CHAPTER 1

Algorithms (Analysis and Design)

1.1 PROBLEM SOLVING

In order to solve a problem using computer, one needs to write step by step solution first. This may be done by writing simple instructions for each operation. There might be a number of methods to solve the problem and hence solutions will differ from person to person. However, in all cases, basic steps for solving a problem would remain the same. These steps are:

a. Formulating the problem and deciding the data types to be entered. ✓
b. Identifying the steps of computation that are necessary for getting the solution. ✓
c. Identifying decision points *i.e.*, under what circumstances a particular operation is to be performed and when not to be carried out. ✓
d. Finding the result and verifying the values. ✓

1.1.1 Procedure for Problem Solving

Problem solving is simply writing the basic steps and putting them in correct sequence to find the result.

☞ *Problem solving is a logical process of breaking down the problem into smaller parts each of which can be solved step by step to obtain the final solution.*

Though, every problem may be unique in itself, yet the procedures for solving such problems are similar. The procedure for solving a problem involves six steps. These steps should be understood thoroughly and practiced effectively. Using these steps, our problem solving capability will improve with time.

Six basic steps in solving a problem are:

1. ✓ First, spend some time in understanding the problem. In this step, you are not required to use a computer. Instead, you should try to answer the question namely, what is expected and how to get it. This means, try to formulate a problem correctly.
2. ✓ Construct a list of *variables* that are needed to find the solution of the problem.
3. ✓ Decide the layout for the output.
4. ✓ Select a programming method best suited to solve the problem and then only carryout the coding, using a suitable programming language.
5. ✓ Test the program. Select test data so that each part of the program would be checked for correctness.

Scanned with CamScanner

6. Finally use data validation steps to guard against processing of wrongly inputted data.

All these six steps are further elaborated in the following sections.

Step 1. Understanding the Problem

Read each *statement* given in the problem carefully, so that you can answer the first question "What is expected by solving the problem?" Do not start drawing a flowchart or decision table straight away. Instead, read each statement of the problem slowly and carefully, by understanding the *keywords*. Use paper and pencil to solve the problem manually for some test data. Let us understand this point by solving the problem given in Example 1.

Example 1 Accept a value M and find the sum of first M even integers.

Solution The solution of this problem requires you to take an input value, say a number 6 is given as the value of M. Then, you should get the sum of first 6 even integers. In the first step, you should be able to answer the following two questions:

"What are the first 6 even integers?"

These are 2, 4, 6, 8, 10, and 12.

"What is their sum?"

The sum is 42. Hence, the solution is to be so made that the sum of first 6 even integers comes out to be 42.

Step 2. Construction of the List of Variables

In this step, you should think in advance the number of variables along with the names of the variables. The names chosen for the variables should be an aid to memory. For example, in the case of the problem stated in step 1 above, the variables may be, I, SUM and COUNT as given below.

1. Generate even integers 2, 4, 6,...(I)
2. Total the sum of even integers 2 + 4 + 6 + ... (SUM)
3. Count the number of even integers, i.e. 1, 2, 3,... (COUNT)

Thus, it is clear that we need to use the above three variables and one more variable "M" whose value will be inputted by the user of the program from the keyboard. Finally, the four variables for this problem would be:

M	to be entered by the user.
I	to generate even integers.
COUNT	to keep a track of the number of even integers that have been summed.
SUM	An accumulator that will hold the current total value of even integers.

Step 3. Output Design

Many ~~a~~ times, the 'output' format is specified in the problem itself, but sometimes, it may not be so. If the output format is not specified, we must keep in mind that the output report should be *easily* understandable by a reader. The headings should not cause any doubt or confusion in the mind of a reader.

In the solution of Example 1, the output format could be as follows:

Scanned with CamScanner

No. of first even integers	Total sum
6	42

You should keep one more point in mind. The programs and the solution to a problem are for other people (teachers, supervisors, contractors, etc). They will appreciate you only if they can understand the results and analyse them. Hence, the output format should have the following characteristics:

a. Attractive,
b. Easy to read and
c. Self-explanatory.

Step 4. Program Development

You should now draw a flowchart for the procedure that you have just made in steps 1, 2 and 3. Standard symbols should be used for drawing a flowchart. If a problem is complex, you should divide it into several simpler parts. Then, draw a flowchart for each part separately and combine them together using connectors.

A flowchart for the problem in Example 1 is drawn as shown in Figure 1.1.
Now write the code in the prescribed high level language to translate the flowchart into a program.

Step 5. Testing the Program

You should give a *dry* run to the developed program that translates the flowchart of step 4. This means by giving known values to the variables and by checking the result and thus comparing results with manually calculated values. Test values are so selected that each part of a flowchart is tested and the program is free from any logical errors.

Step 6. Validating the Program

It is quite likely that the user of your program may enter values, which are not expected by the program. Such values should be rejected by the procedure drawn by you. This is known as validation of data. For example, in the problem of Example 1, in step 1 we may give some upper limit to the value of "M" and state that "M" should be an *integer* value. Such types of checks can be included to validation a program.

1.1.2 Problem Definition and use of Examples for Problem Solving

Let us take another example of a problem faced in day-to-day life. Suppose you want to reach your college computer laboratory at 8 AM. You would lay out a plan to get ready by 7 AM, then take a bus/rikshaw and reach at the gate of your college. Then climb up the stairs to reach the computer science laboratory. For all these movements, you will note the time taken for each part. If due to some reason, you are unable to get ready by 7 AM and you already know that it takes one hour to reach from house to college by bus/rikshaw, then you will take a faster means of transportation. You may take an autorikshaw or a taxi. Thus, a very simple problem of reaching the computer laboratory of your college by 8 AM will need several steps for solution. Each step is to be accurately defined/marked so that no guess work is

Scanned with CamScanner

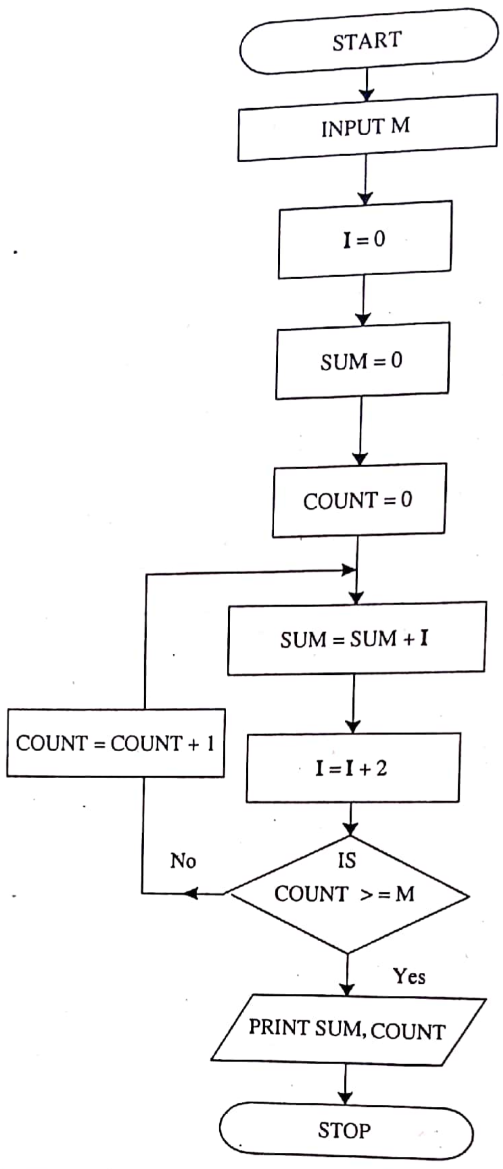


Figure 1.1 Flowchart for solving problem in Example 1

necessary. You can thus represent the solution of this problem in three steps as shown in Figure 1.2.

In Figure 1.2, steps 1, 2 and 3 appear to be very simple. In actual practice when you have to give instructions to a person, who is going for the first time, it may not be so easy. For example, you have to define the word "READY" precisely so that he knows exactly what he has to do by 7AM to get READY. Similarly, in step 2, you may have to clearly specify the bus route number, the bus stop to board the bus, the place to get down from the bus, etc. You may also like to tell that a bus is not to be boarded, if it is overcrowded. The word "overcrowded" needs to be exactly defined. Finally, step 3 needs further elaboration about the room number and floor number; where the computer laboratory is located and how to reach there.

You can explain all this to a person who does not know any thing about the computer laboratory, provided you *know* it correctly. In the same way, you can solve a problem on a computer, if you know exactly what the problem is and how to solve it *manually*.

Scanned with CamScanner

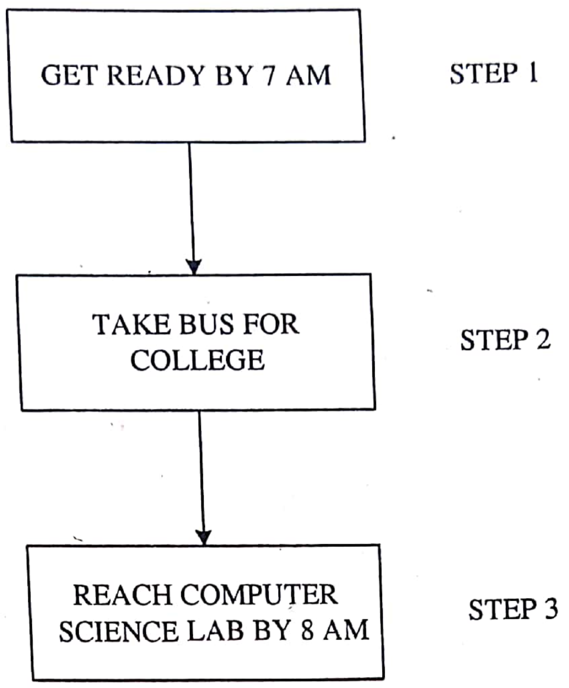


Figure 1.2 Step by step solution to reach college computer science laboratory (LAB)

1.2 TOP-DOWN AND BOTTOM-UP APPROACHES TO ALGORITHM DESIGN

1.2.1 Top-Down Approach of Problem Solving

Top-down design is the technique of breaking down a problem into various sub tasks needed to be performed. Each of these tasks is further broken down into separate *subtasks*, and so on till each subtask is sufficiently simple to be written as a self contained module or procedure. The entire solution of the problem will then consist of a series of simple modules and joining them together will make the complete task of solving the complex problem.

In top-down design, we initially describe the problem we are working with at the highest or most general level. The description of the problem at this level will usually be concerned with what must be done – not how it must be done. The description will be in terms of higher-level operations. We must take all of the operations at this level and individually break them down into simpler steps that begin to describe how to accomplish the tasks. If these simple steps can be represented as acceptable algorithmic steps, we need not split them any further. If that is not the case, then we split each of these second level operations individually into still simpler steps. This stepwise refinement continues until each of the original top-level operations have been described in terms of acceptable shortest (primitive) statements.

The top-down approach, starting at the general levels to gain an understanding of the system and gradually moving down to levels of greater detail, is done in the analysis stage. In the process of moving from top to bottom, each component is exploded into more and more details.

Scanned with CamScanner

Thus, the problem at hand is analysed or broken down into major components, each of which is again broken down if necessary.

☞ *Top-down process involves working from the most general form, down to the most specific form.*

The design of modules is reflected in an hierarchy chart such as the one shown in Figure 1.3. The purpose of the procedure *Main* is to co-ordinate the three branch operations e.g. *Get*, *Process* and *Put* routines. These three routines communicate only through *Main*. Similarly, *Sub1* and *Sub2*, can communicate only through the *Process* routine.

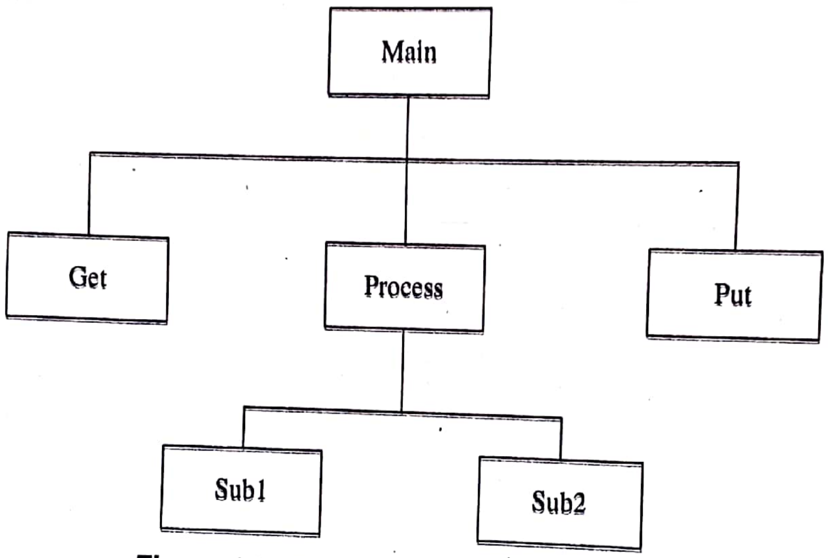


Figure 1.3 Top down modular design

☞ *Using the top-down approach, attention is first focussed on global aspects of the overall system. As the design progresses, the system is decomposed into subsystems and more consideration is given to specific issues.*

Advantages of Top-down approach

a. This approach allows analyst to remain "on top of" a problem and view the developing solution in the context. The solution always proceeds from the highest level to the lowest level. With other techniques, we may find ourselves bogged down with very low-level decisions at a very early stage. It will be difficult to make such decisions if it is not clear as to how they may affect the overall solution of a problem.

b. This would be a very good way to delay decisions on problems whose solution may not be readily available. At each stage in the development, the individual operation will be split up into a number of elementary steps.

c. By dividing the problem into a number of subproblems, it is easier to share problem development. For example, one person may solve one part of the problem and the other person may solve another part of the problem.

Scanned with CamScanner

d. Since the program may have bugs and the debugging time grows quickly when the program is long, it will be easier to debug a long program which is divided into a number of smaller segments or parts. The top-down development process specifies a solution in terms of a group of smaller, individual subtasks. These smaller tasks would become the ideal units of program for testing and debugging.

If we add a new piece of code, say "p" to the overall program "P", and an error condition occurs, we can definitely state that the error must be either in "p" itself or in the interface between "p" and "P", because "P" has been previously checked and certified to be correct.

☞ *By testing a program broken into smaller pieces, we greatly simplify the time consuming debugging process. In addition, we will have the satisfaction of knowing that everything we have coded so far is correct.*

e. Another advantage of top-down development process is that it becomes an ideal structure for managing and implementing a program using a team of programmers. A senior programmer can be made responsible for the design of a high-level task and its decomposition into subtasks. Each of such subtasks can then be "given out" to a junior programmer who works under the direction of the senior staff. Since software projects are carried out by teams of two or more programmers, this top-down characteristic helps in faster completion of the solution of the complex problem.

☞ *In summary, top-down method is a program design technique that analyses a problem in terms of more elementary subtasks. Through the technique of stepwise splitting, we expand and define each of the separate subtasks until the problem is solved completely. Each subtask is tested and verified before it is expanded further.*

In addition to above the following are three more advantages:

a. Increased comprehension of the problem.
b. Unnecessary lower-level detail are removed.
c. Reduced debugging time.

1.2.2 Bottom-up Approach of Problem Solving

When faced with a large and complex problem, it may be difficult to see how the *whole* thing can be solved. It may be easier to solve parts of the problem individually, taking the easier aspects first and thereby gaining the insight and experience to tackle the more difficult tasks, and finally join each of the solutions together to form the complete solution. This is called a bottom-up approach.

This approach suffers from the *disadvantage* that the parts of the solutions or programs may not fit together easily. There may be a lack of consistency among modules, and thus, more re-programming may have to be carried out. Hence, this approach is not very much favoured.

Scanned with CamScanner

1.3 USE OF ALGORITHMS IN PROBLEM SOLVING

A set of instructions which describes the steps to be followed to carry out an activity is called an *algorithm* or *procedure* for solving a problem. If the algorithm is written in a language that the computer can understand, then such a set of instructions is called a *program*.

The task of writing a computer program involves going through several stages. So programming requires more than just writing programs. This also requires writing algorithms, as well as testing the program to locate all types of errors.

We can view an algorithm as a way to solve a problem or in other words, as a set of directions, that tell us exactly how to go about for getting the desired results. For example, we can use following steps or actions to reach the office by 10 AM.

Action

1. Get ready by 8 AM.
2. Take breakfast
3. Board the chartered bus at 9 AM.
4. Reach office at 10 AM.

We can thus, define an *algorithm* as an ordered sequence of well-stated and effective operations that, when executed, will produce the results.

By "ordered sequence" we mean that after the completion of each step in the algorithm, the next step is clearly defined. We must know exactly where to look for the next instruction. The ordering could, for example, be specified by writing a number to the steps with the positive integers and following the sequencing rule.

An algorithm must always have one clearly understood starting point and one or more clearly understood ending points.

The starting point can be implied – we usually assume that we are to start at step 1 – or it can be stated explicitly (one step may be labeled START). More than one starting point would create confusion about where to start, violating the ordering condition just stated.

It is perfectly acceptable to identify one or more of the steps in the algorithms as *terminators* – steps that, when executed, end the execution of the entire algorithm. However, regardless of which section we execute, we wish to stop after completing that section.

The outline, given in Figure 1.4, contains a single clearly identified starting point (step 1) and three clearly defined terminators (steps 5, 8, and 11). The set of steps executed will always be either (1, 2, 3, 4, 5), or (1, 2, 6, 7, 8), or (1, 2, 9, 10, 11).

1. START
2. Make a decision that divides the problem into three possible cases:

	Case 1		Case 2		Case 3
3.	———	6.	———	9.	———
4.	———	7.	———	10.	———
5.	STOP	8.	STOP	11.	STOP

Figure 1.4 Outline for solving a problem

Scanned with CamScanner

The existence of one or more terminators, however, is insufficient to guarantee that execution will eventually stop.

Another fundamental characteristic of algorithms is that each individual operation must be both "effective" and "well defined". By "effective" we mean that some formal method must exist for carrying out that operation and also getting an answer. For example:

a. compute n × (n + 1),
b. determine if *x* is even,
c. wait for one hour and 10 minutes,
d. take the square root to a maximum of three decimal place accuracy.

are all effective operations, and could be easily carried out.

But look at the following statements:

e. Compute n ÷ 0 i.e. divide n by zero.
f. Determine the largest prime number.
g. Write the exact value for the square root of 2.
h. Wait for –1 hour.

The above stated steps are not effective. They either cannot be evaluated or we have no idea how to go about answering them.

Each individual operation must be "well defined" — which means, that they are clearly understandable and without an iota of confusion to the person or machine that is executing the algorithm. A basic question about algorithm concerns the type of well-defined operations we are allowed to write at each step. That is, what are the building blocks from which we can compose an algorithm? These building blocks, are also called *primitives.*

For example:

MIX *x* INTO *y*
MEASURE *x* units OF ingredient
STIR *x*
COOK *x* AT *y*°c for *z* minutes
COOK *x* UNTIL temperature = 100°c

The statements whose meaning is clear to most people would be more realistic examples of cooking *primitives*. However, even these simple statements may not be acceptable *primitives* if there is any *confusion* or *uncertainty* about their meaning . For example, does COOK mean bake, boil, fry, or grill?

1.3.1 Developing an Algorithm

The process of developing an algorithm to solve a specific problem is a trial–and–error process that may need numerous attempts. Programmers will make some initial attempt at a solution and review it to test its correctness. The errors they discover will usually lead to insertions, deletions, or modifications in the existing algorithm. It may need scrapping the old one and beginning a new solution.

Scanned with CamScanner

1.3.2 Characteristics of Algorithmic Language

Some characteristics of the algorithm language are given below:

a. By a judicious choice of algorithmic language primitives, we can ensure that the final algorithm will be closely related to the desired programming language. Thus, it will facilitating the next step of translating the algorithm into that language progam.

b. We will however be bogged down in the restrictive syntax of a specific language. An algorithmic language should be viewed as a set of guidelines for building procedures but not as a rigid set of rules. There will not be any rules for punctuation, spelling, vocabulary, or use of synonyms.

c. The representation of algorithms in our algorithmic language, along with a judicious use of indentation, will clearly indicate the relationships among various statements and thus allow us to gain a better picture of the overall organization and structure of the solution of the problem.

Example 2

Write an algorithm to find the sum of the first k integers, 1, 2, ..., k. The value of k will be an external input to the algorithm. (Be sure to handle the illegal situation of k <= 0.)

```
BEGIN
  read k
  if k <= 0
  then
     write "illegal value for k"
     END
  else
     set i to 1
     set sum to 0
     repeat k times
        add i to sum
        increment i by 1
     end of the repeat loop
     write "the sum of the first", k, "integers is", sum
END
```

Example 3

Develop an algorithm to solve quadratic equations of the form $ax^2 + bx + c = 0$ using the quadratic formula:

$$\text{Roots} = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

Your algorithm should handle the regular cases as well as the special cases of:

a. a double root if $(b^2 - 4ac = 0)$,

b. complex root if $(b^2 - 4ac < 0)$,

c. a nonquadratic equation $(a = 0)$,

Scanned with CamScanner

d. an illegal equation if ($a = 0, b = 0$).

Solution

```
BEGIN
  read a, b, c
  if a = 0 and b = 0
  then write ''illegal equation, cannot solve''
  else
    if a = 0 then
      set root to - c/b
      write ''linear equation, the one root is'', root
    else
      set discriminant to b² - 4ac
    if discriminant < 0 then
      write ''roots are complex, cannot be solved''
    else
      set root1 to $\frac{-b + \sqrt{discriminant}}{2a}$
      set root2 to $\frac{-b - \sqrt{discriminant}}{2a}$
  write ''answers are'' , root1, root2
END
```

1.4 DESIGN OF ALGORITHMS

In designing algorithms we need methods to separate bad algorithms from good ones. This is because there are usually more than one possible ways to solve a problem. It is for us to decide which method will prove the most effective for the solutions of that problem.

An algorithm is clearly specified set of simple instructions to be followed to solve a problem. Once an algorithm is given for a problem and decided to be correct, another important step would be to determine how much time or space, the algorithm will require.

Each of the algorithms will involve a particular data structure. Accordingly, we may not always to able to use the most efficient algorithm, since the choice of data structure depends on many things, including the type of data and the frequency with which various operations on data are applied. The choice of data structure may involve time-space tradeoff, i.e. by increasing the amount of space for storing the data, one may be able to reduce the time needed for processing the data or vice versa.

Short in [illegible] of points

☞ *Once the data structure is chosen for a particular application, program is written by the set of logical instructions that manipulate the related data items. Thus, a study of data structure is also a study of the algorithms that control them.*

Certain qualities can be identified as desirable traits in all such algorithms.

Scanned with CamScanner

First, the algorithm must be expressed in a fashion that is completely free of *ambiguity*. The method we use to express algorithm must be formed to avoid the impercision inherent in a natural language. The algorithm should be flexible enough to allow us to focus on problem-oriented issues.

Second, algorithms should be *efficient*. They should not unnecessarily use memory locations nor should they require an excessive number of logical operations. To analyze the efficiency of an algorithm, we will have to describe numerically the memory and logical operations that they may require.

Third, algorithm should be *concise* and *compact* to facilitate verification of their correction. Verification involves observing the performance of the algorithm with a carefully selected set of test cases. These test cases should attempt to cover all of the exceptional circumstances likely to be encountered by the algorithm.

1.4.1 How to Design an Algorithm?

Algorithm design is a creative activity. The first step in designing an algorithm is to produce a clear specification of the problem. For designing any algorithm, some important things should be considered. These are run-time, space and simplicity of algorithm. In some cases, input data may also be decide in designing an algorithm.

Some common approaches for designing algorithms are:

- **Greedy Algorithm:** The greedy algorithm works in steps. In each step this algorithm selects the best available option until all options finish. This approach is widely used in many places for designing algorithm. For example, the *shortest path algorithm* (discussed in Chapter 8).
- **Divide and Conquer:** Divide and conquer is a design strategy which is well known for breaking down the efficiency barriers. When the method is applied, it often leads to a large improvement in time complexity. In divide and conquer, the big problem is divided into same type of smaller problems and we design the algorithm to combine the implementation of these smaller problems for implementing bigger problem. For example, in *quick sort* (discussed in Chapter 7), we divide initial list into several smaller lists, after sorting those smaller lists, we combine them and get the final list sorted.
- **Non-recursive algorithm:** A set of instructions that perform a logical operation can be grouped together as a function. If a function calls itself, then it is called as *direct recursion*. If a function calls another function, which in turn invokes the calling function, then the technique is called as *indirect recursion*. The recursion is very powerful technique which is supported by programming language C.
- **Randomized algorithm:** In randomized algorithm, we use the feature of random number instead of a fixed number. Performance of some algorithm depends upon the input data. It gives different results with different input data.
- **Backtrack algorithm:** An algorithm technique to find solutions by trying one of several choices. If the choice proves incorrect, computation backtracks or restarts at the point of choice and tries another choice. A backtracking algorithm is to write a function or procedure which traverses the solution space. For example, *game tree*.

Modular programming approach: In industry and commerce, the problems that are to be solved with the help of computer needs thousands or even more number of lines of code. The importance of splitting a problem into a series of self contained modules then becomes important. A module should not exceed about 100 or so lines and should preferably be short enough to be on a single page. Since module is small, it is simpler to understand it as a unit of code. It is therefore, easier to test and debug a shorter module especially if its purpose is clearly defined and documented. In a very large project, several programmers may be working on a single problem. Using modular approach, each programmer can be given a specific set of modules to work on. This enables the whole program to be completed soon.

As soon as one can write an algorithm, it is necessary to learn how to analyze an algorithm. The analysis of an algorithm provides information that gives us a general idea of how long an algorithm will take for solving a problem.

To judge an algorithm, there are many criteria. Some of them are as follows:

- It must work correctly under various conditions.
- It must solve the problem according to the given specification.
- It must be clearly written following the top-down or bottom up strategy.
- It must make efficient use of time and resources available.
- It must be sufficiently documented so that anybody can understand its implementation.
- It must be easy to modify, if required.
- It should not be dependent on being run on a particular type of computer.

1.5 IMPLEMENTATION OF ALGORITHM

After spelling out completely and precisely the requirements for each tasks and sub-tasks (function), it is time to code them into our programming language (say C, C++ or Java etc.). Just as we design from the top down, we should code from the top down. Once the specification at the top levels are complete and precise, we should code the subprograms at these levels and test them appropriately.

1.6 VERIFICATION OF ALGORITHM

Algorithm verification is a proof that the algorithm will accomplish its task. This kind of proof is formulated by looking at the specifications for the subprograms and then arguing that these specifications combine properly to accomplish the task of the whole algorithm.

Verification of algorithm would consist of determining the quality of the output received. In other words, it is a process of measuring the performance of the program with any laid down standards. The feedback so obtained, then sets the basis for making necessary changes in the already designed algorithm specifications. Algorithm verification should precede coding of the programme.

1.7 EFFICIENCY ANALYSIS OF ALGORITHMS

An algorithm analysis provides information that gives us an idea of how long an algorithm will take for solving a problem. For comparing the performance of two algorithms, we have to estimate the time taken to solve the same problem using each algorithm for a set of N

Scanned with CamScanner

input-values. For example, we might determine the number of comparisons a searching algorithm does to find a value in a list of N values. Similarly, we might determine the number of arithmetic operations an algorithm performs to add two matrices of size N * N. A number of algorithms might be able to solve a problem successfully yet the analysis of algorithm gives us the scientific reason to determine which algorithm should be chosen to solve the problem most efficiently.

☞ *The performance of algorithms can be judged by criteria such as whether it satisfies the original specification of task, whether the code is readable. These factors affect the computing time and main memory requirements of the machine.*

1.7.1 Space Complexity

Space complexity of a program is the amount of main memory in a computer. It needs to run for completion. The space needed by a program is the sum of the following components:

a. Fixed part that includes space for the code, space for simple variables and fixed size component variables as well as, space for constants etc..

b. Variable part that consists of the space needed by component variables whose size is dependent on the particular problem instance being solved, and the stack space used by recursive procedures.

1.7.2 Time Complexity

Time complexity of a program is the amount of computer time it needs to run a program to completion. Suppose, space is fixed for one algorithm then only run time will be considered for obtaining the complexity of algorithm. The time complexity may give time for:

a. Best case

b. Worst case

c. Average case

Best Case

Most of the algorithms behave in *best case*. In this case, algorithm searches the elements in first time itself. For example, in linear search, if it finds the element at first time itself then it behaves as best case. Best case takes shortest time to execute, as it causes the algorithms to do the least amount of work.

Worst Case

In *worst case*, we find the element of time at the end or the total time taken when searching of elements fails. This could involve comparing the key to each list value for a total of N comparisons. For example, in linear search, suppose the element for which algorithm is searching, is the last element of an array or the element is not available in array then algorithm behaves as worst case.

Scanned with CamScanner

Average Case

Analysing the average case behaviour of an algorithm is more complex than best case and worst case. Here, we take the probability with the list of data items Average case of algorithm should be the average number of steps but since data can be at any place, so finding exact behaviour of algorithm is difficult. As the volume of data increases, average case of algorithm behaves like worst case algorithm.

1.7.3 Frequency Count

To make an analysis machine independent it can be assumed that every statement will take the same constant amount of time for its execution. Hence, the determination of time complexity of a given program is the summing of the frequency counts of all the statements of that program. The time complexity can also be expressed to represent only the order of magnitude represented by the frequency counts. One such notation for frequency count is Order notation ('O' notation).

'O Notation'

'O' notation is used to measure the performance of any algorithm. Performance of an algorithm depends upon the volume of input data. 'O' notation is used to define the order of the growth for an algorithm.

We write O(1) to mean a computing time that is a constant i.e. when we get data in first time itself. For example, in hash table,

O(n) is called linear, when all the elements of the linear list will be traversed. For example, the best case while using bubble sort method.

O(n^2) is called quadratic, when all the completed list will be traversed for each element. For example, the worst case of bubble sort method.

O(log n) is when we divide linear list to half each time and traverse the middle element. For example, method used in binary searching.

O(n log n) is when we divide list half each time and traverse that half portion. For example, best case of quick sort of an algorithm will take O(log n) time. This method is faster than O(n). However, O(n log n) is better than O(n^2) but not as good as O(n).

1.8 SAMPLE ALGORITHMS

Some examples of algorithms are given in the following sub-sections.

1.8.1 Exchanging the Value of Two Variables

Exchanging the value of two variables means interchanging their values. We can clearly understand this by an example.

Suppose we have two variables x and y. Our aim is to swap or interchange the value of x and y.

The original values of x and y are:

$$\begin{array}{cc} \boxed{11} & \boxed{58} \\ x & y \end{array}$$

Scanned with CamScanner

☞ *For storing a binary number, the electronic elements of a computer need to have only two sta ble states. The output of such electronic circuits at any time is either high or low. These sta represent 1 and 0 respectively. High is represented by 1 and low is represented by 0.*

A computer can directly understand only binary number system. But we use data in th form of decimal digits. All such numbers are converted to binary codes within the comput because computers can operate using binary numbers only.

Conversion of Decimal Number to Binary Number

To convert a decimal number to a binary number, the decimal number is divided by successively. The quotient and remainders are noted down at each stage. The Quotient of t preceding stage is divided by 2 at the next stage. The process is repeated until the quoti becomes zero. Now write the remainders to right hand side (R.H.S) of the previous rema der to get a set of digits. To make binary equivalent, put the remainders in reverse order. example, the binary representation of the decimal number 13 is shown below:

	Quotient	Remainder	
$13 \div 2 =$	6	1	↑ LSB
$6 \div 2 =$	3	0	
$3 \div 2 =$	1	1	
$1 \div 2 =$	0	1	MSB

MSB = Most Significant Bit
LSB = Least Significant Bit

So binary representation of $(13)_{10}$ will be $(1101)_2$.

Algorithm

1. Begin
2. Get the decimal number whose binary equivalent we want.
3. Divide the decimal number by 2.
4. Write the remainder.
5. Quotient of previous stage become the dividend and again divide it by 2 at next stag
6. Write the remainder to right hand side (R.H.S) of previous remainder.
7. Repeat the process successively till quotient is zero.
8. The set of digits formed by the remainders at different stages, when placed in reverse order, will form the binary equivalent of the decimal number.
9. Stop.

1.8.4 Reversing Digits of an Integer Number

Reversing digits basically means changing the digits order backwards. For example:

Input	3	5	6	8
Result	8	6	5	3 (Each digit

After swapping the result should be:

$$\begin{array}{ccc}\boxed{58} & & \boxed{11}\\ x & & y\end{array}$$

To do so, we need one temporary variable, i.e. t. First, copy the value of x into t, i.e.

$$\begin{tikzcd}\boxed{11} \arrow[r] & \boxed{11}\\ x & t\end{tikzcd}$$

then copy the value of y into x, i.e.

$$\begin{tikzcd}\boxed{58} \arrow[r] & \boxed{58}\\ y & x\end{tikzcd}$$

copy the value of t into y i.e.

$$\begin{tikzcd}\boxed{11} \arrow[r] & \boxed{11}\\ t & y\end{tikzcd}$$

At last, we shall get the new values in x and y as seen below:

$$\begin{array}{ccc}\boxed{58} & & \boxed{11}\\ x & & y\end{array}$$

We find that the value of x and y have now been interchanged as desired.

Algorithm

1. Begin
2. Get the values of variables x and y.
3. Assign the value of x to t.
4. Assign the value of y to x. So x has the original value of y now in place of the original value of x.
5. Assign the value of t to y.
6. Show the values of x and y.
7. Stop

1.8.2 Summation of a Set of Numbers

Summation of a set of numbers is like adding numbers given in a series. To add a set of numbers manually, we are used to start adding the digits in the right most column. For example:

```
  782
  222
  565
 -----
  ...9
 -----
```

Scanned with CamScanner

When using computer, we must design an algorithm to perform this task by a different approach. Computer can add two numbers at a time and return the sum of two numbers.

So to add n numbers, we initialize the variable location S, where we are going to store the Sum, by the assignment $S = 0$. Now, we first do the following:

$S = S + a_1$ where a_1 is the first number

then we do $S = S + a_2$ (The new value of S contains $a_1 + a_2$)

then $S = S + a_3$ (The new value of S contains $a_1 + a_2 + a_3$).

and so on till $S = S + a_n$

Here, we are repeating the same process again and again. The only difference is that the values of a and S changes their values with each step.

Algorithm

```
1.  Begin
2.  Read n numbers to be summed.
3.  Initialize sum as 0.
4.  Initialize 'count' as 1.
5.  While 'count' is less than or equal to n, i.e. numbers to be added, repeatedly do:
    a.  Read the number at position 'count', i.e. when count is 1 read 1st number, when
        count is 2, read second number and so on.
    b.  Update the current sum by adding to it the number read.
    c.  Add 1 to 'count'.
6.  Write the sum of n numbers. (After the number at nth count has been added, the control
    will shift to step 7.)
7.  Stop.
```

1.8.3 Decimal Base to Binary Base Conversion

Before doing such a conversion, we need to know definition of decimal number system and binary number system.

Decimal Number System

It is the set of numbers having base 10. Since a decimal number system uses 10 digits from 0 to 9, it has a base 10. The decimal number system is also called base 10 system. For example, 6598, 75, 100567 etc. are examples of decimal numbers.

Binary Number System

Binary means 2. The set of numbers having base 2 is called binary number system.

Binary numbers consist of only two digits, i.e. 0 and 1. Each one of these is also called bit, a short form of **bi**nary digi**t**.

☞ *For storing a binary number, the electronic elements of a computer need to have only two stable states. The output of such electronic circuits at any time is either high or low. These states represent 1 and 0 respectively. High is represented by 1 and low is represented by 0.*

A computer can directly understand only binary number system. But we use data in the form of decimal digits. All such numbers are converted to binary codes within the computer because computers can operate using binary numbers only.

Conversion of Decimal Number to Binary Number

To convert a decimal number to a binary number, the decimal number is divided by 2 successively. The quotient and remainders are noted down at each stage. The Quotient of the preceding stage is divided by 2 at the next stage. The process is repeated until the quotient becomes zero. Now write the remainders to right hand side (R.H.S) of the previous remainder to get a set of digits. To make binary equivalent, put the remainders in reverse order. For example, the binary representation of the decimal number 13 is shown below:

	Quotient	Remainder	
$13 \div 2 =$	6	1	↑ LSB
$6 \div 2 =$	3	0	
$3 \div 2 =$	1	1	
$1 \div 2 =$	0	1	MSB

MSB = Most Significant Bit
LSB = Least Significant Bit

So binary representation of $(13)_{10}$ will be $(1101)_2$.

Algorithm

1. Begin
2. Get the decimal number whose binary equivalent we want.
3. Divide the decimal number by 2.
4. Write the remainder.
5. Quotient of previous stage become the dividend and again divide it by 2 at next stage.
6. Write the remainder to right hand side (R.H.S) of previous remainder.
7. Repeat the process successively till quotient is zero.
8. The set of digits formed by the remainders at different stages, when placed in the reverse order, will form the binary equivalent of the decimal number.
9. Stop.

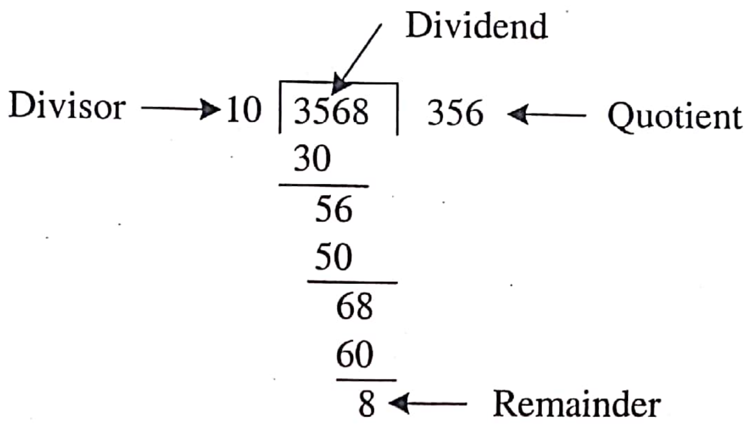
1.8.4 Reversing Digits of an Integer Number

Reversing digits basically means changing the digits order backwards. For example:

Input	3 5 6 8
Result	8 6 5 3 (Each digit placed in the reverse)

Scanned with CamScanner

To do so, we divide the number by 10 and print the remainder.



We can get the remainder by mod() function
$r = 3568 \bmod (10)$
$= 8$
Now remove 8 from the original integer. So we next get 356 (number to be reversed) again
$r = 356 \bmod (10)$
$= 6$

Successively, we find out the remainders until the dividend is less than 10. Now place the digits of the remainders in the reverse.

Therefore, we can use the integer division by 10 to remove the right most digit from the number being reversed and construct the reversed integer by writing the extracted digits to the right hand side of the current reversed number representation. At last, when the dividend is less than 10, then add it to the right side of the current reversed number representation.

Number to be Changed	**Reversed-digits Number**
3568	8
356	86
35	865
3 (i.e. less than 10)	8653

Algorithm

```
1.  Begin
2.  Get positive integer number to be reversed.
3.  While the integer number being reversed is greater than 10 repeatedly do:
    a.  Extract the right most digit of the number to be reversed by remainder function, i.e.
        function mod().
    b.  Construct the reversed integer by writing the extracted digit to right hand side of
        the current reversed number.
```

Scanned with CamScanner

4. Write the integer (dividend which is less than 10) to the R.H.S of the reversed number.
5. Stop

1.8.5 GCD (Greatest Common Divisor) of Two Numbers

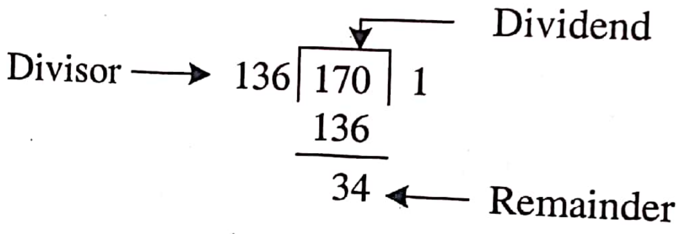
The greatest number which is a common factor of two or more given numbers is called GCD (Greatest Common Divisor).

How to find GCD

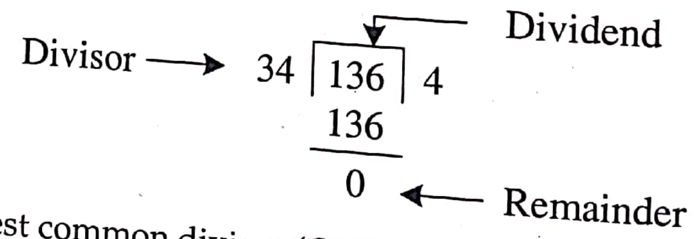
Suppose two numbers are given. Now, divide the greater number by the smaller one. Next, divide the divisor by the remainder. Go on repeating the process of dividing the preceding divisor by the remainder last obtained, till the remainder zero is obtained. Then, the last divisor is the required GCD of the given numbers. Suppose, we want to find GCD of 136 and 170, then

Say, $a = 136$
$b = 170$

Greater number is b i.e. 170; so b will be dividend.
Smaller number is a, i.e. 136; so a will be divisor.



Remainder i.e. 34 is now divisor



Therefore, the greatest common divisor (GCD) is 34.

Algorithm

1. Begin
2. Get 1st and 2nd (non-zero) integers, i.e. a, and b.
3. Check which integer is larger.
4. Get the remainder by dividing the greater integer by the smaller integer.
5. Make smaller integer be dividend and get the remainder.
6. Let remainder be the divisor.
7. Repeat step 4, 5, and 6 until a zero remainder is obtained. Now, the last divisor is the GCD.
8. Stop

Scanned with CamScanner

1.8.6 Test whether the Given Number is a Prime Number!

The natural number 1 has only one factor. Except 1, every decimal number has two or more factors. For example, 2 has two factors 1 and 2, the number 3 has the factors 1 and 3. The number 4 has three factors 1, 2 and 4, the number 5 has two factors 1 and 5. The positive numbers which have only two factors i.e. 1 and that number itself are called prime numbers. The example of prime numbers are 2, 3, 5, 7, 11, 13, 17, etc.

☞ ***Prime numbers** are any of the positive integer numbers greater than 1, each divisible only by itself and by 1. Thus, 2, 3, 5 ,7,....... etc. are prime numbers.*

Our aim here is to find the prime numbers. A Greek mathematician Erotosthenes who lived in the 3rd century BC gave a very simple method for finding a prime number. The method is known as "*Sieve method*". Suppose we have to find all the prime numbers in the range of 1 to 20. To find the prime number, follow the steps:

1. Strike out 1 because it is not a prime number.
2. Encircle 2 and strike out all the multiples of 2, i.e. 4, 6, 8, 10, 12, ... 20.

   ~~1~~ ② 3 4 5 ~~6~~ 7 ~~8~~ 9 ~~10~~
   11 ~~12~~ 13 ~~14~~ 15 ~~16~~ 17 ~~18~~ 19 ~~20~~

3. Encircle 3 and strike out all the multiples of 3.

   ② ③ 5 7 ~~9~~ 11 13 ~~15~~ 17 19

We continue the process of encircling and striking till every number in list is either encircled or struck out.

② ③ ⑤ ⑦ ⑪ ⑬ ⑰ ⑲

All the circled number are thus prime numbers and the numbers that are struck out except 1 are composite numbers. This method is given as an algorithm below.

Algorithm

1. Begin
2. Get the number which we want to check whether it is prime or not.
3. Set value of divisor to 2.
4. Divide the number by divisor (i.e. 2 first)
   If remainder is zero, then the given number is not prime, else the process is to be continued.
5. Increment divisor by 1, divide the number by the incremented divisor and check if remainder is zero. Keep repeating the process till either the remainder becomes zero or until divisor is equal to the number itself. If remainder becomes zero at any stage, the number is not prime else it is prime if carried to last stage, till incremented divisor is the number itself.
6. Stop.

Scanned with CamScanner

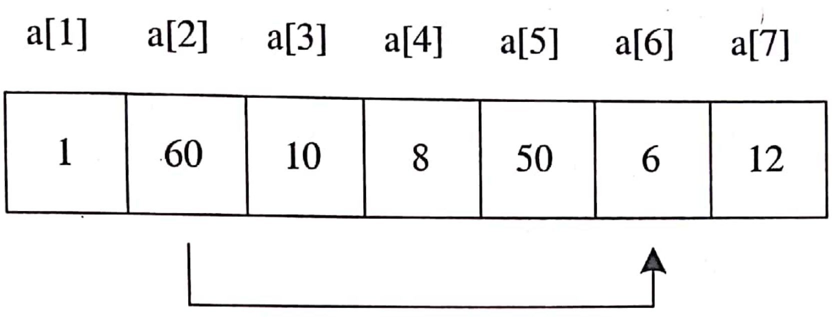


Figure 1.5(d)

Swap the value of *a*[2] and *a*[6] with the help of temporary variable, *temp*. See Figure 1.5(e).

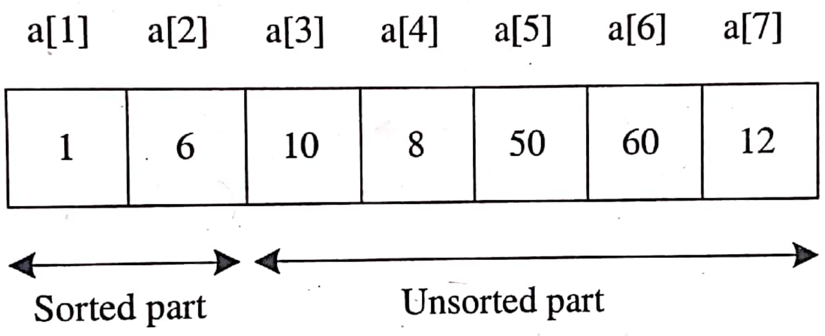
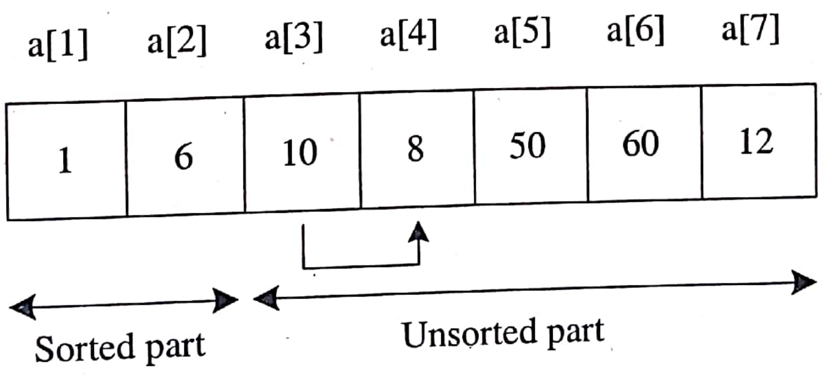


Figure 1.5(e)

Repeat the above steps to get the 3rd element, then 4th, and so on. See Figure 1.5(f).



Note that we need to process the unsorted array only $(n-1)$ times. The last element of the array would now be in the ascending order.

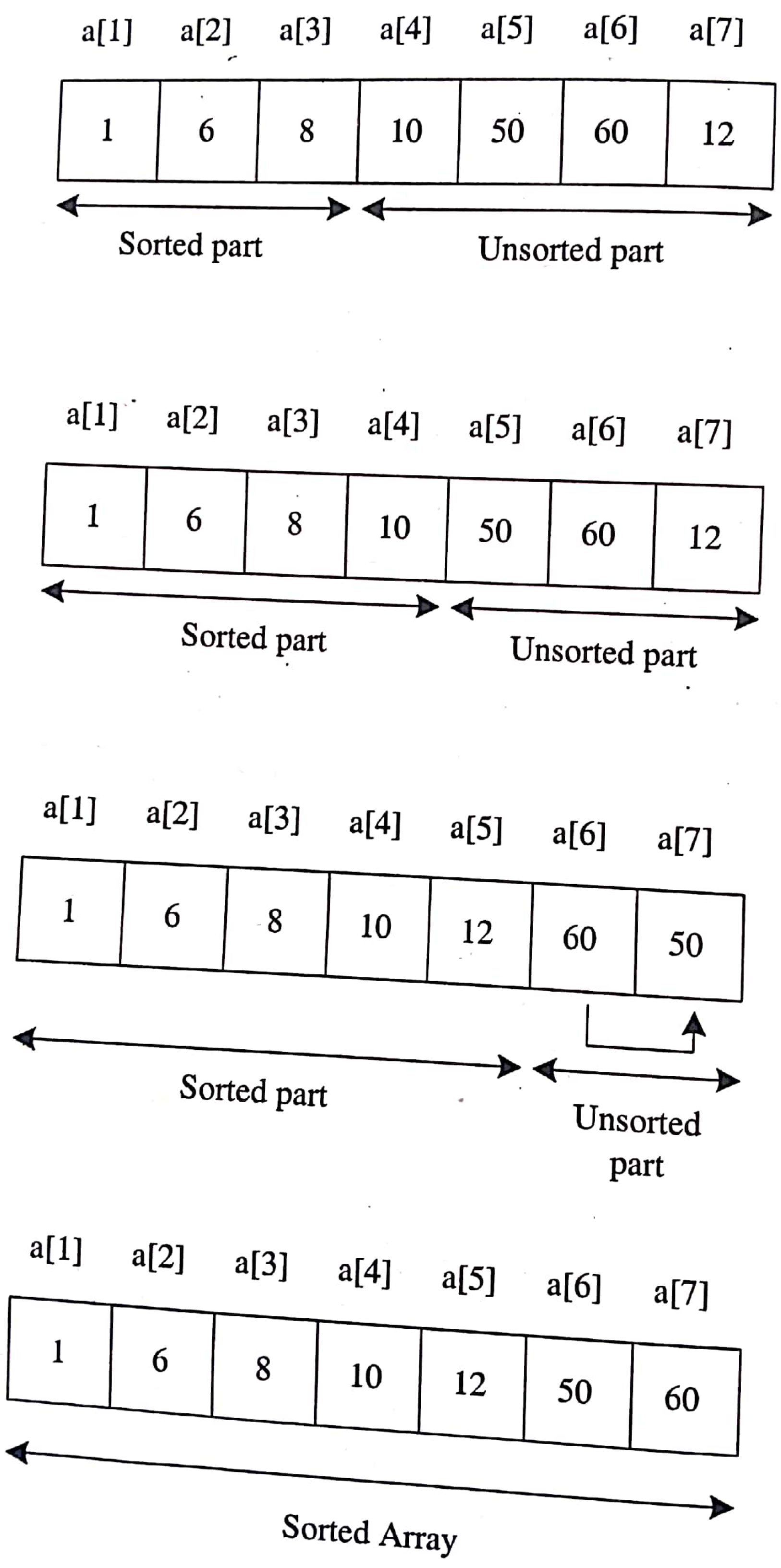


Figure 1.5(f)

Scanned with CamScanner

Algorithm

1. Begin
2. Read numbers and store them in an array of n elements i.e. size of array a is n with elements as $a[1]$, $a[2]$, $a[3]$, ..., $a[n]$.
3. Find the smallest element within unsorted array.
4. Exchange i.e. swap smallest value of the array element and store it in 1st element of the unsorted part of array.
5. Repeat steps 3 and 4 until all the elements in the array are arranged in the ascending order.
6. Stop

1.8.8 Find the Square Root of an Integer Number

Before understanding square root, we must know what we mean by squaring a number. To square a number is to multiply the number by itself. For example:

$$1^2 = 1 \times 1$$
$$= 1$$
$$2^2 = 2 \times 2$$
$$= 4$$
$$3^2 = 3 \times 3$$
$$= 9$$

However, one of the two equal factors of a number is the *square root* of that number.

Let us read some more examples to understand square-root.

$1^2 = 1 \times 1 = 1$ The square root of 1 is 1.

$2^2 = 2 \times 2 = 4$ The square root of 4 is 2.

$3^2 = 3 \times 3 = 9$ The square root of 9 is 3.

Therefore, if we have to find the square root of 49 then we will see "what number multiplied by itself will give 49". Certainly $7 \times 7 = 49$. But (-7×-7) is also 49.

So, ± 7 is the square root of 49.

☞ *A square root of a given number **n** is that natural number which when multiplied by itself gives **n**.*

We can say that if a is the square root of b then b is the square of a. For example,

$\sqrt{49} = \pm 7$ (±7 is square-root of 49; 49 is square of +- 7)

$\sqrt{64} = 8$ (±8 is square-root of 64; 64 is square of +- 8)

Scanned with CamScanner

Algorithm

1. Place bar over every pair of digits starting with the right hand side (R.H.S). The numbe of bars indicates the number of digits in the square root.

$$\overline{5}\,\overline{29}$$

2. Find the largest square root for the first pair that is less than or equal to the first pair. Take the square root of this number as divisor and get quotient. Put the quotient above the period. Subtract the product and bring down the next pair of digits to the right of the remainder.

$$\begin{array}{r|l} & 2 \\ \hline 2 & \overline{5}\,\overline{29} \\ & 4 \\ \hline & 129 \end{array}$$

3. Double the quotient as it appears and enter it with a blank on the right for the next digit, as the next possible divisor.

$$\begin{array}{r|l} & 23 \\ \hline 2 & \overline{5}\,\overline{29} \\ & 4 \\ \hline 43 & 129 \end{array}$$

4. Guess a possible digit to fill the blank and also to become the new digit in the quotient. Enter this digit in both places. Multiply the new digit in the quotient and the new divisor. Subtract and bring down the next period.

$$\begin{array}{r|l} & 23 \\ \hline 2 & \overline{5}\,\overline{29} \\ & 4 \\ \hline 43 & 129 \\ & 129 \\ \hline & 0 \end{array}$$

5. Repeat steps 3 and 4 till the last pair has been used and the remainder comes to zero.

1.8.9 Factorial of a Given Number

The product of all positive integers from 1 to n is called the 'factorial n' or 'n factorial'. It is denoted by $n!$. For example,

$$5! = 1 \times 2 \times 3 \times 4 \times 5$$

$$= 120$$

$$n! = 1 \times 2 \times 3 \ldots..(n-2) \times (n-1) \times n$$

where n is a positive integer.

☞ *It may be noted that 0! = 1, which means factorial zero is equal to 1.*

Scanned with CamScanner

Let us take an example. Suppose we want to find out the value of factorial 5.

Initially assign the value of $i = 1$ and *factorial* = 1

So method will be to successively go on increasing the value of i by 1 and keep updating the value of *factorial* by multiplying the current value of i by the value of factorial till the incremented value of i equals to the number desired for finding the factorial.

Thus, factorial = previous value of factorial $\times\ i$,
i.e. start with $i = 1$, giving factorial $1 = 1 \times 1$

Now increment the value of i by 1, giving updated

$$\text{factorial } 2 = 1 \times 2 = 2$$

Again increase the value of i by 1, giving

$$\text{factorial } 3 = 2 \times 3 = 6$$

Increment the value of i until the value of i becomes 5 and corresponding value of factorial is obtained.

To summarize, the factorial will be calculated in the following way:

$$\begin{aligned}\text{factorial } 1 &= 1 \times 1 = 1\\ \text{factorial } 2 &= 1 \times 2 = 2\\ \text{factorial } 3 &= 2 \times 3 = 6\\ \text{factorial } 4 &= 6 \times 4 = 24\\ \text{factorial } 5 &= 24 \times 5 = 120\end{aligned}$$

So 5! = 120

Algorithm

```
1. Begin
2. Get the number of which the factorial is to be calculated i.e. $n$.
3. Assign the value of $i = 1$ and factorial = 1.
4. Calculate factorial $n$ = factorial (n-1) $\times i$
5. Increment the value of $i$ by 1, i.e. $i = i + 1$.
6. Repeat steps 4 and 5 till step 4 has been executed with the value of $i = n$.
7. Write the value of factorial.
8. Stop.
```

1.8.10 Fibonacci Sequence

In a Fibonacci sequence, the previous two numbers are added to generate the next Fibonacci number. For example:

Scanned with CamScanner

$f_1 = 1$ (1st number $\in$ *the Fibonacci series*)

$f_2 = 2$ (2nd in the Fibonacci series)

$f_3 = f_2 + f_1 = 2 + 1 = 3$

$f_4 = f_3 + f_2 = 3 + 2 = 5$

$f_5 = f_4 + f_3 = 5 + 3 = 8$, and so on.

f_1	f_2	f_3	f_4	f_5	f_6	f_7 ....
1	2	3	5	8	13	21 ....

To get next fibonacci number, we have to do sum of previous two numbers in the series.

Algorithm

1. Assign $sum = 0, A = 0, B = 1, i = 1$
2. Get the number of terms up to which you want to generate the Fibonacci number, i.e. n.
3. Add A and B to get next Fibonacci number.
4. Assign the value of B to A i.e $A = B$.
5. Assign the value of sum to B i.e. $sum = B$.
6. Write the value of sum to get next Fibonacci number in the series.
7. Increment i with 1 i.e. $i = i + 1$ and repeat step 3, 4, 5, 6 with the last value of $i = n$ (n is a number of terms up to which we want to generate Fibonacci number series.)
8. Stop

1.8.11 Evaluate 'sin (x)' as Sum of a Series

We know that

$$\text{Sin}\,x = \frac{x}{1!} - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \frac{x^9}{9!}$$

Where x is in radians.

Understanding the problem

1. Read X
2. Keep accumulating a product of integers (PRODUCT) in such a way that PRODUCT is first 1,
   then $1 \times 2 \times 3$,
   then $1 \times 2 \times 3 \times 4 \times 5$, and so on.
   Product serves as the denominator in terms of the formula.
3. Accumulate the terms:
   $$\frac{X}{\text{PRODUCT}}, \frac{X^3}{\text{PRODUCT}}, \frac{X^5}{\text{PRODUCT}}, \ldots$$
4. Compute $X^1, X^3, X^5 \ldots$ through the use of X^i where i is initially set to 1 and incremented by 2's to obtain X and odd power of X.
5. Generate the oscillating sequence of terms

$X, \ -X^3, \ +X^5, \ -X^7, + \ldots$

To do that, we note that $(X)^1, (-X)^3, (X)^5, (-X)^7$ gives rise to $X, -X^3, X^5, -X^7$. Hence we use –1 as successive multiplier to generate successive terms of the sequence from their magnitude.

The algorithm is given below :

Algorithm

```
1.  Read $x$ in radians.
2.  Read $n$ where '$n$' is the number of terms of series which we may like to add-up to get
    desired precision of sum.
3.  For Ist term,
        sum = $x$
        product = 1  (variable 'product' used for denominator)
        num = $x$ (variable 'num' used for numerator)
        power = 1
4.  For next term,
        num = num × $(-x^2)$
        power = power + 2
        product = product × (power – 1) × power
        next_term = num/product
5.  Then,
        sum = sum + next_term
6.  Repeat step 4 and 5, looping '$n$ – 1' times to get the sum of first '$n$' terms of the series.
7.  Print sum
8.  Stop
```

1.8.12 Reverse the Order of Elements of an Array

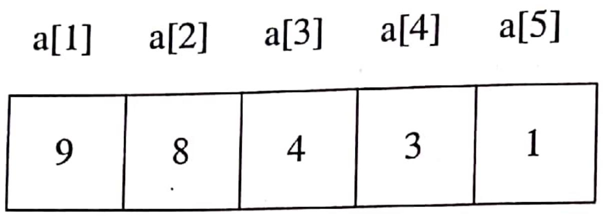
Reverse order of elements means value of 1st position of the array is exchanged with last element of the array, 2nd element of array is exchanged with '2nd last' and so on.

For example, see Figure 1.6(a).

a[1]	a[2]	a[3]	a[4]	a[5]
1	3	4	8	9

Before Reverse

Figure 1.6(a)



After Reverse

Figure 1.6(b)

Here,

$a[1] \Leftrightarrow a[5]$ (Swap element 5 value with element 1 value)
$a[2] \Leftrightarrow a[4]$
$a[3] \Leftrightarrow a[3]$

Here, we can see the suffixes/subscripts on left hand side are in increasing order and the suffixes/subscripts on right hand side are in decreasing order. (See Figure 1.6(b)).

For determining the reverse-array subscript values, we can use the formula $[n - i + 1]$ where 'n' is total number of elements an 'i' is the subscript of the element of original array.

$n - i + 1$
$5 - 1 + 1 = 5$
$5 - 2 + 1 = 4$
$5 - 3 + 1 = 3$

Swapping the values of that elements in the array would thus be reversed.

To exchange the values of the elements of the array, we need one temporary variable. Then we do the swapping as given below:

$\therefore$ $\text{temp} = a[i]$

$a[i] = a[n - i + 1]$

$a[n - i + 1] = \text{temp}$

Algorithm

1. Begin
2. Get the values and store them in an array with elements $a[1], a[2], a[3], a[4], \ldots\ldots, a[n]$ where n is the number of elements.
3. Compute r, the number of exchange needed to reverse the array. r = integer value of 1/2.
4. Exchange the i^{th} element with $[n - i + 1]$ using
   $\text{temp} = a[i]$
   $a[i] = a[n - i + 1]$
   $a[n - i + 1] = \text{temp}$
5. Carry step 4, r times increasing i by 1.

Scanned with CamScanner

6. Print the reversed array elements *a*[1], *a*[2], *a*[3], *a*[4], ..., *a*[*n*].
7. Stop

1.8.13 Find Largest Number in an Array

Largest number means the number which is greater than all other numbers in the array elements. Another number(s) in the set may be equal to this number, but can not be greater than this number. For example, in the following array:

Array	**10**	**11**	**15**	**19**	**2**
	a[1]	**a[2]**	**a[3]**	**a[4]**	**a[5]**

we find that the number 19 is the largest number in the array.

For small set of numbers stored in the array, it is easy to find largest number, but finding the largest value from thousands of numbers stored in the array may take long time. But using following logic, we can get the answer in a short time.

Logic

Assign the value of the first element of the array to variable *temp*, then compare the value of *temp* to those of other elements in the array in order, one by one. If any element value is greater than *temp* then put that value in *temp* and continue comparison.

For the array shown above value of *temp* = 10 taken from first element of the array.

Compare value in *temp* to 2nd element value of the array. Here, 11 is greater than 10, so delete the original value of *temp* and write down the 2nd number i.e. 11 in *temp*. We keep comparing the 3rd element value with the new *temp* value. The procedure will be continued till all the elements in the array have been examined. In the end, we will get the largest value in the *temp* variable.

Algorithm

1. Begin
2. Get the value of element in an array and store them in the array elements a[1], a[2]..... a[n] where *n* is the number of elements.
3. Set temporary variable and assign the value of 1st array element a[1] to *temp*.
4. Compare *temp* with the next element a[2] of the array.
5. If the element is having greater value than *temp*, assign that value to *temp*.
6. Repeat steps 4 and 5 till last element a[n] of the array is encountered.
7. Final value of *temp* is the largest value stored in the array.
8. Stop

Scanned with CamScanner