  - o Convex hull problem
  - o Disjoint sets problem
  - o Online median problem
  - o Multiplication of triangular matrices
  - o Inverting the lower triangular matrix
  - o Computing the transitive closure

## 6.2 INTRODUCTION TO LOWER BOUND THEORY

As you know, the different number of algorithms can exist for solving the given problem. However, it is always difficult to find out which algorithm can be more efficient for the given problem. The lower bound theory provides some techniques that can be used in identifying the most efficient algorithm from the given algorithms. This can be done by identifying the function $g(n)$, which will be the lower bound on the running time of all the algorithms. It can be used to solve the given problem.

Therefore, if $f(n)$ denotes the running time of some algorithm then the mathematical notation for defining lower bound can be written as follows:

$$f(n) = \Omega(g(n)) \tag{6.1}$$

where, $g(n)$ denotes the lower bound for $f(n)$.

The equation (6.1) will only be satisfied if there exists two positive constants a and $n_0$ such that $f(n) \geq cg(n)$ and $n > n_0$.

You not only have to develop the lower bounds on the running time of the algorithms in some given problem but also have to find the most correct lower bound. However, it is always difficult to identify the most efficient lower bound for the given problem.

NOTES

The identification of lower bound for some problems is always easy to find out that can be either equal to the number of inputs or outputs of the given problem. For example, consider the problem of multiplying the two $m \times m$ matrices. The lower bound for any of the algorithm that satisfies this problem is $\Omega(m^2)$ because there are $2m^2$ inputs and $m^2$ outputs for this problem. The lower bounds that is easy to find for some problem is generally referred as trivial lower bounds.

In this chapter, you are going to learn about the different techniques for determining the lower bounds, which are as follows:

- Comparison trees
- Oracle
- Reduction technique

## 6.3 COMPARISON TREES

A comparison tree is the computational model that can be used to obtain the lower bounds for sorting and searching the algorithms. Comparison-based algorithms do not involve in the arithmetic computations but make the compression between the elements to identify the lower bounds for the given problem. The different problems that can be solved by using the comparison tree model are as follows:

- Ordered searching
- Sorting
- Selection

### 6.3.1 Ordered Searching

Suppose there is a set $S$ that contains distinct elements and follows the less than denoted by $<$ comparison relation. Let $A$ be the set that contains $n$ elements denoted by $A[1: n]$ such that $A[1] < A[2] \ldots < A[n]$. The ordered searching problem determines whether any element $x \in S$ is present within the elements in $A[1: n]$. Therefore, the comparison between the two elements of $S$ is actually the comparison between $x$ and $A[i]$. This comparison between the elements of $A$ and $x$ can produce the three outcomes, which are as follows:

- $x < A[i]$
- $x = A[i]$
- $x > A[i]$

if $x < A[i]$ then the left branch of the tree will be taken into account and if $x > A[i]$ then the right branch of the tree will be considered. However, the algorithm will terminate if $x = A[i]$.

Figure 6.1 shows the comparison tree for the linear search algorithm.

NOTES

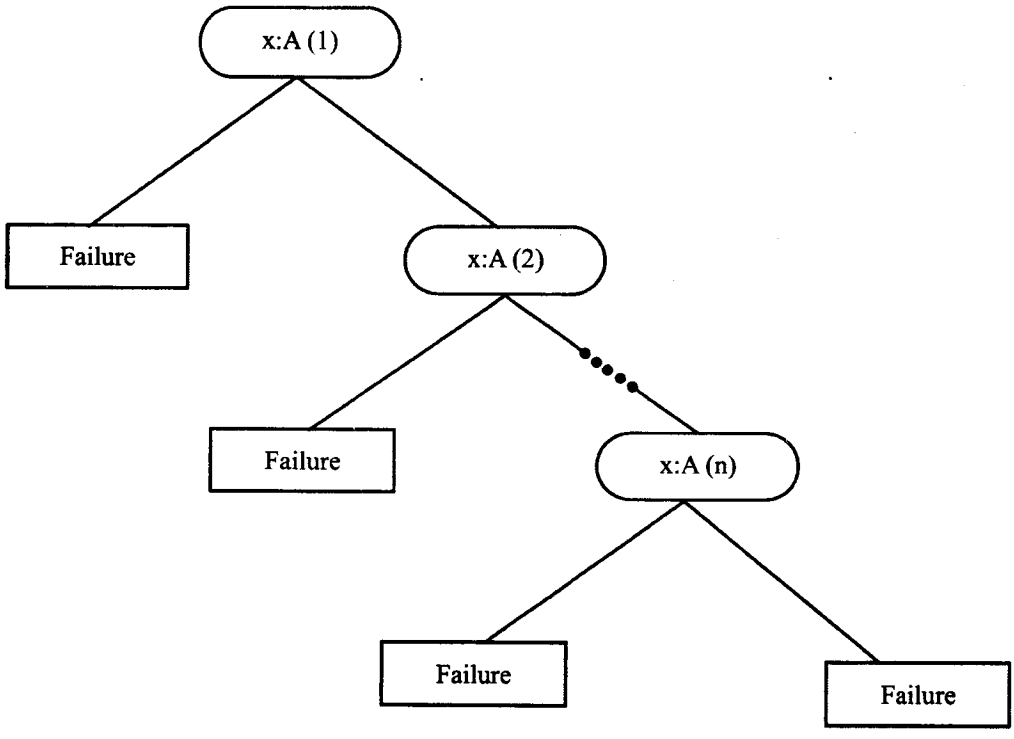



***Figure 6.1*** *Comparison Tree for the Linear Search Algorithm*

The running time for the linear search algorithm is $O(n)$, where n is the number of elements in the list. Therefore, this search algorithm proves to be very expensive.

Figure 6.2 shows the comparison tree for a binary search algorithm.

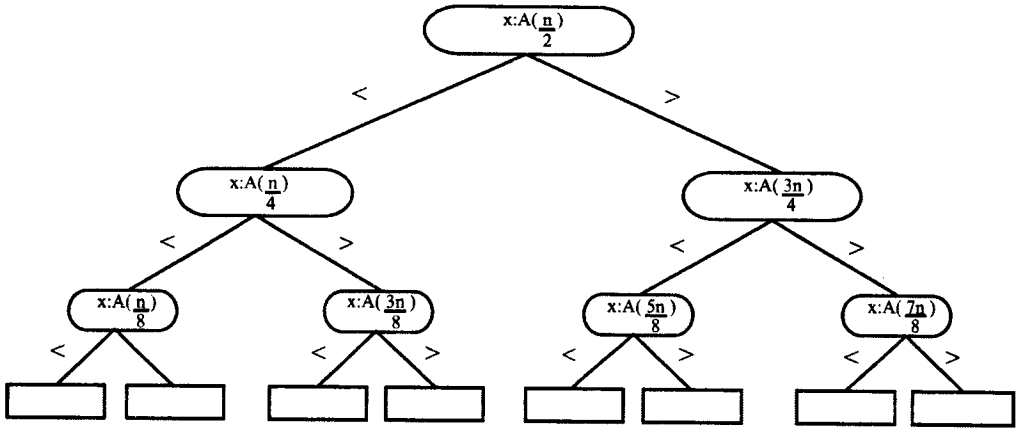



***Figure 6.2*** *Comparison Tree for the Binary Search Algorithm*

The running time for the binary search algorithm is $O(\log n)$.

Therefore, both the linear search algorithm and binary search algorithm contains at least $n$ internal nodes and consequent with the $n$ different values of $i$ against which $x$ is matched and at least one of the external node will be generated when the search becomes unsuccessful.

Now you have to find the best optimal algorithm between the linear search and binary search algorithm. Consider any comparison tree that has to search the element between the set $A[1: n]$. Therefore, there are $n$ internal nodes that can

NOTES

- Online median problem
- Multiplication of triangular matrices
- Inverting a lower triangular matrix
- Computing the transitive closure

## 6.5.1 Convex Hull Problem

The convex hull of a set S in n dimensions refers to the intersection of all the convex sets that contain S. Therefore, the convex hull problem refers to the method for finding the vertices of hull either in clockwise or anticlockwise direction.

Consider the convex hull problem as B and the sorting problem as *A*. The lower bound for the problem *A* can be obtained by reducing it in problem B in O(n) time. You now know that the running time for any algorithm that is used to sort the n elements is nlogn.

Let S $(s_1, s_2,\ldots,s_n)$ be the set that contains n elements that are to be arranged in sorting order. The problem *A* considers these values of set as numbers. However, the problem B when takes these values as inputs consider them as points in the plane. Therefore, convert these numbers of the set S into n points as $(s_1 , s_1^2)$ ,$(s_2 , s_2^2)$ ,...., $(s_n , s_n^2)$. The running time for any algorithm that arranges these numbers into points is n.

The convex hull created with these points would have n vertices corresponding to these n points and they will be arranged in sorting order according to the x-coordinate values. If the convex hull generates the output T = $(t_1 , t_1^2)$ ,$(t_2 , t_2^2)$ ,...., $(t_n , t_n^2)$ and $(t , t^2)$ is the point with minimum x-coordinate value then the sorting order of S can be determined by starting from the t and moving in anticlockwise direction of T.

For example, consider the set S having numbers 4, 2, 3 and 6. These numbers have to be arranged in ascending order. The four points can be created from these numbers, which are (4, 16), (2, 4), (3, 9) and (6, 36). Figure 6.6 shows the convex hull created by these points.

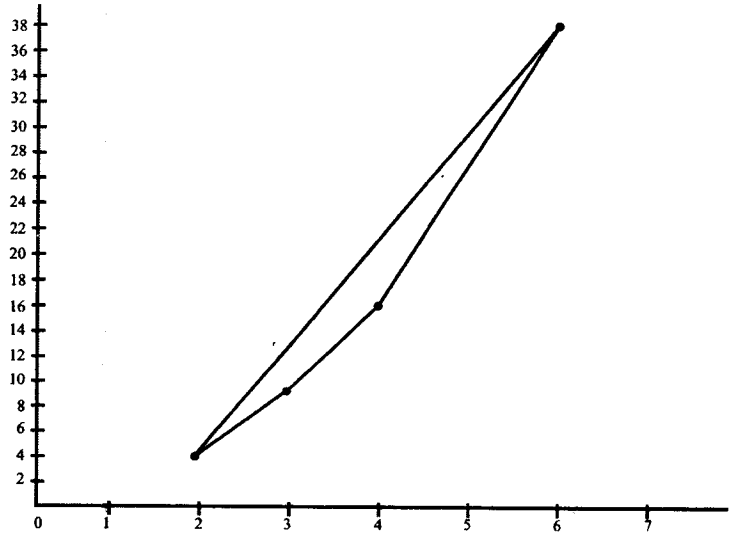



***Figure 6.6** Convex Hull Reduction*

NOTES

The anticlockwise direction of all the points on the hull will give you the sorted order of the points. Therefore, the convex hull of n given points can be determined in Ω (n log n) time.

## 6.5.2 Disjoint Sets Problem

As you know that, the two given sets are said to be disjoint if both of them do not have any common element. For example, {4, 7, 9} and {2, 5, 8} are the two disjoint sets since both of them do not have any common element. Suppose S1 = $\{a_1, a_2, a_3, \ldots, a_n\}$and S2 =$\{b_1, b_2, b_3, \ldots, b_n\}$ are the two given sets each having n elements. Therefore, the problem *A* is to find out whether the sets S1 and S2 are disjoint or not. Let B be another sorting problem in which the problem *A* has to be reduced. Now you have to show that $A \propto B$ in O(n) time.

Let the instance of B is m = 2n and the set S3 contains the keys that have to be sorted, which are $(a_1, 1)$, $(a_2, 1)$,..., $(a_n, 1)$, $(b_1, 2)$, $(b_2, 2)$,..., $(b_n, 2)$. Therefore, the creation of the set S3 takes O(n) time because, 2n tuples have to be created.

Now you have to show that the problem *A* can be solved by using the solution of the problem B in O(n) time. Prepare a set S3' that contains the elements of S3 in sorted order. Then scan the elements of S3' from left to right and analyze if there are any elements (x, 1) and (y, 2) having x = y. If any such element is present in S3' then the two given sets are disjoint otherwise they are not.

## 6.5.3 Online Median Problem

Online median problem deals with the computation of the median value at different steps, where at each step you are given a new key value. Therefore, if n key values are given to you then you have to calculate n medians.

Foe example, if the first key value given to you is six then the computed median value would be six because there is only a single key value. Suppose after this, you are given fourteen, then the median value of both the key values is either of them.

Table 6.2 shows the calculation of online median value when you are given different key values.

***Table 6.2** Calculation of Median Values at Different Steps*

| Step | Key value | Median |
|---|---|---|
| 1 | 6 | 6 |
| 2 | 14 | 6 or 14 |
| 3 | 4 | 6 |
| 4 | 18 | 6 or 14 |
| 5 | 9 | 9 |
| 6 | 12 | 9 or 12 |
| 7 | 7 | 9 |

NOTES

In the above table, at the first step, you are given a key value six. The median value of this key value should be six itself. After this you are given the value fourteen. Then the median value of six and fourteen will be either of them. The median value of six, fourteen and four will be six because it is in the middle of the three key values.

Now you have to find the lower bound for the algorithm that solves the on-line median problem. Suppose the problem of online median finding is declared as the problem B and the problem of sorting is declared as the problem *A*. The problem *A* has the n number of inputs say $S = s_1, s_2, s_3, \ldots, s_n$ that have to be arranged in a specific order. The instance of the problem B should be selected in such a way that each key value of problem B denotes online median value. This instance of problem B can be created by extending the inputs of the problem *A* from $-\infty$ to $\infty$. The running time for generating such an instance of the problem B is n. When the first $\infty$ becomes the input then the median value will be the minimum key value of S and when the third $\infty$ is given as the input then the second largest value of K would be the median value. This shows that the solution of problem B helps in solving the problem *A* and hence the problem *A* reduces to problem B in O(n) time.

## 6.5.4 Multiplication of Triangular Matrices

The triangular matrix is the square matrix which contains the zero value below or above the main diagonal. However, triangular matrices can be either the upper triangular or lower triangular. A matrix *A* with elements $\{a_{ij}\}$, $1 \leq i, j \leq n$ is said to be upper triangular if $a_{ij} = 0$ with the condition $i > j$. Similarly, the matrix *A* is said to be the lower triangular if $a_{ij} = 0$ with the condition $j > i$.

The computation of the lower bound for both the upper and lower triangular matrices are generally the same. Therefore, if you derive the lower bound for one kind of triangular matrix then it also holds for the other kind of triangular matrix too. Suppose we are computing the lower bound for the lower triangular matrices. Suppose *A* is the problem that is related with the multiplication of two full square matrices and the time needed to calculate such multiplication is *A*(n), where n is the order of the matrices. Suppose B denotes the problem of multiplying the two lower triangular matrices that takes B(n) time. However, the computation of the multiplication of two lower triangular square matrices is not an easy task and it can be shown that $B(n) = \Omega(A(n))$, which means the time complexity for any algorithm that solves the problem *A* is generally the same as the time complexity of any algorithm that is used to solve problem B.

To show $B(n) = \Omega(A(n))$, you have to prove that the problem *A* can be reduced in problem B in $O(n^2)$ time. The lower bound for the problem *A* will be $n^2$ because the input contains $2n^2$ elements and the output contains $n^2$ elements. The instance for the problem B will be two $3n \times 3n$ matrices denoted by *A*' and B' as shown:

$$A' = \begin{pmatrix} O & O & O \\ O & O & O \\ O & A & O \end{pmatrix} \text{ and } B' = \begin{pmatrix} O & O & O \\ B & O & O \\ O & O & O \end{pmatrix}$$

where O denoted the zero matrix.

Multiplication of *A*' and B' will give you the following matrix:

NOTES

$$A'B' = \begin{pmatrix} O & O & O \\ O & O & O \\ AB & O & O \end{pmatrix}$$

Therefore, it is very easy to obtain the product AB from the multiplication of $A'$ and B'. This means that the problem $A$ has been reduced to problem B in $O(n^2)$ time.

Therefore, $A(n) \leq B(3n) + O(n^2)$

$\Rightarrow B(n) \geq A(n/3) - O(n^2)$

But the running time for the problem $A$ is $n^2$, therefore, $A(n/3) = \Omega(A(n))$.

Hence, $B(n) = \Omega(A(n))$.

## 6.5.5 Inverting a Lower Triangular Matrix

Inverse of a matrix can be defined in the following way:

Let $A(n)$ be a square matrix of size n, with elements $A_{ij}$, where $1 \leq i \leq n$ and $1 \leq j \leq n$.

And, let I(n) be an identity matrix of size n, with elements $I_{ij}$, $1 \leq i \leq n$ and $1 \leq j \leq n$, such that $I_{ij} = 1$ for $i = j$ and $I_{ij} = 0$ for $i \neq j$.

If there exits another matrix B(n) of size n, with elements $B_{ij}$, where $1 \leq i \leq n$ and $1 \leq j \leq n$, such that the following relation holds:

$AB = I$

Then, the matrix B(n) is said to be the inverse of the matrix $A(n)$ and is denoted by $A^{-1}$. Also, you can say that the matrix $A(n)$ is invertible.

Now, consider the problem of inverting a lower triangular matrix, which is represented by $P_1$. A triangular matrix is invertible only if all the diagonal elements are non-zero. Let us consider another problem, $P_2$, to multiple two matrixes. To determine the lower bound for the problem $P_1$, you can reduce the problem $P_2$ to the problem $P_1$. If the time complexities of the problems $P_1$ and $P_2$ be $T_1(n)$ and $T_2(n)$, then it can be shown that $T_2(n) = O(T_1(n))$.

It has been observed that the problem $P_1$ can be reduced to problem $P_2$ in $O(n^2)$ time. The following the lower triangular matrix, L, can be constructed by using two full matrix $A(n)$ and B(n) each of size n:

$$L = \begin{bmatrix} I & O & O \\ B & I & O \\ O & A & I \end{bmatrix}$$

Where, L represents a 3n x 3n matrix, O represents zero matrix of size n and I represents identity matrix of size n. The inverse of the matrix L can be represented as follows:

$$L^{-1} = \begin{bmatrix} I & O & O \\ -B & I & O \\ AB & -A & I \end{bmatrix}$$

**CHECK YOUR PROGRESS**

6. What do you understand by lower bound through reduction?
7. What is the closest pair problem?
8. What is the convex hull problem?
9. What is the online median problem?

From the above, it is clear that the product of $A$ and B can be easily obtained from the inverse of L. Therefore,

$$T_2(n) \leq T_1(3n) + O(n^2)$$

$$\Rightarrow T_2(n) = O(T_1(n)) \quad (6.4)$$

NOTES

Similarly, it can be shown that $T_1(n) = O(T_2(n))$. Let us consider $L_A(n)$ to be the lower triangular matrix of size n. Group the elements in $L_A$ into four square sub-matrices, each of size n/2. The entire matrix $L_A$ can be written in terms of these four sub-matrices as follows:

$$L_A = \begin{bmatrix} L_{A11} & O \\ L_{A21} & L_{A22} \end{bmatrix}$$

Here, $L_{A11}$ and $L_{A22}$ represent the lower triangular matrices and $L_{A21}$ may be a full matrix. The inverse of $L_A$ can be expressed as follows:

$$L_A^{-1} = \begin{bmatrix} L_{A11}^{-1} & O \\ -L_{A22}^{-1} L_{A21} L_{A11}^{-1} & L_{A22}^{-1} \end{bmatrix}$$

For the above equation, it is clear that to invert a lower triangular matrix, $L_A$, of size n, it is sufficient to invert two lower triangular matrices $L_{A11}$ and $L_{A2,}$ each of size n/2. Then, perform two consecutive multiplications $R_1 = L_{A21} * L_{A11}^{-1}$ and then $R_2 = L_{A22}^{-1} * R_1$. Then, negate the result $R_2$. The approximate time to negate the matrix $R_2$ is $n^2 / 4$. The inverse of each of the lower triangular matrix can be obtained by repeating the above procedure. Therefore, the above equations provides a divide and conquer approach for the computation of the matrix $L_A$. The running time for this type of divide and conquer approach is given by the following recurrence relation:

$$T_1(n) \leq 2T_1\left(\frac{n}{2}\right) + 2T_2\left(\frac{n}{2}\right) + \frac{n^2}{4}$$

$$= O(T_2(n) + n^2)$$

$$\Rightarrow T_1(n) = O(T_2(n)) \quad (6.5)$$

By combining the result of (6.4) and (6.5), you can derive the following result:

$$T_1(n) = \Theta(T_2(n))$$

## 6.6 SUMMARY

In this unit, you have learnt about the lower bound theory that is used to find the most efficient algorithm in terms of running time for a given problem. The lower bounds for some problems can be easily found and for some problems, it is difficult. However, there are some techniques that help to find lower bounds. Some of these techniques are comparison tree, oracle and reduction.

The comparison tree technique for finding the lower bound a the problem makes use of the comparison between different elements rather than any arithmetic calculation. This technique is generally used in searching and sorting problems.

NOTES

In a searching problem, the comparison tree for both the searching techniques is given. On the basis of both the trees, you can see that the binary search algorithm is more efficient than the linear search algorithm. The comparison tree also helps in determining the lower bound for a sorting algorithm.

You also learnt about oracle, which is another technique to identify the lower bound for the given problem. The different problems that can be solved using this technique are as follows:

- Merging problem
- Largest and second largest problem

Apart from these problems, the lower bound for selection problem can also be identified with the concept of oracles.

However, the lower bound through reduction is the best technique for finding the lower bound. In this technique, a problem whose lower bound is known to us, is taken into account as a solution for the given problem. The different problems that can be solved by this technique are as follows:

- Convex hull problem
- Disjoint sets problem
- Online median problem
- Multiplication of triangular matrices
- Inverting a lower triangular matrix

## 6.7 ANSWERS TO 'CHECK YOUR PROGRESS'

- **Lower bound theory:** The lower bound theory provides some techniques that can be used in identifying the most efficient algorithm from the given algorithms.
- **Comparison tree:** A comparison tree is the computational model that can be used to obtain the lower bounds for sorting and searching the algorithms.
- **Oracle:** Oracle is another technique that can be used to get the lower bound. When you use the comparison model to identify the lower bound, the oracle can tell the outcome of the each comparison.
- **Convex hull problem:** The convex hull of a set $S$ in $n$ dimensions refers to the intersection of all the convex sets that contain $S$. Therefore, this problem refers to the method for finding the vertices of hull either in clockwise or anticlockwise direction.
- **Online median problem:** Online median problem deals with the computation of the median value at different steps, where at each step you are given a new key value. Therefore, if $n$ key values are given to you then you have to calculate n median.
- **Multiplication of triangular matrices:** The triangular matrix is the square matrix that contains the zero value below or above the main diagonal. However, triangular matrices can be either the upper triangular matrices or lower triangular.

NOTES

## 6.8 ANSWERS TO 'CHECK YOUR PROGRESS'

1. The different techniques that are used to determine the lower bound are as follows:
   A. Comparison trees
   B. Oracle
   C. Reduction technique
2. A comparison tree is a computational model that can be used to obtain the lower bound for the sorting and searching algorithms. Comparison-based algorithms do not involve in the arithmetic computations but they make the compression between the elements to identify the lower bounds for the given problem.
3. The different problems that can be solved using the comparison technique are as follows:
   A. Ordered searching
   B. Sorting
   C. Selection
4. Oracle is a technique that can be used to get the lower bound. When you use the comparison model to identify the lower bound, the oracle can tell the outcome of each comparison.
5. The state space method is another technique that can be used to find the lower bound for the algorithms of a problem with the use of oracles.
6. The lower bound through the reduction refers to the method of finding the lower bound for a given problem by reducing it into some other problem whose lower bound is already known.
7. The closest pair problem deals with the computation of the smallest mutual distance between the different points contained in a set.
8. The convex hull of a set S in n dimensions refers to the intersection of all the convex sets that contain S. Therefore, the convex hull problem refers to the method of finding the vertices of hull either in clockwise or anticlockwise direction.
9. Online median problem deals with the computation of the median value at the different steps, where, at each step you are given a new key value.

## 6.9 QUESTIONS AND EXERCISES

### Short-Answer Questions

1. How does the lower bound help in finding the most efficient algorithm for a given problem?
2. What are the different techniques for finding the lower bound of a given problem?
3. What is a comparison tree?
4. What is the state space method?

NOTES

## Long-Answer Questions

1. How can an ordered searching problem be solved by using the comparison tree method?
2. How is oracle used for finding the lower bound? Explain in detail.
3. How can a merging problem be solved by using oracles?
4. Explain the technique of reduction in detail.

## 6.10 FURTHER READING

Horowitz, Ellis, Sartaj Sahni, Sanguthevar Rajasekaran. 2006. *Fundamentals of Computer Algorithms*. New Delhi: Galgotia Publication Pvt. Ltd.