CHAPTER-1

The role of ALGORITHM in computing

This chapter holds on the topics of alogorithm,its importance and its role .Now I am going to know about it to you in short time .

Algorithms and data structures :

This course will examine various data structures for storing and accessing information together with relationships between the items being stored, and algorithms for efficiently finding solutions to various problems, both relative to the data structures and queries and operations based on the relationships between the items stored.

To know all about it at first we have to understand what is alogorithm.Lets start with the definition of alogorithm.

1.ALGORITHM:

An alogorithm is thus a sequence of computational steps that transform the input into the output.

2. Algorithm analysis:

We describe containers to store items, relationships between items we may also want to record, the concept of abstract data types, data structures and algorithms that will implement these structures and solve problems, and the asymptotic analysis we will use to analyze our algorithms.

Lets know about algo with a sorting problem : Sorting is by no means the only computational problem for which alogorithms have been developed.

INPUT:A sequence of n numbers(a1,a2,…..,an).

OUTPUT:A permutation(reordering )(a1’,a2’,……an’) of the input sequence such that a1’<=a2’…..<=an’.

Data Structure : is a systematic way to organize data in order to use it efficiently. Following terms are the foundation terms of a data structure.

* **Interface** − Each data structure has an interface. Interface represents the set of operations that a data structure supports. An interface only provides the list of supported operations, type of parameters they can accept and return type of these operations.
* **Implementation** − Implementation provides the internal representation of a data structure. Implementation also provides the definition of the algorithms used in the operations of the data structure.

Characteristics of a Data Structure

* **Correctness** − Data structure implementation should implement its interface correctly.
* **Time Complexity** − Running time or the execution time of operations of data structure must be as small as possible.
* **Space Complexity** − Memory usage of a data structure operation should be as little as possible.

Need for Data Structure

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

* **Data Search** − Consider an inventory of 1 million(106) items of a store. If the application is to search an item, it has to search an item in 1 million(106) items every time slowing down the search. As data grows, search will become slower.
* **Processor speed** − Processor speed although being very high, falls limited if the data grows to billion records.
* **Multiple requests** − As thousands of users can search data simultaneously on a web server, even the fast server fails while searching the data.

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

Execution Time Cases

There are three cases which are usually used to compare various data structure's execution time in a relative manner.

* **Worst Case** − This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's worst case time is ƒ(n) then this operation will not take more than ƒ(n) time where ƒ(n) represents function of n.
* **Average Case** − This is the scenario depicting the average execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution, then m operations will take mƒ(n) time.
* **Best Case** − This is the scenario depicting the least possible execution time of an operation of a data structure. If an operation takes ƒ(n) time in execution, then the actual operation may take time as the random number which would be maximum as ƒ(n).

Basic Terminology

* **Data** − Data are values or set of values.
* **Data Item** − Data item refers to single unit of values.
* **Group Items** − Data items that are divided into sub items are called as Group Items.
* **Elementary Items** − Data items that cannot be divided are called as Elementary Items.
* **Attribute and Entity** − An entity is that which contains certain attributes or properties, which may be assigned values.
* **Entity Set** − Entities of similar attributes form an entity set.
* **Field** − Field is a single elementary unit of information representing an attribute of an entity.
* **Record** − Record is a collection of field values of a given entity.
* **File** − File is a collection of records of the entities in a given entity set.

CHAPTER-2

This chapter will familiarize you with the framework we shall use throught the book to think about the design and analysis of algorithms.

A Sorting Algorithm is used to rearrange a given array or list elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of element in the respective data structure.

**For example**: The below list of characters is sorted in increasing order of their ASCII values. That is, the character with lesser ASCII value will be placed first than the character with higher ASCII value.

Input : g e e k f o

Output : e e f g o

Lets start with insertion sort…

Insertion Sort

Algorithm

for i = 0 to size-1 do

temp = data[i]

x = first location from 0 to i with a value greater or equal to temp

shift all values from x to i-1 one location forwards

data[x] = temp

end

Complexity

 Interesting operations: comparison and shift

 i-th step performs i comparison and shift operations 

Total cost : 1 + 2 + ... + (n-1) + n = n\*(n+1)/2. 

Algorithm is O(n2).

Algorithm analysis Determining the run time of code requires us to consider the various components. All operators in C++ run in Θ(1) time. If two blocks of code run in O(f(n)) and O(g(n)) time, respectively, if they are run in series, the run time is O(f(n) + g(n)). Therefore, any finite and fixed set of operators run serially may also be said to run in Θ(1) time. A loop that cycles n times will run in Θ(n) time if the body is Θ(1), but if there is the possibility of finishing early, we would say it runs in O(n) time. If the body of the loop also has a run time depending on n, the run time is O(n f(n)). If, however, the body of the loop iterates over the sequence a ≤ k ≤ b, and the run time of the body depends on k, say O(f(k)), the run time may be calculated as O( ( )) b k a f k ∑ = . If a function is called and we are not aware of its run time, we may represent it by a placeholder T. If we are aware that the run time depends on a parameter, we may write T(n). In some cases, we may simply determine that the run time of a function S(n) = O(n) + T(n). For example, one function may iterate through an array of size n and then call another function. When a function calls itself, however, we call that a recursive function. For example, T(n) = T(n – 1) + Θ(1) or S(n) = S(n/2) + Θ(1) when n > 1. In general, we assume that T(1) = Θ(1); that is, the time it takes to solve a trivial sized problem is Θ(1). In the case of these two examples, we can solve the recurrence relations to determine that T(n) = Θ(n 2 ) while S(n) = Θ(ln(n)).

**Algorithm design** refers to a method or a mathematical process for problem-solving and engineering **algorithms**. The **design** of **algorithms** is part of many solution theories of operation research, such as dynamic programming and divide-and-conquer.

CHAPTER-3

# GROTH OF FUNCTIONS

When studying the complexity of an algorithm, we are concerned with the growth in the number of operations required by the algorithm as the size of the problem increases. In order to get a handle on its complexity, we first look for a function that gives the number of operations in terms of the size of the problem, usually measured by a positive integer n, to which the algorithm is applied. We then try to compare values of this function, for large n, to the values of some known function, such as a power function, exponential function, or logarithm function. Thus, the growth of functions refers to the relative size of the values of two functions for large values of the independent variable. This is one of the main areas in this course in which experience with the concept of a limit from calculus will be of great help. Before we begin, one comment concerning notation for logarithm functions is in order. Most algebra and calculus texts use log x to denote log10 x (or, perhaps, loge x), but in computer science base 2 is used more prevalently. So we shall use log x to denote log2 x. As we shall see, in the context of this module it actually doesn’t matter which base you use, since loga x = logb x logb a for any acceptable bases a and b.

Growth of Functions. Given functions f and g, we wish to show how to quantify the statement: “g grows as fast as f”. The growth of functions is directly related to the complexity of algorithms. We are guided by the following principles. • We only care about the behavior for “large” problems. • We may ignore implementation details such as loop counter incrementation.

The Big-O Notation :

Let f and g be functions from the natural numbers to the real numbers. Then g asymptotically dominates f, or f is big-O of g if there are positive constants C and k such that |f(x)| ≤ C|g(x)| for x ≥ k.

If f is big-O of g, then we write f(x) is O(g(x)) or f ∈ O(g).

If limx→∞ |f(x)| |g(x)| = L, where L ≥ 0, then f ∈ O(g).

If limx→∞ |f(x)| |g(x)| = ∞, then f is not O(g) (f 6∈ O(g))