Data and File Structures

Sumeet Sharma

This reference book can be useful for

BBA, MBA, B.Com, BMS, M.Com, BCA, MCA

and many more courses for Various Universities

NEERAJ
PUBLICATIONS
www.neerajbooks.com

Published by:

NEERAJ PUBLICATIONS

(Publishers of Educational Books)

Sales Office : 1507, 1st Floor,
Nai Sarak, Delhi-110 006
E-mail: info@neerajbooks.com
Website: www.neerajbooks.com

Typesetting by: Competent Computers

CONTENTS

DATA AND FILE STRUCTURES

INTRODUCTION TO ALGORITHMS AND DATA STRUCTURES

Analysis of Algorithms

1

INTRODUCTION

Since the beginning of civilization in the world, man has been in a continuous process of inventing different methods for expressing data in the form of text or pictures, depending upon the requirements. This led to the subject of data structures. The basic terminology and concepts will be defined in this chapter with relevant examples. We shall also discuss in detail regarding the various operations that can be applied to these data structures. Along with, we shall try to understand the terms like algorithm, its complexity and how to select an algorithm and data structure for a given problem.

Q. 1. What do you understand by Algorithms?

Ans. As we know that the field of computer science centers around writing programs to solve various problems in different domains. A program is a product of both the data structures and an algorithm.

*An **algorithm** is a well defined sequence of steps required for solving a particular problem.*

The concept of using algorithms is to study and provide the solution for a problem by using a computer. It not only specifies the sequence in which the steps are to be performed, but once these steps are performed in the prescribed sequence on a sample data representing the instance of the problem, the expected results are also obtained.

Moreover, while working on these algorithms two terms: *Complexity* and *time-space tradeoff*, come into picture. While processing the data, the time and space it uses are the key measures of the efficiency of an algorithm. On the other hand, the complexity of an algorithm is the function that gives the running time and/or space in terms of the input size.

Example 1. : Let us take an example of finding a factorial of a number which will be an input. The input to the algorithm that solves the given problem will be a natural number. The output is also a positive number which is the factorial of that number given as input. You can write a number of algorithms, but one such is given here for your reference.

Algorithm Find FACTORIAL

```
Step 1.  Read a positive number "x".
Step 2.  Assign the value 1 to the variable "f".
Step 3.  Assign the value "x" to the variable "n".
Step 4.  If "x" is less than zero then jump to Step 10.
Step 5.  If "x" is equal to zero then jump to Step 11.
Step 6.  If "x" is greater than zero then perform steps
         7 to 9 else goto Step 11.
Step 7.  Assign the value "x" multiplied with "f" to
         the variable "f".
Step 8.  Assign the value "x" minus 1 to the variable
         "x".
```

Step 9. Go to Step 6.
Step 10. Print "The input number is negative" and goto Step 12.
Step 11. Print "Factorial of number *n* is *f*".
Step 12. Stop.

The above steps mentioned in an algorithmic form, are so simple that anyone carrying out these steps precisely knows what to do in each step. While carrying out these steps of an algorithm on some input data, one must encounter some step containing a statement like "Stop" after a finite number of steps.

The following are the properties which an algorithm should have:

(a) *Input*: For each and every algorithm there are some input data which is externally supplied to the algorithm.

(b) *Output*: The algorithm should at least specify one output as a result.

(c) *Finiteness*: If the instructions or steps of an algorithm are carried out, then for all possible combinations of input data, the algorithm should be able to terminate after a finite number of steps. It should strictly include a "Halt" or a "Stop" statement.

(d) *Definiteness*: The steps should be quite clear and least ambiguous.

(e) *Effectiveness*: The steps of an algorithm must be simple. By simple we mean that even if a layman tries to carry out the steps, he should be able to mechanically do that using a pencil and a paper without applying any intelligence.

Q. 2. Write down the guidelines of writing algorithms.

Ans. There are some guidelines for writing algorithms, which are given below:

(a) Expressing an algorithm*:** The basic step is to express the problem in terms of an algorithm. It is the "definiteness" property of an algorithm that demands clarity and avoid ambiguity at each step specified. These steps should be expressed in a natural language (like English). However sometimes it becomes difficult to express in natural language. Therefore some pictorial descriptions of an algorithm are also used. These pictorial descriptions are known as ***Flow Charts.

***(b) Designing an algorithm*:** Problem solving still remains an innovative exercise, i.e. creating an algorithm for solving a problem is an art. There is no preset methodology to create an algorithm. There can be *n* number of methods of creating an algorithm for the same problem. However, new techniques and strategies have evolved over a period of time which help us gain useful ideas which can be applied to devise an algorithm for a new problem.

***(c) Analyzing an algorithm*:** After the design stage of an algorithm, there are two key points which should be dealt with properly. One is the validation of the algorithm and other is evaluating the complexity of the algorithm. It is the foremost important issue that an algorithm yields correct results. This is also called as "*program solving*" or "*program verification*". Another important topic is to determine the amount of time and storage an algorithm may require for an execution.

Q. 3. What do you understand by Complexity?

Ans. Complexity refers to the rate at which the required storage or consumed time grows as a function of the problem size. The absolute growth depends on the machine used to execute the program, the compiler used to construct the program, and many other factors. We would like to have a way of describing the inherent complexity of a program, independent of machine/ compiler considerations. This means that we must not try to describe the absolute time or storage needed. We must instead concentrate on a proportionality approach, expressing the complexity in terms of its relationship to some known function. This type of analysis is known as **Asymptotic Analysis**. It may be noted that we are dealing with complexity of an algorithm not that of a problem. For example, the simple problem could have high order of time complexity and vice versa.

Complexity Classes

All decision problems fall into sets of comparable complexity, called complexity classes.

The complexity class P is a set of decision problems that can be solved by deterministic machine in polynomial time. This class corresponds to set of problems which can be effectively solved in the worst cases. We will consider algorithms belonging to this class for analysis of time complexity. Not all algorithms in these classes make practical sense as many of them have higher complexity.

The complexity class NP is a set of decision problems that can be solved by a non-deterministic machine in polynomial time. This class contains many problems like Boolean satisfiability problem, Hamiltonian path problem and the Vertex cover problem.

Q. 4. What do you understand by Asymptotic Analysis?

Ans. Asymptotic analysis is based on the idea that as the problem grows, the complexity can be described as a simple proportionality to some known function. This idea is incorporated in the "Big O", "Omega", "Theta" notation for asymptotic performance.

The notations like "little Oh" are similar in spirit to "Big Oh"; but are rarely used in computer science for asymptotic analysis.

Analysis of Algorithms

The objective of analysis of an algorithm is to find its efficiency. Efficiency is dependent on the resources that are used by the algorithm. For example:

- CPU Utilization (Time complexity)
- Memory Utilization (Space complexity)
- Disk Usage (I/O)
- Network Usage (bandwidth)

There are four important attributes to an algorithm. They are:

- *Performance*: How much time/memory/disk/network bandwidth is actually used when a program is run. This depends on the algorithm, machine, compiler, etc.
- *Complexity*: How do the resource requirements of a program or algorithm scale (the growth of resource requirements as a function of input). In other words, what happens to the performance of an algorithm, as the size of the problem being solved gets larger and larger? For example, the time and memory requirement of an algorithm which computes the sum of 1000 numbers is larger than the algorithm which computes the sum of 2 numbers.
- *Time Complexity*: The maximum time required by a Turing machine to execute on an input of length *n*.
- *Space Complexity*: The amount of storage required by an algorithm varies with the size of the problem being solved. The space complexity is normally expressed as an order of magnitude of the size of the problem, e.g., $O(n^2)$ means that if the size of the problem (n) doubles then the working storage (memory) requirement will become four times.

As we know that there will be many different algorithms for solving the same problem. Accordingly, a data structure can be represented in a number of ways and there may be number algorithms to implement an operation on the said data structure. In such situations, it is required to compare two algorithms to implement an operation on the said data structure. Your comparison or analysis of the two algorithms should reveal some quantitative metrics regarding the execution of the algorithms. In this context we will mainly discuss the following.

Q. 5. What is Time-Space Trade Off?

Ans. *Time-Space Trade off*: *Space* in this context means storage required in addition to the space required to store the input data. *Time* is the computer time required for the execution of an algorithm, but it again depends upon the size of input. Thus time complexity of an algorithm is often a function input size, "*n*". Moreover the same algorithm may take different time to execute for different inputs having the same size. The different times are measured in terms of best case, worst case and an average case. The *best case* complexity of an algorithm is a measure of minimum time that an algorithm will require for its execution. However the worst case complexity of an algorithm is a measure of maximum time that an algorithm will require for its execution.

However, it is very difficult to compute the exact time taken by an algorithm for its execution. There are several important factors which influence the time required by an algorithm. Accordingly, the programmer has to tradeoff between the size of the input and time taken for the execution of the algorithm. Sometimes where the time is foremost issue the programmer decides in time's favor, however in other cases the programmer may decide in size's favor.

When we write a program that finds the maximum elements in a list of an array, the primary operation of an algorithm is to perform comparison operations. However, if the program is to sort an array of *n* elements, then one more operation other than comparison operation becomes more important is the exchange operation. This step of identification of the primary operations of an algorithm guides us in separating the analysis of an algorithm from its implementation.

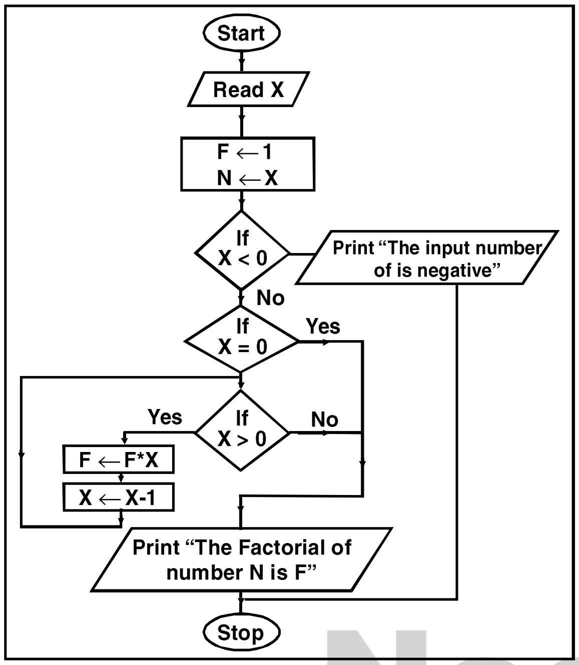
Start
Read X
F ← 1
N ← X
If X < 0
Print "The input number of is negative"
No
If X = 0
Yes
Yes
If X > 0
No
F ← F*X
X ← X-1
Print "The Factorial of number N is F"
Stop

Fig. 1.1

Example 1.2: Let us consider the following function:

```
void addsum()
    {
        int a, b, c;
        c =  a + b;
    }
```

The most important operation that the above function is set to perform is addition of the values of variables *a* and *b* and assign it as the value of the variable *c*. The number of addition that it performs is 1.

Example 1.3: Let us consider another function:

```
void nestedaddsum(int n)
    {
        int a, b, c;
        int i, j;
        for(i=0; i<n; i++)
            for(j=0; j<n; j++)
                c = a + b;
    }
```

The number of assignment operations the function performs are:

$$\Sigma_{i=1,n} (\Sigma_{j=1,n}(1)) = \Sigma_{i=1,n}(n) = n^2$$

Example1.4: Let us consider another function:

```
void anothernestedaddsum(int n)
    {
        int a, b, c;
        int i, j;
        for(i=0; i<n; i++)
            for(j=i; j<n; j++)
            {
                if (j%2 == 0)
                    break;

                else
                    c = a + b;
            }
    }
```

In the above function, when control enters into the inner *j* loop, the value of *i* is either even or odd. Hence *j* is initialized with either an odd or an even value. If *j* is initialized to an odd value, the assignment operation takes place. But before the next operation of *j*, where it becomes an even values the loop is broken. However, if *j* is assigned an even number, therefore no assignment operation takes place.

The most important issue is that algorithm in its entirety is not taken up for complexity analysis. Instead some key operations, which will reveal the computational complexity of an algorithm, are considered. Another issue which is also important is that many mathematical tools including mathematical induction are to be rigorously used to carry out the complexity analysis of algorithms.

Q. 6. What is θ–Notation?

Ans. This notation bounds a function to within constant factors. We say *f(n)* = θ*(g(n))* if there exist positive constants n_0, c_1 and c_2, such that to the right of n_0 the values of *f(n)* always lies between $c_1g(n)$ and $c_2g(n)$, both inclusive. Following the figure displays the function *f(n)* and *g(n)* where *f(n)* = θ*(g(n))*. We will say that the function *g(n)* is asymptotically tight bound for *f(n)*.

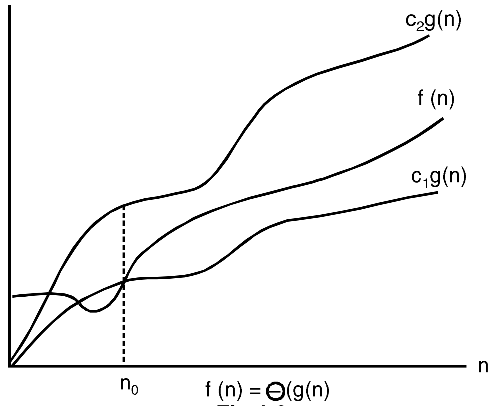

Fig.1.2

For example, let us show that the function *f(n)* = $(1/3)n^2 - 4n = \theta(n^2)$.