UNIT 1 PROBLEM SOLVING

Structure

1.0 INTRODUCTION

In our daily life, we routinely encounter and solve problems. We pose problems that we need or want to solve. For this, we make use of available resources, and solve them. Some categories of resources include: the time and efforts of yours and others; tools; information; and money. Some of the problems that you encounter and solve are quite simple. But some others may be very complex.

In this unit we introduce you to the concepts of problem-solving, especially as they pertain to computer programming**.**

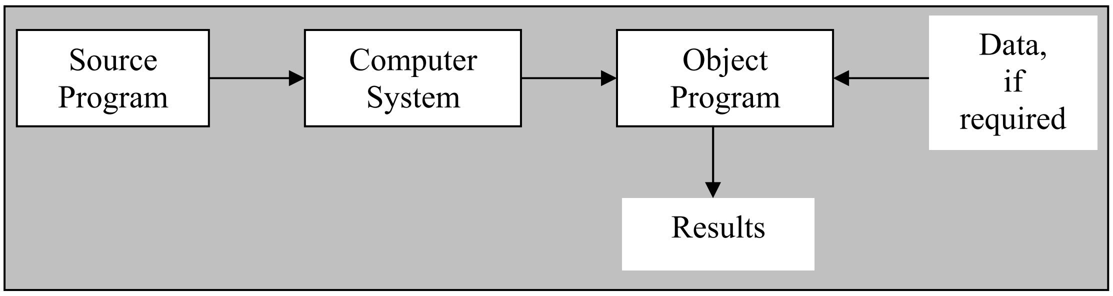
The problem-solving is a skill and there are no universal approaches one can take to solving problems. Basically one must explore possible avenues to a solution one by one until s/he comes across a right path to a solution. In general, as one gains experience in solving problems, one develops one's own techniques and strategies, though they are often intangible. Problem-solving skills are recognized as an integral component of computer programming. It is a demand and intricate process which is equally important throughout the project life cycle especially – study, designing, development, testing and implementation stages. The computer problem solving process requires:

- Problem anticipation
- Careful planning
- Proper thought process
- Logical precision
- Problem analysis
- Persistence and attention.

At the same time it requires personal creativity, analytic ability and expression. The chances of success are amplified when the problem solving is approached in a systematic way and satisfaction is achieved once the problem is satisfactorily solved. The problems should be anticipated in advance as far as possible and properly defined to help the algorithm definition and development process.

Computer is a very powerful tool for solving problems. It is a symbol-manipulating machine that follows a set of stored instructions called a program. It performs these manipulations very quickly and has memory for storing input, lists of commands and output. A computer cannot think in the way we associate with humans. When using the computer to solve a problem, you must specify the needed initial data, the operations which need to be performed (in order of performance) and what results you want for output. If any of these instructions are missing, you will get either no results or invalid results. In either case, your problem has not yet been solved. Therefore, several steps need to be considered before writing a program. These steps may free you from hours of finding and removing errors in your program (a process called **debugging**). It should also make the act of problem solving with a computer a much simpler task.

All types of computer programs are collectively referred to as **software**. Programming languages are also part of it. Physical computer equipment such as electronic circuitry, input/output devices, storage media etc. comes under **hardware**. Software governs the functioning of hardware. Operations performed by software may be built into the hardware, while instructions executed by the hardware may be generated in software. The decision to incorporate certain functions in the hardware and others in the software is made by the manufacturer and designer of the software and hardware. Normal considerations for this are: cost, speed, memory required, adaptability and reliability of the system. Set of instructions of the high level language used to code a problem to find its solution is referred to as **Source Program**. A translator program called **a compiler or interpreter**, translates the source program into the object program. This is the compilation or interpretation phase. All the testing of the source program as regards the correct format of instructions is performed at this stage and the errors, if any, are printed. If there is no error, the source program is transformed into the machine language program called **Object Program**. The Object Program is executed to perform calculations. This stage is the execution phase. Data, if required by the program, are supplied now and the results are obtained on the output device.



1.1 OBJECTIVES

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

- apply problem solving techniques;
- define an algorithm and its features;
- describe the analysis of algorithm efficiency;
- discuss the analysis of algorithm complexity; and
- design flowcharts.

1.2 PROBLEM - SOLVING TECHNIQUES

Problem solving is a creative process which defines systematization and mechanization. There are a number of steps that can be taken to raise the level of one's performance in problem solving.

1.2.1 Steps for Problem - Solving

A problem-solving technique follows certain steps in finding the solution to a problem. Let us look into the steps one by one:

Problem definition phase

The success in solving any problem is possible only after the problem has been fully understood. That is, we cannot hope to solve a problem, which we do not understand. So, the problem understanding is the first step towards the solution of the problem. In *problem definition phase*, we must emphasize *what must be done* rather than *how is it to be done*. That is, we try to extract the precisely defined set of tasks from the problem statement. Inexperienced problem solvers too often gallop ahead with the task of problem - solving only to find that they are either solving the wrong problem or solving just one particular problem.

Getting started on a problem

There are many ways of solving a problem and there may be several solutions. So, it is difficult to recognize immediately which path could be more productive. Sometimes you do not have any idea where to begin solving a problem, even if the problem has been defined. Such block sometimes occurs because you are overly concerned with the details of the implementation even before you have completely understood or worked out a solution. The best advice is not to get concerned with the details. Those can come later when the intricacies of the problem has been understood.

The use of specific examples

To get started on a problem, we can make use of heuristics i.e., the rule of thumb. This approach will allow us to start on the problem by picking a specific problem we wish to solve and try to work out the mechanism that will allow solving this particular problem. It is usually much easier to work out the details of a solution to a specific problem because the relationship between the mechanism and the problem is more clearly defined. This approach of focusing on a particular problem can give us the foothold we need for making a start on the solution to the general problem.

Similarities among problems

One way to make a start is by considering a specific example. Another approach is to bring the experience to bear on the current problem. So, it is important to see if there are any similarities between the current problem and the past problems which we have solved. The more experience one has the more tools and techniques one can bring to bear in tackling the given problem. But sometimes, it blocks us from discovering a desirable or better solution to the problem. A skill that is important to try to develop in problem - solving is the ability to view a problem from a variety of angles. One must be able to metaphorically turn a problem upside down, inside out, sideways, backwards, forwards and so on. Once one has developed this skill it should be possible to get started on any problem.

Working backwards from the solution

In some cases we can assume that we already have the solution to the problem and then try to work backwards to the starting point. Even a guess at the solution to the problem may be enough to give us a foothold to start on the problem. We can systematize the investigations and avoid duplicate efforts by writing down the various steps taken and explorations made. Another practice that helps to develop the problem solving skills is, once we have solved a problem, to consciously reflect back on the way we went about discovering the solution.

1.2.2 Using Computer as a Problem - Solving Tool

The computer is a resource - a versatile tool - that can help you solve some of the problems that you encounter. A computer is a very powerful general-purpose tool. Computers can solve or help to solve many types of problems. There are also many ways in which a computer can enhance the effectiveness of the time and effort that you are willing to devote to solving a problem. Thus, it will prove to be well worth the time and effort you spend to learn how to make effective use of this tool.

In this section, we discuss the steps involved in developing a program. Program development is a multi-step process that requires you to understand the problem, develop a solution, write the program, and then test it. This critical process determines the overall quality and success of your program. If you carefully design each program using good structured development techniques, your programs will be efficient, error-free, and easy to maintain. The following are the steps in detail:

1. Develop an *Algorithm* and a *Flowchart*.
2. Write the program in a computer language (for example say C programming language).
3. Enter the program using some editor.
4. Test and debug the program.
5. Run the program, input data, and get the results.

1.3 DESIGN OF ALGORITHMS

The first step in the program development is to devise and describe a precise plan of what you want the computer to do. This plan, expressed as a sequence of operations, is called an algorithm. An algorithm is just an outline or idea behind a program. something resembling C or Pascal, but with some statements in English rather than within the programming language. It is expected that one could translate each pseudo-code statement to a small number of lines of actual code, easily and mechanically.

1.3.1 Definition

An **algorithm** is a finite set of steps defining the solution of a particular problem. An algorithm is expressed in pseudocode - something resembling C language or Pascal, but with some statements in English rather than within the programming language. Developing an efficient algorithm requires lot of practice and skill. It must be noted that an efficient algorithm is one which is capable of giving the solution to the problem by using minimum resources of the system such as memory and processor's time. Algorithm is a language independent, well structured and detailed. It will enable the programmer to translate into a computer program using any high-level language.

1.3.2 Features of Algorithm

Following features should be present in an algorithm:

Proper understanding of the problem

For designing an efficient algorithm, the expectations from the algorithm should be clearly defined so that the person developing the algorithm can understand the expectations from it. This is normally the outcome of the problem definition phase.

Use of procedures / functions to emphasize modularity

To assist the development, implementation and readability of the program, it is usually helpful to modularize (section) the program. Independent functions perform specific and well defined tasks. In applying modularization, it is important to watch that the process is not taken so far to a point at which the implementation becomes difficult to read because of fragmentation. The program then can be implemented as calls to the various procedures that will be needed in the final implementations.

Choice of variable names

Proper variable names and constant names can make the program more meaningful and easier to understand. This practice tends to make the program more self documenting. A clear definition of all variables and constants at the start of the procedure / algorithm can also be helpful. For example, it is better to use variable *day* for the day of the weeks, instead of the variable *a* or something else.

Documentation of the program

Brief information about the segment of the code can be included in the program to facilitate debugging and providing information. A related part of the documentation is the information that the programmer presents to the user during the execution of the program. Since, the program is often to be used by persons who are unfamiliar with the working and input requirements of the program, proper documentation must be provided. That is, the program must specify what responses are required from the user. Care should also be taken to avoid ambiguities in these specifications. Also the program should "catch" incorrect responses to its requests and inform the user in an appropriate manner.

1.3.3 Criteria to be followed by an Algorithm

The following is the criteria to be followed by an algorithm:

- **Input:** There should be zero or more values which are to be supplied.
- **Output:** At least one result is to be produced.
- **Definiteness:** Each step must be clear and unambiguous.
- **Finiteness:** If we trace the steps of an algorithm, then for all cases, the algorithm must terminate after a finite number of steps.
- **Effectiveness:** Each step must be sufficiently basic that a person using only paper and pencil can in principle carry it out. In addition, not only each step is definite, it must also be feasible.

Example 1.1

Let us try to develop an algorithm to compute and display the sum of two numbers

1. Start
2. Read two numbers *a* and *b*
3. Calculate the sum of *a* and *b* and store it in *sum*
4. Display the value of *sum*
5. Stop

Example 1.2

Let us try to develop an algorithm to compute and print the average of a set of data values.

1. Start
2. Set the sum of the data values and the count to zero.

3. As long as the data values exist, add the next data value to the sum and add 1 to the count.
4. To compute the average, divide the sum by the count.
5. Display the average.
6. Stop

Example 1.3

Write an algorithm to calculate the factorial of a given number.

```
1. Start
2. Read the number n
3. [Initialize]
   i ← 1 , fact ← 1
4. Repeat steps 4 through 6 until i = n
5. fact ← fact * i
6. i ← i + 1
7. Print fact
8. Stop
```

Example 1.4

Write an algorithm to check that whether the given number is prime or not.

```
1. Start
2. Read the number num
3. [Initialize]
   i ← 2 , flag ← 1
4. Repeat steps 4 through 6 until i < num or flag = 0
5. rem ← num mod i
6. if rem = 0 then
   flag ← 0
       else
         i ← i + 1
7. if flag = 0 then
         Print Number is not prime
       Else
         Print Number is prime
8. Stop
```

1.3.4 Top Down Design

Once we have defined the problem and have an idea of how to solve it, we can then use the powerful techniques for designing algorithms. Most of the problems are complex or large problems and to solve them we have to focus on to comprehend at one time, a very limited span of logic or instructions. A technique for algorithm design that tries to accommodate this human limitation is known as **top-down design or stepwise refinement.**

Top down design provides the way of handling the logical complexity and detail encountered in computer algorithm. It allows building solutions to problems in step by step. In this way, specific and complex details of the implementation are encountered only at the stage when sufficient groundwork on the overall structure and relationships among the various parts of the problem.

Before the top down design can be applied to any problem, we must at least have the outlines of a solution. Sometimes this might demand a lengthy and creative investigation into the problem while at another time the problem description may in itself provide the necessary starting point for the top-down design.

Top-down design suggests taking the general statements about the solution one at a time, and then breaking them down into a more precise subtask / sub-problem. These sub-problems should more accurately describe how the final goal can be reached. The process of repeatedly breaking a task down into a subtask and then each subtask into smaller subtasks must continue until the sub-problem can be implemented as the program statement. With each spitting, it is essential to define how sub-problems interact with each other. In this way, the overall structure of the solution to the problem can be maintained. Preservation of the overall structure is important for making the algorithm comprehensible and also for making it possible to prove the correctness of the solution.

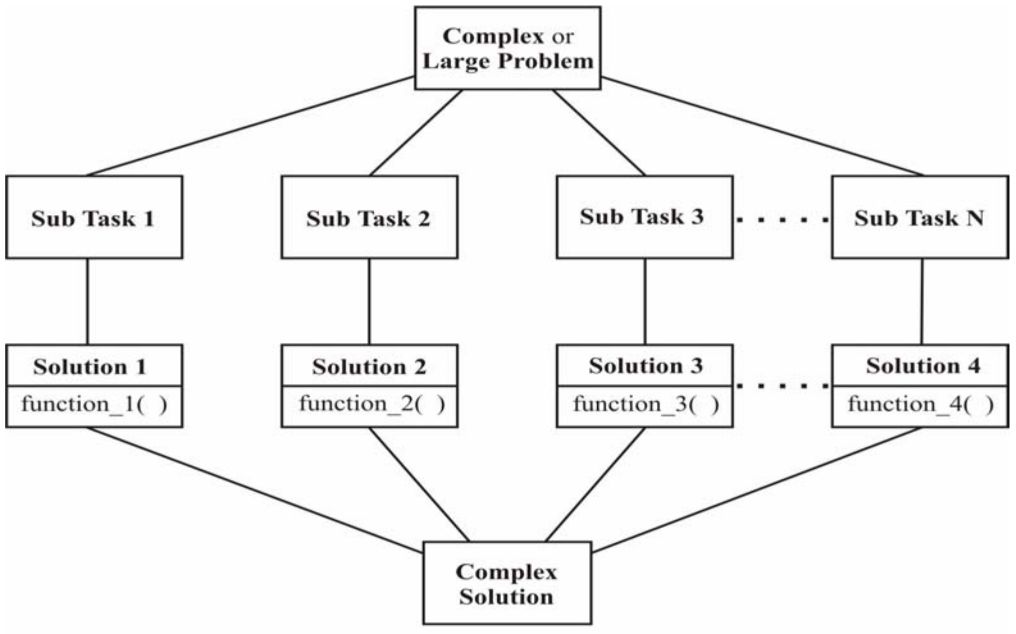


Figure 1.1: Schematic breakdown of a problem into subtasks as employed in top down design

1.4 ANALYSIS OF ALGORITHM EFFICENCY

Every algorithm uses some of the computer's resources like central processing time and internal memory to complete its task. Because of high cost of computing resources, it is desirable to design algorithms that are economical in the use of CPU time and memory. Efficiency considerations for algorithms are tied in with the design, implementation and analysis of algorithm. Analysis of algorithms is less obviously necessary, but has several purposes:

- Analysis can be more reliable than experimentation. If we experiment, we only know the behavior of a program on certain specific test cases, while analysis can give us guarantees about the performance on all inputs.
- It helps one choose among different solutions to problems. As we will see, there can be many different solutions to the same problem. A careful analysis and comparison can help us decide which one would be the best for our purpose, without requiring that all be implemented and tested.

- We can predict the performance of a program before we take the time to write code. In a large project, if we waited until after all the code was written to discover that something runs very slowly, it could be a major disaster, but if we do the analysis first we have time to discover speed problems and work around them.
- By analyzing an algorithm, we gain a better understanding of where the fast and slow parts are, and what to work on or work around in order to speed it up.

There is no simpler way of designing efficient algorithm, but a few suggestions as shown below can sometimes be useful in designing an efficient algorithm.

1.4.1 Redundant Computations

Redundant computations or unnecessary computations result in inefficiency in the implementation of the algorithms. When redundant calculations are embedded inside the loop for the variable which remains unchanged throughout the entire execution phase of the loop, the results are more serious. For example, consider the following code in which the value *a*a*a*c* is redundantly calculated in the loop:

```
x=0;
for i=0 to n
        x=x+1;
        y=(a*a*a*c)*x*x+b*b*x;
        print x,y
next i
```

This redundant calculation can be removed by small modification in the program:

```
x=0;
d=a*a*a*c;
e= b*b;
for i = 0 to n
        x = x+1;
        y = d*x*x+e*x;
        print x,y
next i
```

1.4.2 Referencing Array Elements

For using the array element, we require two memory references and an additional operation to locate the correct value for use. So, efficient program must not refer to the same array element again and again if the value of the array element does not change. We must store the value of array element in some variable and use that variable in place of referencing the array element. For example:

Version (1)

```
x=1;
for i = 0 to n
        if (a[i] > a[x]) x=i;
next i
max = a[x];
```

Version (2)

```
x=1;
max=a[1];
for i = 0 to n
        if(a[i]>max)
                x=i;
                max=a[i];
next i
```

Version (2) is more efficient algorithm than version (1) algorithm.

1.4.3 Inefficiency Due to Late Termination

Another place where inefficiency can come into an implementation is where considerably more tests are done than are required to solve the problem at hand. For example, if in the linear search process, all the list elements are checked for a particular element even if the point is reached where it was known that the element cannot occur later (in case of sorted list). Second example can be in case of the bubble sort algorithm, where the inner loop should not proceed beyond n-i, because last i elements are already sorted (in the algorithm given below).

```
for  i = 0 to n
        for j = 0 to n – 1
                if(a[j] > a[j+1])
                        //swap values a[j], a[j+1]
```

The efficient algorithm in which the inner loop terminates much before is given as:

```
for i=0 to n
        for j=0 to n – 1
                if(a[j]>a[j+1])
                        //swap values a[j], a[j+1]
```

1.4.4 Early Detection of Desired Output Condition

Sometimes the loops can be terminated early, if the desired output conditions are met. This saves a lot of unfruitful execution. For example, in the bubble sort algorithm, if during the current pass of the inner loop there are no exchanges in the data, then the list can be assumed to be sorted and the search can be terminated before running the outer loop for *n* times.

1.4.5 Trading Storage for Efficient Gains

A trade between storage and efficiency is often used to improve the performance of an algorithm. This can be done if we save some intermediary results and avoid having to do a lot of unnecessary testing and computation later on.

One strategy for speeding up the execution of an algorithm is to implement it using the least number of loops. It may make the program much harder to read and debug. It is therefore sometimes desirable that each loop does one job and sometimes it is required for computational speedup or efficiency that the same loop must be used for different jobs so as to reduce the number of loops in the algorithm. A kind of trade off is to be done while determining the approach for the same.

1.5 ANALYSIS OF ALGORITHM COMPLEXITY

Algorithms usually possess the following qualities and capabilities:

- Easily modifiable if necessary.
- They are easy, general and powerful.
- They are correct for clearly defined solution.
- Require less computer time, storage and peripherals i.e. they are more economical.
- They are documented well enough to be used by others who do not have a detailed knowledge of the inner working.
- They are not dependable on being run on a particular computer.
- The solution is pleasing and satisfying to its designer and user.
- They are able to be used as a sub-procedure for other problems.

Two or more algorithms can solve the same problem in different ways. So, quantitative measures are valuable in that they provide a way of comparing the performance of two or more algorithms that are intended to solve the same problem. This is an important step because the use of an algorithm that is more efficient in terms of time, resources required, can save time and money.

1.5.1 Computational Complexity

We can characterize an algorithm's performance in terms of the size (usually n) of the problem being solved. More computing resources are needed to solve larger problems in the same class. The table below illustrates the comparative cost of solving the problem for a range of n values.

Log_2 n	n	n $\log_2$ n	n^2	n^3	2^n
1	2	2	4	8	4
3.322	10	33.22	10^2	10^3	$>10^3$
6.644	10^2	664.4	10^4	10^6	$>>10^{25}$
9.966	10^3	9966.0	10^6	10^9	$>>10^{250}$
13.287	10^4	132877	10^8	10^{12}	$>>10^{2500}$

The above table shows that only very small problems can be solved with an algorithm that exhibit exponential behaviour. An exponential problem with n=100 would take immeasurably longer time. At the other extreme, for an algorithm with logarithmic dependency would merely take much less time (13 steps in case of $\log_2$n in the above table). These examples emphasize the importance of the way in which algorithms behave as a function of the problem size. Analysis of an algorithm also provides the theoretical model of the inherent computational complexity of a particular problem.

To decide how to characterize the behaviour of an algorithm as a function of size of the problem n, we must study the mechanism very carefully to decide just what constitutes the dominant mechanism. It may be the number of times a particular expression is evaluated, or the number of comparisons or exchanges that must be made as n grows. For example, comparisons, exchanges, and moves count most in sorting algorithm. The number of comparisons usually dominates so we use comparisons in computational model for sorting algorithms.

1.5.2 The Order of Notation

The O-notation gives an upper bound to a function within a constant factor. For a given function g(n), we denote by O(g(n)) the set of functions.

O(g(n)) = { f(n) : there exist positive constants c and n0, such that 0 <= f(n) <= cg(n) for all n >= n0 }

Using O-notation, we can often describe the running time of an algorithm merely by inspecting the algorithm's overall structure. For example a double nested loop structure of the following algorithm immediately yields O(n2) upper bound on the worst case running time.

```
for i=0 to n
        for j=0 to n
                print i,j
        next j
next i
```

What we mean by saying "the running time is O(n^2)" is that the worst case running time (which is a function of n) is O(n^2). Or equivalently, no matter what particular input of size n is chosen for each value of n, the running time on that set of inputs is O(n^2).

1.5.3 Rules for using the Big-O Notation

Big-O bounds, because they ignore constants, usually allow for very simple expression for the running time bounds. Below are some properties of big-O that allow bounds to be simplified. The most important property is that big-O gives an upper bound only. If an algorithm is $O(N^2)$, it doesn't have to take N^2 steps (or a constant multiple of N^2). But it can't take more than N^2. So any algorithm that is O(N), is also an $O(N^2)$ algorithm. If this seems confusing, think of big-O as being like "<". Any number that is < N is also $<N^2$.

1. Ignoring constant factors: O(c f(N)) = O(f(N)), where c is a constant; e.g. $O(20 N^3) = O(N^3)$
2. Ignoring smaller terms: If a<b then O(a+b) = O(b), for example, $O(N^2+N) = O(N^2)$
3. Upper bound only: If a<b then an O(a) algorithm is also an O(b) algorithm. For example, an O(N) algorithm is also an $O(N^2)$ algorithm (but not vice versa).
4. N and log N are bigger than any constant, from an asymptotic view (that means for large enough N). So if k is a constant, an O(N + k) algorithm is also O(N), by ignoring smaller terms. Similarly, an O(log N + k) algorithm is also O(log N).
5. Another consequence of the last item is that an O(N log N + N) algorithm, which is O(N(log N + 1)), can be simplified to O(N log N).

1.5.4 Worst and Average Case Behavior

Worst and average case behaviors of the algorithm are the two measures of performance that are usually considered. These two measures can be applied to both space and time complexity of an algorithm. The worst case complexity for a given problem of size n corresponds to the maximum complexity encountered among all problems of size n. For determination of the worst case complexity of an algorithm, we choose a set of input conditions that force the algorithm to make the least possible progress at each step towards its final goal.

In many practical applications it is very important to have a measure of the expected complexity of an algorithm rather than the worst case behavior. The expected complexity gives a measure of the behavior of the algorithm averaged over all possible problems of size n.

As a simple example: Suppose we wish to characterize the behavior of an algorithm that linearly searches an ordered list of elements for some value x.
1 2 3 4 5 … … …. N

In the worst case, the algorithm examines all n values in the list before terminating.

In the average case, the probability that x will be found at position 1 is 1/n, at position 2 is 2/n and so on. Therefore,

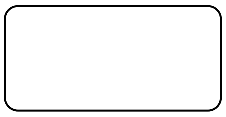
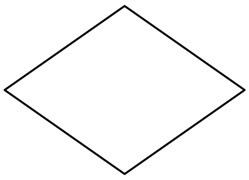
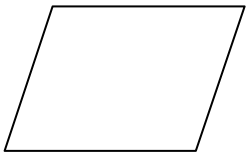
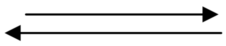
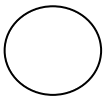
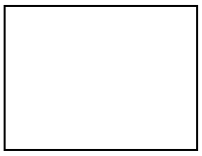
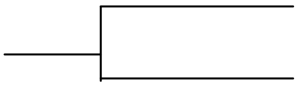
$$\begin{aligned}\text{Average search cost} &= 1/n(1+2+3+ \ldots..+n)\\ &= 1/n(n/2(n+1)) = (n+1)/2\end{aligned}$$

Let us see how to represent the algorithm in a graphical form using a flowchart in the following section.

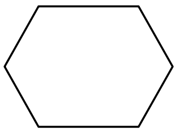
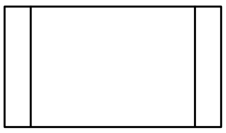
1.6 FLOWCHARTS

The next step after the algorithm development is the flowcharting. Flowcharts are used in programming to diagram the path in which information is processed through a computer to obtain the desired results. Flowchart is a graphical representation of an algorithm. It makes use of symbols which are connected among them to indicate the flow of information and processing. It will show the general outline of how to solve a problem or perform a task. It is prepared for better understanding of the algorithm.

1.6.1 Basic Symbols used in flowchart design

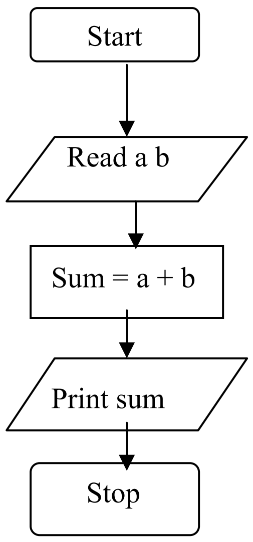
	Start/Stop
	Question, Decision (Use in Branching)
	Input/Output
	Lines or arrows represent the direction of the flow of control.
	Connector (connect one part of the flowchart to another)
	Process, Instruction
	Comments, Explanations, Definitions.

Additional Symbols Related to more advanced programming

	Preparation (may be used with “do Loops”)
	Refers to separate flowchart

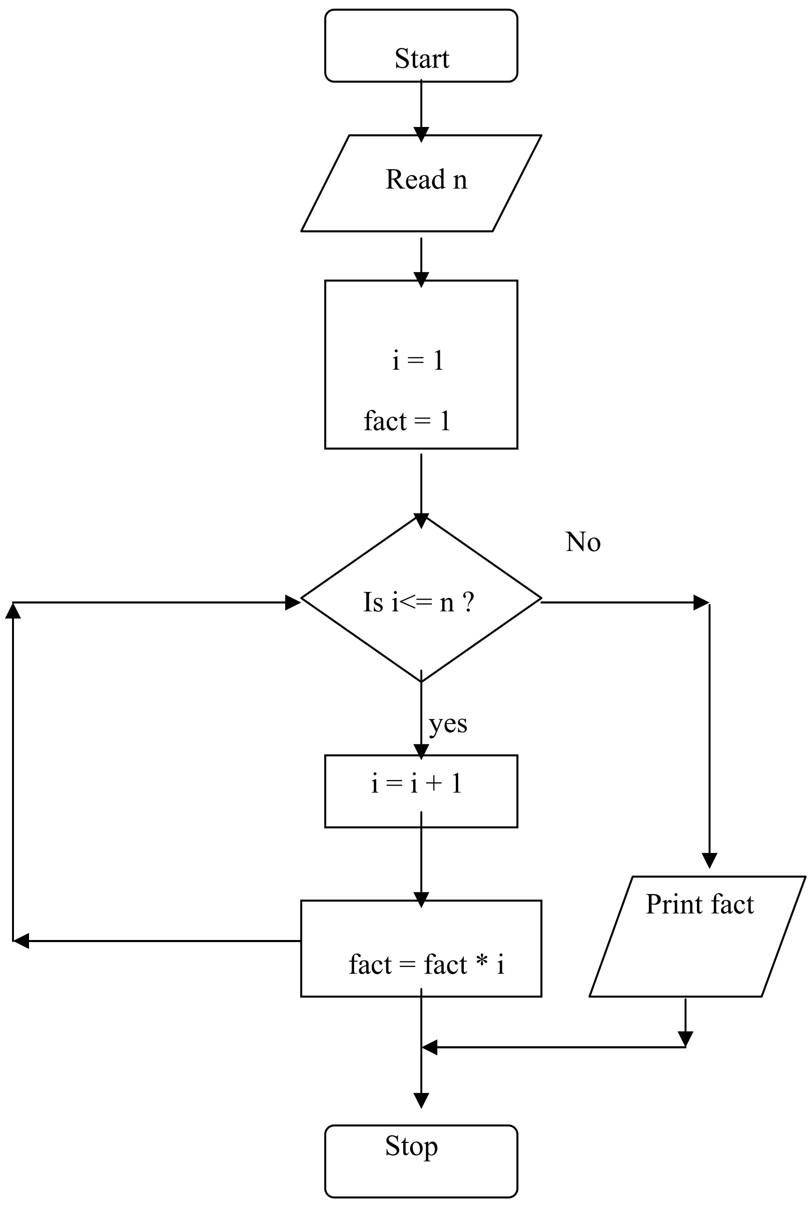
Example 1.5

The flowchart for the Example 1.1 is shown below:



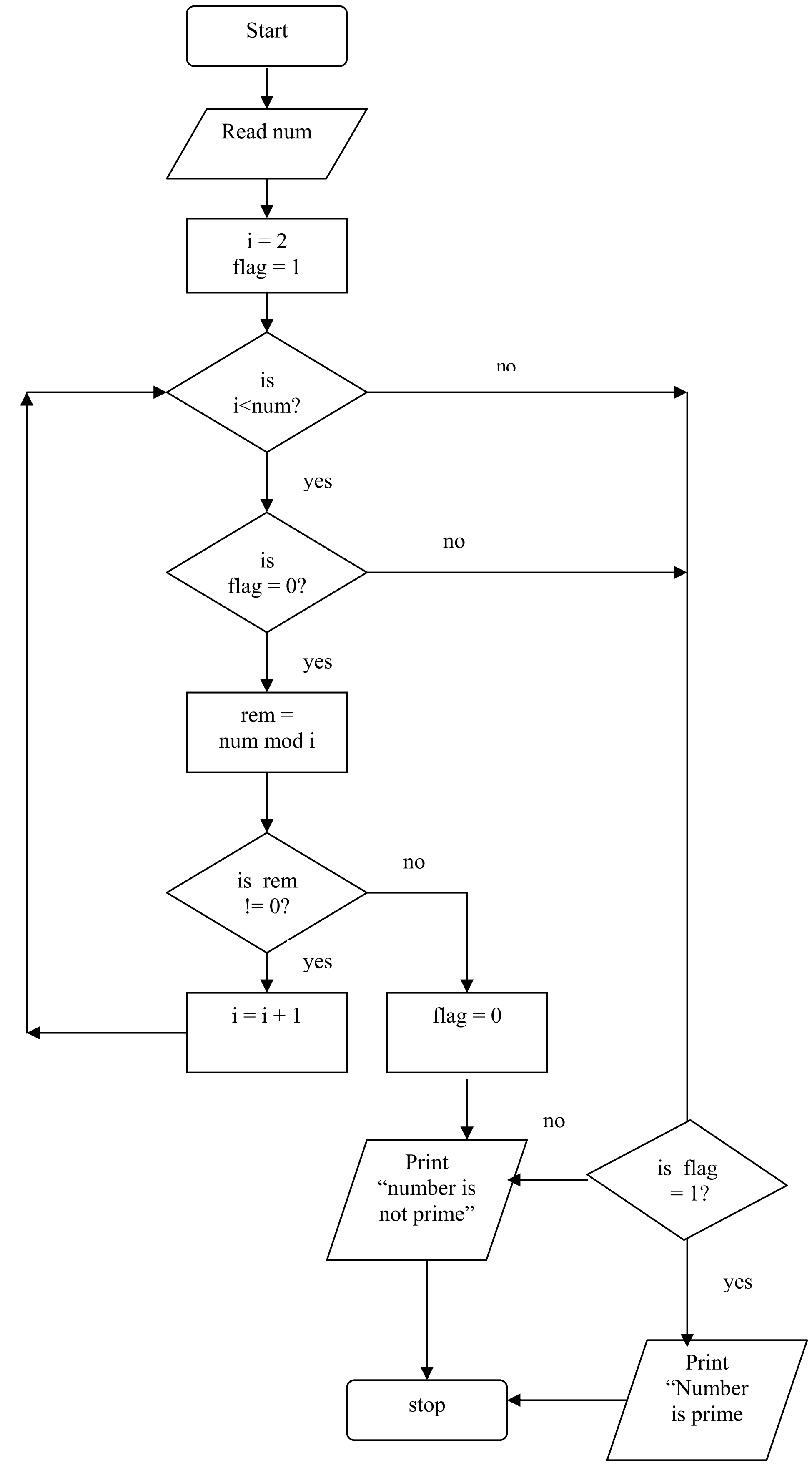
Example 1.6

The flowchart for the Example 1.3 (to find factorial of a given number) is shown below:



Example 1.7:

The flowchart for Example 1.4 is shown below:



Check Your Progress

1. Differentiate between flowchart and algorithm.

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

2. Compute and print the sum of a set of data values.

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

3. Write the following steps are suggested to facilitate the problem solving process using computer.

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

   …………………………………………………………………………………

4. Draw an algorithm and flowchart to calculate the roots of quadratic equation Ax^2 + Bx + C = 0.

   ……………………………………………………………………………..…

   ……………………………………………………………………………….

   …………………………………………………………………………………

   …………………………………………………………………………………

1.7 SUMMARY

To solve a problem different problem - solving tools are available that help in finding the solution to problem in an efficient and systematic way. Steps should be followed to solve the problem that includes writing the algorithm and drawing the flowchart for the solution to the stated problem**.** Top down design provides the way of handling the logical complexity and detail encountered in computer algorithm. It allows building solutions to problems in a stepwise fashion. In this way, specific and complex details of the implementation are encountered only at the stage when sufficient groundwork on the overall structure and relationships among the carious parts of the problem. We present C language - a standardized, industrial-strength programming language known for its power and portability as an implementation vehicle for these problem solving techniques using computer.

1.8 SOLUTIONS / ANSWERS

Check Your Progress

1. The process to devise and describe a precise plan (in the form of sequence of operations) of what you want the computer to do, is called an **algorithm**. An algorithm may be symbolized in a flowchart or pseudocode.

2. 1. Start
   2. Set the sum of the data values and the count of the data values to zero.
   3. As long as the data values exist, add the next data value to the sum and add 1 to the count.
   4. Display the average.
   5. Stop

3. The following steps are suggested to facilitate the problem solving process:

   a) Define the problem
   b) Formulate a mathematical model
   c) Develop an algorithm
   d) Design the flowchart
   e) Code the same using some computer language
   f) Test the program

1.9 FURTHER READINGS

1. How to solve it by Computer, 5th Edition, *R G Dromey*, PHI, 1992.
2. Introduction to Computer Algorithms, Second Edition, *Thomas H. Cormen*, MIT press, 2001.
3. Fundamentals of Algorithmics, *Gilles Brassword, Paul Bratley,* PHI, 1996.
4. Fundamental Algorithms, Third Edition, *Donald E Knuth*, Addison-Wesley, 1997.