

SUBJECT CODE : 210242

As per Revised Syllabus of
SAVITRIBAI PHULE PUNE UNIVERSITY
Choice Based Credit System (CBCS)
S.E. (Computer) Semester - I

FUNDAMENTALS OF DATA STRUCTURES

(For IN SEM Exam - 30 Marks)

Mrs. Anuradha A. Puntambekar

M.E. (Computer)
Formerly Assistant Professor in
P.E.S. Modern College of Engineering,
Pune

Dr. Priya Jeevan Pise

Ph.D. (Computer Engineering)
Associate Professor & Head
Indira College of Engineering
& Management, Pune

Dr. Prashant S. Dhotre

Ph.D. (Computer Engineering)
Associate Professor
JSPM's Rajarshi Shahu College of Engineering,
Tathawade, Pune



FUNDAMENTALS OF DATA STRUCTURES

(For IN SEM Exam - 30 Marks)

Subject Code : 210242

S.E. (Computer) Semester - I

First Edition : August 2020

© Copyright with A. A. Puntambekar

All publishing rights (printed and ebook version) reserved with Technical Publications. No part of this book should be reproduced in any form, Electronic, Mechanical, Photocopy or any information storage and retrieval system without prior permission in writing, from Technical Publications, Pune.

Published by :



Amit Residency, Office No.1, 412, Shaniwar Peth,
Pune - 411030, M.S. INDIA, Ph.: +91-020-24495496/97
Email : sales@technicalpublications.org Website : www.technicalpublications.org

Printer :

Yogiraj Printers & Binders
Sr.No. 10/1A,
Ghule Industrial Estate, Nanded Village Road,
Tal. - Haveli, Dist. - Pune - 411041.

ISBN 978-93-90041-85-5



SPPU 19

PREFACE

The importance of **Fundamentals of Data Structures** is well known in various engineering fields. Overwhelming response to our books on various subjects inspired us to write this book. The book is structured to cover the key aspects of the subject **Fundamentals of Data Structures**.

The book uses plain, lucid language to explain fundamentals of this subject. The book provides logical method of explaining various complicated concepts and stepwise methods to explain the important topics. Each chapter is well supported with necessary illustrations, practical examples and solved problems. All the chapters in the book are arranged in a proper sequence that permits each topic to build upon earlier studies. All care has been taken to make students comfortable in understanding the basic concepts of the subject.

Representative questions have been added at the end of each section to help the students in picking important points from that section.

The book not only covers the entire scope of the subject but explains the philosophy of the subject. This makes the understanding of this subject more clear and makes it more interesting. The book will be very useful not only to the students but also to the subject teachers. The students have to omit nothing and possibly have to cover nothing more.

We wish to express our profound thanks to all those who helped in making this book a reality. Much needed moral support and encouragement is provided on numerous occasions by our whole family. We wish to thank the **Publisher** and the entire team of **Technical Publications** who have taken immense pain to get this book in time with quality printing.

Any suggestion for the improvement of the book will be acknowledged and well appreciated.

Authors

*A. A. Puntambekar
Dr. Priya Jeevan Pise
Dr. Prashant S. Dhotre*

Dedicated to God.

SYLLABUS

Fundamentals of Data Structures - (210242)

Credit Scheme	Examination Scheme and Marks
03	Mid_Semester (TH) : 30 Marks

Unit I Introduction to Algorithm and Data Structures

Introduction : From Problem to Program (Problem, Solution, Algorithm, Data Structure and Program). Data Structures : Data, Information, Knowledge, and Data structure, Abstract Data Types (ADT), Data Structure Classification (Linear and Non-linear, Static and Dynamic, Persistent and Ephemeral data structures).

Algorithms : Problem Solving, Introduction to algorithm, Characteristics of algorithm, Algorithm design tools : Pseudo-code and flowchart. **Complexity of algorithm** : Space complexity, Time complexity, Asymptotic notation- Big-O, Theta and Omega, Finding complexity using step count method, Analysis of programming constructs-Linear, Quadratic, Cubic, Logarithmic.

Algorithmic Strategies : Introduction to algorithm design strategies - Divide and Conquer, and Greedy strategy. **(Chapter - 1)**

Unit II Linear Data Structure using Sequential Organization

Concept of Sequential Organization, Overview of Array, Array as an Abstract Data Type, Operations on Array, Merging of two arrays, Storage Representation and their Address Calculation : Row major and Column Major, Multidimensional Arrays : Two-dimensional arrays, n-dimensional arrays. Concept of Ordered List, **Single Variable Polynomial** : Representation using arrays, Polynomial as array of structure, Polynomial addition, Polynomial multiplication. **Sparse Matrix** : Sparse matrix representation using array, Sparse matrix addition, Transpose of sparse matrix- Simple and Fast Transpose, Time and Space tradeoff. **(Chapter - 2)**

TABLE OF CONTENTS

Unit - I

Chapter - 1 Introduction to Algorithm and Data Structures (1 - 1) to (1 - 58)

Part I : Introduction to Data Structures

1.1 From Problem to Data Structure (Problem, Logic, Algorithm, and Data Structure)	1 - 2
1.2 Data Structures : Data, Information, Knowledge, and Data Structure.....	1 - 2
1.3 Abstract Data Types (ADT)	1 - 3
1.3.1 Realization of ADT	1 - 3
1.3.2 ADT for Arrays	1 - 4
1.4 Data Structure Classification	1 - 5
1.4.1 Linear and Non linear Data Structure.	1 - 5
1.4.2 Static and Dynamic Data Structure.	1 - 5
1.4.3 Persistent and Ephimeral Data Structure.	1 - 6

Part II : Introduction to Algorithms

1.5 Problem Solving	1 - 7
1.6 Difficulties in Problem Solving	1 - 9
1.7 Introduction to Algorithm	1 - 10
1.7.1 Characteristics of Algorithm	1 - 10
1.8 Algorithm Design Tools	1 - 11
1.8.1 Pseudocode	1 - 11
1.8.2 Flowchart	1 - 16
1.9 Step Count Method	1 - 19
1.10 Complexity of Algorithm	1 - 19

1.10.1 Space Complexity	1 - 19
1.10.2 Time Complexity	1 - 20
1.11 Asymptotic Notations	1 - 21
1.11.1 Big oh Notation	1 - 21
1.11.2 Omega Notation.	1 - 23
1.11.3 Θ Notation	1 - 24
1.11.4 Properties of Order of Growth	1 - 25
1.11.5 How to Choose the Best Algorithm ?	1 - 26
1.12 Analysis of Programming Constructs	1 - 28
1.13 Recurrence Relation	1 - 32
1.14 Solving Recurrence Relation	1 - 32
1.14.1 Substitution Method	1 - 32
1.14.2 Master's Method	1 - 40
1.15 Best, Worst and Average Case Analysis	1 - 45
1.16 Introduction to Algorithm Design Strategies	1 - 47
1.16.1 Divide and Conquer	1 - 48
1.16.1.1 Merge Sort	1 - 48
1.16.2 Greedy Strategy	1 - 54
1.16.2.1 Example of Greedy Method	1 - 55

Unit - II

Chapter - 2 Linear Data Structure using Sequential Organization (2 - 1) to (2 - 66)

2.1 Concept of Sequential Organization	2 - 2
2.2 Array as an Abstract Data Type	2 - 2
2.3 Array Overview	2 - 3
2.4 Operations on Array	2 - 4
2.5 Merging of Two Arrays	2 - 9
2.6 Storage Representation and their Address Calculation	2 - 12

2.7 Multidimensional Arrays	2 - 15
2.7.1 Two Dimensional Array	2 - 15
2.7.2 Three Dimensional Array	2 - 21
2.8 Concept of Ordered List	2 - 23
2.8.1 Ordered List Methods	2 - 24
2.8.2 Built in Functions	2 - 26
2.8.3 List Comprehension	2 - 27
2.9 Single Variable Polynomial.....	2 - 28
2.9.1 Representation	2 - 28
2.9.2 Polynomial Addition	2 - 28
2.9.3 Polynomial Multiplication	2 - 37
2.9.4 Polynomial Evaluation	2 - 38
2.10 Sparse Matrix	2 - 42
2.10.1 Sparse Matrix Representation using Array	2 - 43
2.10.2 Sparse Matrix Addition	2 - 45
2.10.3 Transpose of Sparse Matrix	2 - 51
2.10.4 Fast Transpose	2 - 56
2.11 Time and Space Tradeoff	2 - 64

Unit - I

1

Introduction to Algorithm and Data Structures

Syllabus

Introduction : From Problem to Program (Problem, Solution, Algorithm, Data Structure and Program). Data Structures : Data, Information, Knowledge, and Data structure, Abstract Data Types (ADT), Data Structure Classification (Linear and Non-linear, Static and Dynamic, Persistent and Ephemeral data structures).

Algorithms : Problem Solving, Introduction to algorithm, Characteristics of algorithm, Algorithm design tools : Pseudo-code and flowchart. **Complexity of algorithm :** Space complexity, Time complexity, Asymptotic notation- Big-O, Theta and Omega, Finding complexity using step count method, Analysis of programming constructs-Linear, Quadratic, Cubic, Logarithmic.

Algorithmic Strategies : Introduction to algorithm design strategies - Divide and Conquer, and Greedy strategy.

Introduction : From Problem to Program (Problem, Solution, Algorithm, Data Structure and Program). Data Structures : Data, Information, Knowledge, and Data structure, Abstract Data Types (ADT), Data Structure Classification (Linear and Non-linear, Static and Dynamic, Persistent and Ephemeral data structures).

Algorithms : Problem Solving, Introduction to algorithm, Characteristics of algorithm, Algorithm design tools : Pseudo-code and flowchart. **Complexity of algorithm :** Space complexity, Time complexity, Asymptotic notation- Big-O, Theta and Omega, Finding complexity using step count method, Analysis of programming constructs-Linear, Quadratic, Cubic, Logarithmic.

Algorithmic Strategies : Introduction to algorithm design strategies - Divide and Conquer, and Greedy strategy.

Contents

1.1	From Problem to Data Structure (Problem, Logic, Algorithm, and Data Structure)	
1.2	Data Structures : Data, Information, Knowledge, and Data Structure	
1.3	Abstract Data Types (ADT)	Dec.-10, 11, May-10, 11, 12, 13, 19, Marks 6
1.4	Data Structure Classification	May-17, Dec.-18, 19, Marks 4
1.5	Problem Solving	Dec.-09, 10, May-10, 11, Marks 12
1.6	Difficulties in Problem Solving	Dec.-10, May-16, Marks 4
1.7	Introduction to Algorithm.	Dec.-16, May-18, 19, Marks 4
1.8	Algorithm Design Tools	May-10, 11, Dec.-10, Marks 8
1.9	Step Count Method	
1.10	Complexity of Algorithm	Dec.-09, 10, 11, 19, May-12, 13, Marks 8
1.11	Asymptotic Notations	Dec.-09, 10, 11, 16, 17, 19, May-10, 12, 13, 18, 19, Marks 8
1.12	Analysis of Programming Constructs	May-10, 13, 14, Dec.-11, 13, Marks 8
1.13	Recurrence Relation	Dec.-18, Marks 2
1.14	Solving Recurrence Relation	
1.15	Best, Worst and Average Case Analysis	
1.16	Introduction to Algorithm Design Strategies . .	May-17, 18, Dec.-17, 19, Marks 6

- | | | |
|------|--|---|
| 1.1 | From Problem to Data Structure (Problem, Logic, Algorithm, and Data Structure) | |
| 1.2 | Data Structures : Data, Information, Knowledge, and Data Structure | |
| 1.3 | Abstract Data Types (ADT) | Dec.-10, 11,
May-10, 11, 12, 13, 19, Marks 6 |
| 1.4 | Data Structure Classification | May-17, Dec.-18, 19, Marks 4 |
| 1.5 | Problem Solving | Dec.-09, 10, May-10, 11, . . . Marks 12 |
| 1.6 | Difficulties in Problem Solving | Dec.-10, May-16, Marks 4 |
| 1.7 | Introduction to Algorithm. | Dec.-16, May-18, 19, Marks 4 |
| 1.8 | Algorithm Design Tools | May-10, 11, Dec.-10, Marks 8 |
| 1.9 | Step Count Method | |
| 1.10 | Complexity of Algorithm | Dec.-09, 10, 11, 19,
May-12, 13, Marks 8 |
| 1.11 | Asymptotic Notations | Dec.-09, 10, 11, 16, 17, 19,
May-10, 12, 13, 18, 19, Marks 8 |
| 1.12 | Analysis of Programming Constructs | May-10, 13, 14, Dec.-11, 13, . . Marks 8 |
| 1.13 | Recurrence Relation | Dec.-18, Marks 2 |
| 1.14 | Solving Recurrence Relation | |
| 1.15 | Best, Worst and Average Case Analysis | |
| 1.16 | Introduction to Algorithm Design Strategies . . | May-17, 18, Dec.-17, 19, Marks 6 |

Part I : Introduction to Data Structures

1.1 From Problem to Data Structure (Problem, Logic, Algorithm, and Data Structure)

- Problem can be solved using series of actions. The steps that are followed in order to solve the problem are collectively called as **algorithm**.
- There are some problems for which the direct solution does not exist. For building the solution to such problems some reasoning is required. This reasoning is usually based on knowledge and prior experience. Hence the process of trial and error is followed to solve the given problem. This process is called **logic building** for particular solution.
- For example – For displaying the list of numbers in reverse direction, we should read and display the last element of the list, then last but one and so on.
- In computer science, when we want to solve any problem, then we need Data, the **data structure** that will arrange the data in some specific manner and the algorithm which will handle this data structure in some **systematic manner**.
- Data structure is the means by which we can **model the problem**.
- For example - Arrays - It contains integer elements(data) which are arranged in sequential organization (data structure) and then algorithms such as - addition of all elements, or for sorting the elements in specific manner are applied on this data structure.
- Thus Data structure is used for arranging the data and algorithm is used to solve the problem by systematic execution of each step.

1.2 Data Structures : Data, Information, Knowledge, and Data Structure

Data : Data refers to raw data or unprocessed data.

Information : Information is data that has been processed in such a way that it will become meaningful to the person who receives it. The information has structure and context.

Knowledge : Knowledge is basically something which person knows. Knowledge requires a person to understand what information is, based on their experience and knowledge base

Data structure : A data structure is a particular way of organizing data in a computer so that it can be used effectively.

For example – Consider set of elements which can be stored in array data structure. The representation of array containing set of elements is as follows –

0	1	2	3	4
10	20	30	40	50

Any element in above array can be referred by an index. For instance the element 40 can be accessed as array[3].

Difference between Data and Information

Sr. No.	Data	Information
1.	Data is in raw form.	Information is processed form of data.
2.	Data may or may not be meaningful.	Information is always meaningful.
3.	Data may not be in some specific order.	Information generally follows specific ordering.
4.	For example - each Student's marks in exam is one piece of data.	For example - The average score of a class is information that can be derived from given data.

1.3 Abstract Data Types (ADT)

SPPU : Dec.-10, 11, May-10, 11, 12, 13, 19, Marks 6

The abstract data type is a triple of D - Set of domains, F - Set of functions, A - axioms in which only what is to be done is mentioned but how is to be done is not mentioned.

In ADT, all the implementation details are hidden. In short

ADT = Type + Function names + Behavior of each function

We will discuss in further chapters how the ADT can be written

1.3.1 Realization of ADT

The abstract datatype consists of following things

1. Data used along with its data type.
2. Declaration of functions which specify only the purpose. That means "What is to be done" in particular function has to be mentioned but "how is to be done" must be hidden.
3. Behavior of function can be specified with the help of data and functions together.

Thus ADT allows programmer to hide the implementation details. Hence it is called **abstract**. For example : If we want to write ADT for a set of integers, then we will use following method

AbstractDataType Set {

Instances : Set is a collection of integer type of elements.

Preconditions : none

Operations :

1. **Store ()** : This operation is for storing the integer element in a set.
 2. **Retrieve ()** : This operation is for retrieving the desired element from the given set.
 3. **Display ()** : This operation is for displaying the contents of set.
- }

There is a specific method using which an ADT can be written. We begin with keyword **AbstractDataType** which is then followed by name of the data structure for which we want to write an ADT. In above given example we have taken **Set** data structure.

- Then inside a pair of curly brackets ADT must be written.
- We must first write **instances** in which the basic idea about the corresponding data structure must be given. Generally in this section definition of corresponding data structure is given.
- Using **Preconditions** or **Postconditions** we can mention specific conditions that must be satisfied before or after execution of corresponding function.
- Then a listing of all the required operations must be given. In this section, we must specify the purpose of the function. We can also specify the data types of these functions.

1.3.2 ADT for Arrays

AbstractDataType Array

{

Instances : An array A of some size, index i and total number of elements in the array n.

Operations :

Store () – This operation stores the desired elements at each successive location.

display () – This operation displays the elements of the array.

}

Review Questions

1. Explain what is abstract data type ?

SPPU : Dec.-10, Marks 4

2. What is ADT ? Write ADT for an arrays.

SPPU : May-10,11, Dec.-11, Marks 6, May-12, Marks 4

3. Explain following terminologies : i) Data type ii) Data structure iii) Abstract data type.

SPPU : May-13, Marks 6

4. Define (1) ADT (2) Data Structure.

SPPU : May-19, Marks 2

1.4 Data Structure Classification

SPPU : May-17, Dec.-18, 19, Marks 4

The data structures can be divided into two basic types primitive data structure and non primitive data structure. The Fig. 1.4.1. shows various types of data structures.

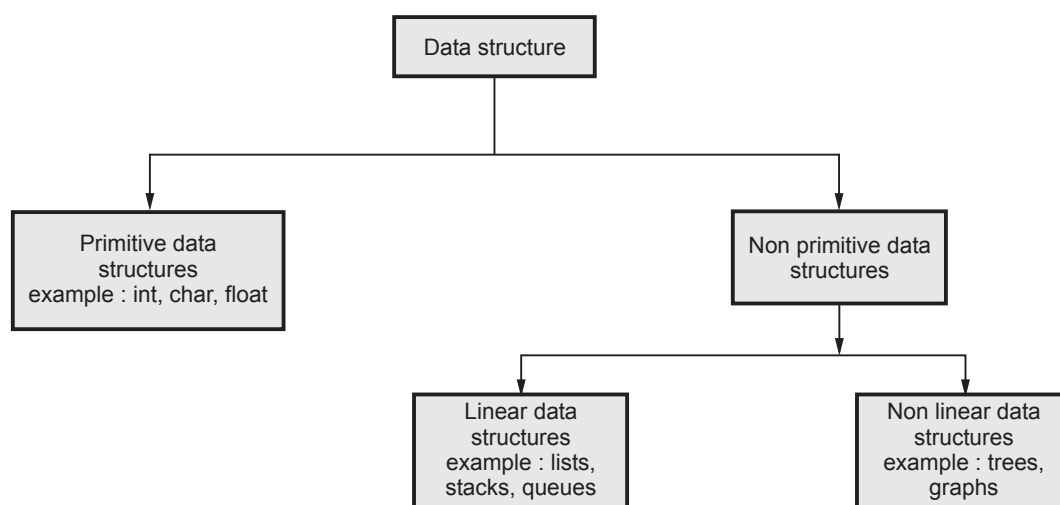


Fig. 1.4.1 Classification of data structure

1.4.1 Linear and Non linear Data Structure

Linear data structures are the data structures in which data is arranged in a list or in a straight sequence.

For example : arrays, list.

Non linear data structures are the data structures in which data may be arranged in hierarchical manner.

For example : trees, graphs.

1.4.2 Static and Dynamic Data Structure

The static data structures are data structures having fixed size memory utilization. One has to allocate the size of this data structure before using it.

For example (Static Data Structure) :

Arrays in C is a static data structure. We first allocate the size of the arrays before its actual use. Sometimes what ever size we have allocated for the arrays may get wasted or we may require larger than the one which is existing. Thus the data structure is static in nature.

The dynamic data structure is a data structure in which one can allocate the memory as per his requirement. If one does not want to use some block of memory, he can deallocate it.

For example (Dynamic Data Structure (Dynamic)) :

Linked list, the linked list is a collection of many nodes. Each node consists of data and pointer to next node. In linked list user can create as much nodes (memory) as he wants, he can dellocate the memory which cannot be utilized further.

The advantage of dynamic data structure over the static data structure is that there is no wastage of memory.

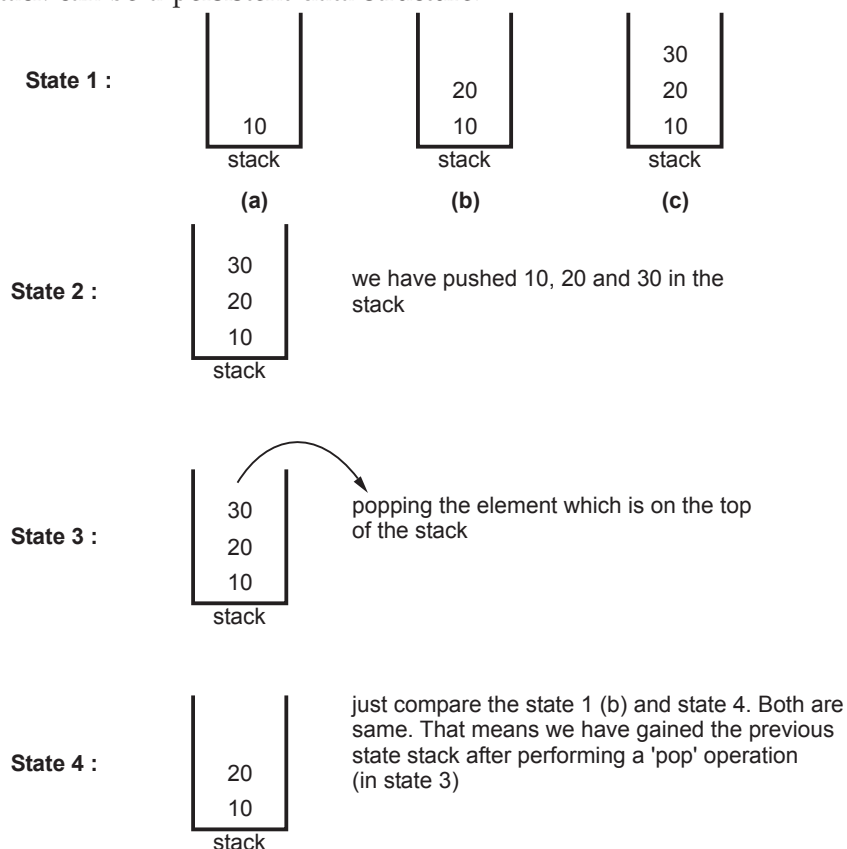
1.4.3 Persistent and Ephimeral Data Structure

The persistent data structures are the data structures which retain their previous state and modifications can be done by performing certain operations on it.

For example stack can be a persistent data structure in following case

If we push 10, 20, 30 onto it and by performing pop operation we can gain its previous state. That means -

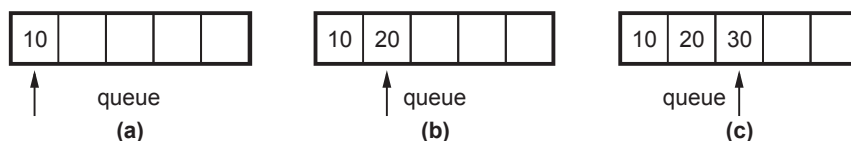
Thus stack can be a persistent data structure.



The ephemeral data structures are the data structures in which we cannot retain its previous state.

For example Queues

State 1 : In queues elements are inserted by one end and deleted by other end. Let us insert 10, 20, 30 in the queue.



State 2 : Now if we delete the element, we can delete it only by other end. That means 10 can be deleted. And queue will be -



Now this state is not matching with any of the previous states. That means we can not retain the previous state of the data structure even after performing certain operations. Such a data structure is called ephemeral data structure.

Review Questions

1. Differentiate between linear and non linear data structure with example.
2. Explain static and dynamic data structures with examples
3. Write short note on Linear and Non-Linear data structure with example

SPPU : May-17, Marks 3

SPPU : Dec.-18, Marks 4

SPPU : Dec.-19, Marks 4

Part II : Introduction to Algorithms

1.5 Problem Solving

SPPU : Dec.-09, 10, May-10, 11, Marks 12

Problem solving is based on the decisions that are taken. Following are the **six steps of problem solving** -

1. **Identify the problem :** Identifying the problem is the first step in solving the problem. Problem identification is very essential before solving any problem.
2. **Understand the problem :** Before solving any problem it is important to understand it. There are three aspects based on which the problem can be understood.

- **Knowledgebase** : While solving the problem the knowledgebase can be related to a **person or a machine**. If the problem is to be solved for the person then it is necessary to know **what the person knows**. If the problem is to be solved for the machine then its **instruction set** must be known. Along with this the problem solver can make use of his/her own instruction set.
 - **Subject** : Before solving the problem the subject on which the problem is based must be known. For instance : To solve the problem involving Laplace, it is necessary to know about the Laplace transform.
 - **Communication** : For understanding the problem, the developer must communicate **with the client**.
3. **Identify the alternative ways to solve the problem** : The alternative way to solve the problem must be known to the developer. These alternatives can be decided by **communicating** with the customer.
 4. **Select the best way to solve the problem from list of alternative solutions** : For selecting the best way to solve the problem, the merits and demerits of each problem must be analysed. The criteria to evaluate each problem must be predefined.
 5. **List the instructions using the selected solution** : Based on the knowledgebase(created/used in step 2) the step by step instructions are listed out. Each and every instruction must be understood by the person or the machine involved in the problem solving process.
 6. **Evaluate the solution** : When the solution is evaluated then - i) check whether the solution is correct or not ii) check whether it satisfies the requirements of the customer or not.

Example 1.5.1 *An admission charge for the cinemax theatre varies according to the age of the person. Complete the six problem solving steps to calculate the ticket charge given the age of the person. The charges are as follows :*

- | | | |
|----------------------|---------------------|-------------------|
| i) Over 55 : ₹ 50.00 | ii) 21-54 : ₹ 75.00 | |
| iii) 13-20 : ₹ 50.00 | iv) 3-12 : ₹ 25.00 | v) Under 3 : free |

(Hint: No need to draw flowchart)

SPPU : Dec.-09, Marks 12

Solution : Step 1 : Identify the problem : What are the charges for the person for the cine ticket?

Step 2 : Understand the problem : Here the age of the person must be known so that the charge for the ticket can be calculated.

Step 3 : Identify alternatives : No alternative is possible for this problem.

Step 4 : Select the best way to solve the problem : The only way to solve this problem is to calculate ticket charge based on the age of the person.

Step 5 : Prepare the list of selected solutions :

- i. Enter the age of the person.
- ii. If age > 55 charge for the ticket is ₹ 50.00
- iii. If age >= 21 and <= 54 charge for the ticket is ₹ 75.00
- iv. If age >= 13 and <= 20 charge for the ticket is ₹ 50.00
- v. If age >= 3 and <= 12 charge for the ticket is ₹ 25.00
- vi. If age <= 3 charge for the ticket is ₹ 00.00
- vii. Print charge

Step 6 : Evaluate Solution : The charge for the ticket should be according to the age of the person.

Review Questions

1. State a reason why each of the six problem solving steps is important in developing the best solution for a problem. Give one reason for each step. **SPPU : May-10, Marks 8**
2. Consider any one problem and solve that problem using six steps of problem solving. Explain each step in detail. **SPPU : Dec.-10, Marks 8**
3. Describe the six steps in problem solving with example. **SPPU : May-11, Marks 8**

1.6 Difficulties in Problem Solving

SPPU : Dec.-10, May-16, Marks 4

Various difficulties in solving the problem are -

- People do **not know** how to solve particular problem.
- Many times people get **afraid of** taking decisions
- While following the problem solving steps people complete one or two **steps inadequately**.
- People do **not define** the problem statements correctly.
- They do not generate the **sufficient list of alternatives**. Sometimes good alternatives might get eliminated. Sometimes the merits and demerits of the alternatives is defined hastily.
- The **sequence of solution** is not defined **logically**, or the focus of the design is sometimes on **detailed work** before the framework solution.

- When solving the problems on **computer** then **writing the instructions** for the computer is most crucial task. With **lack of knowledgebase** one can not write the proper instruction set for the computers.

Review Questions

1. State and explain any four difficulties with problem solving

SPPU : Dec.-10, Marks 4

2. What are the difficulties in problem solving ? Explain any four steps in problem solving with suitable example.

SPPU : May-16, Marks 4

1.7 Introduction to Algorithm

SPPU : Dec.-16, May-18, 19, Marks 4

Definition of Algorithm : An algorithm is a finite set of instructions for performing a particular task. The instructions are nothing but the statements in simple English language.

Example : Let us take a very simple example of an algorithm which adds the two numbers and store the result in a third variable.

Step 1 : Start.

Step 2 : Read the first number is variable 'a'

Step 3 : Read the second number in variable 'b'

Step 4 : Perform the addition of both the numbers i.e. and store the result in variable 'c'.

Step 5 : Print the value of 'c' as a result of addition.

Step 6 : Stop.

1.7.1 Characteristics of Algorithm

1. Each algorithm is supplied with **zero or more inputs**.
2. Each algorithm must **produce** at least one **output**
3. Each algorithm should have **definiteness** i.e. each instruction must be **clear and unambiguous**.
4. Each algorithm should have **finiteness** i.e. if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after finite number of steps.

Each algorithm should have **effectiveness** i.e. every instruction must be sufficiently basic that it can in principal be carried out by a person using only pencil and paper. Moreover each instruction of an algorithm must also be feasible.

Review Questions

1. Define algorithm and its characteristics.

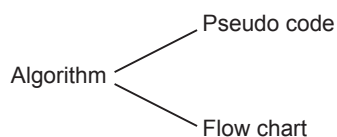
SPPU : Dec.-16, Marks 4, May-19, Marks 2

2. Define and explain following terms – (i) Data (ii) Data structure (iii) Algorithm.

SPPU : May-18, Marks 3

1.8 Algorithm Design Tools

There are various ways by which we can specify an algorithm.



Let us discuss these techniques

1.8.1 Pseudocode

SPPU : May-10, 11, Dec.-10, Marks 8

- **Definition :** Pseudo code is nothing but an informal way of writing a program. It is a combination of algorithm written in simple English and some programming language.
- In pseudo code, there is no restriction of following the syntax of the programming language.
- Pseudo codes cannot be compiled. It is just a previous step of developing a code for given algorithm.
- Even sometimes by examining the pseudo code one can decide which language has to select for implementation.

Algorithm heading
It consists of name of algorithm, problem description, input and output

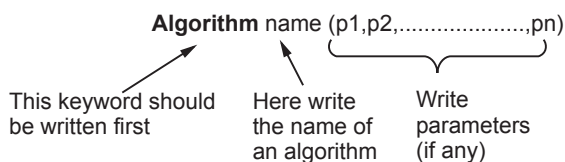
Algorithm body
It consists of logical body of the algorithm by making use of various programming constructs and assignment statement.

The algorithm is broadly divided into two sections. (Refer Fig. 1.8.1)

Fig. 1.8.1 Structure of algorithm

Let us understand some rules for writing the algorithm.

1. Algorithm is a procedure consisting of heading and body. The heading consists of keyword **Algorithm** and name of the algorithm and parameter list. The syntax is



2. Then in the heading section we should write following things :

```
//Problem Description :  
//Input :  
//Output :
```

3. Then body of an algorithm is written, in which various programming constructs like if, for, while or some assignment statements may be written.
4. The compound statements should be enclosed within { and } brackets.
5. Single line comments are written using // as beginning of comment.
6. The **identifier** should begin by letter and not by digit. An identifier can be a combination of alphanumeric string.

It is not necessary to write data types explicitly for identifiers. It will be represented by the context itself. Basic data types used are integer, float, char, Boolean and so on. The pointer type is also used to point memory location. The compound data type such as structure or record can also be used.

7. Using assignment operator \leftarrow an assignment statement can be given.

For instance :

```
Variable  $\leftarrow$  expression
```

8. There are other types of operators such as Boolean operators such as true or false. Logical operators such as **and**, **or**, **not**. And relational operators such as $<$, $<=$, $>$, $>=$, $=$, \neq .
9. The array indices are stored with in square brackets '[' '']. The index of array usually start at zero. The multidimensional arrays can also be used in algorithm.
10. The inputting and outputting can be done using **read** and **write**.

For example :

```
write("This message will be displayed on console");  
read(val);
```

11. The conditional statements such as if-then or if-then-else are written in following form :

```
if (condition) then statement  
if (condition) then statement else statement
```

If the **if-then** statement is of compound type then { and } should be used for enclosing block.

12. **while** statement can be written as :

```
while (condition) do  
{  
    statement 1  
    statement 2
```

```

        :
        :
    statement n
}

```

While the condition is true the block enclosed with { } gets executed otherwise statement after } will be executed.

13. The general form for writing for loop is :

```

for variable  $\leftarrow$  value1 to valuen do
{
    statement 1
    statement 2
    :
    :
    statement n
}

```

Here **value₁** is initialization condition and **value_n** is a terminating condition.

Sometime a keyword **step** is used to denote increment or decrement the value of variable for example

```

for i  $\leftarrow$  1 to n step 1
{
    Write (i)
}

```

Here variable i is incremented by 1 at each iteration

14. The **repeat - until** statement can be written as :

```

repeat
    statement 1
    statement 2
    :
    :
    statement n
until (condition)

```

15. The **break** statement is used to exit from inner loop. The **return** statement is used to return control from one point to another. Generally used while exiting from function.

Note that statements in an algorithm executes in sequential order i.e. in the same order as they appear-one after the other.

A sub-algorithm is complete and independently defined algorithmic module. This module is actually called by some main algorithm or some another sub-algorithm.

There are two types of sub-algorithms -

1. Function sub-algorithm
2. Procedure sub-algorithm

Example of function sub-algorithm

```
Function sum(a,b:integer):integer
{
    //body of function
    //return statement
}
```

Example of Procedure sub-algorithm

```
Procedure sum(a,b:integer):integer
{
    //body of procedure
}
```

The difference between function sub-algorithm and procedure sub-algorithm is that, the function can return only one value where as the procedure can return more than one value.

Examples of Pseudo Code

Example 1.8.1 Write an pseudo code to count the sum of n numbers.

Solution :

```
Algorithm sum (1, n)
//Problem Description: This algorithm is for finding the
//sum of given n numbers
//Input: 1 to n numbers
//Output: The sum of n numbers
    result  $\leftarrow$  0
    for i  $\leftarrow$  1 to n do i  $\leftarrow$  i+1
        result  $\leftarrow$  result+i
    return result
```

Example 1.8.2 Write a pseudo code to check whether given number is even or odd.

Solution :

```
Algorithm eventest (val)
//Problem Description: This algorithm test whether given
//number is even or odd
//Input: the number to be tested i.e. val
//Output: Appropriate messages indicating even or oddness
if (val%2=0) then
    write ("Given number is even")
else
    write("Given number is odd")
```

Example 1.8.3 Write a pseudo code for sorting the elements.

Solution :

```
Algorithm sort (a,n)
//Problem Description: sorting the elements in ascending order
//Input:An array a in which the elements are stored and n
//is total number of elements in the array
//Output: The sorted array
for i  $\leftarrow$  1 to n do
  for j  $\leftarrow$  i+1 to n - 1 do
  {
    if (a[i]>a[j]) then
    {
      temp  $\leftarrow$  a[i]
      a[i]  $\leftarrow$  a[j]
      a[j]  $\leftarrow$  temp
    }
  }
write ("List is sorted")
```

Example 1.8.4 Write a pseudo code to find factorial of n number. (n!)

Solution :

```
Algorithm fact (n)
//Problem Description: This algorithm finds the factorial
//of given number n
//Input: The number n of which the factorial is to be
//calculated.
//Output:factorial value of given n number.
if (n  $\leftarrow$  1) then
  return 1
else
  return n*fact(n - 1)
```

Example 1.8.5 Write a pseudo code to perform multiplication of two matrices.

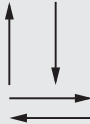


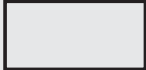
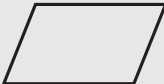
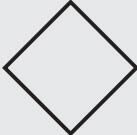

Solution :

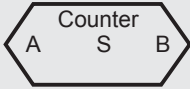


```
Algorithm Mul(A,B,n)
//Problem Description:This algorithm is for computing
//multiplication of two matrices
//Input:The two matrices A,B and order of them as n
//Output:The multiplication result will be in matrix C
for i  $\leftarrow$  1 to n do
  for j  $\leftarrow$  1 to n do
    C[i,j]  $\leftarrow$  0
    for k  $\leftarrow$  1 to n do
      C[i,j]  $\leftarrow$  C[i,j]+A[i,k]B[k,j]
```

1.8.2 Flowchart

- Flowcharts are the graphical representation of the algorithms.
- The algorithms and flowcharts are the final steps in organizing the solutions.
- Using the algorithms and flowcharts the programmers can find out the bugs in the programming logic and then can go for coding.
- Flowcharts can show errors in the logic and set of data can be easily tested using flowcharts.

Symbols used in Flowchart

Flow lines are used to indicate the flow of data. The arrow heads are important for flowlines. The flowlines are also used to connect the different blocks in the flowchart.	 <p>Flowline</p>
These are termination symbols. The start of the flowchart is represented by the name of the module in the ellipse and the end of the flowchart is represented by the keywords End or Stop or Exit	 <p>Start</p>  <p>End/Stop/Exit</p>
The rectangle indicates the processing. It includes calculations, opening and closing files and so on.	 <p>Processing</p>
The parallelogram indicates input and output.	 <p>I/O</p>
The diamond indicates the decision. It has one entrance and two exits. One exit indicates the true and other indicates the false.	 <p>Decision</p>
The process module has only one entrance and one exit.	 <p>Process Module</p>

<p>This polygon indicates the loop</p> <p>A indicates the starting of the counter</p> <p>S indicates the step by which the counter is incremented or decremented.</p> <p>B indicates the ending value of the counter</p> <p>Using the counter the number of times the looping instruction gets executed.</p>	
<p>The on-page connector connects the two different sections on the same page. A letter is written inside the circle.</p> <p>The off-page connector connects the two different sections on the different pages. The page numbers are used in off-page connector.</p> <p>These two symbols should be used as little as possible because then the readability of the flowchart may get affected.</p>	 <p>On page connector</p>  <p>Off page connector</p>

Example 1.8.6 What do you mean by flow chart ? Give the meaning of each symbol used in flowchart. Draw flowchart to compute the sum of elements from a given integer array

SPPU : May-10, Marks 8

Solution : Refer section 1.8.2 for flowchart and symbols used in it.

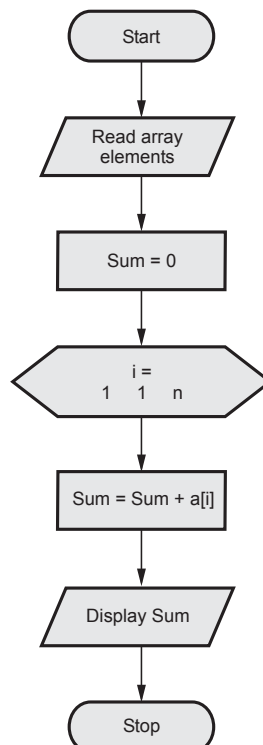


Fig. 1.8.2

Example 1.8.7 Design and explain an algorithm to find the sum of the digits of an integer number.

SPPU : Dec.-10, Marks 6

Solution :

```

Read N
Remainder=0
Sum=0
Repeat
    Remainder=N mod 10
    Sum=Sum+remainder
    N=N/10
Until N<0
Display Sum
End

```

Example 1.8.8 What is a difference between flowchart and algorithm ? Convert the algorithm for computing factorial of given number into flowchart.

SPPU : May-11, Marks 8

Solution : Algorithm is a set of instructions written in natural language or pseudo code and flowchart is a graphical representation of an **algorithm**.

```

Read N
Set i and F to 1
While i <= N
    F = F * i
    Increase the value of i by 1
Display F
End

```

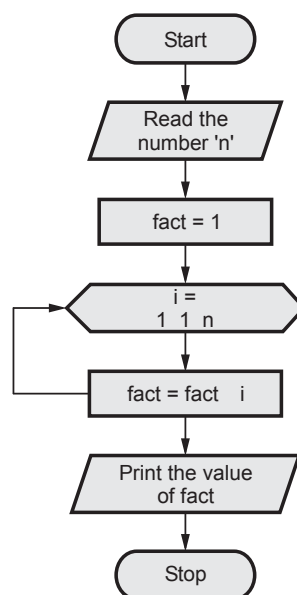


Fig. 1.8.3 Flowchart for factorial

1.9 Step Count Method

Definition : The frequency count is a count that denotes how many times particular statement is executed.

For Example : Consider following code for counting the frequency count

```
void fun()
{
    int a;
    a=10; .....1
    printf("%d",a); .....1
}
```

The frequency count of above program is 2.

1.10 Complexity of Algorithm SPPU : Dec.-09, 10, 11, 19, May-12, 13, Marks 8

1.10.1 Space Complexity

The space complexity can be defined as amount of memory required by an algorithm to run.

To compute the space complexity we use two factors : **constant and instance characteristics**. The space requirement $S(p)$ can be given as

$$S(p) = C + S_p$$

where C is a constant i.e. fixed part and it denotes the space of inputs and outputs. This space is an amount of space taken by instruction, variables and identifiers. And S_p is a space dependent upon instance characteristics. This is a variable part whose space requirement depends on particular problem instance.

There are two types of components that contribute to the space complexity - Fixed part and variable part.

The **fixed part** includes space for

- Instructions
- Variables
- Array size
- Space for constants

The **variable part** includes space for

- The variables whose size is dependent upon the particular problem instance being solved. The control statements (such as for, do, while, choice) are used to solve such instances.
- Recursion stack for handling recursive call.

Consider an example of algorithm to compute the space complexity.

Example 1.10.1 Compute the space needed by the following algorithms justify your answer.

Algorithm Sum(a,n)

```
{
    s := 0.0 ;
    For i := 1 to n do
        s := s + a[i] ;
    return s
}
```

In the given code we require space for

```
s := 0      ← O(1)
For i := 1 to n  ← O(n)
    s := s + a[i] ; ← O(n)
returns ;      ← O(1)
```

Hence the space complexity of given algorithm can be denoted in terms of big-oh notation. It is **O(n)**. We will discuss big oh notation concept in section 1.11.

1.10.2 Time Complexity

The amount of time required by an algorithm to execute is called the time complexity of that algorithm.

For determining the time complexity of particular algorithm following steps are carried out -

1. Identify the basic operation of the algorithm
2. Obtain the frequency count for this basic operation.
3. Consider the order of magnitude of the frequency count and express it in terms of big oh notation.

Example 1.10.2 Write an algorithm to find smallest element in a array of integers and analyze its time complexity

SPPU : May-13, Marks 8

Solution :

Algorithm MinValue(int a[n])

```
{
    min_element=a[0];
    for(i=0;i<n;i++)
    {
        if (a[i]<min_element) then
            min_element=a[i];
    }
    Write(min_element);
}
```

We will first obtain the frequency count for the above code

By neglecting the constant terms and by considering the order of magnitude we can express the frequency count in terms of Omega notation as $O(n)$. Hence the frequency count of above code is $O(n)$.

Statement	Frequency Count
<code>min_element=a[0]</code>	1
<code>for(i=0;i<n;i++)</code>	$n+1$
<code>if (a[i]<min_element) then</code>	n
<code>Write(min_element);</code>	1
Total	$2n+3$

Review Questions

1. What is space complexity of an algorithm? Explain its importance with example.

SPPU : Dec.-10, Marks 4

2. What do you mean by frequency count and its importance in analysis of an algorithm

SPPU : Dec.-09,10, May-12,13, Marks 6

3. What is time complexity? How is time complexity of an algorithm computed ?

SPPU : Dec.-11, Marks 6

4. What is complexity of algorithm ? Explain with an example.

SPPU : Dec.-19, Marks 3

1.11 Asymptotic Notations

SPPU : Dec.-09, 10, 11, 16, 17, 19, May-10, 12, 13, 18, 19, Marks 8

To choose the best algorithm, we need to check efficiency of each algorithm. The efficiency can be measured by computing time complexity of each algorithm. Asymptotic notation is a shorthand way to represent the time complexity.

Using asymptotic notations we can give time complexity as “fastest possible”, “slowest possible” or “average time”.

Various notations such as Ω , Θ and O used are called **asymptotic notations**.

1.11.1 Big oh Notation

The **Big oh** notation is denoted by 'O'. It is a method of representing the **upper bound** of algorithm's running time. Using big oh notation we can give longest amount of time taken by the algorithm to complete.

Definition

Let $f(n)$ and $g(n)$ be two non-negative functions.

Let n_0 and constant c are two integers such that n_0 denotes some value of input and $n > n_0$. Similarly c is some constant such that $c > 0$. We can write

$$f(n) \leq c \cdot g(n)$$

then $f(n)$ is big oh of $g(n)$. It is also denoted as $f(n) \in O(g(n))$. In other words $f(n)$ is less than $g(n)$ if $g(n)$ is multiple of some constant c .

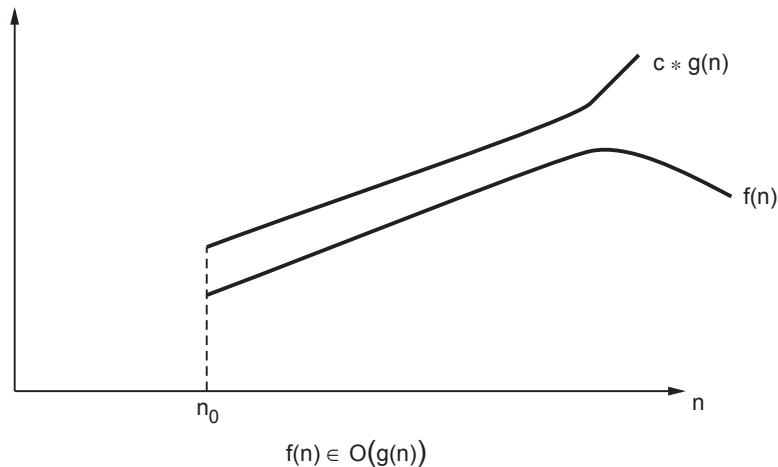


Fig. 1.11.1 Big oh notation

Example : Consider function $f(n) = 2n + 2$ and $g(n) = n^2$. Then we have to find some constant c , so that $f(n) \leq c * g(n)$. As $F(n) = 2n + 2$ and $g(n) = n^2$ then we find c for $n = 1$ then

$$\begin{aligned} f(n) &= 2n + 2 \\ &= 2(1) + 2 \end{aligned}$$

$$\begin{aligned} f(n) &= 4 \\ \text{and } g(n) &= n^2 \\ &= (1)^2 \end{aligned}$$

$$g(n) = 1$$

$$\text{i.e. } f(n) > g(n)$$

If $n = 2$ then,

$$\begin{aligned} f(n) &= 2(2) + 2 \\ &= 6 \end{aligned}$$

$$\begin{aligned} g(n) &= (2)^2 \\ g(n) &= 4 \end{aligned}$$

$$\text{i.e. } f(n) > g(n)$$

If $n = 3$ then,

$$\begin{aligned} f(n) &= 2(3) + 2 \\ &= 8 \end{aligned}$$

$$g(n) = (3)^2$$

$$g(n) = 9$$

i.e. $f(n) < g(n)$ is true.

Hence we can conclude that for $n > 2$, we obtain

$$f(n) < g(n)$$

Thus always **upper bound** of existing time is obtained by big oh notation.

1.11.2 Omega Notation

Omega notation is denoted by ' Ω '. This notation is used to represent the **lower bound** of algorithm's running time. Using omega notation we can denote shortest amount of time taken by algorithm.

Definition

A function $f(n)$ is said to be in $\Omega(g(n))$ if $f(n)$ is bounded below by some positive constant multiple of $g(n)$ such that

$$f(n) \geq c * g(n)$$

For all $n \geq n_0$

It is denoted as $f(n) \in \Omega(g(n))$. Following graph illustrates the curve for Ω notation.

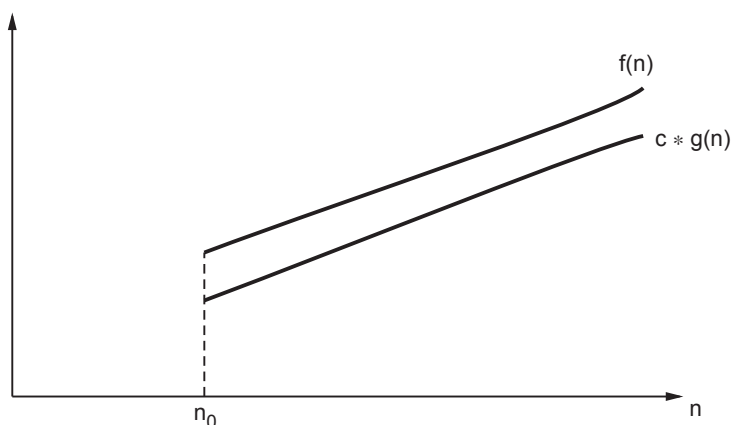


Fig. 1.11.2 Omega notation $f(n) \in \Omega(g(n))$

Example :

Consider $f(n) = 2n^2 + 5$ and $g(n) = 7n$

Then if $n = 0$

$$f(n) = 2(0)^2 + 5$$

$$= 5$$

$$g(n) = 7(0)$$

$$= 0$$

i.e. $f(n) > g(n)$

But if $n = 1$

$$f(n) = 2(1)^2 + 5$$

$$= 7$$

$$g(n) = 7(1)$$

$$7 \text{ i.e. } f(n) = g(n)$$

If $n = 3$ then,

$$f(n) = 2(3)^2 + 5$$

$$= 18 + 5$$

$$= 23$$

$$g(n) = 7(3)$$

$$= 21$$

$$\text{i.e. } f(n) > g(n)$$

Thus for $n > 3$ we get $f(n) > c * g(n)$.

It can be represented as

$$2n^2 + 5 \in \Omega(n)$$

Similarly any

$$n^3 \in \Omega(n)^2$$

1.11.3 Θ Notation

The theta notation is denoted by Θ . By this method the running time is **between upper bound and lower bound**.

Definition

Let $f(n)$ and $g(n)$ be two non negative functions. There are two positive constants namely c_1 and c_2 such that

$$c_1 \leq g(n) \leq c_2 g(n)$$

Then we can say that

$$f(n) \in \Theta(g(n))$$

Example :

If $f(n) = 2n + 8$ and $g(n) = 7n$.

where $n \geq 2$

Similarly $f(n) = 2n + 8$

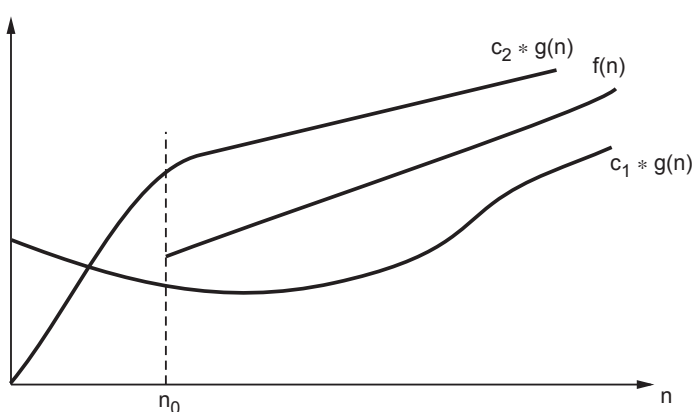


Fig. 1.11.3 Theta notation $f(n) \in \Theta(g(n))$

$$g(n) = 7n$$

i.e. $5n < 2n + 8 < 7n$ For $n \geq 2$

Here $c_1 = 5$ and $c_2 = 7$ with $n_0 = 2$.

The theta notation is more precise with both big oh and omega notation.

Some Examples of Asymptotic Order

1) $\log_2 n$ is $f(n)$ then

$$\log_2 n \in O(n) \quad \because \log_2 n \leq O(n), \text{ the order of growth of } \log_2 n \text{ is slower than } n.$$

$$\log_2 n \in O(n^2) \quad \because \log_2 n \leq O(n^2), \text{ the order of growth of } \log_2 n \text{ is slower than } n^2 \text{ as well.}$$

But

$$\log_2 n \notin \Omega(n) \quad \because \log_2 n \leq \Omega(n) \text{ and if a certain function } F(n) \text{ is belonging to } \Omega(n) \text{ it should satisfy the condition } F(n) \geq c * g(n)$$

Similarly $\log_2 n \notin \Omega(n^2)$ or $\Omega(n^3)$

2) Let $f(n) = n(n-1)/2$

Then

$$n(n-1)/2 \notin O(n) \quad \because f(n) > O(n) \text{ we get } f(n) = n(n-1)/2 = \frac{n^2-1}{2}$$

i.e. maximum order is n^2 which is $> O(n)$.

Hence $F(n) \notin O(n)$

But $n(n-1)/2 \in O(n^2)$

As $f(n) \leq O(n^2)$

and $n(n-1)/2 \in O(n^3)$

Similarly,

$$n(n-1)/2 \in \Omega(n) \quad \because f(n) \geq \Omega(n)$$

$$n(n-1)/2 \in \Omega(n^2) \quad \because f(n) \geq \Omega(n^2)$$

$$n(n-1)/2 \notin \Omega(n^3) \quad \because f(n) < \Omega(n^3)$$

1.11.4 Properties of Order of Growth

1. If $f_1(n)$ is order of $g_1(n)$ and $f_2(n)$ is order of $g_2(n)$, then $f_1(n) + f_2(n) \in O(\max(g_1(n), g_2(n)))$.

2. Polynomials of degree $m \in \Theta(n^m)$.

That means maximum degree is considered from the polynomial.

For example : $a_1n^3 + a_2n^2 + a_3n + c$ has the order of growth $\Theta(n^3)$.

3. $O(1) < O(\log n) < O(n) < O(n^2) < O(2^n)$.

4. Exponential functions a^n have different orders of growth for different values of a .

Key Points

i) $O(g(n))$ is a class of functions $F(n)$ that grows less fast than $g(n)$, that means $F(n)$ possess the time complexity which is always lesser than the time complexities that $g(n)$ have.

ii) $\Theta(g(n))$ is a class of functions $F(n)$ that grows at same rate as $g(n)$.

iii) $\Omega(g(n))$ is a class of functions $F(n)$ that grows faster than or atleast as fast as $g(n)$. That means $F(n)$ is greater than $\Omega(g(n))$.

Example 1.11.1 Analyze time complexity of the following code segments :

```

i) for (i = 1 ; i <= n ; i++)
    for (j = 1 ; j <= m ; j++)
        for (k = 1 ; k <= p ; k++)
            x = x + 1 ;

ii) i = 1
    while (i <= n)
    {
        x++ ;
        i++ ;
    }

iii) int process (int no)
    {
        if (no <= 0)
            return (0) ;
        else
            return (no + process (no - 1));
    }

```

SPPU : Dec.-10, Marks 8

Solution : i) $O(n^3)$

ii) $O(n)$

iii) $O(n)$

1.11.5 How to Choose the Best Algorithm ?

If we have two algorithms that perform same task and the first one has a computing time of $O(n)$ and the second of $O(n^2)$, then we will usually prefer the first one.

The reason for this is that as n increases the time required for the execution of second algorithm will get far more than the time required for the execution of first.

n	$\log_2 n$	$n \log_2 n$	n^2	n^3	2^n
1	0	0	1	1	2
2	1	2	4	8	4
4	2	8	16	64	16
8	3	24	64	512	256
16	4	64	256	4096	65536
32	5	160	1024	32768	2147483648

We will study various values for computing function for the constant values. The graph given below will indicate the rate of growth of common computing time functions.

Notice how the times $O(n)$ and $O(n \log n)$ grow much more slowly than the others. For large data sets algorithms with a complexity greater than $O(n \log n)$ are often impractical. The very slow algorithm will be the one having time complexity 2^n .

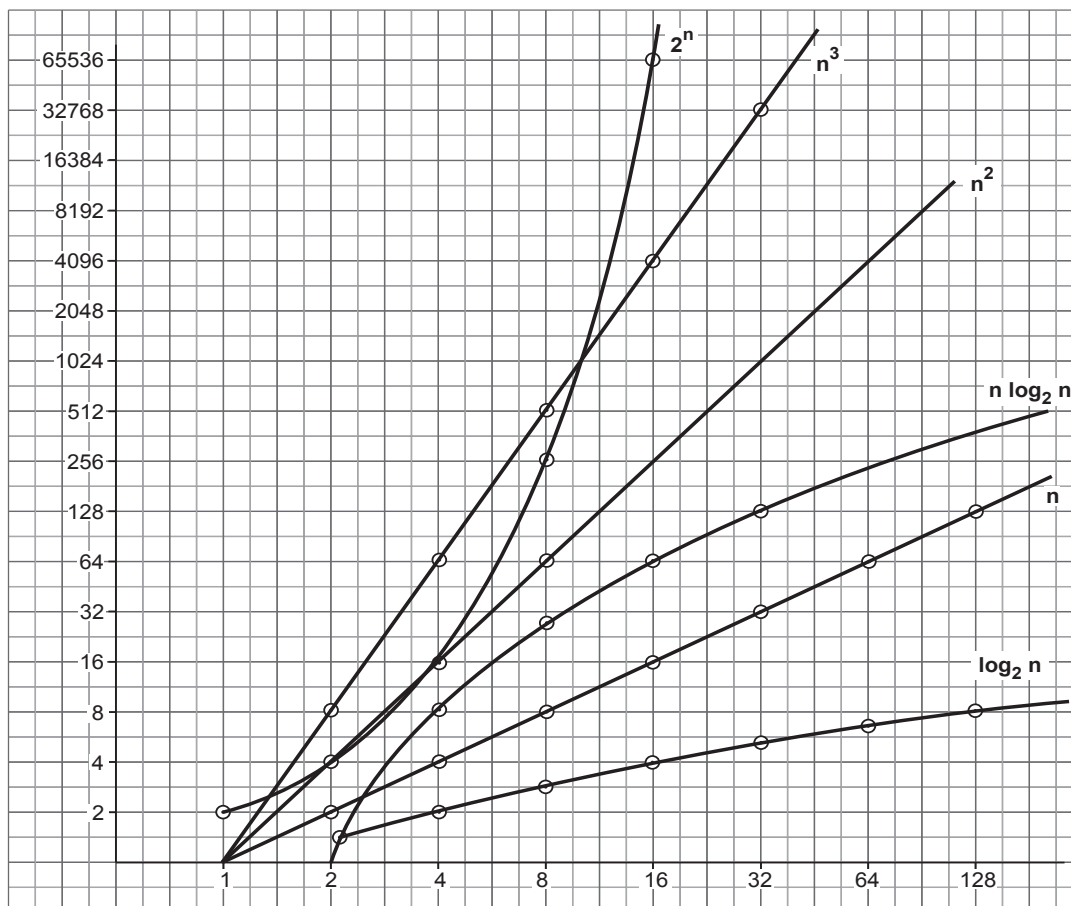


Fig. 1.11.4 Rate of growth of common computing time function

Review Questions

1. Explain different asymptotic notation

SPPU : Dec.-09, May-12, Marks 4, May-13, Marks 6

2. With respect to algorithm analysis, explain the following terms :

i) Omega notation ii) Theta notation iii) Big oh notation

SPPU : May-10, Dec.-11, Marks 6

3. Explain asymptotic notations Big O, Theta and Omega with one example each.

SPPU : Dec.-16, 17, May-18, Marks 6, Dec.-19, Marks 3

4. What is complexity analysis of an algorithm ? Explain the notations used in complexity analysis.

SPPU : May-19, Marks 6

1.12 Analysis of Programming Constructs

SPPU : May-10, 13, 14, Dec.-11, 13, Marks 8

Example 1.12.1 Obtain the frequency count for the following code

Solution :

```
void fun()
{
    int a;
    a=0; .....1
    for(i=0;i<n;i++) .....n+1
    {
        a = a+i; .....n
    }
    printf("%d",a); .....1
}
```

The frequency count of above code is $2n+3$

The for loop in above given fragment of code is executed n times when the condition is true and one more time when the condition becomes false. Hence for the for loop the frequency count is $n+1$. The statement inside the for loop will be executed only when the condition inside the for loop is true. Therefore this statement will be executed for n times. The last printf statement will be executed for once.

Example 1.12.2 Obtain the frequency count for the following code

Solution :

```
void fun(int a[][][],int b[][][])
{
    int c[3][3];
    for(i=0;i<m;i++) .....m+1
    {
        for(j=0;j<n;j++) .....m(n+1)
        {
            c[i][j]=a[i][j]+b[i][j]; .....m.n
        }
    }
}
```

The frequency count $= (m+1) + m(n+1) + mn = 2m + 2mn + 1 = 2m(1+n) + 1$

Example 1.12.3 Obtain the frequency count for the following code

```
for(i=1;i<=n;i++)
{
    for(j=1;j<=n;j++)
    {
        c[i][j]=0;
        for(k=1;k<=n;k++)
            c[i][j]=c[i][j]+a[i][k]*b[k][j];
    }
}
```

SPPU : May-13, Marks 8; Dec.-13, Marks 3

Solution : After counting the frequency count, the constant terms can be neglected and only the order of magnitude is considered. The time complexity is denoted in terms of algorithmic notations. The Big oh notation is a most commonly used algorithmic notation. For the above frequency count all the constant terms are neglected and only the order of magnitude of the polynomial is considered. Hence the time complexity for the above code can be $O(n^3)$. The higher order is the polynomial is always considered.

Statement	Frequency Count
for(i=1;i<=n;i++)	n+1
for(j=1;j<=n;j++)	n.(n+1)
c[i][j]=0;	n.(n)
for(k=1;k<=n;k++)	n.n(n+1)
c[i][j]=c[i][j]+a[i][k]*b[k][j];	n.n.n
Total	$2n^3+3n^2+2n+1$

Example 1.12.4 Obtain the frequency count for the following code $i=1$;

```
do
{
    a++;
    if(i==5)
        break;
    i++;
}while(i<=n)
```

Solution :

Statement	Frequency Count
i=1;	1
a++;	5
if(i==5)	5
break;	1
i++;	5
while(i<=n)	5
Total	22

Example 1.12.5 Obtain the frequency count for the following code

```
m=n/2;
for(i=0;i+m<m;i++)
{
    a++;
    k++;
}
```

Solution :

Statement	Frequency Count
m=n/2	1
for(i=0;i+m<m;i++)	(n/2)+1
a++;	n/2
k++;	n/2
Total	(3n/2)+2

If we consider only the degree of the polynomial for this code then the time complexity in terms of Big-oh notation can be specified as **O(n/2)**.

Example 1.12.6 Find the frequency count (F.C.) of the given code :

Explain each step.

```
double IterPow(double X, int N)
{
    double Result = 1;
    while (N > 0)
    {
        Result = Result *X;
        N--;
    }
    return Result;
}
```

SPPU : May-10, Marks 6

Solution :

Statement	Frequency Count
double Result=1	1
while(N>0)	N+1
Result=Result*X	N
N--	N
return Result	1
Total	3N+3

Example 1.12.7 What is frequency count of a statement ? Analyze time complexity of the following code :

```
i)    for(i = 1; i <= n; i++)
        for(j = 1; j <= m; j++)
            for(k = 1; k <= p; k++)
                sum = sum + i;
ii)   i = n;
        while(i >= 1)
            {i--;
```

SPPU : Dec.-11, Marks 6, May-14, Marks 3

Solution : i)

Statement	Frequency Count
for(i=1;i<=n;i++)	n+1
for(j=1;j<=m;j++)	n(m+1)
for(k=1;k<=p;k++)	n.m.(p+1)
sum=sum+i	n.m.p
Total	$2(n+nm+nmp)+1$

ii)

Statement	Frequency Count
i=n	1
while(i>=1)	n+1
i--	n
Total	$2(n+1)$

Example 1.12.8 Determine the frequency counts for all the statements in the following program segment

```
i=10;
for(i=10;i<=n;i++)
    for(j=1;j<i;j++)
        x=x+1;
```

SPPU : May-13, Marks 6

Solution : Assume $(n-10+2)=m$.
Then the total frequency count is -

The outer loop will execute for m times. The inner loop is dependant upon the outer loop. Hence it will be $(m+1)/2$. Hence overall frequency count is $(1+m+m^2)(m(m+1)/2)$

Statement	Frequency Count
i=10;	1
for(i=10;i<=n;i++)	m (we assume the count of execution is m)
for(j=1;j<i;j++)	$m((m+1)/2)$
x=x+1;	$m(m)$

If we consider only the order of magnitude of this code then overall time complexity in terms of big oh notation is $O(n^2)$.

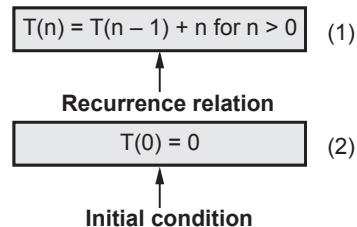
If an algorithm takes time $O(\log n)$ then it is faster than any other algorithm, for larger value of n. Similarly $O(n \log n)$ is better than $O(n^2)$. Various computing time can be $O(1)$, $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^3)$ and $O(2^n)$.

1.13 Recurrence Relation

SPPU : Dec.-18, Marks 2

- **Definition :**

The recurrence equation is an equation that defines a sequences resursively. It is normally in following form :



- The recurrence equation can have infinite number of sequence.
- The **general solution** to the recursive function specifies some formula.
- **For example :** consider a recurrence relation

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

$$T(1) = 1$$

By solving this relation we will get $T(n) = 2^n - 1$ where $n > 1$.

Review Question

1. What is recurrence relation ? Explain with example.

SPPU : Dec.-18, Marks 2

1.14 Solving Recurrence Relation

The recurrence relation can be solved by following methods -

1. Substitution method
2. Master's method.

Let us discuss methods with suitable examples -

1.14.1 Substitution Method

The substitution method is a kind of method in which a guess for the solution is made

There are two types of substitutions -

- Forward substitution
- Backward substitution

Forward Substitution Method - This method makes use of an initial condition in the initial term and value for the next term is generated. This process is continued until some formula is guessed. Thus in this kind of substitution method, we use recurrence equations to generate the few terms.

For example -

Consider a recurrence relation

$$T(n) = T(n - 1) + n$$

With initial condition $T(0) = 0$.

Let,

$$T(n) = T(n - 1) + n \quad \dots (1.14.1)$$

If $n = 1$ then

$$\begin{aligned} T(1) &= T(0) + 1 \\ &= 0 + 1 \end{aligned}$$

\therefore Initial condition

$$\therefore \quad \mathbf{T(1) = 1} \quad \dots (1.14.2)$$

If $n = 2$, then

$$\begin{aligned} T(2) &= T(1) + 2 \\ &= 1 + 2 \end{aligned}$$

\therefore equation (1.14.2)

$$\therefore \quad \mathbf{T(2) = 3} \quad \dots (1.14.3)$$

If $n = 3$ then

$$\begin{aligned} T(3) &= T(2) + 3 \\ &= 3 + 3 \end{aligned}$$

\therefore equation (1.14.4)

$$\therefore \quad \mathbf{T(3) = 6} \quad \dots (1.14.5)$$

By observing above generated equations we can derive a formula

$$T(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

We can also denote $T(n)$ in terms of big oh notation as follows -

$$T(n) = O(n^2)$$

But, in practice, it is difficult to guess the pattern from forward substitution. Hence this method is not very often used.

Backward Substitution : In this method backward values are substituted recursively in order to derive some formula.

For example - Consider, a recurrence relation

$$T(n) = T(n - 1) + n \quad \dots (1.14.6)$$

With initial condition $T(0) = 0$

$$T(n-1) = T(n-1-1) + (n-1) \quad \dots (1.14.7)$$

Putting equation (1.14.7) in equation (1.14.6) we get

$$T(n) = T(n-2) + (n-1) + n \quad \dots (1.14.8)$$

Let

$$T(n-2) = T(n-2-1) + (n-2) \quad \dots (1.14.9)$$

Putting equation (1.14.9) in equation (1.14.8) we get.

$$\begin{aligned} T(n) &= T(n-3) + (n-2) + (n-1) + n \\ &\vdots \\ &= T(n-k) + (n-k+1) + (n-k+2) + \dots + n \end{aligned}$$

if $k = n$ then

$$T(n) = T(0) + 1 + 2 + \dots + n$$

$$T(n) = 0 + 1 + 2 + \dots + n \quad \because T(0) = 0$$

$$T(n) = \frac{n(n+1)}{2} = \frac{n^2}{2} + \frac{n}{2}$$

Again we can denote $T(n)$ in terms of big oh notation as

$$T(n) \in O(n^2)$$

Example 1.14.1 Solve the following recurrence relation $T(n) = T(n-1) + 1$ with $T(0) = 0$ as initial condition. Also find big oh notation.

Solution : Let,

$$T(n) = T(n-1) + 1$$

By backward substitution,

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

$$\begin{aligned} \therefore T(n) &= T(n-1) + 1 \\ &= (T(n-2) + 1) + 1 \end{aligned}$$

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

$$\begin{aligned} \text{Again } T(n-2) &= T(n-2-1) + 1 \\ &= T(n-3) + 1 \end{aligned}$$

$$\begin{aligned} \therefore T(n) &= T(n-2) + 2 \\ &= (T(n-3) + 1) + 2 \end{aligned}$$

$$T(n) = T(n - 3) + 3$$

$$\vdots$$

$$T(n) = T(n - k) + k \quad \dots (1)$$

If $k = n$ then equation (1) becomes

$$T(n) = T(0) + n$$

$$= 0 + n$$

$$\because T(0) = 0$$

$$T(n) = n$$

\therefore We can denote $T(n)$ in terms of big oh notation as

$$T(n) = O(n)$$

Example 1.14.2 Solve the following recurrence :

$$T(1) = 1$$

$$T(n) = 4T(n/3) + n^2 \text{ for } n \geq 2$$

Solution :

$$\begin{aligned} T(n) &= 4T\left(\frac{n}{3}\right) + n^2 & T\left(\frac{n}{3}\right) &= 4T\left(\frac{n}{9}\right) + \left(\frac{n}{3}\right)^2 \\ &= 4\left[4 \cdot T\left(\frac{n}{3^2}\right) + \frac{n^2}{3^2}\right] + n^2 \\ &= 4^2 T\left(\frac{n}{3^2}\right) + 4 \cdot \frac{n^2}{3^2} + n^2 \\ &= 4^2 T\left(\frac{n}{3^2}\right) + \frac{13n^2}{3^2} \\ &= 16 \cdot T\left(\frac{n}{9}\right) + \frac{13n^2}{9} & \because T\left(\frac{n}{9}\right) &= 4T\left(\frac{n}{27}\right) + \left(\frac{n}{9}\right)^2 \\ &= 16\left[4 \cdot T\left(\frac{n}{27}\right) + \frac{n^2}{81}\right] + \frac{13n^2}{9} & \because T\left(\frac{n}{9}\right) &= 4T\left(\frac{n}{27}\right) + \frac{n^2}{81} \\ &= 64T\left(\frac{n}{27}\right) + 16 \cdot \frac{n^2}{81} + \frac{13n^2}{9} \\ &= 64T\left(\frac{n}{27}\right) + \frac{133n^2}{81} \end{aligned}$$

$$\begin{aligned}
&= 4^3 T\left(\frac{n}{3^3}\right) + 133 \cdot \left(\frac{n}{3^2}\right)^2 \\
&\vdots \\
&= 4^k T\left(\frac{n}{3^k}\right) + C \left(\frac{n}{3^{k-1}}\right)^2 \quad \because 133 + C_1 = C
\end{aligned}$$

Now if we assume $\frac{n}{3^k} = 1$ then $n = 3^k$ and $k = \log_3 n$

$$= 4^k T(1) + C \left(\frac{n}{3^{k-1}}\right)^2 \quad \because \frac{n}{3^k} = 1$$

Purposely we kept constant term in terms of n .

$$= 4^k \cdot 1 + \frac{C}{(3^{k-1})^2} \cdot n^2 \quad \because T(1) = 1$$

$$= 4^k + C_1 \cdot n^2 \quad \because C_1 = \frac{C}{(3^{k-1})^2}$$

$$= 4 \log_3 n + C_1 n^2$$

$$= n \log_3 4 + C_1 n^2 \quad \because a \log_b n = n \log_b a$$

$$= n^{1.26} + C \cdot n^2$$

$$T(n) \approx \Theta(n^2)$$

Example 1.14.3 Solve the following recurrence relations -

$$i) T(n) = 2T\left(\frac{n}{2}\right) + C \quad T(1) = 1 \quad ii) T(n) = T\left(\frac{n}{3}\right) + C \quad T(1) = 1$$

Solution : i) Let

$$\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + C \\
&= 2\left(2T\left(\frac{n}{4}\right) + C\right) + C \\
&= 4T\left(\frac{n}{4}\right) + 3C \\
&= 4\left(2T\left(\frac{n}{8}\right) + C\right) + 3C
\end{aligned}$$

$$\begin{aligned}
&= 8T\left(\frac{n}{8}\right) + 7C \\
&= 2^3T\left(\frac{n}{2^3}\right) + (2^3 - 1)C \\
&\vdots \\
T(n) &= 2^kT\left(\frac{n}{2^k}\right) + (2^k - 1)C
\end{aligned}$$

If we $2^k = n$ then

$$\begin{aligned}
T(n) &= nT\left(\frac{n}{n}\right) + (n - 1)C \\
&= nT(1) + (n - 1)C
\end{aligned}$$

$$T(n) = n + (n - 1)C \quad \because T(1) = 1$$

ii) Let,

$$\begin{aligned}
T(n) &= T\left(\frac{n}{3}\right) + C \\
&= \left(T\left(\frac{n}{9}\right) + C\right) + C \\
&= T\left(\frac{n}{9}\right) + 2C \\
&= \left[T\left(\frac{n}{27}\right) + C\right] + 2C \\
&= T\left(\frac{n}{27}\right) + 3C \\
&\vdots \\
&= T\left(\frac{n}{3^k}\right) + kC
\end{aligned}$$

If we put $3^k = n$ then

$$\begin{aligned}
&= T\left(\frac{n}{n}\right) + \log_3 n \cdot C \\
&= T(1) + \log_3 n \cdot C
\end{aligned}$$

$$T(n) = C \cdot \log_3 n + 1 \quad \because T(1) = 1$$

Example 1.14.4 Solve the recurrence relation by iteration :

$$T(n) = T(n-1) + n^4.$$

Solution : Let

$$T(n) = T(n-1) + n^4$$

By backward substitution method,

$$\begin{aligned} T(n) &= [T(n-2) + (n-1)^4] + n^4 & \because T(n-1) &= T(n-2) + (n-1)^4 \\ &= T(n-2) + (n-1)^4 + n^4 \\ &= [T(n-3) + (n-2)^4] + (n-1)^4 + n^4 \\ &= T(n-3) + (n-2)^4 + (n-1)^4 + n^4 \\ &= [T(n-4) + (n-3)^4] + (n-2)^4 + (n-1)^4 + n^4 \\ &= T(n-4) + (n-3)^4 + (n-2)^4 + (n-1)^4 + n^4 \\ &\vdots \\ &= T(n-k) + (n-k+1)^4 + (n-k+2)^4 + (n-k+3)^4 + \dots + n^4 \end{aligned}$$

If $k = n$ then

$$\begin{aligned} &= T(n-n) + (n-n+1)^4 + (n-n+2)^4 + (n-n+3)^4 + \dots + n^4 \\ &= T(0) + 1^4 + 2^4 + 3^4 + \dots + n^4 \\ &= T(0) + \sum_{i=1}^n n^4 \\ &= T(0) + n(n^4) & \because \sum_{i=1}^n n^4 = n^4 \cdot \sum_{i=1}^n 1 = n^4 \cdot n \\ &= 0 + n^5 & \because \text{Assume } T(0) = 0 \text{ as initial condition} \end{aligned}$$

$$\therefore T(n) \approx \theta(n^5)$$

Example 1.14.5 Consider the following code :

```

int sum (int a [ ], n)
{
    count = count + 1 ;
    if (n <= 0)
    {
        count = count + 1 ;
        return 0 ;
    }
    else
    {
        count = count + 1
        return sum (a, n - 1)+a[n] ;
    }
}

```

Write the recursive formula for the above code and solve this recurrence relation.

Solution : In the above code the following statement gets executed at least twice

count = count + 1

In the else part, there is a recursive call to the function **sum**, by changing the value of n by $n - 1$, each time.

Hence the recursive formula for above code will be

$$T(n) = \underbrace{T(n-1)}_{\substack{\text{For recursive} \\ \text{call to function} \\ \text{sum in else} \\ \text{part.}}} + \underbrace{2}_{\substack{\text{For count} \\ \text{statement}}}$$

$$T(0) = 2$$

We can solve this recurrence relation using backward substitution. It will be

$$\begin{aligned}
 &= [T(n-2) + 2] + 2 \\
 \therefore &= T(n-2) + 2(2) \\
 &= [T(n-3) + 2] + 2(2) \\
 \therefore T(n-2) &= T(n-3) + 3 \cdot (2) \\
 &\vdots \\
 T(n) &= T(n-n) + n \cdot (2)
 \end{aligned}$$

$$= T(0) + 2n$$

$$= 2 + 2 \cdot n$$

$$\therefore T(0) = 2$$

$$T(n) = 2(n+1)$$

We can denote the time complexity in the form of big oh notation as **O(n)**.

1.14.2 Master's Method

We can solve recurrence relation using a formula denoted by Master's method.

$$T(n) = aT(n/b) + F(n) \quad \text{where } n \geq d \text{ and } d \text{ is some constant.}$$

Then the Master theorem can be stated for efficiency analysis as -

If $F(n)$ is $\Theta(n^d)$ where $d \geq 0$ in the recurrence relation then,

$$1. \quad T(n) = \Theta(n^d) \quad \text{if } a < b^d$$

$$2. \quad T(n) = \Theta(n^d \log n) \quad \text{if } a = b$$

$$3. \quad T(n) = \Theta(n^{\log_b a}) \quad \text{if } a > b^d$$

Let us understand the Master theorem with some examples :

Example 1.14.6 Solve the following recurrence relation $T(n) = 4T(n/2) + n$

Solution : We will map this equation with

$$T(n) = aT(n/b) + f(n)$$

Now $f(n)$ is n i.e. n^1 . Hence $d = 1$.

$$a = 4 \text{ and } b = 2 \text{ and}$$

$$a > b^d \text{ i.e. } 4 > 2^1$$

$$\therefore T(n) = \Theta(n^{\log_b a})$$

$$= \Theta(n^{\log_2 4})$$

$$= \Theta(n^2)$$

$$\therefore \log_2 4 = 2$$

Hence time complexity is $\Theta(n^2)$.

For quick and easy calculations of logarithmic values to base 2 following table can be memorized.

m	k
1	0
2	1
4	2
8	3
16	4
32	5
64	6
128	7
256	8
512	9
1024	10

Another variation of Master theorem is

For $T(n) = aT(n/b) + f(n)$ if $n \geq d$

1. If $f(n)$ is $O(n^{\log_b a - \epsilon})$, then

$$T(n) = \Theta(n^{\log_b a})$$
2. If $f(n)$ is $\Theta(n^{\log_b a} \log^k n)$, then

$$T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$
3. If $f(n)$ is $\Omega(n^{\log_b a + \epsilon})$, then

$$T(n) \text{ is } \Theta(f(n))$$

Example 1.14.7 Solve the following recurrence relation. $T(n) = 2T(n/2) + n \log n$

Solution :

Here $f(n) = n \log n$

$a = 2, b = 2$

$\log_2 2 = 1$

According to case 2 given in above Master theorem

$$f(n) = \Theta(n^{\log_2 2} \log^1 n) \quad \text{i.e.} \quad k = 1$$

Then $T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$

$$= \Theta(n^{\log_2 2} \log^2 n)$$

$$= \Theta(n^1 \log^2 n)$$

$$\therefore T(n) = \Theta(n \log^2 n)$$

Example 1.14.8 Solve the following recurrence relation $T(n) = 8T(n/2) + n^2$

Solution :

Here $f(n) = n^2$

$$a = 8 \text{ and } b = 2$$

$$\therefore \log_2 8 = 3$$

Then according to case 1 of above given Master theorem

$$f(n) = O(n^{\log_b a - \epsilon})$$

$$= O(n^{\log_2 8 - \epsilon})$$

$$= O(n^{3-\epsilon}). \text{ If we put } \epsilon = 1 \text{ then } O(n^3 - 1) = O(n^2) = f(n)$$

Then $T(n) = \Theta(n^{\log_b a})$

$$\therefore T(n) = \Theta(n^{\log_2 8})$$

$$T(n) = \Theta(n^3)$$

Example 1.14.9 Solve the following recurrence relation $T(n) = 9T(n/3) + n^3$

Solution :

Here $a = 9, \quad b = 3 \quad \text{and} \quad f(n) = n^3$

And $\log_3 9 = 2$

According to case 3 in above Master theorem

As $f(n)$ is $= \Omega(n^{\log_3 9 + \epsilon})$

$$\text{i.e. } \Omega(n^{2+\epsilon}) \text{ and we have } f(n) = n^3.$$

Then to have $f(n) = \Omega(n)$. We must put $\epsilon = 1$.

Then $T(n) = \Theta(f(n))$

Then $T(n) = \Theta(f(n))$

$$T(n) = \Theta(n^3)$$

Example 1.14.10 Solve the following recurrence relation. $T(n) = T(n/2) + 1$

Solution : Here $a = 1$ and $b = 2$.

and $\log_2 1 = 0$

Now we will analyze $f(n)$ which is $= 1$.

We assume $f(n) = 2^k$

when $k = 0$ then $f(n) = 2^0 = 1$.

That means according to case 2 of above given Master theorem.

$$f(n) = \Theta(n^{\log_b a} \log^k n)$$

$$= \Theta(n^{\log_2 1} \log^0 n)$$

$$= \Theta(n^0 \cdot 1)$$

$$= \Theta(1)$$

$$\because n^0 = 1$$

$$\therefore \text{ We get } T(n) = \Theta(n^{\log_b a} \log^{k+1} n)$$

$$= \Theta(n^{\log_2 1} \log^{0+1} n)$$

$$= \Theta(n^0 \cdot \log^1 n)$$

$$T(n) = \Theta(\log n)$$

$$\because n^0 = 1$$

Example 1.14.11 Find the complexity of the following recurrence relation. $T(n) = 9T(n/3) + n$

Solution : Let $T(n) = 9T(n/3) + n$

$$\begin{array}{ccc} \downarrow & \downarrow & \downarrow \\ a & b & f(n) \end{array}$$

\therefore We get $a = 9$, $b = 3$ and $f(n) = n$.

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

Now $f(n) = n$

i.e. $T(n) = f(n) = \Theta(n^{\log_b a - \epsilon}) = \Theta(n^{2 - \epsilon})$

When $\epsilon = 1$. That means case 1 is applicable. According to case 1, the time complexity of such equations is

$$T(n) = \Theta(n^{\log_b a})$$

$$= \Theta(n^{\log_3 9})$$

$$T(n) = \Theta(n^2)$$

Hence the complexity of given recurrence relation is $\Theta(n^2)$.

Example 1.14.12 Solve recurrence relation $T(n) = k \cdot T(n/k) + n^2$ when $T(1) = 1$, and k is any constant.

Solution : Let,

$$T(n) = k \cdot T\left(\frac{n}{k}\right) + n^2 \text{ be a recurrence relation. Let us use substitution}$$

method

If we assume $k = 2$ then the equation becomes

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2 \qquad T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + \left(\frac{n}{2}\right)^2$$

$$= 2 \left[2 \cdot T\left(\frac{n}{4}\right) + \frac{n^2}{4} \right] + n^2$$

$$= 4T\left(\frac{n}{4}\right) + \frac{n^2}{2} + n^2$$

$$= 4T\left(\frac{n}{4}\right) + \frac{3n^2}{2}$$

$$T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + \left(\frac{n}{4}\right)^2$$

$$= 4 \left[2 \cdot T\left(\frac{n}{8}\right) + \frac{n^2}{16} \right] + \frac{3n^2}{2}$$

$$= 8T\left(\frac{n}{8}\right) + \frac{n^2}{4} + \frac{3n^2}{2}$$

$$= 8T\left(\frac{n}{8}\right) + \frac{7n^2}{4}$$

$$= 2^k T\left(\frac{n}{2^k}\right) + \frac{2^k - 1}{2^{k-1}} n^2$$

$$= 2^k T\left(\frac{n}{2^k}\right) + Cn^2$$

$$\therefore \frac{2^k - 1}{2^{k-1}} = C = \text{Constant}$$

If we put $\frac{n}{2^k} = 1$ then $2^k = n$ and $k = \log_2 n$

$$= nT(1) + Cn^2$$

$$= n + Cn^2$$

$$\therefore T(1) = 1$$

$$T(n) = \Theta(n)^2$$

Alternate Method

We can solve the given recurrence relation using Master's theorem also.

$$\begin{array}{ccccc}
 T(n) & = & k \cdot T\left(\frac{n}{k}\right) & + & n^2 \\
 \downarrow & & \downarrow & & \downarrow \\
 & & a & & b & & f(n)
 \end{array}$$

By Master's theorem $n^{\log_b a} = n^{\log_k k} = n^1$

For $T(n) = f(n)$ we need

$$T(n) = n^{\log_k k + \varepsilon}$$

Applying case 3

$$= n^{1+1}$$

when $\varepsilon = 1$

$$= f(n) \text{ i.e. } n^2$$

$$\therefore T(n) = \Theta(f(n))$$

$$T(n) = \Theta(n^2)$$

1.15 Best, Worst and Average Case Analysis

If an algorithm takes minimum amount of time to run to completion for a specific set of input then it is called **best case** time complexity.

For example : While searching a particular element by using sequential search we get the desired element at first place itself then it is called best case time complexity.

If an algorithm takes maximum amount of time to run to completion for a specific set of input then it is called **worst case** time complexity.

For example : While searching an element by using linear searching method if desired element is placed at the end of the list then we get worst time complexity.

The time complexity that we get for certain set of inputs is as a average same. Then for corresponding input such a time complexity is called **average case** time complexity.

Consider the following algorithm

```
Algorithm Seq_search(X[0 ... n - 1 ],key)
// Problem Description: This algorithm is for searching the
//key element from an array X[0...n - 1] sequentially.
//Input: An array X[0...n - 1] and search key
//Output: Returns the index of X where key value is present
for i ← 0 to n - 1 do
  if(X[i]=key)then
    return i
```

Best case time complexity

Best case time complexity is a time complexity when an algorithm runs for short time. In above searching algorithm the element **key** is searched from the list of **n** elements. If the **key** element is present at first location in the list(X[0...n-1]) then algorithm runs for a very short time and thereby we will get the best case time complexity. We can denote the best case time complexity as

$$C_{\text{best}} = 1$$

Worst case time complexity

Worst case time complexity is a time complexity when algorithm runs for a longest time. In above searching algorithm the element **key** is searched from the list of **n** elements. If the **key** element is present at n^{th} location then clearly the algorithm will run for longest time and thereby we will get the worst case time complexity. We can denote the worst case time complexity as

$$C_{\text{worst}} = n$$

The algorithm guarantees that for any instance of input which is of size n , the running time will not exceed $C_{\text{worst}}(n)$. Hence the worst case time complexity gives important information about the efficiency of algorithm.

Average case time complexity

This type of complexity gives information about the behaviour of an algorithm on specific or random input. Let us understand some terminologies that are required for computing average case time complexity.

Let the algorithm is for sequential search and

P be a probability of getting successful search.

n is the total number of elements in the list.

The first match of the element will occur at i^{th} location. Hence probability of occurring first match is P/n for every i^{th} element.

The probability of getting unsuccessful search is $(1 - P)$.

Now, we can find average case time complexity $C_{avg}(n)$ as -

$$C_{avg}(n) = \text{Probability of successful search (for elements 1 to n in the list)} \\ + \text{Probability of unsuccessful search}$$

$$C_{avg}(n) = \left[1 \cdot \frac{P}{n} + 2 \cdot \frac{P}{n} + \dots + i \cdot \frac{P}{n} \right] + n \cdot (1 - P)$$

$$= \frac{P}{n} [1 + 2 + \dots + i \dots n] + n(1 - P)$$

$$= \frac{P}{n} \frac{n(n+1)}{2} + n(1 - P)$$

$$C_{avg}(n) = \frac{P(n+1)}{2} + n(1 - P)$$

There may be n elements at which chances of 'not getting element' are possible. Hence $n \cdot (1 - P)$

Thus we can obtain the general formula for computing average case time complexity.

Suppose if $P = 0$ that means there is no successful search i.e. we have scanned the entire list of n elements and still we do not found the desired element in the list then in such a situation,

$$C_{avg}(n) = 0(n+1)/2 + n(1 - 0)$$

$$C_{avg}(n) = n$$

Thus the average case running time complexity becomes equal to n.

Suppose if $P = 1$ i.e. we get a successful search then

$$C_{avg}(n) = 1(n+1)/2 + n(1 - 1)$$

$$C_{avg}(n) = (n+1)/2$$

That means the algorithm scans about half of the elements from the list.

For calculating average case time complexity we have to consider probability of getting success of the operation. And any operation in the algorithm is heavily dependent on input elements. Thus computing average case time complexity is difficult than computing worst case and best case time complexities.

1.16 Introduction to Algorithm Design Strategies

SPPU : May-17, 18, Dec.-17, 19, Marks 6

Algorithm design strategy is a general approach by which many problems can be solved algorithmically. These problems may belong to different areas of computing. Algorithmic strategies are also called as **algorithmic techniques** or **algorithmic paradigm**.

- Various algorithm techniques are-
 - **Brute Force** : This is a straightforward technique with naive approach.
 - **Divide-and-Conquer** : The problem is divided into smaller instances.
 - **Greedy Technique** : To solve the problem locally optimal decisions are made.
 - **Backtracking** : In this method, we start with one possible move out from many moves out and if the solution is not possible through the selected move then we backtrack for another move.

1.16.1 Divide and Conquer

- In divide and conquer method, a given problem is,
 - 1) Divided into smaller sub problems.
 - 2) These sub problems are solved independently.
 - 3) If necessary, the solutions of the sub problems are combined to get a solution to the original problem.
- If the sub problems are large enough, then divide and conquer is reapplied.

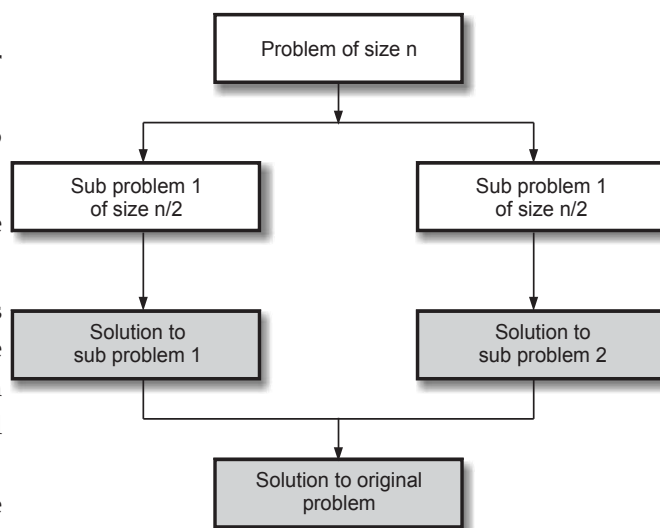


Fig. 1.16.1 Divide and conquer technique

The divide and conquer technique is as shown in Fig. 1.16.1.

The generated sub problems are usually of same type as the original problem. Hence sometimes recursive algorithms are used in divide and conquer strategy.

1.16.1.1 Merge Sort

The merge sort is a sorting algorithm that uses the divide and conquer strategy. In this method division is dynamically carried out.

Merge sort on an input array with n elements consists of three steps:

Divide : partition array into two sub lists $s1$ and $s2$ with $n/2$ elements each.

Conquer : Then sort sub list $s1$ and sub list $s2$.

Combine : merge $s1$ and $s2$ into a unique sorted group.

Consider the elements as

70, 20, 30, 40, 10, 50, 60

Now we will split this list into two sublists.

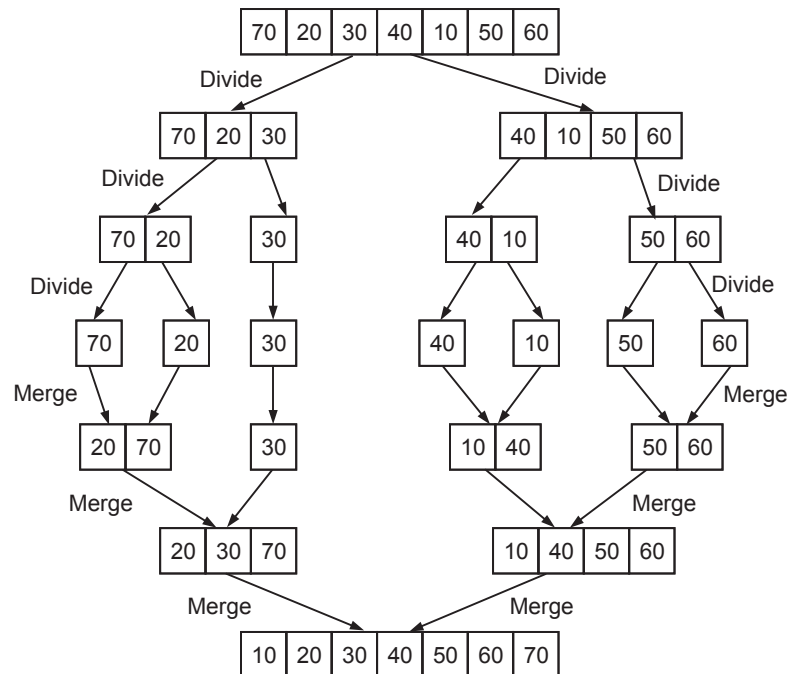


Fig. 1.16.2

Pseudo Code

```

Algorithm MergeSort(int A[0...n-1],low,high)
//Problem Description: This algorithm is for sorting the
//elements using merge sort
//Input: Array A of unsorted elements, low as beginning
//pointer of array A and high as end pointer of array A
//Output: Sorted array A[0...n-1]
    if(low < high)then
    {
        mid ← (low+high)/2          //split the list at mid
        MergeSort(A,low,mid)        //first sublist
        MergeSort(A,mid+1,high)     //second sublist
        Combine(A,low,mid,high)     //merging of two sublists
    }
Algorithm Combine(A[0...n-1],low, mid, high)
{
    k ← low; //k as index for array temp
    i ← low; //i as index for left sublist of array A

```

```

    j ← mid+1 //j as index for right sublist of array A
    while(i <= mid and j <= high)do
    {
        if(A[i]<=A[j])then
//if smaller element is present in left sublist
        {
            //copy that smaller element to temp array
            temp[k] ← A[i]
            i ← i+1
            k ← k+1
        }
        else //smaller element is present in right sublist
        {
            //copy that smaller element to temp array
            temp[k] ← A[j]
            j ← j+1
            k ← k+1
        }
    }
    //copy remaining elements of left sublist to temp
    while(i<=mid)do
    {
        temp[k] ← A[i]
        i ← i+1
        k ← k+1
    }
    //copy remaining elements of right sublist to temp
    while(j<=high)do
    {
        temp[k] ← A[j]
        j ← j+1
        k ← k+1
    }

```

Logic Explanation

To understand above algorithm consider a list of elements as

70	20	30	40	10	50	60
0	1	2	3	4	5	6
↑			↑			↑
low			mid			high

Then we will first make two sublists as

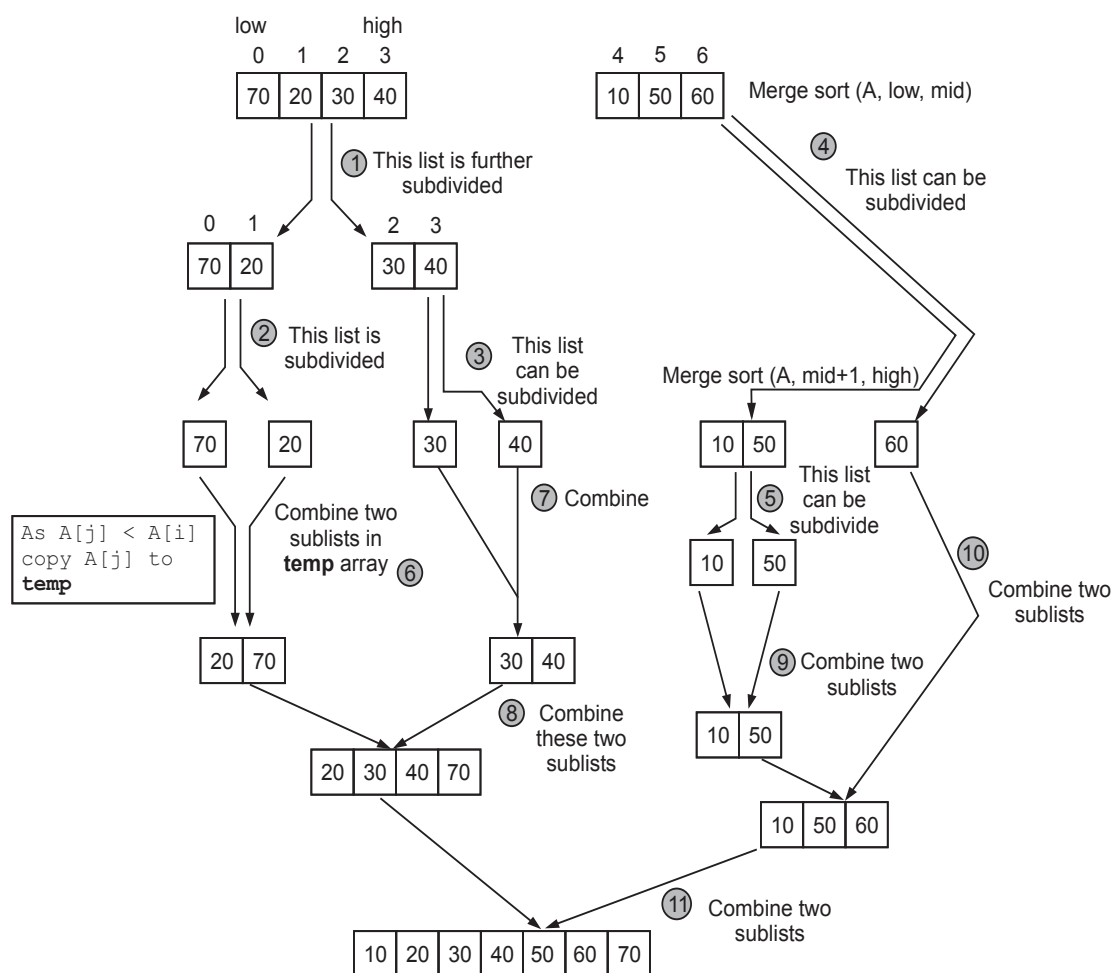
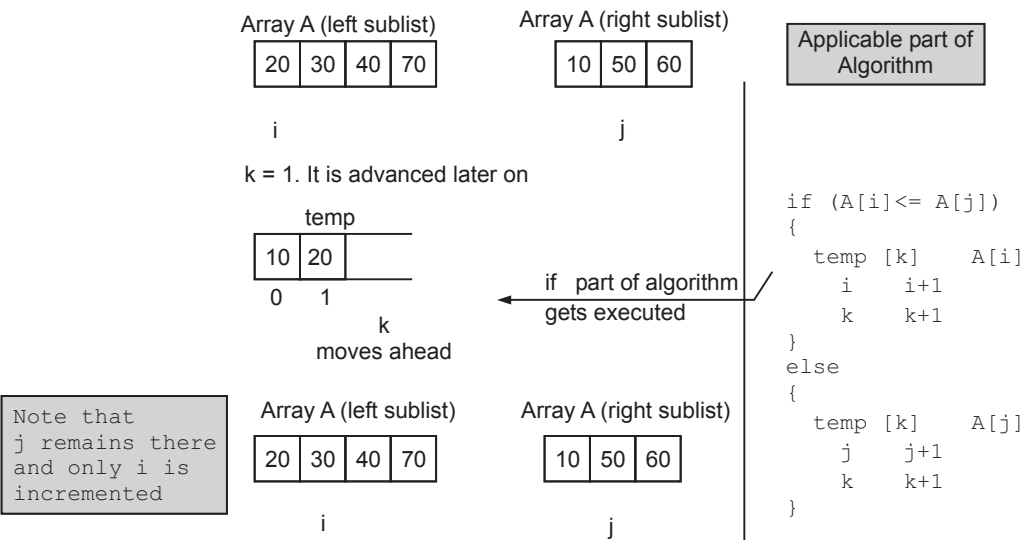
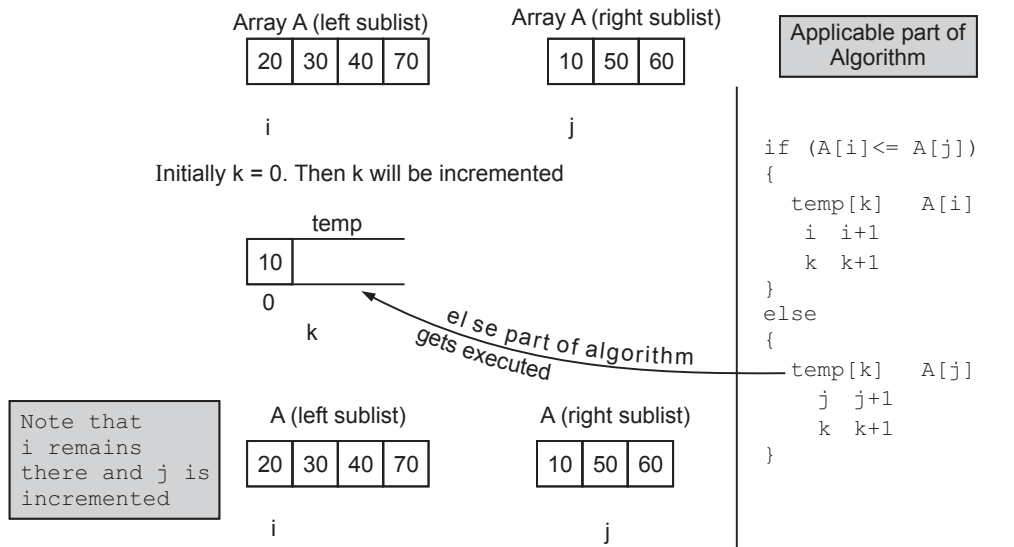
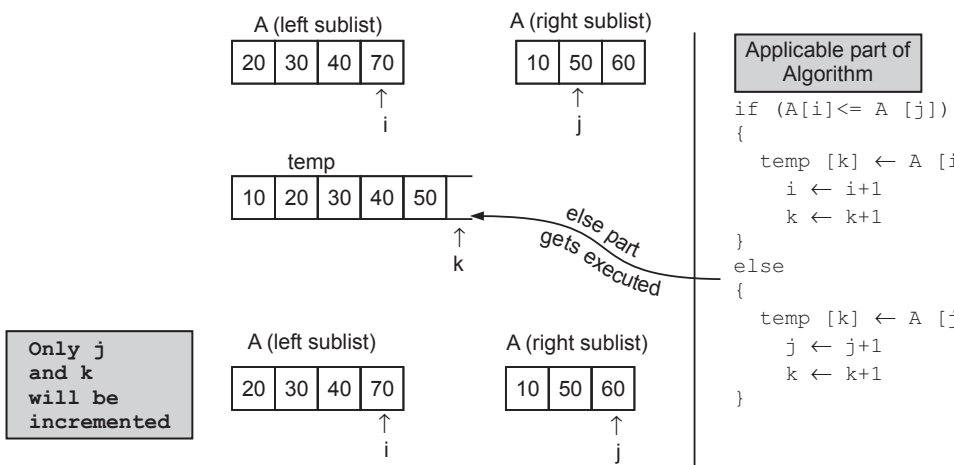
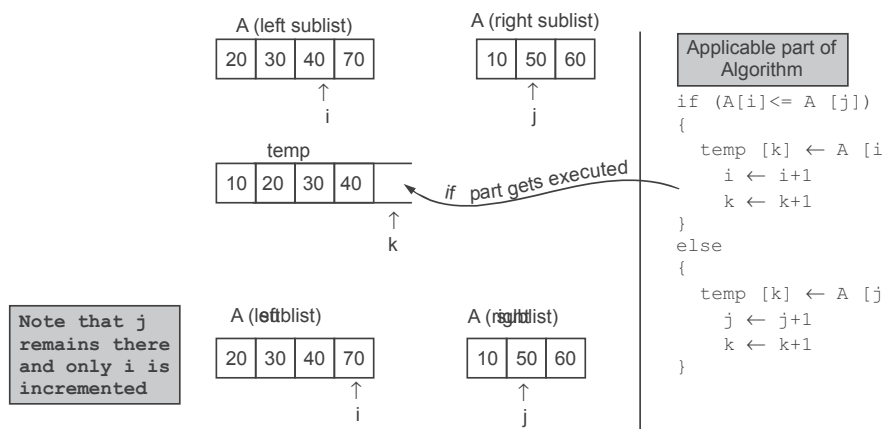
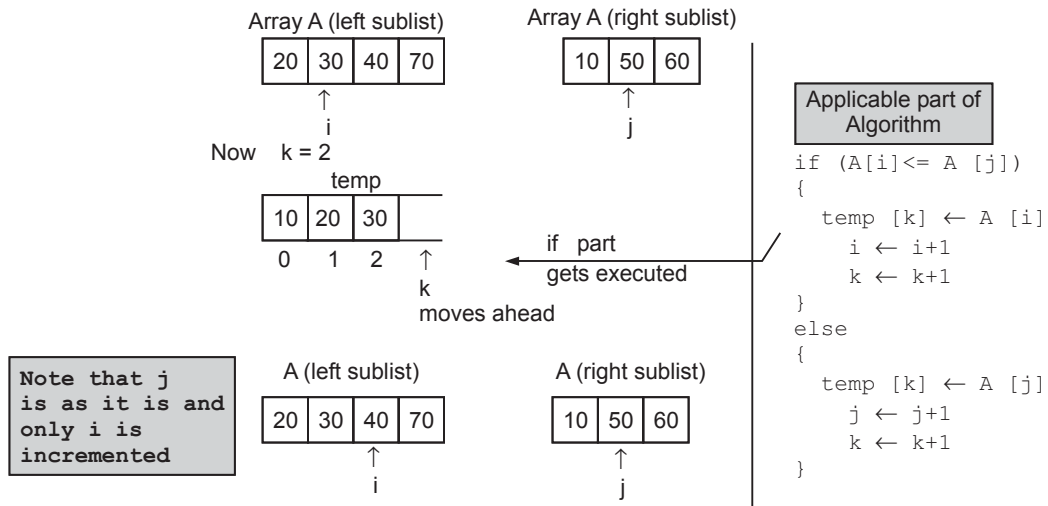


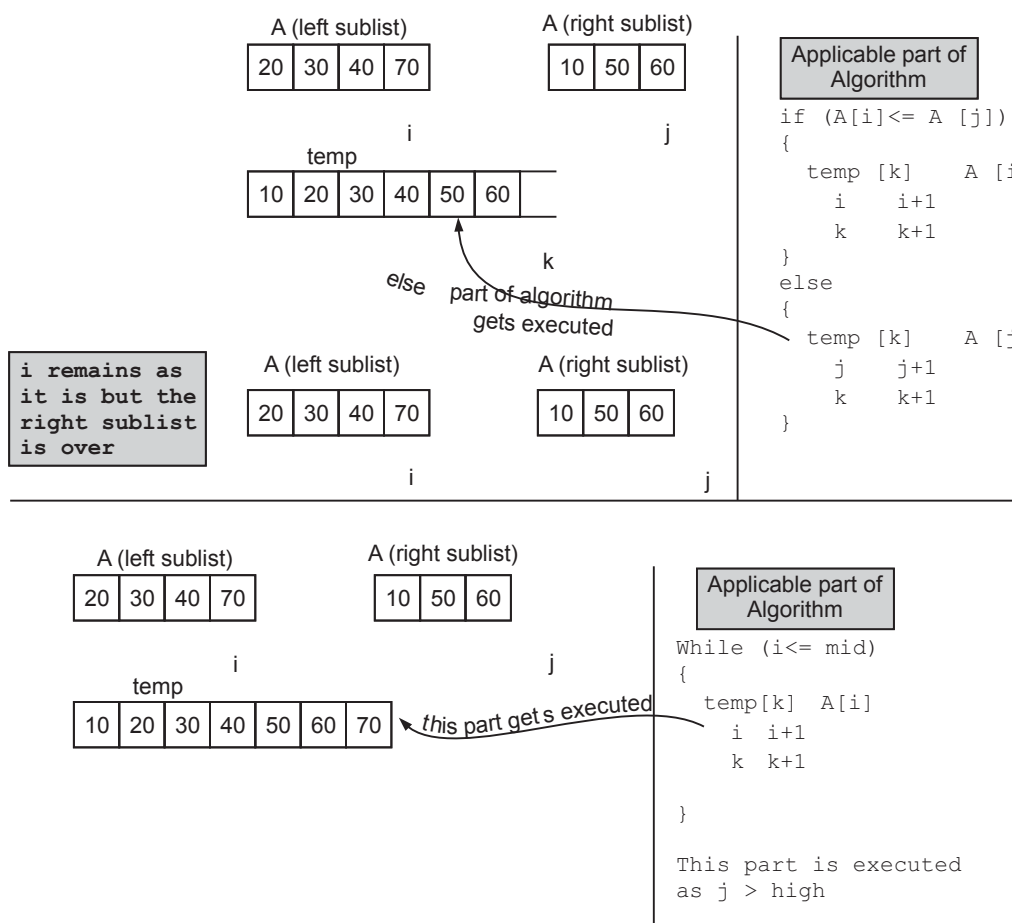
Fig. 1.16.3

Let us see the **combine** operation more closely with the help of some example.

Consider that at some instance we have got two sublists 20, 30, 40, 70 and 10, 50, 60, then







Finally we will copy all the elements of array **temp** to array **A**. Thus array **A** contains sorted list.

A						
10	20	30	40	50	60	70

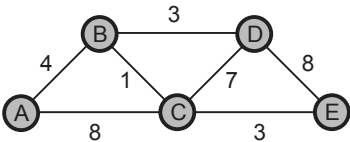
1.16.2 Greedy Strategy

- This method is popular for obtaining the **optimized solutions**.
- In Greedy technique, the solution is constructed through a sequence of steps, each expanding a partially constructed solution obtained so far, until a complete solution to the problem is reached.
- In Greedy method following activities are performed.
 1. First we select some solution from input domain.
 2. Then we check whether the solution is feasible or not.

- 3. From the set of feasible solutions, particular solution that satisfies or nearly satisfies the objective of the function. Such a solution is called optimal solution.
- 4. As Greedy method works in stages. At each stage only one input is considered at each time. Based on this input it is decided whether particular input gives the optimal solution or not.

1.16.2.1 Example of Greedy Method

- Dijkstra's Algorithm is a popular algorithm for finding shortest path using Greedy method. This algorithm is called **single source shortest path** algorithm.
 - In this algorithm, for a given vertex called source the shortest path to all other vertices is obtained.
 - In this algorithm the main focus is not to find only one single path but to find the shortest paths from any vertex to all other remaining vertices.
 - This algorithm applicable to graphs with non-negative weights only.
- Consider a weighted connected graph as given below.



Now we will consider each vertex as a source and will find the shortest distance from this vertex to every other remaining vertex. Let us start with vertex A.

Source vertex	Distance with other vertices	Path shown in graph
A	A-B, path = 4 A-C, path = 8 A-D, path = ∞ A-E, path = ∞	
B	B-C, path = 4 + 1 B-D, path = 4 + 3 B-E, path = ∞	
C	C-D, path = 5 + 7 = 12 C-E, path = 5 + 3 = 8	
D	D-E, path = 7 + 8 = 15	

But we have one shortest distance obtained from A to E and that is A - B - C - E with path length = $4 + 1 + 3 = 8$. Similarly other shortest paths can be obtained by choosing appropriate source and destination.

Pseudo Code

Algorithm Dijkstra(int cost[1...n,1...n],int source,int dist[])

```

for i  $\leftarrow$  0 to tot_nodes
{
    dist[i]  $\leftarrow$  cost[source,i]//initially put the
    s[i]  $\leftarrow$  0 //distance from source vertex to i
    //i is varied for each vertex
    path[i]  $\leftarrow$  source//all the sources are put in path
}

s[source]  $\leftarrow$  1 Start from each source node
for(i  $\leftarrow$  1 to tot_nodes)
{
    min_dist  $\leftarrow$  infinity;
    v1  $\leftarrow$  -1//reset previous value of v1
    for(j  $\leftarrow$  0 to tot_nodes-1)
    {
        if(s[j]=0)then
        {
            if(dist[j]<min_dist)then
            {
                min_dist  $\leftarrow$  dist[j]
                v1  $\leftarrow$  j
            }
        }
    }
    s[v1]  $\leftarrow$  1
    for(v2  $\leftarrow$  0 to tot_nodes-1)
    {
        if(s[v2]=0)then
        {
            if(dist[v1]+cost[v1][v2]<dist[v2])then
            {
                dist[v2]  $\leftarrow$  dist[v1]+cost[v1][v2]
                path[v2]  $\leftarrow$  v1
            }
        }
    }
}

```

Finding minimum distance from selected source node. That is : source-j represents min_dist. edge

v₁ is next selected destination vertex with shortest distance. All such vertices are accumulated in array path[]

Review Questions

1. Explain divide and conquer strategy with example. Also comment on the time analysis.

SPPU : May-17, 18, Marks 6

2. Explain the greedy strategy with suitable example. Comment on its time complexity.

SPPU : Dec.-17, Marks 6

3. What is divide and conquer strategy ?

SPPU : Dec.-19, Marks 3



Notes

Unit - II

2

Linear Data Structure using Sequential Organization

Syllabus

*Concept of Sequential Organization, Overview of Array, Array as an Abstract Data Type, Operations on Array, Merging of two arrays, Storage Representation and their Address Calculation : Row major and Column Major, Multidimensional Arrays : Two-dimensional arrays, n-dimensional arrays. Concept of Ordered List, **Single Variable Polynomial** : Representation using arrays, Polynomial as array of structure, Polynomial addition, Polynomial multiplication. **Sparse Matrix** : Sparse matrix representation using array, Sparse matrix addition, Transpose of sparse matrix- Simple and Fast Transpose, Time and Space tradeoff.*

Contents

2.1	Concept of Sequential Organization	
2.2	Array as an Abstract Data Type	
2.3	Array Overview	
2.4	Operations on Array	
2.5	Merging of Two Arrays	
2.6	Storage Representation and their Address Calculation	
	Dec.-06, 09, 16, 18, Marks 6
2.7	Multidimensional Arrays	
2.8	Concept of Ordered List	
2.9	Single Variable Polynomial	May-17, 18, Marks 3
2.10	Sparse Matrix	May-17, 19, Dec.-19, Marks 6
2.11	Time and Space Tradeoff	

2.1 Concept of Sequential Organization

Arrays is referred as the **sequential organization** that means the data in arrays is stored in some sequence.

For example : If we want to store names of all the students in a class we can make use of an array to store the names in **sequential** form.

Definition of Arrays : Array is a set of consecutive memory locations which contains similar data elements.

Array is basically a **set of pair-index and the value**.

Syntax

```
data_type  name_of_array [size] ;
```

For example, `int a [10]; double b[10] [10];`

Here 'a' is the name of the array inside the square bracket size of the array is given. This array is of integer type i.e. all the elements are of integer type in array 'a'.

Advantages of sequential organization of data structure

1. Elements can be retrieved or stored very efficiently in sequential organization with the help of index or memory location.
2. All the elements are stored at continuous memory locations. Hence searching of element from sequential organization is easy.

Disadvantages of sequential organization of data structure

1. Insertion and deletion of elements becomes complicated due to sequential nature.
2. For storing the data large continuous free block of memory is required.
3. Memory fragmentation occurs if we remove the elements randomly.

2.2 Array as an Abstract Data Type

The abstract data type is written with the help of instances and operations.

We make use of the reserved word **AbstractDataType** while writing an ADT.

AbstractDataType Array

{

Instances : An array A of some size, index i and total number of elements in the array n.

Operations :

1. **Create ()** – This operation creates an array.

2. **Insert()** - This operation is for inserting the element in an array
3. **Delete()** - This operation is for deleting the elements from the array.
Only logical deletion of the element is possible.
4. **display ()** - This operation displays the elements of the array.

}

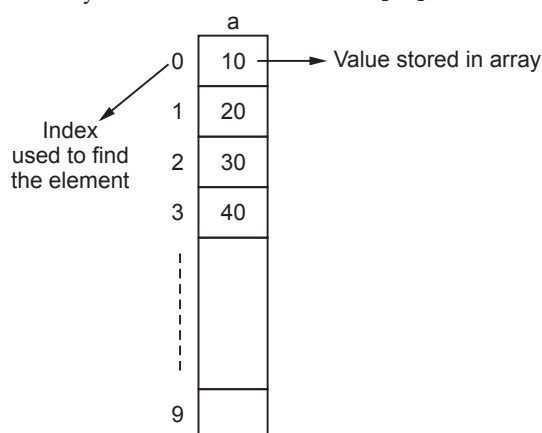
2.3 Array Overview

Definition: Array is a collection of similar type of elements.

The arrays can be one dimensional, two dimensional, or multidimensional.

One dimensional array :

The one dimensional array 'a' is declared as `int a[10];`



Two dimensional array :

If we declare a two dimensional array as

`int a[10] [3];`

Then it will look like this -

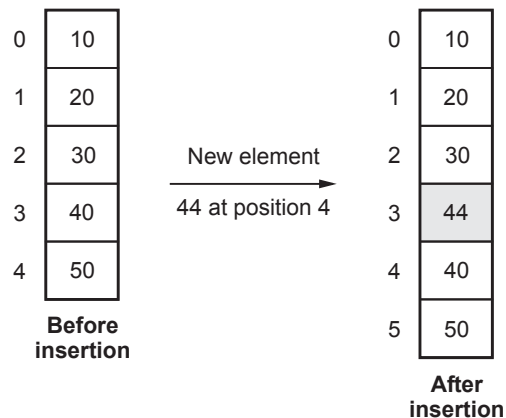
		Column ↓		
		0	1	2
row →	0	10	20	30
	1	40	50	60
	2			
	.			
	.			
	.			
	9			

The two dimensional array should be in row-column form.

2.4 Operations on Array

(1) Insertion of Element in Array

- Inserting an element in an array is complex activity because we have to shift the elements ahead in order to create a space for new element.
- That means after inserting an element array size gets incremented by one.



- We can implement array using Python program. Python does not support the concept of array, but we can implement the array using **List**
- List is a sequence of values.
- String is also sequence of values. But in string this sequence is of characters. On the other hand, in case of list the values can be of any type.
- The values in the list are called **elements** or **items**. These elements are separated by commas and enclosed within the square bracket.

For example

```
[10,20,30,40] # list of integers  
['aaa','bbb','ccc'] #list of strings
```

- The list that contains no element is called **empty list**. The empty list is represented by [].

Python Program

```
print("\nHow many elements are there in Array?")  
n = int(input())  
array = []  
i=0  
for i in range(n):  
    print("\n Enter element in Array")  
    item = int(input())  
    array.append(item)
```

```
print("Enter the location where you want to insert an element")
position = int(input())

print("Enter the value to insert")
value = int(input())
array=array[:position]+[value]+array[position:]
print("Resultant array is\n")
print(array)
```

Output

```
How many elements are there in Array?
5

Enter element in Array
10

Enter element in Array
20

Enter element in Array
30

Enter element in Array
40

Enter element in Array
50
Enter the location where you want to insert an element
4
Enter the value to insert
44
Resultant array is

[10, 20, 30, 40, 44, 50]
>>>
```

Logic Explanation

For insertion of any element in the array, we can make use of list slicing technique. Here is an illustration.

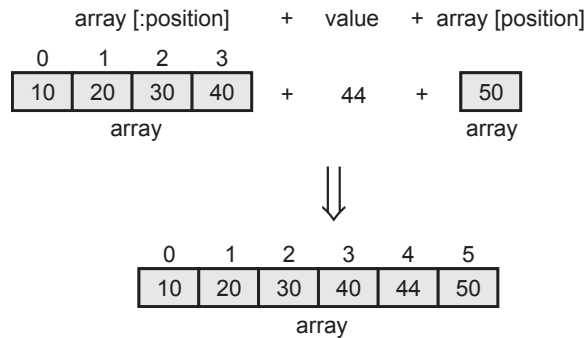
Suppose the array is as follows -

0	1	2	3	4
10	20	30	40	50

array

Position = 4

Element to be inserted = 44



C++ Code

```
#include <iostream>
using namespace std;
int main()
{
    int array[100], position, i, n, value;
    cout<<"\n How many elements are there in array";
    cin>>n;
    cout<<"Enter the elements\n";
    for (i = 0; i < n; i++)
        cin>>array[i];
    cout<<"\n Enter the location where you wish to insert an element: ";
    cin>> position;
    cout<<"\n Enter the value to insert: ";
    cin>>value;
    for (i = n - 1; i >= position - 1; i--)//creating space by shifting the element down
        array[i + 1] = array[i];
    array[position - 1] = value;//at the desired space inserting the element
    printf("Resultant array is\n");
    for (i = 0; i <= n; i++)
        cout<<"\n"<< array[i];
    return 0;
}
```

(2) Traversing List

- The loop is used in list for traversing purpose. The **for** loop is used to traverse the list elements.

Syntax

```
for VARIABLE in LIST :
    BODY
```


Example

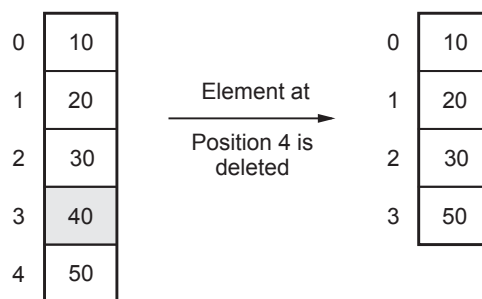
```
>>> a=['a','b','c','d','e'] # List a is created
>>> for i in a:
print(i)
will result into
a
b
c
d
e
>>>
```

- We can traverse the list using **range()** function. Using range() function we can access each element of the list using index of a list.
- If we want to increment each element of the list by one, then we must pass index as argument to for loop. This can be done using range() function as follows -

```
>>> a=[10,20,30,40]
>>> for i in range(len(a)):
a[i]=a[i]+1 #incremented each number by one
>>> a
[11, 21, 31, 41]
>>>
```

(3) Deleting an element from array

- Deleting an element from array is complex activity because have to shift the elements to previous position.
- That means after deleting an element size gets decremented by one.

**Python Program**

```
print("\nHow many elements are there in Array?")
n = int(input())
array = []
i=0
```

```
for i in range(n):
    print("\n Enter element in Array")
    item = int(input())
    array.append(item)

print("Enter the index from where you want to delete an element")
position = int(input())
array=array[:position]+array[position+1:]
print("Resultant array is\n")
print(array)
```

Output

```
How many elements are there in Array?
5

Enter element in Array
10

Enter element in Array
20

Enter element in Array
30

Enter element in Array
40

Enter element in Array
50
Enter the index from where you want to delete an element
2
Resultant array is

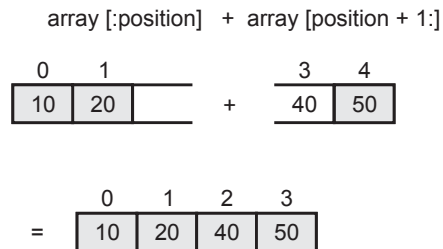
[10, 20, 40, 50]
>>> |
```

Logic Explanation :

Suppose array is

0	1	2	3	4
10	20	30	40	50

Position = 2. That means we want to delete an element 30, then



C++ Code

```

#include <iostream>
using namespace std;
int main()
{
    int array[100], position, i, n;
    cout<<"\n How many elements are there in array ";
    cin>>n;
    cout<<"\nEnter the elements:\n";
    for (i = 0; i < n; i++)
        cin>>array[i];
    cout<<"\n Enter the location of the element which is to be deleted ";
    cin>>position;
    if (position >= n + 1)
        cout<<"Element can not be deleted\n";
    else
    {
        for (i = position - 1; i < n - 1; i++)
            array[i] = array[i + 1];
        cout<<"\nArray is\n";
        for (i = 0; i < n - 1; i++)
            cout<<"\n"<< array[i];
    }
    return 0;
}

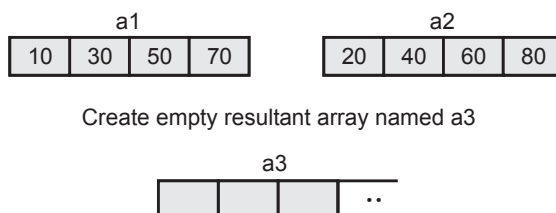
```

2.5 Merging of Two Arrays

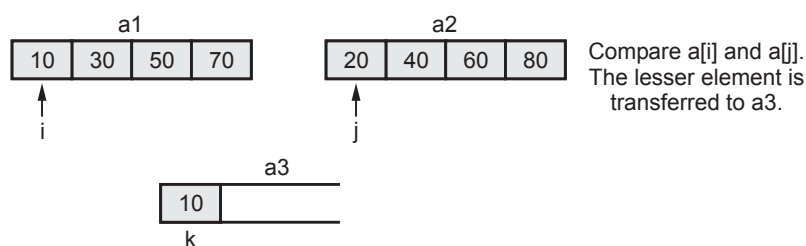
Merging of two arrays result into a single array in which elements are arranged in sorted order.

For example - consider two arrays as follows

Step 1 :

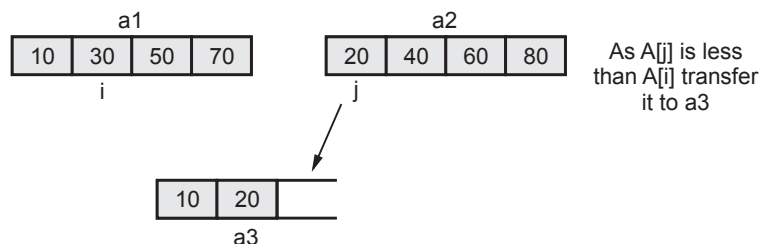


Step 2 :



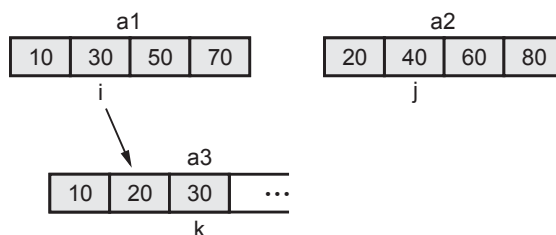
As $a1[i] < a2[j]$, transfer element $a1[i]$ to $a3[k]$. Then increment i and k pointer

Step 3 :



As $a2[j]$ is transferred to $a3$ array increment j and k pointer.

Step 4 :



Increment i and k pointers. In this way we can transfer the elements from two arrays $a1$ and $a2$ to get merged array $a3$.

Finally we get



Python Program

```
def mergeArr(a1,a2,n,m):
    a3 = [None]*(n+m)
    i = 0
    j = 0
    k = 0
    #traverse both arrays
    #if element of first array is less then store it in third array
    #if element of second array is less then store it in third array
    while i < n and j < m:
        if a1[i] < a2[j]:
            a3[k] = a1[i]
            k = k + 1
            i = i + 1
        else:
            a3[k] = a2[j]
            k = k + 1
            j = j + 1
    #if elements of first array are remaining
    #then transfer them to third array
    while i < n:
        a3[k] = a1[i]
        k = k + 1
        i = i + 1

    #if elements of second array are remaining
    #then transfer them to third array
    while j < m:
        a3[k] = a2[j]
        k = k + 1
        j = j + 1

    #display the resultant merged array
    print("Merged Array is ...")
    for i in range (n+m):
        print(str(a3[i]), end = " ")
```

#Driver Code

```
a1 = []
n = int(input("Enter total number of elements in first array:"))
for i in range(0,n):
    item = int(input("Enter the element: "))
    a1.append(item)

a2 = []
m = int(input("Enter total number of elements in second array:"))
for i in range(0,m):
    item = int(input("Enter the element: "))
```

```
a2.append(item)
mergeArr(a1,a2,n,m)
```

Output

```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help
Enter total number of elements in first array:4
Enter the element: 10
Enter the element: 30
Enter the element: 50
Enter the element: 70
Enter total number of elements in second array:4
Enter the element: 20
Enter the element: 40
Enter the element: 60
Enter the element: 80
Merged Array is ...
10 20 30 40 50 60 70 80
>>>
```

2.6 Storage Representation and their Address Calculation**SPPU : Dec.-06, 09, 16, 18, Marks 6**

The array can be represented using i) Row Major ii) Column Major Representation.

Row Major Representation

If the elements are stored in **rowwise manner** then it is called row major representation.

For example : If we want to store elements

10 20 30 40 50 60 then in a two dimensional array

		0	1	2
	0	10	20	30
	1	40	50	60
	.			
	.			
	.			
	9			

The \Rightarrow elements will
be stored
horizontally

To access any element in two dimensional array we must specify both its row number and column number. That is why we need two variables which act as row index and column index.

Column Major Representation

If elements are stored in **column wise manner** then it is called column major representation.

For example : If we want to store elements

10 20 30 40 50 60 then the elements will be filled up by columnwise manner as follows (consider array a[3] [2]). Here 3 represents number of rows and 2 represents number of columns.

	0	1
0	10	40
1	20	50
2	30	60

Each element is occupied at successive locations if the element is of integer type then 2 bytes of memory will be allocated, if it is of floating type then 4 bytes of memory will be allocated and so on.

For example :

```
int a [3] [2] = { {10, 20 }
                 {30, 40}
                 {50, 60} }
```

Then in row major matrix

a[0][0]	a[0][1]	a[1][0]	a[1][1]	a[2][0]	a[2][1]	
10	20	30	40	50	60	...
100	102	104	106	108	110	

And in column major matrix

a[0][0]	a[1][0]	a[2][0]	a[0][1]	a[1][1]	a[2][1]	
10	30	50	20	40	60	...
100	102	104	106	108	110	

Here each element occupies 2 bytes of memory base address will be 100.

Address calculation for any element will be as follows

In row major matrix, the element a[i] [j] will be at

base address + (row_index * total number of columns + column_index)
* element_size.

In column major matrix, the element at $a[i][j]$ will be at

base address + (column_index * total number of rows + row_index) * element.

In C normally the row major representation is used.

Example 2.6.1 Consider integer array `int arr[3][4]` declared in 'C' program. If the base address is 1050, find the address of the element `arr[2][3]` with row major and column major representation of array.

SPPU : Dec.-06, Marks 6

Solution : Row Major Representation

The element $a[i][j]$ will be at

$$a[i][j] = \text{base address} + (\text{row_index} * \text{total number of columns} + \text{column_index}) * \text{element_size}$$

$$= (\text{base address} + (i * \text{col_size} + j) * \text{element_size})$$

when $i=2$ and $j=3$, element_size=int occupies 2 bytes of memory hence it is 2

total number of columns=4,

$$a[2][3] = 1050 + (2 * 4 + 3) * 2$$

$$= 1050 + 22$$

$$a[2][3] = 1072$$

Column Major Representation

The element $a[i][j]$ will be at

$$a[i][j] = \text{base address} + (\text{col_index} * \text{total number of rows} + \text{row_index}) * \text{element_size}$$

$$= (\text{base address} + (j * \text{row_size} + i) * \text{element_size})$$

when $i=2$ and $j=3$, element_size=int occupies 2 bytes of memory hence it is 2

total number of rows=3

$$a[2][3] = 1050 + (3 * 3 + 2) * 2$$

$$= 1050 + 22$$

$$a[2][3] = 1072$$

Example 2.6.2 Consider integer array `int arr[4][5]` declared in 'C' program. If the base address is 1020, find the address of the element `arr[3][4]` with row major and column major representation of array.

SPPU : Dec.-09, Marks 6

Solution : Row Major Representation

The element $a[i][j]$ will be at

$$\begin{aligned}a[i][j] &= \text{base address} + (\text{row_index} * \text{total number of columns} + \\ &\quad \text{col_index}) * \text{element_size} \\ &= (\text{base address} + (i * \text{col_size} + j) * \text{element_size})\end{aligned}$$

when $i=3$ and $j=4$, $\text{element_size}=\text{int}$ occupies 2 bytes of memory hence it is 2
total number of columns=5

$$\begin{aligned}a[3][4] &= 1020 + (3*5 + 4)*2 \\ &= 1020 + 38 \\ a[3][4] &= 1058\end{aligned}$$

Column Major Representation

The element $a[i][j]$ will be at

$$\begin{aligned}a[i][j] &= \text{base address} + (\text{col_index} * \text{total number of} \\ &\quad \text{rows} + \text{row_index}) * \text{element_size} \\ &= (\text{base address} + (j * \text{row_size} + i) * \text{element_size})\end{aligned}$$

when $i=3$ and $j=4$, $\text{element_size}=\text{int}$ occupies 2 bytes of memory hence it is 2
total number of rows = 4

$$\begin{aligned}a[3][4] &= 1020 + (4*4 + 3)*2 \\ &= 1020 + 38 \\ a[3][4] &= 1058\end{aligned}$$

Review Questions

1. Derive address calculation formula for one dimensional array with one example.

SPPU : Dec.-16, Marks 6

2. Explain two dimensional arrays with row and column major implementation. Explain address calculation in both cases with example

SPPU : Dec.-18, Marks 6

2.7 Multidimensional Arrays

Multidimensional array is an array having more than one dimension. Popularly, two and three dimensional arrays are used.

2.7.1 Two Dimensional Array

The two dimensional array is used to represent the matrix.

Various operations that can be performed on matrix are –

(1) Matrix Addition (2) Matrix Multiplication and (3) Transpose of Matrix

Let us discuss the implementation of various matrix operations

Example 2.7.1 Write a python program for representation two dimensional matrix.

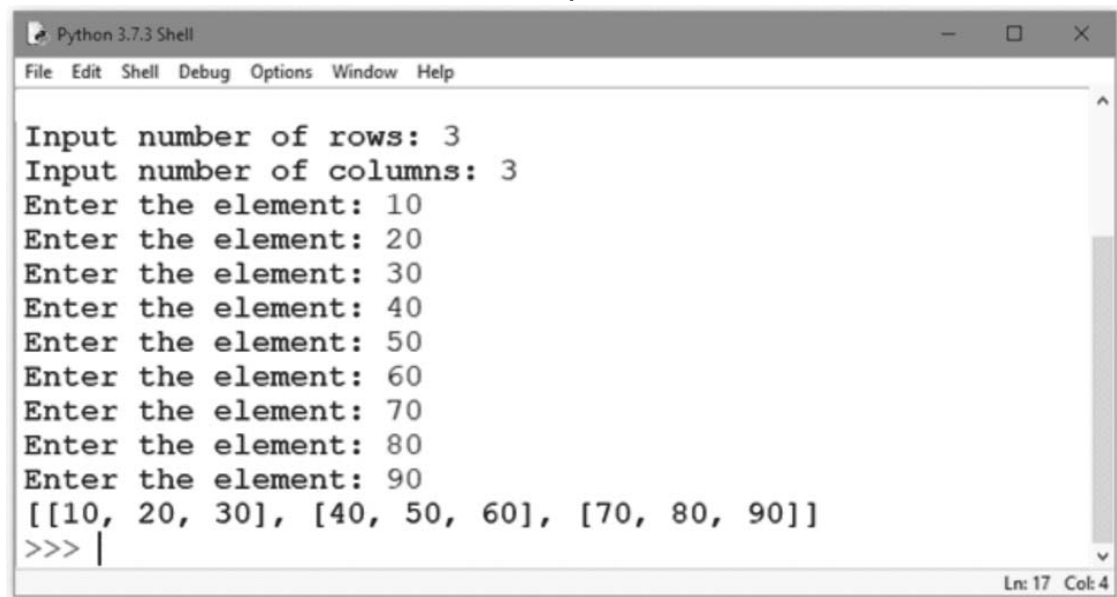
Solution :

```
# This program stores and displays the elements of two dimensional Array
row_num = int(input("Input number of rows: "))
col_num = int(input("Input number of columns: "))
arr = [[0 for col in range(col_num)] for row in range(row_num)]

for row in range(row_num):
    for col in range(col_num):
        item = int(input("Enter the element: "))
        arr[row][col] = item

print(arr)
```

Output



```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help

Input number of rows: 3
Input number of columns: 3
Enter the element: 10
Enter the element: 20
Enter the element: 30
Enter the element: 40
Enter the element: 50
Enter the element: 60
Enter the element: 70
Enter the element: 80
Enter the element: 90
[[10, 20, 30], [40, 50, 60], [70, 80, 90]]
>>> |
```

Ln: 17 Col: 4

C++ Code

```
// This program stores and displays the elements of two dimensional Array
cout<<"\nInput number of rows: ";
cin>>row_num;
cout<<"\nInput number of columns: ";
cin>>col_num;
```

```

int arr[row_num][col_num];
for (row=0;row<row_num;row++)
{
    for (col=0;col<col_num;col++)
    {
        cout<<"Enter the element: ";
        cin>>arr[row][col];
    }
}
for (row=0;row<row_num;row++)
{
    for (col=0;col<col_num;col++)
    {
        cout<<"\t"arr[row][col];
    }
    cout<<"\n";
}

```

(1) Addition of Two Matrices

Consider matrix A and B as follows –

Matrix A

1	2	3
4	5	6
7	8	9

Matrix B

1	1	1
2	2	2
3	3	3

Addition of Matrices is

2	3	4
6	7	8
10	11	12

Example 2.7.2 Write a python program for performing addition of two matrices

Solution :

```

def add_matrix(arr1,arr2):
    result = [[arr1[i][j] + arr2[i][j] for j in range(len(arr1[0]))] for i in range(len(arr1))]

    print("The Addition of Two Matrices...")
    print(result)
row_num = int(input("Input number of rows: "))
col_num = int(input("Input number of columns: "))
arr1 = [[0 for col in range(col_num)] for row in range(row_num)]
for row in range(row_num):
    for col in range(col_num):
        item = int(input("Enter the elements in first matrix: "))
        arr1[row][col]= item

```

```
print("The first matrix is...")
print(arr1)

arr2 = [[0 for col in range(col_num)] for row in range(row_num)]
for row in range(row_num):
    for col in range(col_num):
        item = int(input("Enter the elements in second matrix: "))
        arr2[row][col] = item

print("The second matrix is...")
print(arr2)

#Driver Code
add_matrix(arr1,arr2,)
```

Output

```
Input number of rows: 3
Input number of columns: 3
Enter the elements in first matrix: 1
Enter the elements in first matrix: 2
Enter the elements in first matrix: 3
Enter the elements in first matrix: 4
Enter the elements in first matrix: 5
Enter the elements in first matrix: 6
Enter the elements in first matrix: 7
Enter the elements in first matrix: 8
Enter the elements in first matrix: 9
The first matrix is...
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Enter the elements in second matrix: 1
Enter the elements in second matrix: 1
Enter the elements in second matrix: 1
Enter the elements in second matrix: 2
Enter the elements in second matrix: 2
Enter the elements in second matrix: 2
Enter the elements in second matrix: 3
Enter the elements in second matrix: 3
Enter the elements in second matrix: 3
The second matrix is...
[[1, 1, 1], [2, 2, 2], [3, 3, 3]]
The Addition of Two Matrices...
[[2, 3, 4], [6, 7, 8], [10, 11, 12]]
>>>
```

(2) Matrix Multiplication**Consider matrix A**

1	2	3
4	5	6
7	8	9

Consider matrix B

1	1	1
2	2	2
3	3	4

The resultant matrix is

14	14	17
32	32	38
50	50	59

Example 2.7.3 Write a python program to implement matrix multiplication operation.

Solution :

```

A      =  [[1,2,3],
           [4,5,6],
           [7,8,9]]

B      =  [[1,1,1],
           [2,2,2],
           [3,3,4]]

result =  [[0,0,0],
           [0,0,0],
           [0,0,0]]

print("Matrix A is ...")
print(A)

print("Matrix B is ...")
print(B)

# iterate through rows of X
for i in range(len(A)):
    for j in range(len(B[0])):
        for k in range(len(B)):
            result[i][j] += A[i][k] * B[k][j]

print("Matrix Multiplication is ...")
for r in result:
    print(r)

```

Output

```

Matrix A is ...
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Matrix B is ...
[[1, 1, 1], [2, 2, 2], [3, 3, 4]]

```

Matrix Multiplication is ...

[14, 14, 17]

[32, 32, 38]

[50, 50, 59]

>>>

(3) Transpose of Matrix

Consider matrix A

1	2	3
4	5	6
7	8	9

Transposed matrix

1	4	7
2	5	8
3	6	9

Example 2.7.4 Write a Python program for performing transpose of matrix.

Solution :

```
A = [[1,2,3],
      [4,5,6],
      [7,8,9]]

result = [[0,0,0],
          [0,0,0],
          [0,0,0]]

print("Original Matrix is...")
print(A)

# iterate through rows
for i in range(len(A)):
    for j in range(len(A[0])):
        result[j][i] = A[i][j]

print("Transposed Matrix is ...")
for r in result:
    print(r)
```

Output

```
Original Matrix is...
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
Transposed Matrix is ...
[1, 4, 7]
[2, 5, 8]
[3, 6, 9]
>>>
```

2.7.2 Three Dimensional Array

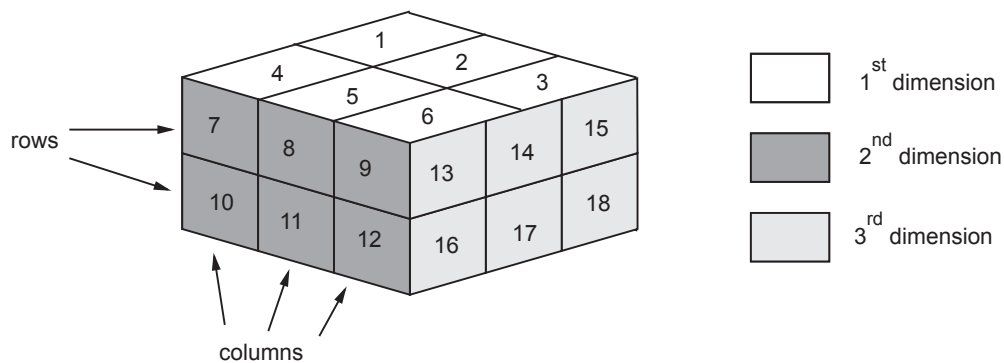
The multidimensional array is similar to the two dimensional array with multiple dimensions for example Here is 3-D array

$$a[3][2][3] = \{ \{ \{1,2,3\}, \{4,5,6\} \}, \{ \{7,8,9\}, \{10,11,12\} \}, \{ \{13,14,15\}, \{16,17,18\} \} \}$$

← dimension
↓
↓

rows
column

We can represent it graphically as



Python Program

```
row_num = int(input("Input number of rows: "))
col_num = int(input("Input number of columns: "))
dim_num = int(input("Input number of dimensions: "))
arr1 = [] #creating an empty list
for dim in range(dim_num):
    arr1.append([])
    for row in range(row_num):
        arr1[dim].append([])
        for col in range(col_num):
            item = int(input("Enter Element: "))
            arr1[dim][row].append(item)

print("\n Elements in 3D Array are...")
for dim in range(dim_num):
    for row in range(row_num):
        for col in range(col_num):
            print(arr1[dim][row][col], end = " ")
        print()
    print("_____")
```

Output

```
Input number of rows: 2
Input number of columns: 3
Input number of dimensions: 3
Enter Element: 1
Enter Element: 2
Enter Element: 3
Enter Element: 4
Enter Element: 5
Enter Element: 6
Enter Element: 7
Enter Element: 8
Enter Element: 9
Enter Element: 10
Enter Element: 11
Enter Element: 12
Enter Element: 13
Enter Element: 14
Enter Element: 15
Enter Element: 16
Enter Element: 17
Enter Element: 18

  Elements in 3D Array are...
1  2  3
4  5  6
-----
7  8  9
10 11 12
-----
13 14 15
16 17 18
-----
>>> |
```

C++ Code

```
/*
    This program is for storing and retrieving the elements in a 3-D array
*/
#include<iostream>
using namespace std;
int main()
```



```
{
    int a[3][2][3]
    int i,j,k;
    cout<<"\n Enter the elements";
    for (i=0; i<3; i++)
    {
        for(j=0; j<2; j++)
        {
            for(k=0; k<3; k++)
            {
                cin>>a[i][j][k];
            }
        }
    }
    cout<<"\n Printing the elements \n";
    for(i=0; i<3; i++)
    {
        for (j=0; j<2; j++)
        {
            for(k=0; k<3; k++)
            {
                cout<<"\t"<<a[i][j][k];
            }
            cout<<"\n";
        }
        cout<<"\n-----";
    }
    return 0;
}
```

2.8 Concept of Ordered List

Ordered list is nothing but a set of elements. Such a list sometimes called as linear list.

For example

1. List of one digit numbers
(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
2. Days in a week.
(Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday)

With this concept in mind let us formally define the ordered list.

Definition : An ordered list is set of elements where set may be empty or it can be written as a collection of elements such as (a₁, a₂, a₃ an).

Operations on ordered list

Following operations can be possible on an ordered list.

1. Display of list
2. Searching a particular element from the list
3. Insertion of any element in the list
4. Deletion of any element from the list

Ordered List can be implemented using the List data structure in Python.

Various operations on List are possible using following methods

2.8.1 Ordered List Methods

(1) append

The append method adds the element at the end of the list. For example

```
>>> a=[10,20,30]
>>> a.append(40)  #adding element 40 at the end
>>> a
[10, 20, 30, 40]
>>> b=['A','B','C']
>>> b.append('D') #adding element D at the end
>>> b
['A', 'B', 'C', 'D']
>>>
```

(2) extend

The extend function takes the list as an argument and appends this list at the end of old list. For example

```
>>> a=[10,20,30]
>>> b=['a','b','c']
>>> a.extend(b)
>>> a
[10, 20, 30, 'a', 'b', 'c']
>>>
```

(3) sort

The sort method arranges the elements in increasing order. For example

```
>>> a=['x','z','u','v','y','w']
>>> a.sort()
>>> a
['u', 'v', 'w', 'x', 'y', 'z']
>>>
```

The methods `append`, `extend` and `sort` does not return any value. These methods simply modify the list. These are void methods.

(4) Insert

This method allows us to insert the data at desired position in the list. The syntax is **`insert(index,element)`**

For example -

```
>>> a=[10,20,40]
>>> a.insert(2,30)
>>> print(a)
[10, 20, 30, 40]
>>>
```

(5) Delete

The deletion of any element from the list is carried out using various functions like **`pop`**, **`remove`**, **`del`**.

If we know the index of the element to be deleted then just pass that index as an argument to **`pop`** function.

For example

```
>>> a=['u','v','w','x','y','z']
>>> val=a.pop(1) #the element at index 1 is v, it is deleted
>>> a
['u', 'w', 'x', 'y', 'z']    #list after deletion
>>> val    #deleted element is present in variable val
'v'
>>>
```

If we do not provide any argument to the `pop` function then the last element of the list will be deleted.

For example -

```
>>> a=['u','v','w','x','y','z']
>>> val=a.pop()
>>> a
['u', 'v', 'w', 'x', 'y']
>>> val
'z'
>>>
```

If we know the value of the element to be deleted then the **`remove`** function is used. That means the parameter passed to the `remove` function is the actual value that is to be removed from the list. Unlike, **`pop`** function the **`remove`** function does not return any value. The execution of **`remove`** function is shown by following screenshot.

```
>>> a=['a','b','c','d','e']
>>> a.remove('c')
>>> a
['a', 'b', 'd', 'e']
>>>
```

In python, it is possible to remove more than one element at a time using **del** function.

For example

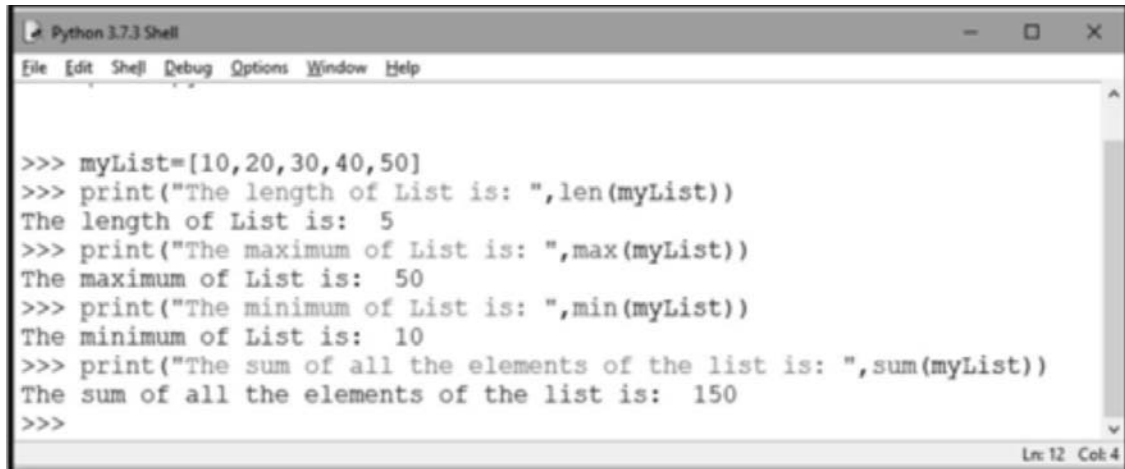
```
>>> a=['a','b','c','d','e']
>>> del a[2:4]
>>> a
['a', 'b', 'e']
>>>
```

2.8.2 Built in Functions

- There are various built in functions in python for supporting the list operations. Following table shows these functions

Built-in function	Purpose
all()	If all the elements of the list are true or if the list is empty then this function returns true.
any()	If the list contains any element true or if the list is empty then this function returns true.
len()	This function returns the length of the string.
max()	This function returns maximum element present in the list.
min()	This function returns minimum element present in the list.
sum()	This function returns the sum of all the elements in the list.
sorted()	This function returns a list which is sorted one.

For example - Following screenshot of python shell represents execution of various built in functions –



```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help

>>> myList=[10,20,30,40,50]
>>> print("The length of List is: ",len(myList))
The length of List is: 5
>>> print("The maximum of List is: ",max(myList))
The maximum of List is: 50
>>> print("The minimum of List is: ",min(myList))
The minimum of List is: 10
>>> print("The sum of all the elements of the list is: ",sum(myList))
The sum of all the elements of the list is: 150
>>>
```

2.8.3 List Comprehension

- List comprehension is an elegant way to create and define new lists using existing lists.
- This is mainly useful to make new list where each element is obtained **by applying some operations to each member** of another sequence.

Syntax

```
List=[expression for item in the list]
```

Example 2.8.1 Write a python program to create a list of even numbers from 0 to 10.

Solution :

```
even = [] #creating empty list
for i in range(11):
    if i % 2 ==0:
        even.append(i)
print("Even Numbers List: ",even)
even = [] #creating empty list
for i in range(11):
    if i % 2 ==0:
        even.append(i)
print("Even Numbers List: ",even)
```

Output

```
Even Numbers List: [0, 2, 4, 6, 8, 10]
```

Program explanation : In above program,

- 1) We have created an empty list first.
- 2) Then using the range of numbers from 0 to 11 we append the empty list with even numbers. The even number is test using the if condition. i.e. if $I \% 2 == 0$. If so then that even number is appended in the list.
- 3) Finally the comprehended list will be displayed using print statement.

Example 2.8.2 Write a python program to combine and print two lists using list comprehension.

Solution :

```
print([(x,y) for x in ['a','b'] for y in ['b','d'] if x!=y])
```

Output

```
[('a', 'b'), ('a', 'd'), ('b', 'd')]
```

2.9 Single Variable Polynomial

SPPU : May-17, 18, Marks 3

Definition : Polynomial is the sum of terms where each term consists of variable, coefficient and exponent.

Various operations on polynomial are – addition, subtraction, multiplication and evaluation.

2.9.1 Representation

For representing a single variable polynomial one can make use of one dimensional array. In single dimensional array the index of an array will act as the exponent and the coefficient can be stored at that particular index which can be represented as follows :

For e.g. : $3x^4 + 5x^3 + 7x^2 + 10x - 19$

This polynomial can be stored in single dimensional array.

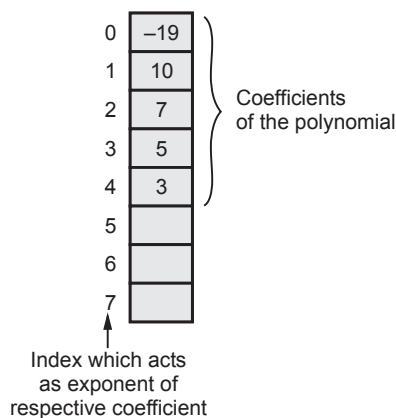


Fig. 2.9.1 Polynomial representation

2.9.2 Polynomial Addition

Algorithm :

Assume that the two polynomials say A and B and the resultant polynomial storing the addition result is stored in one large array.

1. Set a pointer i to point to the first term in a polynomial A.

2. Set a pointer j to point to first term in a polynomial B.
3. Set a pointer k to point to the first position in array C.
4. Read the number of terms of A polynomial in variable t1 and read the number of terms of B polynomial in variable t2.
5. While i < t1 and j < t2 then
 - { if exponent at ith position of A poly is equal to the exponent at jth position of polynomial B then
 - Add the coefficients at that position from both the polynomial and store the result in C arrays coefficient field at position k copy the exponent either pointed by i or j at position k in C array.
 - Increment i, j and k to point to the next position in the array A, B and C.
 - }
 - else
 - {
 - if the exponent position i is greater than the exponent at position j in polynomial B then
 - {
 - copy the coefficient at position i from A polynomial into coefficient field at position k of array C copy the exponent pointed by i into the exponent field at position k in array C.
 - Increment i, k pointers.
 - }
 - else
 - {
 - Copy the coefficient at position j of B polynomial into coefficient field at position k in C array.
 - Copy the exponent pointed by j into exponent field at position k in C array.
 - Increment j and k pointers to point to the next position in array B and C.
 - }

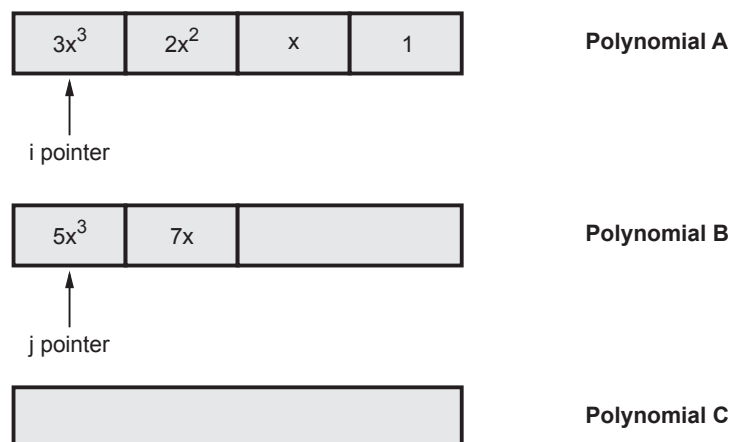
6. While $i < t1$
 - {
 - Copy the coefficient at position i of into the coefficient field at position k in C .
 - Copy the exponent pointed by i into the exponent field at position k in C array.
 - Increment i and k to point to the next position in arrays A and C .
 - }
7. While $j < t2$
 - {
 - Copy the coefficient at position j of B into the coefficient field at position k in C .
 - Copy the exponent pointed by j into exponent field at position k in C .
 - Increment j , k to point to the next position in B and C arrays.
 - }
8. Display the complete array C as the addition of two given polynomials.
9. Stop.

Logic of Addition of two polynomials

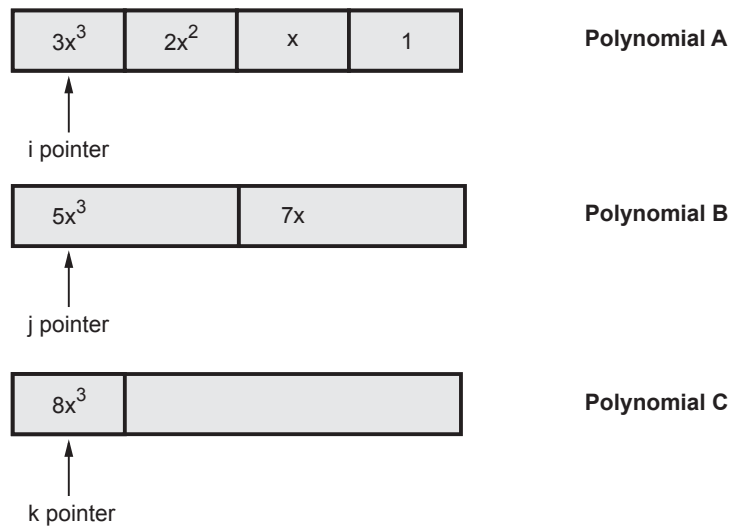
Let us take polynomial A and B as follows :

$$3x^3 + 2x^2 + x + 1$$

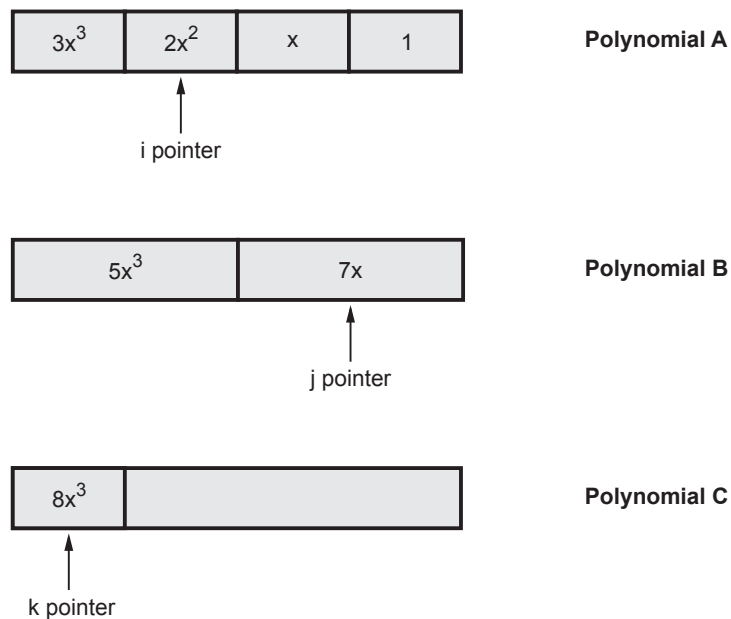
$$5x^3 + 7x$$



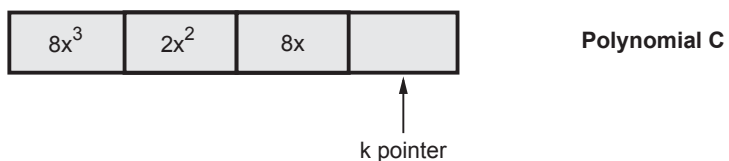
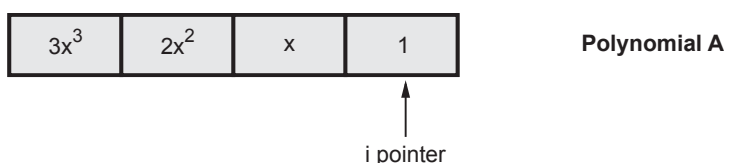
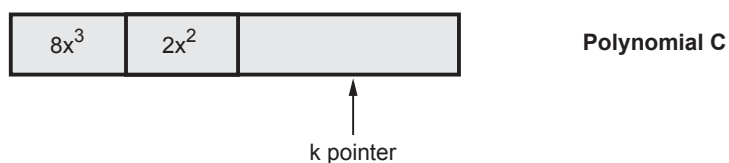
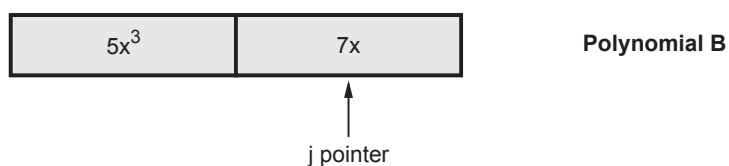
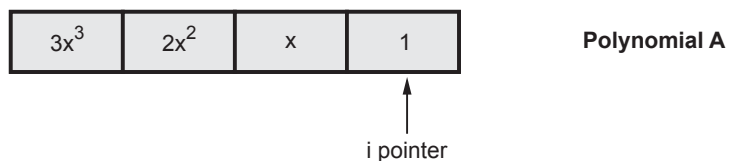
As the terms pointed by i and j shows that both the terms have equal exponent, we can perform addition of these terms and we will store the result in polynomial C.



Now increment i, j, and k pointers.



Now pointer i points to $2x^2$ and pointer j points to $7x$. As $2x^2 > 7x$. We will copy $2x^2$ in the polynomial C. And we will simply increment i and k pointers



Now terms in polynomial B are over. Hence we will copy the remaining terms from polynomial A to polynomial C.

$3x^3$	$2x^2$	x	1
--------	--------	-----	---

Polynomial A

$5x^3$	$7x$
--------	------

Polynomial B

$8x^3$	$2x^2$	$8x$	1
--------	--------	------	---

Polynomial C

Thus the addition of polynomials A and B is in polynomial C.

Python Program

```
# A[] represents coefficients of first polynomial
# B[] represents coefficients of second polynomial
# m and n are sizes of A[] and B[] respectively
def add(A, B, m, n):
    size = max(m, n);
    C = [0 for i in range(size)]

    for i in range(0, m, 1):
        #Each term from first polynomial to C array
        C[i] = A[i]
    # Add each term of second poly and add it to C
    for i in range(n):
        C[i] += B[i]

    return C

# A function to print a polynomial
def display(poly, n):
    for i in range(n):
        print(poly[i], end = "")
        if (i != 0):
            print("x^", i, end = "")
        if (i != n - 1):
            print(" + ", end = "")

# Driver Code
if __name__ == '__main__':

    # The following array represents
    # polynomial 1 + 1x + 2x^2 + 3x^3
    A = [1, 1, 2, 3]

    # The following array represents
    # polynomial 7x + 5x^3
```

```
B = [0, 7, 0, 5]
m = len(A)
n = len(B)

print("\nFirst polynomial is")
display(A, m)
print("\nSecond polynomial is")
display(B, n)

C = add(A, B, m, n)
size = max(m, n)

print("\n Addition of polynomial is")
display(C, size)
```

Output

```
First polynomial is
1 + 1x^ 1 + 2x^ 2 + 3x^ 3
Second polynomial is
0 + 7x^ 1 + 0x^ 2 + 5x^ 3
Addition of polynomial is
1 + 8x^ 1 + 2x^ 2 + 8x^ 3
>>>
```

C++ Code

```
/******
Program To Perform The Polynomial Addition And To Print
The Resultant Polynomial.
******/
#include<iostream>
using namespace std;
class APADD
{
    private:
        struct p
        {
            int coeff;
            int expo;
        };
    public:
        p p1[10],p2[10],p3[10];
        int Read(p p1[10]);
        int add(p p1[10],p p2[10],int t1,int t2,p p3[10]);
        void Print(p p2[10],int t2);
};
```

```
/*
-----
The Read Function Is For Reading The Two Polynomials
-----
*/
int APADD::Read(p p[10])
{
    int t1,i;
    cout<<"\n Enter The Total number Of Terms in The Polynomial: ";
    cin>>t1;
    cout<<"\n Enter The Coef and Exponent In Descending Order";
    for(i=0;i<t1;i++)
    {
        cout<<"\n Enter Coefficient and exponent: ";
        cin>>p[i].coeff;
        cin>>p[i].expo;
    }
    return(t1);
}

/*
-----
The add Function Is For adding The Two Polynomials
-----*/
int APADD::add(p p1[10],p p2[10],int t1,int t2,p p3[10])
{
    int i,j,k;
    int t3;
    i=0;
    j=0;
    k=0;
    while(i<t1 && j<t2)
    {
        if(p1[i].expo==p2[j].expo)
        {
            p3[k].coeff=p1[i].coeff+p2[j].coeff;
            p3[k].expo =p1[i].expo;
            i++;j++;k++;
        }
        else if(p1[i].expo>p2[j].expo)
        {
            p3[k].coeff=p1[i].coeff;
            p3[k].expo =p1[i].expo;
            i++;k++;
        }
        else
        {

```

```

        p3[k].coeff=p2[j].coeff;
        p3[k].expo =p2[j].expo;
        j++;k++;
    }
}
while(i<t1)
{
    p3[k].coeff=p1[i].coeff;
    p3[k].expo =p1[i].expo;
    i++;k++;
}
while(j<t2)
{
    p3[k].coeff=p2[j].coeff;
    p3[k].expo =p2[j].expo;
    j++;k++;
}
t3=k;
return(t3);
}
/*
-----
The Print Function Is For Printing The Two Polynomials
-----
*/
void APADD::Print(p pp[10],int term)
{
    int k;
    cout<<"\n Printing The Polynomial";
    for(k=0;k<term-1;k++)
        cout<<" "<<pp[k].coeff<<"X ^"<<pp[k].expo<<" + ";
    cout<<pp[k].coeff<<"X ^"<<pp[k].expo;
}
/*
-----
The main function
-----
*/
int main()
{
    APADD obj;
    int t1,t2,t3;
    cout<<"\n Enter The First Polynomial";
    t1=obj.Read(obj.p1);
    cout<<"\n The First Polynomial is: ";
    obj.Print(obj.p1,t1);
    cout<<"\n Enter The Second Polynomial";

```

```

t2=obj.Read(obj.p2);
cout<<"\n The Second Polynomial is: ";
obj.Print(obj.p2,t2);
cout<<"\n The Addition is: ";
t3=obj.add(obj.p1,obj.p2,t1,t2,obj.p3);
obj.Print(obj.p3,t3);
return 0;
}

```

2.9.3 Polynomial Multiplication

We will now discuss another operation on polynomials and that is multiplication.

For example

If two polynomials are given as

$$\begin{array}{r}
 3x^3 + 2x^2 + x + 1 \\
 \times \quad 5x^3 + 7x \\
 \hline
 \end{array}$$

Then we will perform multiplication by multiplying each term of polynomial A by each term of polynomial B.

$$\begin{array}{r}
 3x^3 + 2x^2 + x + 1 \\
 \times \quad 5x^3 + 7x \\
 \hline
 \underbrace{15x^6 + 10x^5 + 5x^4 + 5x^3}_{\text{Multiplied poly A by } 5x^3} + \underbrace{21x^4 + 14x^3 + 7x^2 + 7x}_{\text{Multiplied poly A by } 7x} \\
 \hline
 \end{array}$$

Now rearranging the terms

$$15x^6 + 10x^5 + 26x^4 + 19x^3 + 7x^2 + 7x$$

is a resultant polynomial

Python Program

```

# A[] represents coefficients of first polynomial
# B[] represents coefficients of second polynomial
# m and n are sizes of A[] and B[] respectively
def mul(A, B, m, n):

    #allocating total size for resultant array
    C = [0]*(m+n-1)
    for i in range(0, m, 1):
        #multiplying by current term of first polynomial
        #to each term of second polynomial
        for j in range(n):
            C[i+j] += A[i]*B[j]

    return C

```

```
# A function to print a polynomial
def display(poly, n):
    for i in range(n):
        print(poly[i], end = "")
        if (i != 0):
            print("x^", i, end = "")
        if (i != n - 1):
            print(" + ", end = "")

# Driver Code
if __name__ == '__main__':

    # The following array represents
    # polynomial 1 + 1x + 2x^2 + 3x^3
    A = [1, 1, 2, 3]

    # The following array represents
    # polynomial 7x + 5x^3
    B = [0, 7, 0, 5]
    m = len(A)
    n = len(B)

    print("\nFirst polynomial is")
    display(A, m)
    print("\nSecond polynomial is")
    display(B, n)

    C = mul(A, B, m, n)

    print("\n Multiplication of polynomial is...")
    display(C, m+n-1)
```

Output

```
First polynomial is
1 + 1x^ 1 + 2x^ 2 + 3x^ 3
Second polynomial is
0 + 7x^ 1 + 0x^ 2 + 5x^ 3
Multiplication of polynomial is...
0 + 7x^ 1 + 7x^ 2 + 19x^ 3 + 26x^ 4 + 10x^ 5 + 15x^ 6
>>>
```

2.9.4 Polynomial Evaluation

We will now discuss the algorithm for evaluating the polynomial.

Consider the polynomial for evaluation –

$$-10x^7 + 4x^5 + 3x^2$$

where $x = 1$,

$$= -10 + 4 + 3 = -6 + 3$$

$= 3$ is the result of polynomial evaluation.

Now, let us see the algorithm for polynomial.

Algorithm :

Step 1 : Read the polynomial array A

Step 2 : Read the value of x.

Step 3 : Initialize the variable sum to zero.

Step 4 : Then calculate $\text{coeff} * \text{pow}(x, \text{exp})$ of each term and add the result to sum.

Step 5 : Display " sum"

Step 6 : Stop

Python Program

```
def evalPoly(A,n,x):
    result = A[0]
    for i in range(1,n):
        result = result + A[i]*x**i
    return result

def display(poly, n):
    for i in range(n):
        print(poly[i], end = "")
        if (i != 0):
            print("x^", i, end = "")
        if (i != n - 1):
            print(" + ", end = "")

# Driver Code
if __name__ == '__main__':

    # The following array represents
    # polynomial 1 + 1x + 2x^2 + 3x^3
    A = [1, 1, 2, 3]
    n = len(A)
    print("\n Polynomial is: ");
    display(A,n)

    x = int(input("\nEnter the value of x: "))
    print("\nThe result of evaluation of polynomial is: ",evalPoly(A,n,x))
```

Output

```

Polynomial is:
1 + 1x^ 1 + 2x^ 2 + 3x^ 3
Enter the value of x: 2
The result of evaluation of polynomial is: 35
>>>

```

C++ Code

```

/*****
    Program to evaluate a polynomial in single variable for a
    given value of x.
*****/
#include <iostream>
#include <cmath>
using namespace std;
#define size 20
class APEVAL
{
    private:
    struct p
    {
        int coef;
        int expo;
    };
    public:
    p p1[size];
    int get_poly(p p1[size]);
    void display(p p1[size],int n1);
    float eval(int n1, p p1[]);
};

/*-----
Function get_poly
-----*/
int APEVAL::get_poly( p p1[] )
{
    int term,n;

    cout<<"\nHow Many Terms are there? ";
    cin>>n;
    if (n>size)
    {
        cout<<"Invalid number Of Terms\n";
        getch();
        return;
    }
    cout<<"\nEnter the terms of the Polynomial ";
    cout<<"in descending order of the exponent\n";

```

```

    for (term = 0; term < n ; term ++ )
    {
        cout<<"Enter coefficient and exponenet: ";
        cin>>p1[term].coef;
        cin>>p1[term].expo;
    }
    return( n);
}

/*
-----
This Function is to display the polynomial
Parameter Passing: By reference and value
-----*/
void APEVAL::display(p p1[ ],int n)
{
    int term;
    for (term = 0; term < n-1 ; term ++ )
        cout<<p1[term].coef<<"x^"<<p1[term].expo<<" + ";
        cout<<p1[term].coef<<"x^"<<p1[term].expo;
}

/*-----
Function eval
-----*/
float APEVAL::eval(int n1, p p1[])
{
    int i,sum, x;
    cout<<"\nEnter the value for x for evaluation : ";
    cin>>x;
    sum = 0;
    for(i=0; i < n1;i++ )
        sum= sum+p1[i].coef *pow( x, p1[i].expo);
    return( sum );
}

/*
-----
Function main
-----
*/
int main()
{
    int n1;
    int value;
    APEVAL obj;

```

```

cout<<"\n Enter the Polynomial";
n1 =obj.get_poly(obj.p1);
cout<<"The First polynomial is : \n";
obj.display(obj.p1,n1);
value = obj.eval(n1,obj.p1);
cout<<"\n The Resultant Value of the polynomial is : "<<value<<"\n";
return 0;
}

```

Review Question

1. Explain polynomial representation using arrays with suitable example.

SPPU : May-17, 18, Marks 3

2.10 Sparse Matrix

SPPU : May-17, 19, Dec.-19, Marks 6

- In a matrix, if there are m rows and n columns then the space required to store the numbers will be $m \times n \times s$ where s is the number of bytes required to store the value. Suppose, there are 10 rows and 10 columns and we have to store the integer values then the space complexity will be bytes.

$$10 \times 10 \times 2 = 200 \text{ bytes}$$

(Here 2 bytes are required to store an integer value.)

- It is observed that, many times we deal with matrix of size $m \times n$ and values of m and n are reasonably higher and only a few elements are non zero. Such matrices are called **sparse matrices**.
- **Definition:** Sparse matrix is a matrix containing few non zero elements.
- For example - if the matrix is of size 100×100 and only 10 elements are non zero. Then for accessing these 10 elements one has to make 10000 times scan. Also only 10 spaces will be with non-zero elements remaining spaces of matrix will be filled with zeros only. i.e. we have to allocate the memory of $100 \times 100 \times 2 = 20000$.
- Hence sparse matrix representation is a kind of representation in which only non zero elements along with their rows and columns is stored.

1	2	3
4	5	6
7	8	9

Dense Matrix

1	0	0
0	0	0
0	1	0

Sparse Matrix

2.10.1 Sparse Matrix Representation using Array

- The representation of sparse matrix will be a **triplet** only. That means it stores rows, columns and values.
- The 0th row will store total rows of the matrix, total columns of the matrix, and total non-zero values.
- For example – Suppose a matrix is 6×7 and number of non zero terms are say 8. In our sparse matrix representation the matrix will be stored as

Index	Row No.	Col. No.	Value
0	6	7	8
1	0	6	– 10
2	1	0	55
3	2	5	– 23
4	3	1	67
5	3	6	88
6	4	3	14
7	4	4	– 28
8	5	0	99

Python Program

```
# function display a matrix
def display(matrix):
    for row in matrix:
        for element in row:
            print(element, end = " ")
        print()

# function to convert the matrix
# into a sparse matrix
def convert(matrix):
    SP = []
    # searching values greater
    # than zero
    for i in range(len(matrix)):
        for j in range(len(matrix[0])):
            if matrix[i][j] != 0 :

                # creating a temporary sublist
                temp = []

                # appending row value, column
                # value and element into the
```

```
# sublist
temp.append(i)
temp.append(j)
temp.append(matrix[i][j])

# appending the sublist into
# the sparse matrix list
SP.append(temp)

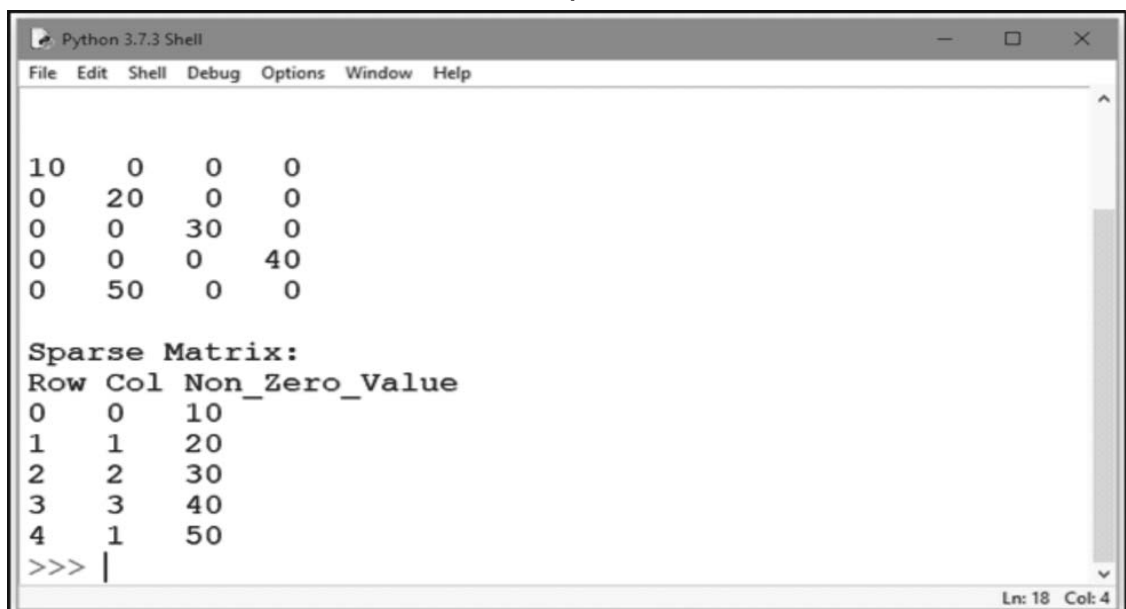
# displaying the sparse matrix
print("\nSparse Matrix: ")
print("Row Col Non_Zero_Value")
display(SP)

# Driver code
#Original Matrix
A = [ [10, 0, 0, 0],
      [0, 20, 0, 0],
      [0, 0, 30, 0],
      [0, 0, 0, 40],
      [0, 50, 0, 0]]

# displaying the matrix
display(A)

# converting the matrix to sparse
convert(A)
```

Output



```
Python 3.7.3 Shell
File Edit Shell Debug Options Window Help

10    0    0    0
0    20    0    0
0    0    30    0
0    0    0    40
0    50    0    0

Sparse Matrix:
Row Col Non_Zero_Value
0    0    10
1    1    20
2    2    30
3    3    40
4    1    50
>>> |

Ln: 18 Col: 4
```

2.10.2 Sparse Matrix Addition

Algorithm :

1. Start
2. Read the two sparse matrices say SP1 and SP2 respectively.
3. The index for SP1 and SP2 will be i and j respectively. Non zero elements are n1 and n2 for SP1 and SP2.
4. The k index will be for sparse matrix SP3 which will store the addition of two matrices.
5. If $SP1[0][0] > SP2[0][0]$ i.e. total number of rows then
$$SP3[0][0] = SP1[0][0]$$
else
$$SP3[0][0] = SP2[0][0]$$
Similarly check in SP1 and SP2 the value of total number of columns which ever value is greater transfer it to SP3.
Set $i = 1, j = 1, k = 1$.
6. When $i < n1$ and $j < n2$.
7. a) If row values of SP1 and SP2 are same then check also column values are same, if so add the non-zero values of SP1 and SP2 and store them in SP3, increment i, j, k by 1.
b) Otherwise if colno of SP1 is less than colno of SP2 copy the row, col and nonzero value of SP1 to SP3. Increment i and k by 1.
c) Otherwise copy the row, col and non-zero value of SP2 to SP3. Increment j and k by 1.
8. When rowno of SP1 is less than SP2 copy row, col and non-zero element of SP1 to SP3. Increment i and k by 1.
9. When rowno of SP2 is less than SP1 copy row, col and non-zero element of SP2 to SP3. Increment j, k by 1. Repeat step 6 to 9 till condition is true.
10. Finally whatever k value that will be total number of non-zero values in SP3 matrix.
 $\therefore SP3[0][2] = k$
11. Print SP3 as a result.
12. Stop.

Python Program

```
def create(s,row_num,col_num,non_zero_values):
    s[0][0]= row_num
    s[0][1] = col_num
    s[0][2] = non_zero_values
    for k in range(1,non_zero_values+1):
        row = int(input("Enter row value: "))
        col = int(input("Enter col value: "))
        element = int(input("Enter the element: "))
        s[k][0]= row
        s[k][1] = col
        s[k][2] = element

def display(s):
    print("Row\tcol\t Non_Zero_values")
    for i in range(0,(s[0][2]+1)):
        for j in range(0,3):
            print(s[i][j], "\t", end="")
        print()

def add(s1,s2):
    i = 1
    j = 1
    k = 1
    s3 = []
    if ((s1[0][0] == s2[0][0]) and (s1[0][1] == s2[0][1])):
        #traversing thru all the terms
        while ((i <= s1[0][2]) and (j <= s2[0][2])):
            if (s1[i][0] == s2[j][0]):
                temp =[]
                if (s1[i][1] == s2[j][1]):
                    temp.append(s1[i][0])
                    temp.append(s1[i][1])
                    temp.append(s1[i][2]+s2[j][2])
                    s3.append(temp)
                    i += 1
                    j += 1
                    k += 1
                elif (s1[i][1]<s2[j][1]):
                    temp.append(s1[i][0])
                    temp.append(s1[i][1])
                    temp.append(s1[i][2])
                    s3.append(temp)
                    i += 1
                    k += 1
            else:
                temp.append(s2[j][0])
```



```
        temp.append(s2[j][1])
        temp.append(s2[j][2])
        s3.append(temp)
        j += 1
        k += 1
    elif (s1[i][0]<s2[j][0]):
        temp = []
        temp.append(s1[i][0])
        temp.append(s1[i][1])
        temp.append(s1[i][2])
        s3.append(temp)
        i += 1
        k += 1
    else:
        temp = []
        temp.append(s1[j][0])
        temp.append(s1[j][1])
        temp.append(s1[j][2])
        s3.append(temp)
        j += 1
        k += 1
#copying remaining terms
while (i <= s1[0][2]): #s1 is greater than s2
    temp = []
    temp.append(s1[i][0])
    temp.append(s1[i][1])
    temp.append(s1[i][2])
    s3.append(temp)
    i += 1
    k += 1
while (j <= s2[0][2]): #s2 is greater than s1
    temp = []
    temp.append(s2[j][0])
    temp.append(s2[j][1])
    temp.append(s2[j][2])
    s3.append(temp)
    j += 1
    k += 1
#assigning total rows, total columns
#and total non zero values as a first row
#in resultant matrix
s3.insert(0,[s1[0][0],s1[0][1],k-1])
else:
    print("\n Addition is not possible")

print(" Addition of Sparse Matrix ...")
print("\nRow Col Non_Zero_values")
```

```
    for row in s3:
        for element in row:
            print(element, end = "  ")
        print()
#Driver Code
row_num1 = int(input("Input total number of rows for first matrix: "))
col_num1 = int(input("Input total number of columns for first matrix: "))
non_zero_values1 = int(input("Input total number of non-zero values: "))
cols = 3
s1 = [[0 for col in range(cols)] for row in range(non_zero_values1+1)]
#creating first sparse matrix
create(s1,row_num1,col_num1,non_zero_values1)
print("First sparse matrix is")
display(s1)

row_num2 = int(input("Input total number of rows for second matrix: "))
col_num2 = int(input("Input total number of columns for second matrix: "))
non_zero_values2 = int(input("Input total number of non-zero values: "))
s2 = [[0 for col in range(cols)] for row in range(non_zero_values2+1)]
#creating second sparse matrix
create(s2,row_num2,col_num2,non_zero_values2)
print("Second sparse matrix is")
display(s2)
#Performing and displaying addition of two sparse matrices
add(s1,s2)
```

Output

```
Input total number of rows for first matrix: 2
Input total number of columns for first matrix: 2
Input total number of non-zero values: 3
Enter row value: 1
Enter col value: 1
Enter the element: 10
Enter row value: 1
Enter col value: 2
Enter the element: 20
Enter row value: 2
Enter col value: 1
Enter the element: 30
First sparse matrix is
Row  col    Non_Zero_values
2    2      3
1    1     10
1    2     20
2    1     30
Input total number of rows for second matrix: 2
Input total number of columns for second matrix: 2
```

Input total number of non-zero values: 2

Enter row value: 2

Enter col value: 1

Enter the element: 5

Enter row value: 2

Enter col value: 2

Enter the element: 40

Second sparse matrix is

Row	col	Non_Zero_values
2	2	2
2	1	5
2	2	40

Addition of Sparse Matrix ...

Row	Col	Non_Zero_values
2	2	4
1	1	10
1	2	20
2	1	35
2	2	40

>>>

C++ code

Following is a C++ function for addition of two sparse matrices

```
void add(int s1[10][3],int s2[10][3],int s3[10][3])
{
    int i, j, k;
    i = 1; j = 1; k = 1;
    if ((s1[0][0] == s2[0][0]) && (s1[0][1] == s2[0][1]))
    {
        s3[0][0] = s1[0][0];
        s3[0][1] = s1[0][1];
        //traversing thru all the terms
        while ((i <= s1[0][2]) && (j <= s2[0][2]))
        {
            if (s1[i][0] == s2[j][0])
            {
                if (s1[i][1] == s2[j][1])
                {
                    s3[k][2] = s1[i][2] + s2[j][2];
                    s3[k][1] = s1[i][1];
                    s3[k][0] = s1[i][0];
                    i++;
                    j++;
                    k++;
                }
                else if (s1[i][1] < s2[j][1])
```

```
        {
            s3[k][2] = s1[i][2];
            s3[k][1] = s1[i][1];
            s3[k][0] = s1[i][0];
            i++;
            k++;
        }
    else
    {
        s3[k][2] = s2[j][2];
        s3[k][1] = s2[j][1];
        s3[k][0] = s2[j][0];
        j++;
        k++;
    }
} //end of 0 if
else if (s1[i][0] < s2[j][0])
{
    s3[k][2] = s1[i][2];
    s3[k][1] = s1[i][1];
    s3[k][0] = s1[i][0];
    i++;
    k++;
}
else
{
    s3[k][2] = s2[j][2];
    s3[k][1] = s2[j][1];
    s3[k][0] = s2[j][0];
    j++;
    k++;
}
} //end of while
//copying remaining terms
while (i <= s1[0][2])
{
    s3[k][2] = s1[i][2];
    s3[k][1] = s1[i][1];
    s3[k][0] = s1[i][0];
    i++;
    k++;
}
while (j <= s2[0][2])
{
    s3[k][2] = s2[j][2];
    s3[k][1] = s2[j][1];
    s3[k][0] = s2[j][0];
```

```

        j++;
        k++;
    }
    s3[0][2] = k - 1;
}
else
    cout<<"\n Addition is not possible";
}

```

2.10.3 Transpose of Sparse Matrix

We will now discuss the algorithm for performing transpose of sparse matrix.

For eg :

Index	Row No.	Col. No.	Value
0	6	7	8
1	0	6	- 10
2	1	0	55
3	2	5	- 23
4	3	1	67
5	3	6	88
6	4	3	14
7	4	4	- 28
8	5	0	99

will become

Index	Row No.	Col. No.	Value
0	7	6	8
1	0	1	55
2	0	5	99
3	1	3	67
4	3	4	14
5	4	4	- 28
6	5	2	- 23
7	6	0	- 10
8	6	3	88

During the transpose of matrix (i) Interchange rows and columns (ii) Also maintain the rows in sorted order.

Python Program

```
row_num = int(input("Input number of rows: "))
col_num = int(input("Input number of columns: "))
non_zero_values = int(input("Input total number of non-zero values: "))
cols = 3
s1 = [[0 for col in range(cols)] for row in range(non_zero_values+1)]
s2 = [[0 for col in range(cols)] for row in range(non_zero_values+1)]

def create(s1,row_num,col_num,cols,non_zero_values):
    s1[0][0]= row_num
    s1[0][1] = col_num
    s1[0][2] = non_zero_values
    for k in range(1,non_zero_values+1):
        row = int(input("Enter row value: "))
        col = int(input("Enter col value: "))
        element = int(input("Enter the element: "))
        s1[k][0]= row
        s1[k][1] = col
        s1[k][2] = element

def display(s,cols,non_zero_values):
    print("Row\tcol\t Non_Zero_values")
    for i in range(0,non_zero_values+1):
        for j in range(0,cols):
            print(s[i][j], "\t", end="")
        print()

def transpose(s1,row_num,col_num,s2,cols,non_zero_values):
    s2[0][0]= col_num
    s2[0][1] = row_num
    s2[0][2] = non_zero_values
    nxt=1
    for c in range(0,col_num):
        # for each column scan all the terms for a 'term' in that column
        for Term in range(1,non_zero_values+1):
            if (s1[Term][1] == c):
                # Interchange Row and Column
                s2[nxt][0] = s1[Term][1]
                s2[nxt][1] = s1[Term][0]
                s2[nxt][2] = s1[Term][2]
                nxt=nxt+1

#Driver Code
create(s1,row_num,col_num,cols,non_zero_values)
print("Original sparse matrix is")
display(s1,cols,non_zero_values)
transpose(s1,row_num,col_num,s2,cols,non_zero_values)
print("Transposed sparse matrix is")
display(s2,cols,non_zero_values)
```

Output

```
Input number of rows: 3
Input number of columns: 3
Input total number of non-zero values: 4
Enter row value: 0
Enter col value: 0
Enter the element: 10
Enter row value: 0
Enter col value: 2
Enter the element: 20
Enter row value: 1
Enter col value: 1
Enter the element: 30
Enter row value: 2
Enter col value: 1
Enter the element: 40
Original sparse matrix is
Row      col      Non_Zero_values
3        3        4
0        0        10
0        2        20
1        1        30
2        1        40
Transposed sparse matrix is
Row      col      Non_Zero_values
3        3        4
0        0        10
1        1        30
1        2        40
2        0        20
```

Logic Explanation :

In above program we look for column values of s1 array starting from 0 to total number of column value and interchange row and column one by one. For instance – Consider the output of above program

s1[0][0] =10, the column value is 0 here, we swap the row and column and copy the column, row and non zero value to s2 array

Locate zero in **Col**, swap row, col and copy it to s2. At the same time copy corresponding non-zero-value to s2

s1

Row	Col	Non_Zero_values
0	0	10
0	2	20
1	1	30
2	1	40

s2

Row	Col	Non_Zero_values
0	0	10

Locate 1 in **Col**, swap row, col and copy it to s2. At the same time copy corresponding non-zero-value to s2

Row	Col	Non_Zero_values
0	0	10
0	2	20
1	1	30
2	1	40

Row	Col	Non_Zero_values
0	0	10
1	1	30

Row	Col	Non_Zero_values
0	0	10
0	2	20
1	1	30
2	1	40

Row	Col	Non_Zero_values
0	0	10
1	1	30
1	2	40

Locate 2 in **Col**, swap row, col and copy it to s2. At the same time copy corresponding non-zero-value to s2

Row	Col	Non_Zero_values
0	0	10
0	2	20
1	1	30
2	1	40

Row	Col	Non_Zero_values
0	0	10
1	1	30
1	2	40
2	1	20

Thus we get transposed matrix in the form of sparse matrix representation.

C++ Code

```
void transp(int s1[size + 1][3], int s2[size + 1][3])
{
    int  nxt, c, Term;
    int  n, m, terms;
/*Read Number of rows, columns and terms of given matrix */
    n = s1[0][0];
    m = s1[0][1];
    terms = s1[0][2];

    /* Interchange Number of rows with number of columns */
    s2[0][0] = m;
    s2[0][1] = n;
    s2[0][2] = terms;
    if (terms > 1) /* if nonzero matrix then */
    {
        nxt = 1; /* Gives next position in the transposed matrix */
        /* Do the transpose columnwise */
        for (c = 0; c <= m; c++)
        {
            /* for each column scan all the terms for a term in that column */
            for (Term = 1; Term <= terms; Term++)
            {
                if (s1[Term][1] == c)
```

```

        {
            /* Interchange Row and Column */
            s2[nxt][0] = s1[Term][1];
            s2[nxt][1] = s1[Term][0];
            s2[nxt][2] = s1[Term][2];
            nxt++;
        }
    }
}

```

Time complexity of simple transpose:

The simple transpose of sparse matrix can be performed using two nested for loops. Hence the time complexity is $O(n^2)$

2.10.4 Fast Transpose

The fast transpose is a transpose method in which matrix transpose operation is performed efficiently. In this method an auxiliary array are used to locate the position of the elements to be transposed sequentially.

Here is an implementation of fast transpose of sparse matrix in Python.

Python Program

```

row_num = int(input("Input number of rows: "))
col_num = int(input("Input number of columns: "))
non_zero_values = int(input("Input total number of non-zero values: "))
cols = 3
s1 = [[0 for col in range(cols)] for row in range(non_zero_values+1)]
s2 = [[0 for col in range(cols)] for row in range(non_zero_values+1)]

def create(s1,row_num,col_num,cols,non_zero_values):
    s1[0][0]= row_num
    s1[0][1] = col_num
    s1[0][2] = non_zero_values
    for k in range(1,non_zero_values+1):
        row = int(input("Enter row value: "))
        col = int(input("Enter col value: "))
        element = int(input("Enter the element: "))
        s1[k][0]= row
        s1[k][1] = col
        s1[k][2] = element

def display(s,cols,non_zero_values):
    print("Row\tcol\t Non_Zero_values")
    for i in range(0,non_zero_values+1):

```

```
    for j in range(0,cols):
        print(s[i][j], "\t", end="")
    print()

def transpose(s1,row_num,col_num,s2,cols,non_zero_values):
    s2[0][0]= col_num
    s2[0][1] = row_num
    s2[0][2] = non_zero_values
    rterm = []
    rpos = []
    if non_zero_values > 0:
        for i in range(col_num):
            rterm.insert(i,0)
        for i in range(1,non_zero_values+1):
            #rterm[s1[i][1]]++
            index = s1[i][1]
            val = rterm.pop(index)
            rterm.insert(index,val+1)
        rpos.insert(0,1)
        for i in range(1,col_num+1):
            #rpos[i]=rpos[i-1]+ rterm[(i - 1)]
            rpos_val=rpos[i-1]
            rterm_val=rterm[i-1]
            rpos.insert(i,(rpos_val+rterm_val))
        for i in range(1,non_zero_values+1):
            j = rpos[s1[i][1]]
            s2[j][0] = s1[i][1]
            s2[j][1] = s1[i][0]
            s2[j][2] = s1[i][2]
            rpos[s1[i][1]] = j + 1
#Driver Code
create(s1,row_num,col_num,cols,non_zero_values)
print("Original sparse matrix is")
display(s1,cols,non_zero_values)
transpose(s1,row_num,col_num,s2,cols,non_zero_values)
print("Transposed sparse matrix is")
display(s2,cols,non_zero_values)
```

Output

```
Input number of columns: 3
Input total number of non-zero values: 4
Enter row value: 0
Enter col value: 1
Enter the element: 10
Enter row value: 1
Enter col value: 0
Enter the element: 20
Enter row value: 2
Enter col value: 0
Enter the element: 30
Enter row value: 2
Enter col value: 1
Enter the element: 40
Original sparse matrix is
Row      col      Non_Zero_values
3         3         4
0         1        10
1         0        20
2         0        30
2         1        40
Transposed sparse matrix is
Row      col      Non_Zero_values
3         3         4
0         1        20
0         2        30
1         0        10
1         2        40
>>> |
```

Logic for Fast Transpose

Consider the sparse matrix representative as

S1

Index	Row	Col	Non-Zero values
0	3	3	4
1	0	1	10
2	1	0	20
3	2	0	30
4	2	1	40

We will first consider one dimensional array, named **rterm[]**. In this array we will store non zero terms present in each column.

At 0th column there are two non zero terms. At 1st column also there are two non zero terms but there is non zero term in 2nd column.

rterm	
0	2
1	2
2	0

Similarly we will take another one dimensional array named **rpos[]**. We initialize 0th location of **rpos[]** by 1. so,

rpos	
0	1

Now we use following formula to fill up the **rpos** array.

$$\text{rpos}[i] = \text{rpos}[i - 1] + \text{rterm}[i - 1]$$

$$i = 1$$

$$\begin{aligned} \text{rpos}[1] &= \text{rpos}[0] + \text{rterm}[0] \\ &= 1 + 2 \end{aligned}$$

$$\text{rpos} = 3$$

$$i = 2$$

$$\begin{aligned} \text{rpos}[2] &= \text{rpos}[1] + \text{rterm}[1] \\ &= 3 + 2 \end{aligned}$$

$$\text{rpos}[2] = 5$$

$$i = 3$$

$$\begin{aligned} \text{rpos}[3] &= \text{rpos}[2] + \text{rterm}[2] \\ &= 5 + 0 = 5 \end{aligned}$$

$$\text{rpos}[3] = 5$$

rpos	
0	1
1	3
2	5
3	5

Now we will read values of S1 array from 1 to 4.

We must find the triplet from S1 array which is at index 1. it is (0, 1, 10)

index	row	col	value
1	0	1	10

S1

Just interchange
row and col.

row	col	value
1	0	10

↑
Since value is 1,
check rpos[1]

rpos[1] points to value 3. That means place the triplet (1, 0, 10) at index 3 in S2 array.

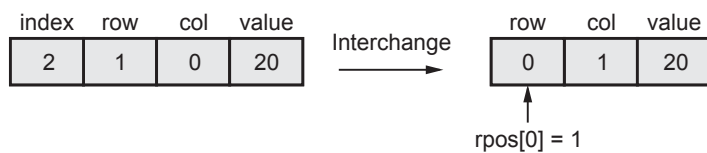
S2			
	Row	Col	Non-Zero values
0			
1			
2			
3	1	0	10
4			

Now since rpos[1] is read just now, increment rpos[1] value by 1.

Hence

rpos	
0	1
1	3 4
2	5
3	5

Read next element from S1 array



That mean place triplet (0, 1, 20) at index 1 in S2 array

S2

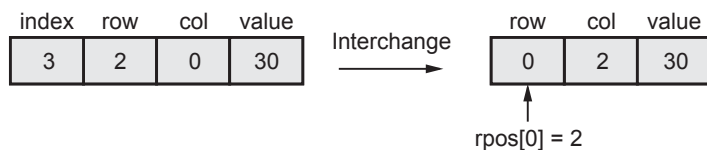
	Row	Col	Non-Zero values
0			
1	0	1	20
2			
3	1	0	10
4			

Since rpos[0] is read just now, increment rpos[0] by 1.

rpos

0	2
1	4
2	5
3	5

Read next element from S1 array



That means place triplet (0, 2, 30) at index 2 in S2 array.

S2

	Row	Col	Non-Zero values
0			
1	0	1	20
2	0	2	30
3	1	0	10
4			

Since rpos[0] is read just now increment rpos[0] by 1.

rpos

0	2	3
1	4	
2	5	
3	5	

Read next element from S1 array

index	row	col	value
4	2	1	40

Interchange →

row	col	value
1	2	40

↑
rpos[1] = 4

That means place triple (1, 2, 40) at index 2 in S2 array.

S2

	Row	Col	Non-Zero values
0			
1	0	1	20
2	0	2	30
3	1	0	10
4	1	2	40

Thus we get transposed sparse matrix.

Finally we fill up S2[0] by total number of rows, total number of columns and total number of non-zero values.

Thus we get

	Row	Col	Non-Zero values
0	3	3	4
1	0	1	20
2	0	2	30
3	1	0	10
4	1	2	40

C++ Code

```
void trans (int s1[max1][3],int s2[max1][3] )
{
    int rterm[max1],rpos[max1];
    int j,i ;
    int row,col,num ;

    row= s1[0][0];
    col= s1[0][1];
    num= s1[0][2];

    s2[0][0] = col;
    s2[0][1] = row;
    s2[0][2] = num;
    if ( num > 0 )
    {
        for ( i = 0; i <= col ; i ++ )
            rterm[i] = 0;
        for ( i = 1; i <= num ; i ++ )
            rterm[s1[i][1]] ++;
        rpos[0] = 1; /*setting the rowwise position*/
        for ( i = 1; i <= col; i ++ )
            rpos[i]=rpos[i-1]+ rterm[(i - 1)];
        for ( i = 1; i <= num ; i ++ )
        {
            j = rpos[s1[i][1]];
            s2[j][0] = s1[i][1];
            s2[j][1] = s1[i][0];
            s2[j][2] = s1[i][2];
            rpos[s1[i][1]] = j + 1;
        }
    }
}
```

Time Complexity of Fast Transpose

For transposing the elements using simple transpose method we need two nested for loops but in case of fast transpose we are determining the position of the elements that get transposed using only one for loop. Hence the time complexity of fast transpose is $O(n)$

Review Questions

1. Explain fast transpose of sparse matrix with suitable example. Discuss time complexity of fast transpose. **SPPU : May-17, Marks 6**
2. Write pseudo code to perform simple transpose of sparse matrix. **SPPU : May-19, Marks 4**
3. What is sparse matrix ? Explain its representation with an example. **SPPU : May-19, Marks 4**
4. Write a pseudo code to perform the simple transpose of sparse matrix. Also discuss the time complexity. **SPPU : Dec.-19, Marks 6**

2.11 Time and Space Tradeoff

Basic concept : Time space trade-off is basically a situation where either a space efficiency (memory utilization) can be achieved at the cost of time or a time efficiency (performance efficiency) can be achieved at the cost of memory.

Example 1 : Consider the programs like compilers in which symbol table is used to handle the variables and constants. Now if entire symbol table is stored in the program then the time required for searching or storing the variable in the symbol table will be reduced but memory requirement will be more. On the other hand, if we do not store the symbol table in the program and simply compute the table entries then memory will be reduced but the processing time will be more.

Example 2 : Suppose, in a file, if we store the uncompressed data then reading the data will be an efficient job but if the compressed data is stored then to read such data more time will be required.

Example 3 : This is an example of reversing the order of the elements. That is, the elements are stored in an ascending order and we want them in the descending order. This can be done in two ways -

- We will use another array **b[]** in which the elements in descending order can be arranged by reading the array **a[]** in reverse direction. This approach will actually increase the memory but time will be reduced.
- We will apply some extra logic for the same array **a[]** to arrange the elements in descending order. This approach will actually reduce the memory but time of execution will get increased.

The illustration of above mentioned example is given by following C program -

C Program

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int a[10],b[10],n,mid,i,j,temp;
    clrscr();
    printf("\n How many elements do you want? ");
    scanf("%d",&n);
    printf("\n Enter the elements in the ascending order ");
    for(i=0;i<n;i++)
    {
        printf("\n Enter the element ");
        scanf("%d",&a[i]);
    }
    printf("\nThe elements in descending order(By Method 1) are ...\n");
    j=0;
    for(i=n-1;i>=0;i--)//    reading the array in    reverse direction
    {
        b[j]=a[i];//storing them in another array
        j++;
    }
    for(j=0;j<n;j++)
        printf(" %d",b[j]);

    printf("\nThe elements in descending
order(By Method 2) are ...\n");
    mid=n/2;
    for(i=0;i<mid;i++)
    {
        j=(n-1)-i;
        temp=a[i];
        a[i]=a[j];
        a[j]=temp;
    }
    for(i=0;i<n;i++)
        printf(" %d",a[i]);
    getch();
}
```

This method makes use of some **computations** for storing the elements in descending order. It makes use of the same array. Hence **less amount of space** is needed at the cost of **addition computational time**.

Output (Run 1)

How many elements do you want? 5

Enter the elements in the ascending order

Enter the element 11

```
Enter the element 22
```

```
Enter the element 33
```

```
Enter the element 44
```

```
Enter the element 55
```

```
The elements in descending order(By Method 1) are ...
```

```
55 44 33 22 11
```

```
The elements in descending order(By Method 2) are ...
```

```
55 44 33 22 11
```

Output (Run 2)

```
How many elements do you want? 6
```

```
Enter the elements in the ascending order
```

```
Enter the element 11
```

```
Enter the element 22
```

```
Enter the element 33
```

```
Enter the element 44
```

```
Enter the element 55
```

```
Enter the element 66
```

```
The elements in descending order(By Method 1) are ...
```

```
66 55 44 33 22 11
```

```
The elements in descending order(By Method 2) are ...
```

```
66 55 44 33 22 11
```

Thus time space trade-off is a situation of compensating one performance measure at the cost of another.

