LECTURE NOTES
ON
DATA STRUCTURES USING C (BCA-204)

For
BCA 1st year

by

Dr. Himanshu Pandey
Assistant Professor
Department of Computer Science and Engineering

FACULTY OF ENGINEERING & TECHNOLOGY
University of Lucknow
LUCKNOW, UTTAR PRADESH

2019-2020

CONTENTS

CHAPTER 1 BASIC CONCEPTS

1.1 Introduction to Data Structures
1.2 Data structures: organizations of data
1.3. Abstract Data Type (ADT)
1.4. Selecting a data structure to match the operation
1.5. Algorithm
1.6. Practical Algorithm design issues
1.7. Performance of a program
1.8. Classification of Algorithms
1.9. Complexity of Algorithms

CHAPTER 2 RECURSION

2.1. Introduction to Recursion
2.2. Differences between recursion and iteration
2.3. Factorial of a given number
2.4. The Towers of Hanoi
2.5. Fibonacci Sequence Problem
2.6. Program using recursion to calculate the NCR of a given number
2.7. Program to calculate the least common multiple of a given number
2.8. Program to calculate the greatest common divisor
Exercises
Multiple Choice Questions

CHAPTER 3 LINKED LISTS

3.1. Linked List Concepts
3.2. Types of Linked Lists
3.3. Single Linked List
3.3.1. Source Code for the Implementation of Single Linked List
3.4. Using a header node
3.5. Array based linked lists
3.6. Double Linked List
3.6.1. A Complete Source Code for the Implementation of Double Linked List
3.7. Circular Single Linked List
3.7.1. Source Code for Circular Single Linked List
3.8. Circular Double Linked List
3.8.1. Source Code for Circular Double Linked List
3.9. Comparison of Linked List Variations
Exercise
Multiple Choice Questions

CHAPTER 4 STACK AND QUEUE

4.1. Stack
    4.1.1. Representation of Stack
    4.1.2. Program to demonstrate a stack, using array
    4.1.3. Program to demonstrate a stack, using linked list
4.2. Algebraic Expressions
4.3. Converting expressions using Stack
    4.3.1. Conversion from infix to postfix
    4.3.2. Program to convert an infix to postfix expression
    4.3.3. Conversion from infix to prefix
    4.3.4. Program to convert an infix to prefix expression
    4.3.5. Conversion from postfix to infix
    4.3.6. Program to convert postfix to infix expression
    4.3.7. Conversion from postfix to prefix
    4.3.8. Program to convert postfix to prefix expression
    4.3.9. Conversion from prefix to infix
    4.3.10. Program to convert prefix to infix expression
    4.3.11. Conversion from prefix to postfix
    4.3.12. Program to convert prefix to postfix expression
4.4. Evaluation of postfix expression
4.5. Applications of stacks
4.6. Queue
    4.6.1. Representation of Queue
    4.6.2. Program to demonstrate a Queue using array
    4.6.3. Program to demonstrate a Queue using linked list
4.7. Applications of Queue
4.8. Circular Queue
    4.8.1. Representation of Circular Queue
4.9. Deque
4.10. Priority Queue
    Exercises
    Multiple Choice Questions

CHAPTER 5 BINARY TREES

5.1. Trees
5.2. Binary Tree
5.3. Binary Tree Traversal Techniques
    5.3.1. Recursive Traversal Algorithms
    5.3.2. Building Binary Tree from Traversal Pairs
    5.3.3. Binary Tree Creation and Traversal Using Arrays
    5.3.4. Binary Tree Creation and Traversal Using Pointers
    5.3.5. Non Recursive Traversal Algorithms
5.4. Expression Trees
    5.4.1. Converting expressions with expression trees
5.5. Threaded Binary Tree
5.6. Binary Search Tree
5.7. AVL Tree

5.8. Search and Traversal Techniques for m-ary trees
    5.8.1. Depth first search
    5.8.2. Breadth first search
5.9. Sparse Matrices
    Exercises
    Multiple Choice Questions

CHAPTER 6 GRAPHS

6.1. Introduction to Graphs
6.2. Representation of Graphs
6.3. Minimum Spanning Tree
6.3.1. Kruskal's Algorithm
6.3.2. Prim's Algorithm
6.4. Reachability Matrix
6.5. Traversing a Graph
6.5.1. Breadth first search and traversal
6.5.2. Depth first search and traversal
Exercises
Multiple Choice Questions

CHAPTER 7 SEARCHING AND SORTING

7.1. Linear Search
7.1.1. A non-recursive program for Linear Search
7.1.1. A Recursive program for linear search
7.2. Binary Search
7.1.2. A non-recursive program for binary search
7.1.3. A recursive program for binary search
7.3. Bubble Sort
7.3.1. Program for Bubble Sort
7.4. Selection Sort
7.4.1 Non-recursive Program for selection sort
7.4.2. Recursive Program for selection sort
7.5. Quick Sort
7.5.1. Recursive program for Quick Sort
Exercises
Multiple Choice Questions

Chapter 1

Basic Concepts

The term data structure is used to describe the way data is stored, and the term algorithm is used to describe the way data is processed. Data structures and algorithms are interrelated. Choosing a data structure affects the kind of algorithm you might use, and choosing an algorithm affects the data structures we use.

An Algorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions.

1.1. Introduction to Data Structures:

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structure is used to describe the way data is stored.

To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as:

Algorithm + Data structure = Program

A data structure is said to be *linear* if its elements form a sequence or a linear list. The linear data structures like an array, stacks, queues and linked lists organize data in linear order. A data structure is said to be *non linear* if its elements form a hierarchical classification where, data items appear at various levels.

Trees and Graphs are widely used non-linear data structures. Tree and graph structures represents hierarchial relationship between individual data elements. Graphs are nothing but trees with certain restrictions removed.

Data structures are divided into two types:

- Primitive data structures.
- Non-primitive data structures.

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers. Integers, floating point numbers, character constants, string constants and pointers come under this category.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item. Arrays, lists and files come under this category. Figure 1.1 shows the classification of data structures.

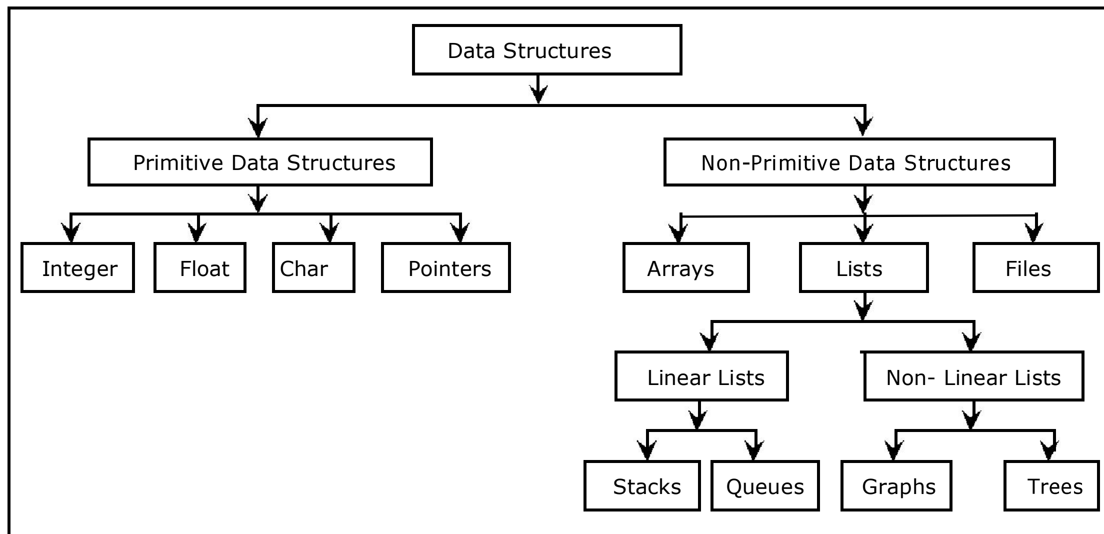

Figure 1. 1. Classification of Data Structures

1.2. Data structures: Organization of data

The collection of data you work with in a program have some kind of structure or organization. No matte how complex your data structures are they can be broken down into two fundamental types:

- Contiguous
- Non-Contiguous.

In contiguous structures, terms of data are kept together in memory (either RAM or in a file). An array is an example of a contiguous structure. Since each element in the array is located next to one or two other elements. In contrast, items in a non-contiguous structure and scattered in memory, but we linked to each other in some way. A linked list is an example of a non-contiguous data structure. Here, the nodes of the list are linked together using pointers stored in each node. Figure 1.2 below illustrates the difference between contiguous and non-contiguous structures.

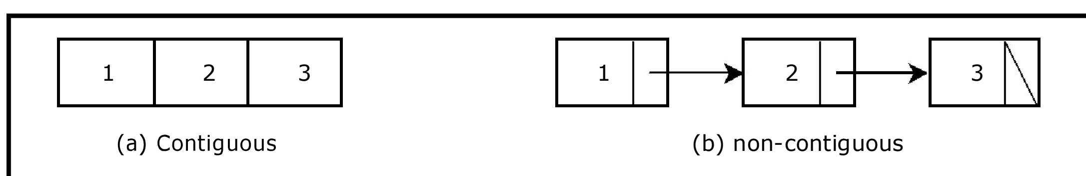

Figure 1.2 Contiguous and Non-contiguous structures compared

Contiguous structures:

Contiguous structures can be broken drawn further into two kinds: those that contain data items of all the same size, and those where the size may differ. Figure 1.2 shows example of each kind. The first kind is called the array. Figure 1.3(a) shows an example of an array of numbers. In an array, each element is of the same type, and thus has the same size.

The second kind of contiguous structure is called structure, figure 1.3(b) shows a simple structure consisting of a person's name and age. In a struct, elements may be of different data types and thus may have different sizes.

For example, a person's age can be represented with a simple integer that occupies two bytes of memory. But his or her name, represented as a string of characters, may require many bytes and may even be of varying length.

Couples with the atomic types (that is, the single data-item built-in types such as integer, float and pointers), arrays and structs provide all the "mortar" you need to built more exotic form of data structure, including the non-contiguous forms.

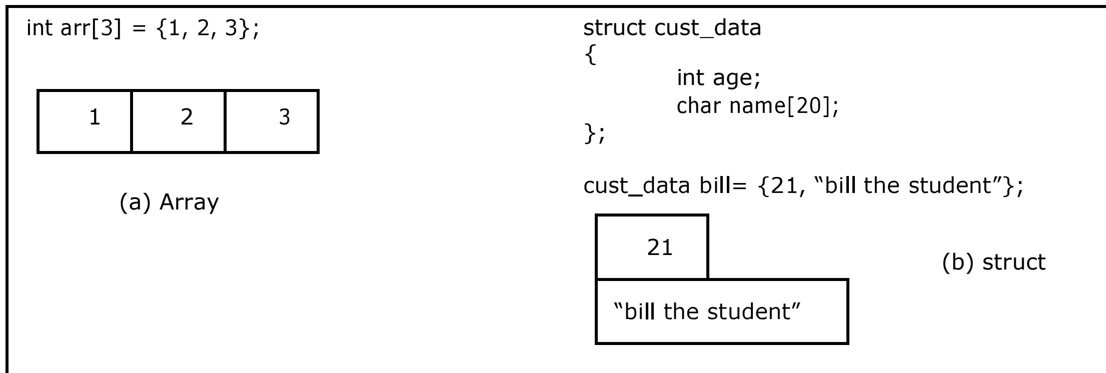

Figure 1.3 Examples of contiguous structures.

Non-contiguous structures:

Non-contiguous structures are implemented as a collection of data-items, called nodes, where each node can point to one or more other nodes in the collection. The simplest kind of non-contiguous structure is linked list.

A linked list represents a linear, one-dimension type of non-contiguous structure, where there is only the notation of backwards and forwards. A tree such as shown in figure 1.4(b) is an example of a two-dimensional non-contiguous structure. Here, there is the notion of up and down and left and right.

In a tree each node has only one link that leads into the node and links can only go down the tree. The most general type of non-contiguous structure, called a graph has no such restrictions. Figure 1.4(c) is an example of a graph.

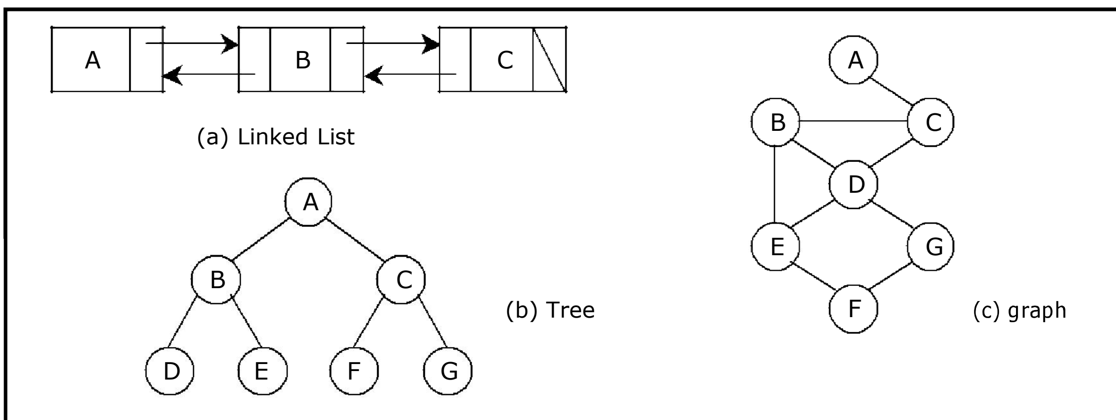

Figure 1.4. Examples of non-contiguous structures

Hybrid structures:

If two basic types of structures are mixed then it is a hybrid form. Then one part contiguous and another part non-contiguous. For example, figure 1.5 shows how to implement a double–linked list using three parallel arrays, possibly stored a past from each other in memory.

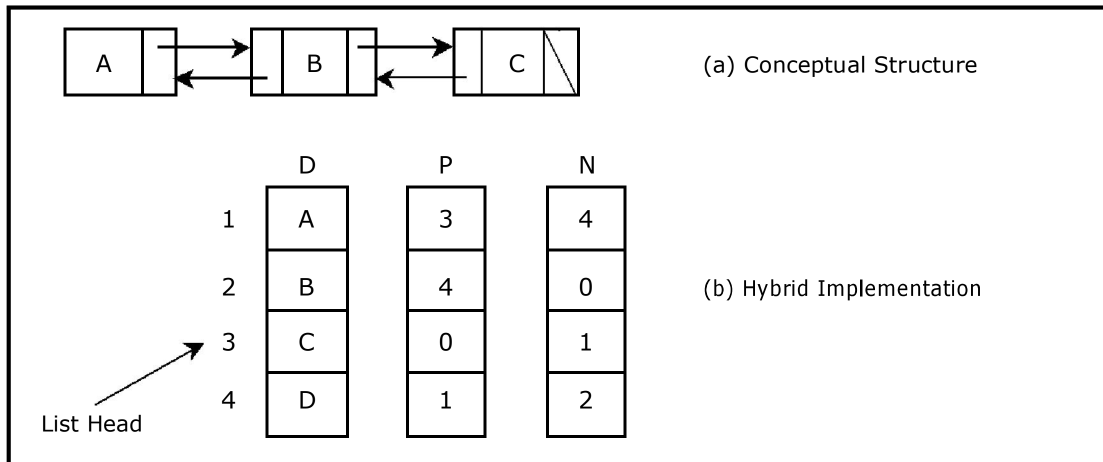

Figure 1.5. A double linked list via a hybrid data structure

The array D contains the data for the list, whereas the array P and N hold the previous and next "pointers". The pointers are actually nothing more than indexes into the D array. For instance, D[i] holds the data for node i and p[i] holds the index to the node previous to i, where may or may not reside at position i–1. Like wise, N[i] holds the index to the next node in the list.

1.3. Abstract Data Type (ADT):

The design of a data structure involves more than just its organization. You also need to plan for the way the data will be accessed and processed – that is, how the data will be interpreted actually, non-contiguous structures – including lists, tree and graphs – can be implemented either contiguously or non- contiguously like wise, the structures that are normally treated as contiguously - arrays and structures – can also be implemented non-contiguously.

The notion of a data structure in the abstract needs to be treated differently from what ever is used to implement the structure. The abstract notion of a data structure is defined in terms of the operations we plan to perform on the data.

Considering both the organization of data and the expected operations on the data, leads to the notion of an abstract data type. An *abstract data type* in a theoretical construct that consists of data as well as the operations to be performed on the data while hiding implementation.

For example, a stack is a typical abstract data type. Items stored in a stack can only be added and removed in certain order – the last item added is the first item removed. We call these operations, pushing and popping. In this definition, we haven't specified have items are stored on the stack, or how the items are pushed and popped. We have only specified the valid operations that can be performed.

For example, if we want to read a file, we wrote the code to read the physical file device. That is, we may have to write the same code over and over again. So we created what is known

today as an ADT. We wrote the code to read a file and placed it in a library for a programmer to use.

As another example, the code to read from a keyboard is an ADT. It has a data structure, character and set of operations that can be used to read that data structure.

To be made useful, an abstract data type (such as stack) has to be implemented and this is where data structure comes into ply. For instance, we might choose the simple data structure of an array to represent the stack, and then define the appropriate indexing operations to perform pushing and popping.

1.4. Selecting a data structure to match the operation:

The most important process in designing a problem involves choosing which data structure to use. The choice depends greatly on the type of operations you wish to perform.

Suppose we have an application that uses a sequence of objects, where one of the main operations is delete an object from the middle of the sequence. The code for this is as follows:

```
void delete (int *seg, int &n, int posn)
// delete the item at position from an array of n elements.
{
        if (n)
        {
                 int i=posn;
                 n--;
                while (i < n)
                {
                        seq[i] = seg[i+1];
                        i++;
                }
        }
        return;
}
```

This function shifts towards the front all elements that follow the element at position *posn*. This shifting involves data movement that, for integer elements, which is too costly. However, suppose the array stores larger objects, and lots of them. In this case, the overhead for moving data becomes high. The problem is that, in a contiguous structure, such as an array the logical ordering (the ordering that we wish to interpret our elements to have) is the same as the physical ordering (the ordering that the elements actually have in memory).

If we choose non-contiguous representation, however we can separate the logical ordering from the physical ordering and thus change one without affecting the other. For example, if we store our collection of elements using a double–linked list (with previous and next pointers), we can do the deletion without moving the elements, instead, we just modify the pointers in each node. The code using double linked list is as follows:

```
void delete (node * beg, int posn)
//delete the item at posn from a list of elements.
{
        int i = posn;
        node *q = beg;
        while (i && q)
        {
```

```
                i--;
                q = q -> next;
        }

        if (q)
        {/* not at end of list, so detach P by making previous and
                        next nodes point to each other */
                node *p = q -> prev;
                node *n = q -> next;
                if (p)
                        p -> next = n;
                if (n)
                        n -> prev = P;
        }
        return;
}
```

The process of detecting a node from a list is independent of the type of data stored in the node, and can be accomplished with some pointer manipulation as illustrated in figure below:

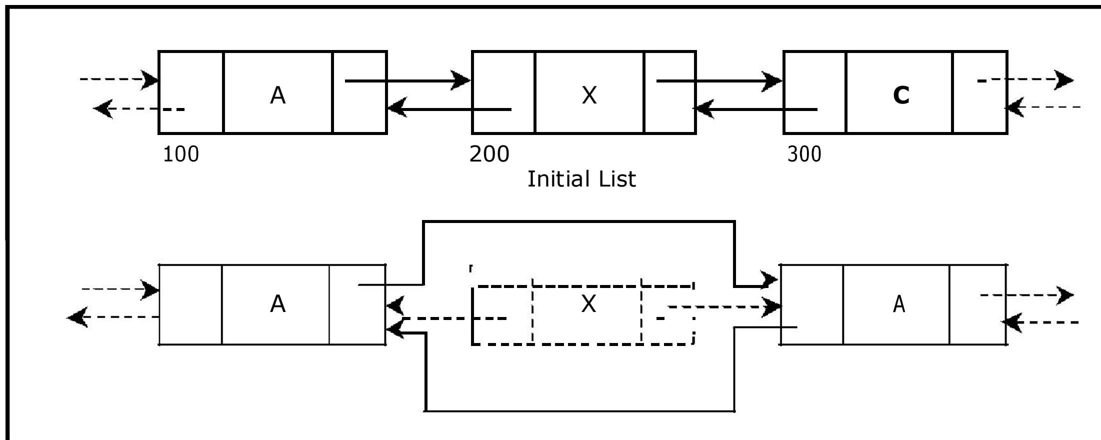

Figure 1.6 Detaching a node from a list

Since very little data is moved during this process, the deletion using linked lists will often be faster than when arrays are used.

It may seem that linked lists are superior to arrays. But is that always true? There are trade offs. Our linked lists yield faster deletions, but they take up more space because they require two extra pointers per element.

1.5. Algorithm

An **a**lgorithm is a finite sequence of instructions, each of which has a clear meaning and can be performed with a finite amount of effort in a finite length of time. No matter what the input values may be, an algorithm terminates after executing a finite number of instructions. In addition every algorithm must satisfy the following criteria:

Input: there are zero or more quantities, which are externally supplied;

Output: at least one quantity is produced;

Definiteness: each instruction must be clear and unambiguous;

Finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;

Effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

In formal computer science, one distinguishes between an algorithm, and a program. A program does not necessarily satisfy the fourth condition. One important example of such a program for a computer is its operating system, which never terminates (except for system crashes) but continues in a wait loop until more jobs are entered.

We represent an algorithm using pseudo language that is a combination of the constructs of a programming language together with informal English statements.

1.6. Practical Algorithm design issues:

Choosing an efficient algorithm or data structure is just one part of the design process. Next, will look at some design issues that are broader in scope. There are three basic design goals that we should strive for in a program:

1. Try to save time (Time complexity).
2. Try to save space (Space complexity).
3. Try to have face.

A program that runs faster is a better program, so saving time is an obvious goal. Like wise, a program that saves space over a competing program is considered desirable. We want to "save face" by preventing the program from locking up or generating reams of garbled data.

1.7. Performance of a program:

The performance of a program is the amount of computer memory and time needed to run a program. We use two approaches to determine the performance of a program. One is analytical, and the other experimental. In performance analysis we use analytical methods, while in performance measurement we conduct experiments.

Time Complexity:

The time needed by an algorithm expressed as a function of the size of a problem is called the ***TIME COMPLEXITY*** of the algorithm. The time complexity of a program is the amount of computer time it needs to run to completion.

The limiting behavior of the complexity as size increases is called the asymptotic time complexity. It is the asymptotic complexity of an algorithm, which ultimately determines the size of problems that can be solved by the algorithm.

Space Complexity:

The space complexity of a program is the amount of memory it needs to run to completion. The space need by a program has the following components:

Instruction space: Instruction space is the space needed to store the compiled version of the program instructions.

Data space: Data space is the space needed to store all constant and variable values. Data space has two components:

- Space needed by constants and simple variables in program.
- Space needed by dynamically allocated objects such as arrays and class instances.

Environment stack space: The environment stack is used to save information needed to resume execution of partially completed functions.

Instruction Space: The amount of instructions space that is needed depends on factors such as:

- The compiler used to complete the program into machine code.
- The compiler options in effect at the time of compilation
- The target computer.

1.8. Classification of Algorithms

If 'n' is the number of data items to be processed or degree of polynomial or the size of the file to be sorted or searched or the number of nodes in a graph etc.

1 Next instructions of most programs are executed once or at most only a few times. If all the instructions of a program have this property, we say that its running time is a constant.

Log n When the running time of a program is logarithmic, the program gets slightly slower as n grows. This running time commonly occurs in programs that solve a big problem by transforming it into a smaller problem, cutting the size by some constant fraction., When n is a million, log n is a doubled whenever n doubles, log n increases by a constant, but log n does not double until n increases to n^2.

n When the running time of a program is linear, it is generally the case that a small amount of processing is done on each input element. This is the optimal situation for an algorithm that must process n inputs.

n. log n This running time arises for algorithms but solve a problem by breaking it up into smaller sub-problems, solving them independently, and then combining the solutions. When n doubles, the running time more than doubles.

n^2 When the running time of an algorithm is quadratic, it is practical for use only on relatively small problems. Quadratic running times typically arise in algorithms that process all pairs of data items (perhaps in a double nested loop) whenever n doubles, the running time increases four fold.

n^3 Similarly, an algorithm that process triples of data items (perhaps in a triple–nested loop) has a cubic running time and is practical for use only on small problems. Whenever n doubles, the running time increases eight fold.

2^n Few algorithms with exponential running time are likely to be appropriate for practical use, such algorithms arise naturally as "brute–force" solutions to problems. Whenever n doubles, the running time squares.

1.9. Complexity of Algorithms

The complexity of an algorithm M is the function f(n) which gives the running time and/or storage space requirement of the algorithm in terms of the size 'n' of the input data. Mostly, the storage space required by an algorithm is simply a multiple of the data size 'n'. Complexity shall refer to the running time of the algorithm.

The function f(n), gives the running time of an algorithm, depends not only on the size 'n' of the input data but also on the particular data. The complexity function f(n) for certain cases are:

1. Best Case : The minimum possible value of f(n) is called the best case.
2. Average Case : The expected value of f(n).
3. Worst Case : The maximum value of f(n) for any key possible input.

The field of computer science, which studies efficiency of algorithms, is known as analysis of algorithms.

Algorithms can be evaluated by a variety of criteria. Most often we shall be interested in the rate of growth of the time or space required to solve larger and larger instances of a problem. We will associate with the problem an integer, called the size of the problem, which is a measure of the quantity of input data.