Chapter 1

Introduction to Programming & Problem Solving

Learning Outcomes

After reading this chapter, students will be able to

- explain the concept of programming
- describe the concept of problem-solving and its need
- explain the concept of process and its types – *adhoc* and *defined*
- describe different approaches to problem-solving – *top-down* and *bottom-up*
- explain the concept of structured programming
- explain various control structures adhering to the requirement of structured programming
- explain the notion of an algorithm
- explain the desirable characteristics of a good algorithms
- represent an algorithm using a flowchart or a pseudocode
- solve variety of numerical and logical problems

1.1 INTRODUCTION TO PROGRAMMING

Programming is a way to *instruct the computer to perform various tasks.*

Instruct the computer, basically means that you need to provide the computer with a set of instructions that are written in a language that the computer can understand.

Perform various tasks, basically means finding the solution to a problem (task). The tasks could be small & simple that require few instructions to obtain their solution or large & complex one that may involve a large number of instructions

Hence, in brief, *Programming* is a way to tell computers to do a specific task.

The real problem with the current teaching & learning of programming (using any of C/C++/Java/Python/Kotlin) is that ultimately it reduces to teaching & learning of programming language only.

Programming is not just about learning the syntax of the language, it is more about accomplishing the task – *Problem Solving*.

1.1.1 Problem Solving

Problem-solving is a skill that anybody can learn with practice.

Let us pause for while, and ask ourselves a simple question - *what is a problem?*

The obvious answer is the absence of a solution. The moment you get a solution, the problem is no more a problem for you.

Therefore, I wish that our education system must produce individuals, who can seek solutions to the problems they are facing or may face in future, *the solution seekers*.

We will learn about problem-solving, in more detail, in the next section.

1.1.2 Process

A process is a series of steps or activities that interact to produce a solution.

Whatever we do in our day-to-day life, we are always following certain series to steps to accomplish a task.

I am leaving this as a task for you – *enumerate most of the daily activities you do, and what are the steps you follow to accomplish them*.

1.1.2.1 Adhoc Process

A process where activities are not *well-defined*, and the solution is sought in a *hit-and-trial* manner.

Characteristics of an adhoc process:

- may or may not produce solution
- the outcome may not be traceable
- the solution may not be repeatable

1.1.2.2 Defined Process

A process that is well-defined and documented, i.e., every step is clear and had been written in a form that one can understand easily.

Characteristics of a defined process:

- solution is sought by following the steps prescribed in the document
- will always produce the solution
- the outcome is always traceable
- the solution is always repeatable

1.1.2.3 Tools to Document a Process

Any one of the following tools can be used to document a defined process:

- Algorithm
- Flowchart
- Pseudocode
- Decision table
- Decision tree

In this textbook, we will discuss about *algorithm*, *flowchart*, and *pseudocode*. The remaining tools – *decision table* and *decision tree* you will learn in the subjects where they possible used.

1.2 INTRODUCTION TO PROBLEM-SOLVING

The ability to solve problems is a most basic life skill and is essential to our day-to-day lives, at home, at school, and at the workplace.

We solve problems every day without really thinking about how we solve them.

For example, it is raining and you need to go to the market.

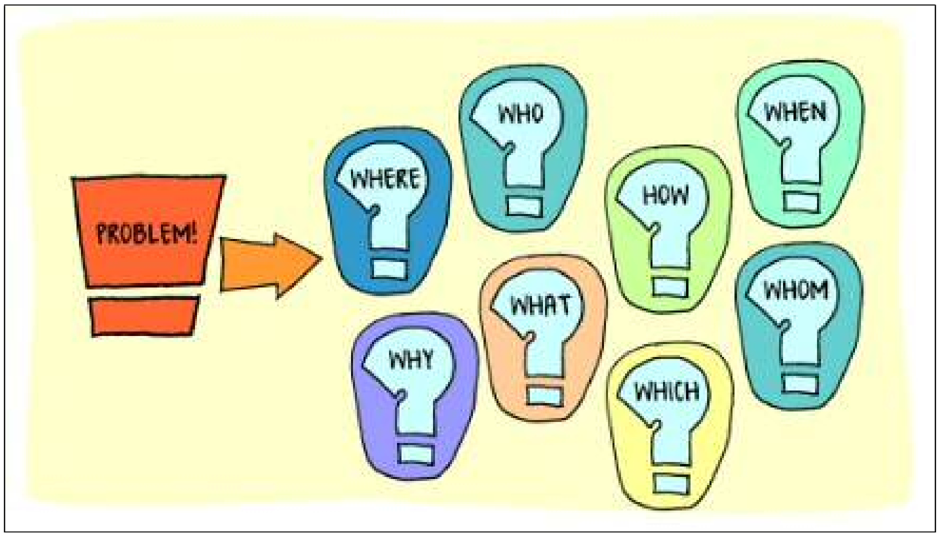
What do you do?

There are a variety of possible solutions:

- You can take your umbrella and walk down to the market.
- If you don't want to get wet, you can drive, or take the bus.
- You might decide to call a friend for a ride, or you might decide to go to the market another day.

The important point to note down here is that there is no right way to solve this problem and different people may solve it differently.

Problem-solving is the process of identifying a problem, developing possible solution alternatives, and taking the appropriate course of action.



Why is problem-solving important?

Good problem-solving skills empower you not only in your personal life, but are very critical in your professional life.

In the currently fast-changing global economy, employers often identify everyday problem solving as crucial to the success of their organizations.

For employees, problem solving can be used to develop practical and creative solutions and to show independence and initiative to employers.

1.3 APPROACHES TO PROBLEM SOLVING

There are two approaches to problem solving:

- Top-down approach
- Bottom-up approach

1.3.1 Top-Down Approach

The basic idea of the top-down approach is to divide a complex problem into smaller sub-problems, this process is also called *decomposition*. The sub-problems are further divided into sub-problems and this process is continued until each sub-problem is atomic (can't be divided further) and can be solved independently of other sub-problems.

The top-down way of solving a program is the step-by-step process of breaking down the problem into chunks for organizing and solving the sole problem.

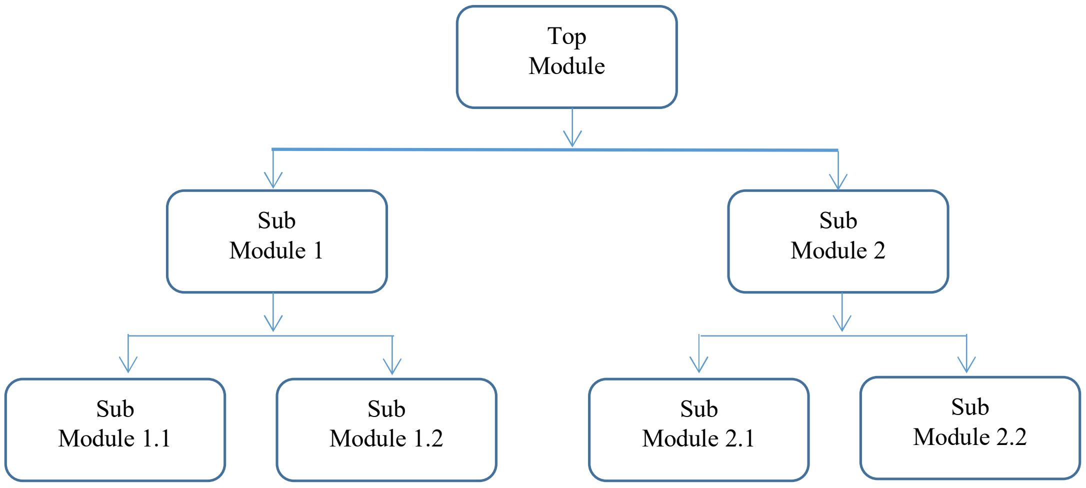


Figure 1.1: Top-down process

Structured programming languages, like the C programming language, use the top-down approach to solving a problem in which the flow of control is in the downward direction.

1.3.2 Bottom-Up Approach

As the name suggests, this method of solving a problem works exactly opposite to the top-down approach.

In this approach, we start working from the most basic level of problem solving and moving up in conjugation of several parts of the solution to achieve the required results. The most fundamental units, modules, and sub-modules are designed and solved individually, and these units are then integrated together to get a more concrete base for problem-solving.

This bottom-up approach works in different phases or layers. Each module designed is tested at a fundamental level which means unit testing is done before the integration of the individual modules to get the solution.

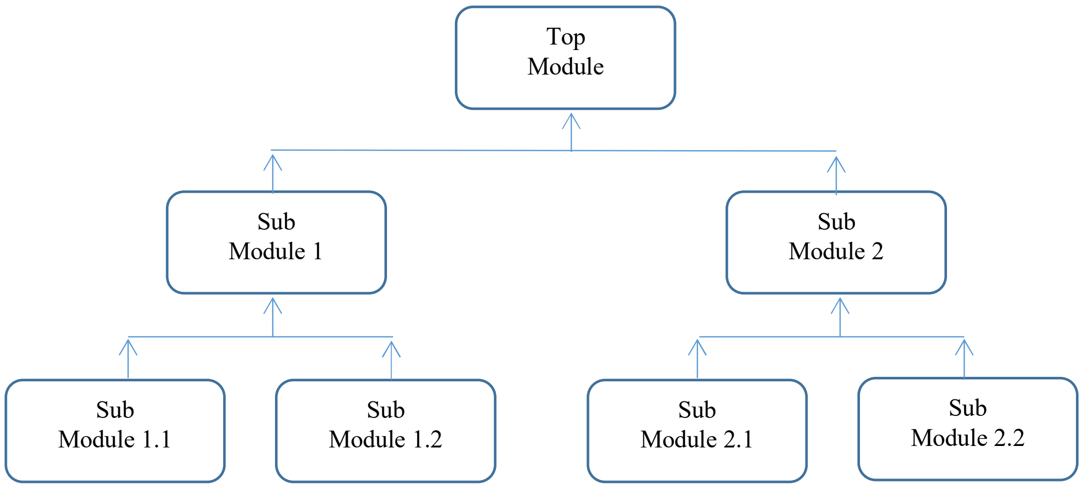


Figure 1.2: Bottom-up process

The object oriented programming languages, like the C++ or Java programming language, uses the bottom-up approach to solving a problem in which the flow of control is in the upward direction.

1.3.3 Top-down v/s Bottom-up Approach

Table 1.1 summarizes the key differences between top-down approach and bottom-up approach.

Table 1.1: Top-down V/S Bottom-up Approach

Top-down Approach	**Bottom-up Approach**
Divides a problem into smaller units and then solves it.	Starts by solving small modules and adding them up together.
This approach may contain redundant information.	Redundancy can easily be eliminated.
A well-established communication is not required.	Communication among steps is mandatory.
The individual modules are thoroughly analyzed.	Works on the concept of data-hiding and encapsulation.
Structured programming languages such as C uses a top-down approach.	OOP languages like C++ and Java etc. uses a bottom-up mechanism.
Relation among modules is not always required.	The modules must be related for better communication and workflow.
Primarily used in code implementation, test case generation, debugging, and module documentation.	Finds use primarily in testing.

The top-down approach is the conventional approach in which the decomposition of the higher-level system into a lower-level system takes place respectively while the bottom-up approach starts by designing lower abstraction modules and then integrating them into a higher-level system.

1.4 STRUCTURED PROGRAMMING

Structured programming is a technique devised to improve the reliability and clarity of programs.

In structured programming, control of program flow is restricted to the following three structures:

- sequence
- selection
- iteration

or to a structure derivable from a combination of these basic three structures.

Each of these structures is described overleaf.

1.4.1 Sequence Structure

In *sequence structure*, instructions are followed or executed one after another in sequence in which they appear. The flow of logic is from top to bottom.

```
     :
     :
instruction-1

instruction-2

instruction-3
     :
     :
```

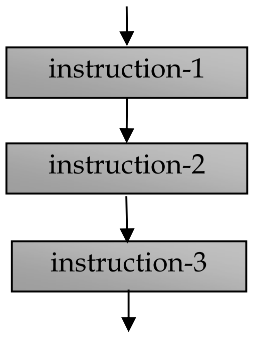


Figure 1.3: Pseudocode and flowchart for sequence structure

1.4.2 Selection Structure

Selection structure is used for making a decision. It is used for selecting a proper path out of the alternative paths in the program logic.

Selection structure may take the form as either *If . . . Endif* or *If . . . Else . . . Endif* or *If . . . Else If . . . Else . . . Endif* structure.

The *If . . . Endif* structure says that if the expression is true, then execute statement else (if the expression is false) skip over the statement.

```
     :
     :
If ( expression ) then
     statement
Endif
     :
     :
```

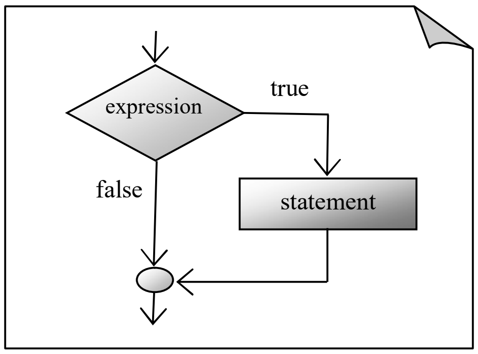


Figure 1.4: Pseudocode and flowchart for *If . . . Endif* selection structure

The *If . . . Else. . . Endif* structure says that if the expression is true then execute statement-1, else (if the expression is false) execute statement-2.

Depending on the outcome of the expression being tested, if there are multiple alternatives (execution paths), then *If . . . Else If . . . Else . . . Endif* structure is a very handy structure.

```
    :
    :
If ( expression ) then
   statement-1
Else
   statement-2
Endif
    :
    :
```

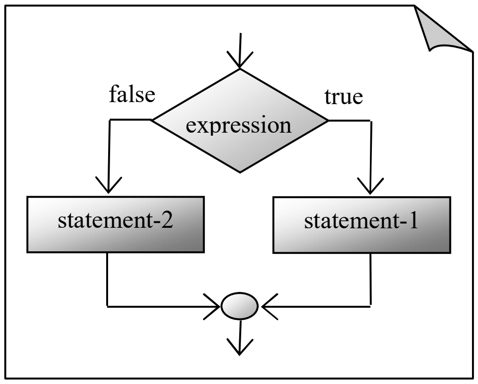


Figure 1.5: Pseudocode and flowchart for *If . . . Else . . . Endif* selection structure

```
If ( expression-1 ) then
   statement-1
Else If ( expression-2 ) then
   statement-2
Else If ( expression-3 ) then
   statement-3
       ⋮
Else If ( expression-n ) then
   statement-n
Else
   statement-s
Endif
```

Figure 1.6: Syntax of *If . . . Else If . . . Else . . . Endif* structure

The expressions are evaluated in order, and if any expression is true then the statement block associated with it is executed, and this terminates the whole chain.

The last *else* part handles *none of the above* where none of the specified expressions are satisfied.

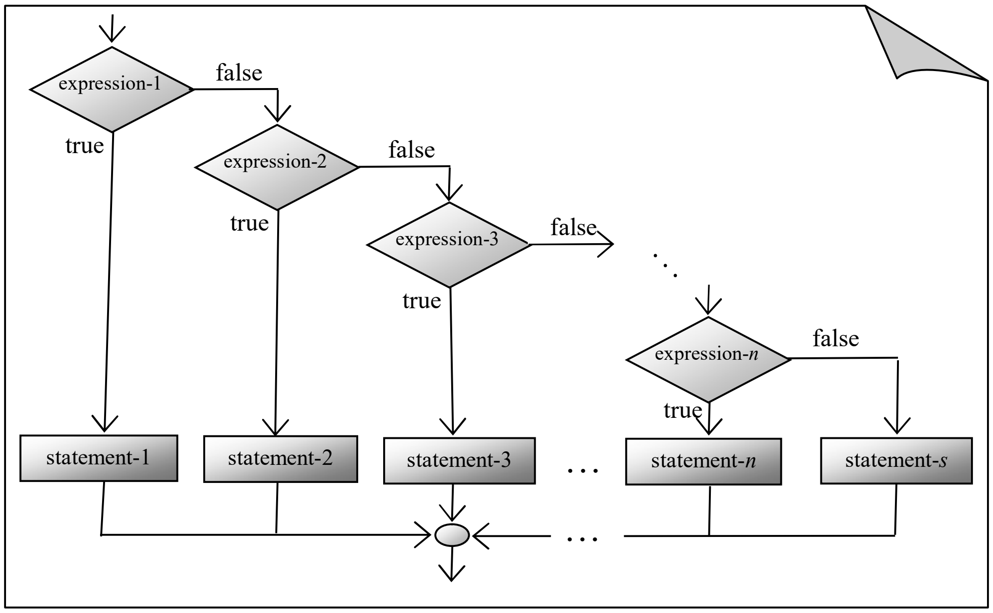


Figure 1.7: Logic flow of *If . . . Else If . . . Else . . . Endif* structure

1.4.3 Iterative Structure

The *iterative structure* is used to produce loops when one or more instructions are to be executed either a given number of times or till a certain condition is met.

The following two iterative structures are used:

- *While . . . Endwhile*
- *For . . . Endfor*

```
    :
    :
While ( expression )
   statement
Endwhile
    :
    :
```

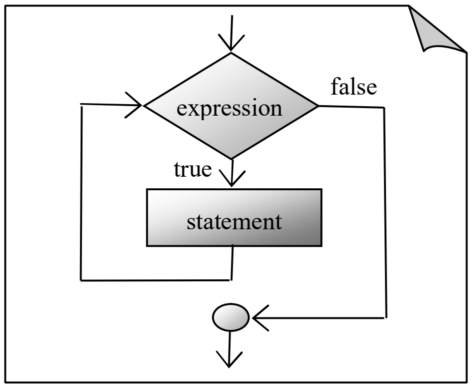


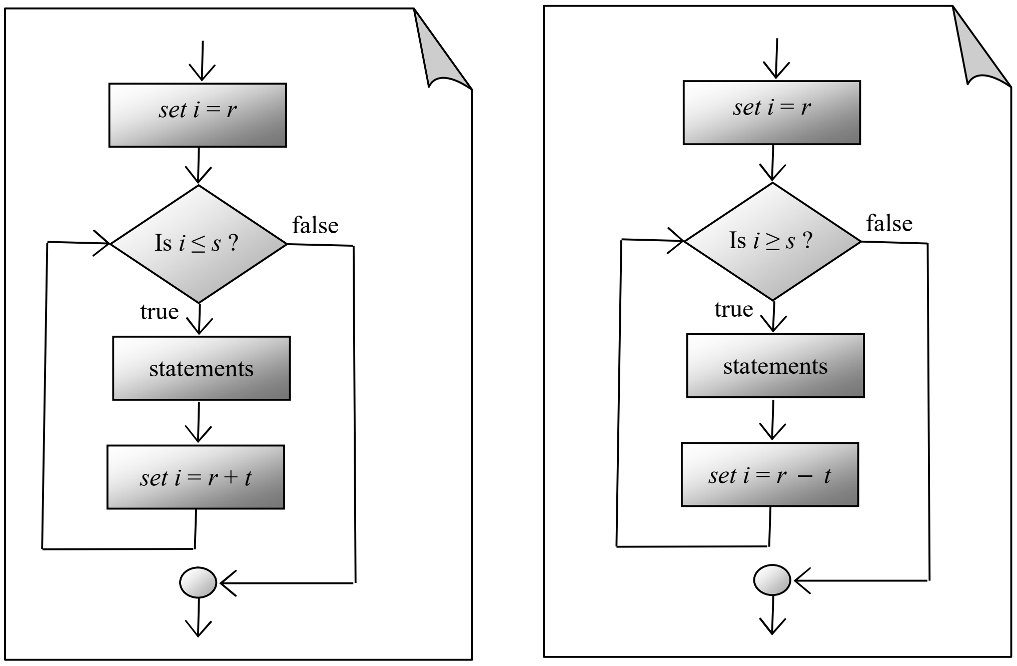
Figure 1.8: Pseudocode and flowchart for *While . . . Endwhile* iterative structure

The *While . . . Endwhile* iterative structure will continue executing until the expression is true. However, if statement or a certain group of statements are to be executed for a known number of times, the *For . . . Endfor* iterative structure is a better choice.

```
      :
      :
For i = r to s in steps of t
   statement
Endfor
      :
      :
```

Figure 1.9: Syntax for *For . . . Endfor* iterative structure

It uses an index variable *i* to control the loop. Here *r* is called the initial value, *s* is called the final value, and *t* is called the step size, which may be positive (increment) or negative (decrement).



(*a*) When step size *t* is positive (*b*) When step size *t* is negative

Figure 1.10: Working of *for* statement for positive and negative step size

1.5 ALGORITHMS

An *algorithm* is a finite sequence of steps defining the solution of a particular problem.

Characteristics of a good algorithm:

There are five important characteristics of an algorithm that should be considered while designing an algorithm for a problem.

- ***Input*:** An algorithm must have zero or more but a finite number of inputs, which are externally supplied. An example of zero input algorithms can be to find the sum of the first 100 natural numbers. Here, the user doesn't need to supply any external input since it is already specified to find the sum of the first 100 natural numbers. However, if the above problem is re-stated as finding the sum of first *n* natural numbers, the user is required to provide single input denoting the value for *n*.
- ***Output*:** An algorithm must have at least one desirable outcome, *i.e.*, output.
- ***Definiteness (No ambiguity)*:** Each step must be clear and unambiguous, *i.e.*, having one and only one meaning.
- ***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 it can in principle be carried out by a person using only paper and pencil. In addition, not only should each step be definite, but it must also be feasible.

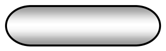
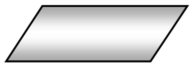
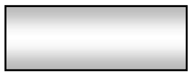
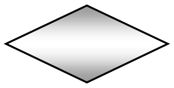
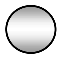
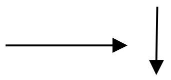
An algorithm can be represented using a flowchart or pseudocode.

1.5.1 Flowchart

A flowchart is a pictorial representation of an algorithm. It uses different shapes to denote different types of instructions. The actual instructions are written within the shapes using clear and concise statements. These shapes are connected by directed lines to indicate the sequence in which instructions are to be executed.

Table 1.2 shows various symbols used in flowcharts along with their name and brief description.

Table 1.2: Various flowchart symbols and their brief description

Symbol	Name	Purpose
	Oval	*Terminal* - to mark the beginning and end of the program logic flow.
	Parallelogram	*Input/Output* - to denote input to the program or output from the program.
	Rectangle	*Processing* - to denote arithmetic operations and movement of data.
	Diamond	*Decision* - to denote a point where decision has to be made to branch to one of the alternatives.
	Small circle	*Connector* - To provide a logical link between segments of a flowchart.
	Directed lines	*Flow Lines* - To indicate the sequence in which instructions are to be executed.

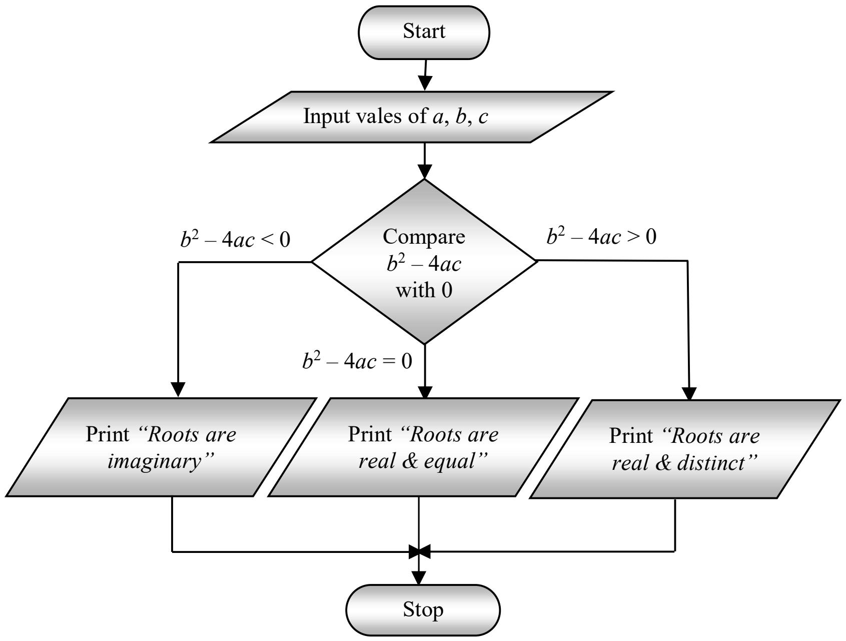

Start
Input vales of a, b, c
Compare $b^2 - 4ac$ with 0
$b^2 - 4ac < 0$
$b^2 - 4ac > 0$
$b^2 - 4ac = 0$
Print *"Roots are imaginary"*
Print *"Roots are real & equal"*
Print *"Roots are real & distinct"*
Stop

Figure 1.11: Flowchart to find the nature of roots of a quadratic equation

1.5.2 Pseudocode

The word "*pseudo*" means imitation or false and the word "*code*" refers to the instruction written in a programming language. Pseudocode, therefore, is an imitation of actual computer instruction. Pseudo instructions are phrases written in English like statements. Instead of using symbols to describe the logic of the program, as in flowcharts, pseudocode uses a structure that resembles computer instructions. Because, it emphasizes the design of the program, pseudocode is also called *Program Design Language* (PDL).

Pseudocode is made up of the following basic logic structures that have proved to be sufficient for writing any computer program:

1. Sequence
2. Selection (*If . . . Endif, If . . . Else . . . Endif, If . . . Else If . . . Endif*)
3. Iteration (*While . . . Endwhile, Do . . . While*)

We have already discussed about these logic structures under the heading of structured programming.

Pseudocode Description

Comments

Each instruction may be followed by a comment. The comments begin with a double slash, and the explain the purpose of the instruction, such as

```
Read: n        // Enter the value of variable n
```

Appropriate use of comments enhances the readability of the pseudocode, which in turn helps in maintaining the pseudocode.

Variable Names

For variable names, we will use italicized lowercase letters such as *max*, *loc*, etc., whereas for defined constants, if any, we will use uppercase letters.

Assignment Statement

The assignment statement will use the notation as

```
Set max = a     OR          max = a
```

to assign the value of *a* to *max*. The right hand side of the assignment statement can have a *value*, *a variable* or *an expression*.

However, if several assignment statements appear in the same line, such as

Set $k = 1$, $loc = 1$, $max = a_i$ OR $k = 1$, $loc = 1$, $max = a_i$

then they are executed from left to right.

Input/Output

Data may be input and assigned to variables by means of a *read* statement with the following format

Read: *Variable list*

where the *Variable list* consists of one or more variables separated by comma.

Similarly, the data held by the variables and the messages, if any, enclosed in double quotes can be output by means of a *print* statement with the following format

Print: *message* and/or *Variable list*

where the *message* and the variables in the *Variable list* are separated by comma.

Execution of Instructions

The instructions are usually executed one after the other as they appear in the pseudocode. However, there may be instances when some instructions are skipped or some instructions may be repeated as a result of certain expressions.

Completion of the Algorithm

A pseudocode is completed with execution of the last instruction. However, it can be terminated at any intermediate state using the *exit* instruction.

Pseudocode to display the nature of roots of a quadratic equation of the type

$ax^2 + bx + c = 0$ provided $a \neq 0$

Pseudocode 1.1

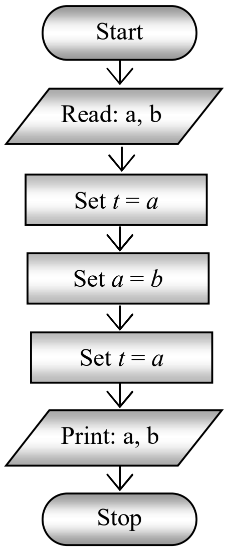
```
Begin
    Read: $a, b, c$
    Set $disc = b^2 - 4ac$
    If ( $disc = 0$ ) then
        Print: “Roots are real and equal”
    Else If ( $disc > 0$ ) then
        Print: “Roots are real and distinct”
    Else
        Print: “Roots are imaginary”
    Endif
End.
```

ILLUSTRATIVE EXAMPLES

Example 1.1: Draw a flowchart and write a pseudocode to swap (interchange) two variables say *a* and *b*.

Solution: *Think of the scenario* – we have water in one glass and juice in another glass. We want to have water in a glass in which we have juice, and juice in a glass in which we have water. *How can this task be accomplished*?

In a similar way, we have to use a third variable say *t*, to facilitate the swapping of values of two variables.



Pseudocode 1.2

```
Begin
    Read: a, b
    Set t = a
    Set a = b
    Set b = t
    Print: a, b
End.
```

Figure 1.12: Flowchart and pseudocode to swap two variables

Example 1.2: Draw a flowchart and write a pseudocode to test whether a given natural number ‘n’ is even or odd.

Solution: You all may know that any natural number is even if it is exactly divisible by 2, *i.e.*, division by 2 gives 0 as remainder. The operation of obtaining remainder is called modulo (mod in short) operation.

Pseudocode 1.3

```
Begin
    Read: n
    If ( n mod 2 = 0 ) then
        Print: n, " is Even"
    Else
        Print: n, " is Odd"
    Endif
End.
```

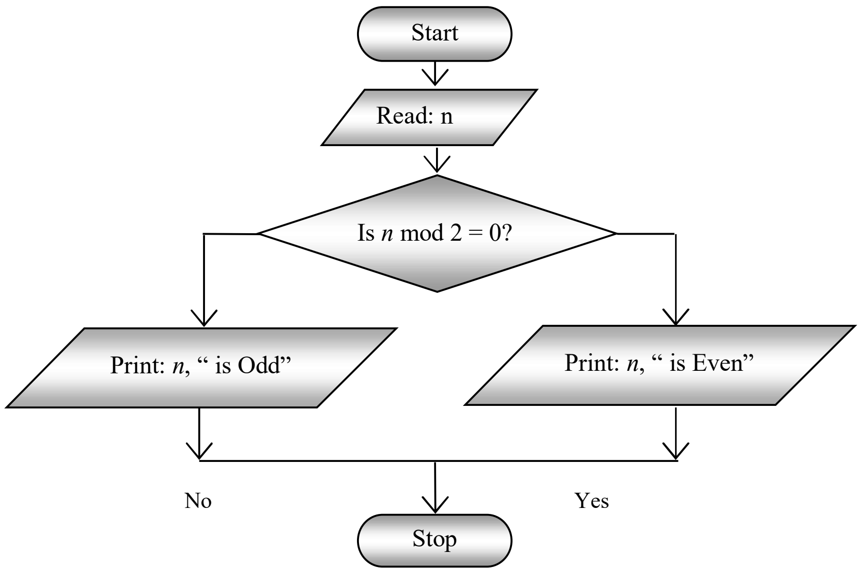


Figure 1.13: Flowchart to test whether a given number is Even or Odd

Example 1.3: Draw a flowchart and write pseudocode to find the largest of three numbers; say *a*, *b*, *c*.

Solution: We first compare *a* with *b*. If *a* is greater than *b* then we compare *a* with *c*. If *a* is greater than *c*, then *a* is taken as the largest number otherwise we take c as the largest number.

However, if *a* is not greater than *b*, we compare *b* with *c*. If *b* is greater than *c* then *b* is taken as the largest number otherwise we take *c* as the largest number.

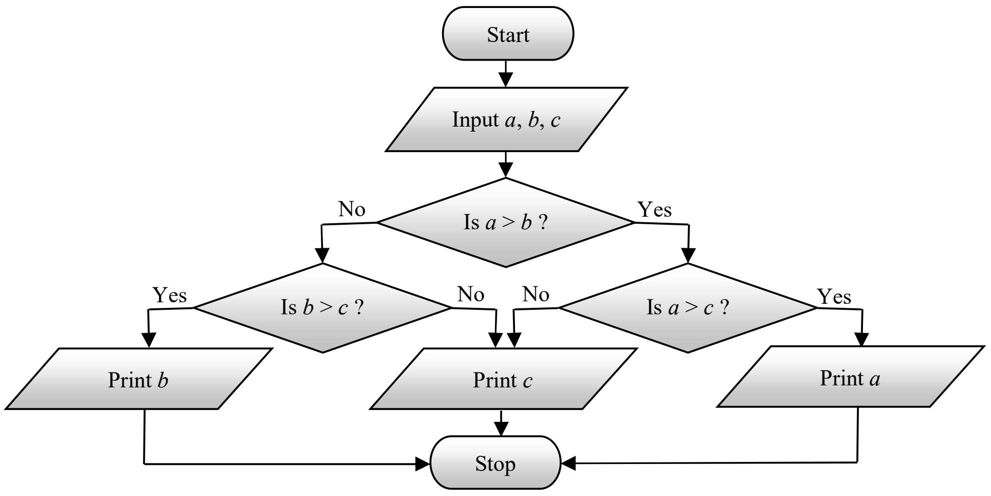


Figure 1.14: Flowchart and pseudocode to find largest of three numbers

Pseudocode 1.4

```
Begin
    Read: a, b, c
    If ( a > b )
        If ( a > c ) then
            Print: a
        Else
            Print: c
        Endif
    Else
        If ( b > c ) then
            Print: b
        Else
            Print: c
        Endif
    Endif
End.
```

Example 1.4: Based on the percentage of marks in a subject, letter grade is assigned to a student as per the following examination policy:

Percentage of Marks	Grade
percentage ≥ 90	A+
90 > percentage ≥ 80	A
80 > percentage ≥ 70	B
70 > percentage ≥ 60	C
60 > percentage ≥ 50	D
percentage < 50	F

Write pseudocode to assign a letter grade to a student whose percentage of marks in a subject is given.

Pseudocode 1.5

```
Begin
    Read: percentage
    If ( percentage >= 90 )
        Print: "Grade = A+"
    Else If ( percentage >= 80 )
        Print: "Grade = A"
    Else If ( percentage >= 70 )
        Print: "Grade = B"
    Else If ( percentage >= 60 )
        Print: "Grade = C"
    Else If ( percentage >= 50 )
        Print: "Grade = D"
    Else
        Print: "Grade = F"
    Endif
End.
```

Example 1.5: Commission on sales by a salesman is calculated as per following policy:

Amount of Sale (in Rs.)	Commission Rate
0 – 5000	Nil
5001 – 10000	5 % excess of 5000
10001 – 15000	7.5 % excess of 10000
> 15000	10 % excess of 15000

Draw a flowchart and write a pseudocode that accepts sales made by a salesman and displays the commission due.

Solution:

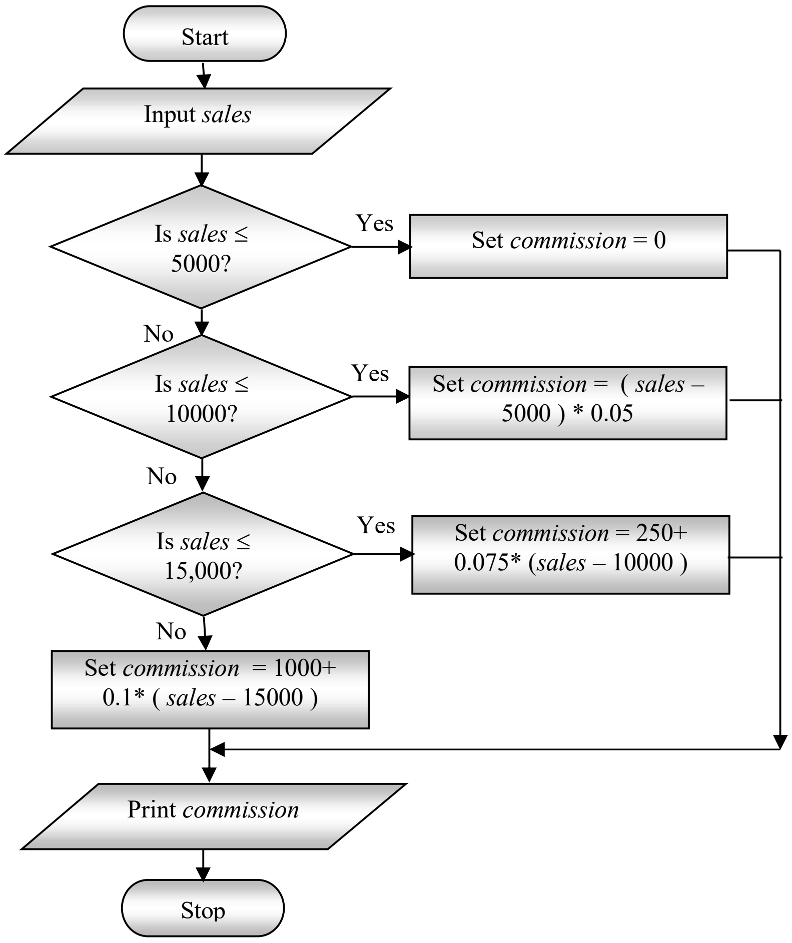

Start
Input *sales*
Is *sales* ≤ 5000?
Yes
Set *commission* = 0
No
Is *sales* ≤ 10000?
Yes
Set *commission* = (*sales* – 5000) * 0.05
No
Is *sales* ≤ 15,000?
Yes
Set *commission* = 250+ 0.075* (*sales* – 10000)
No
Set *commission* = 1000+ 0.1* (*sales* – 15000)
Print *commission*
Stop

Figure 1.15: Flowchart to compute the commission

Pseudocode 1.6

```
Begin
    Read: sales
    If ( sales <= 5000 )
        Set commission = 0
    Else If (sales <= 10000  )
        Set commission = ( sales − 5000 ) * 0.05;
    Else If ( sales <= 15000 )
        Set commission = 250 + ( sales − 10000 ) * 0.075;
    Else
        Set commission = 1000 + ( sales − 15000 ) * 0.1;
    Endif
    Print: "Computed commission = ", commission
End.
```

Example 1.6: Draw a flowchart and write pseudocode to find the sum of digits of a number *n*.

Solution:

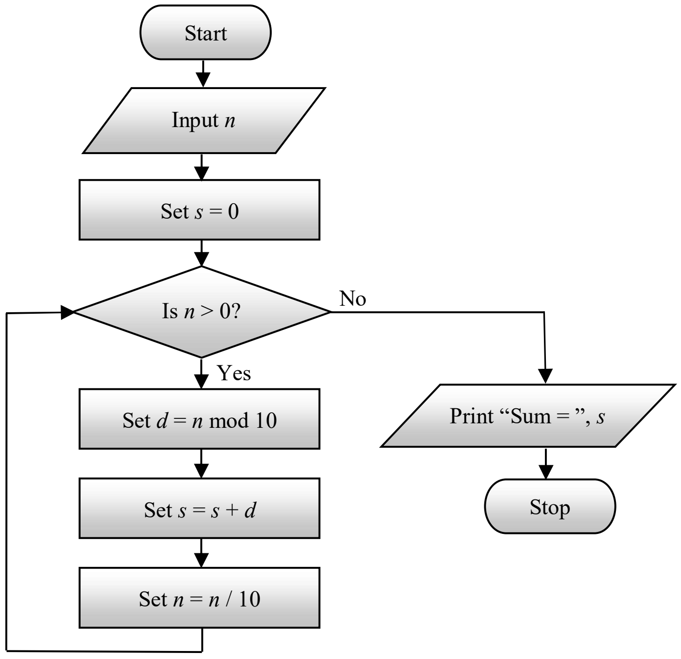


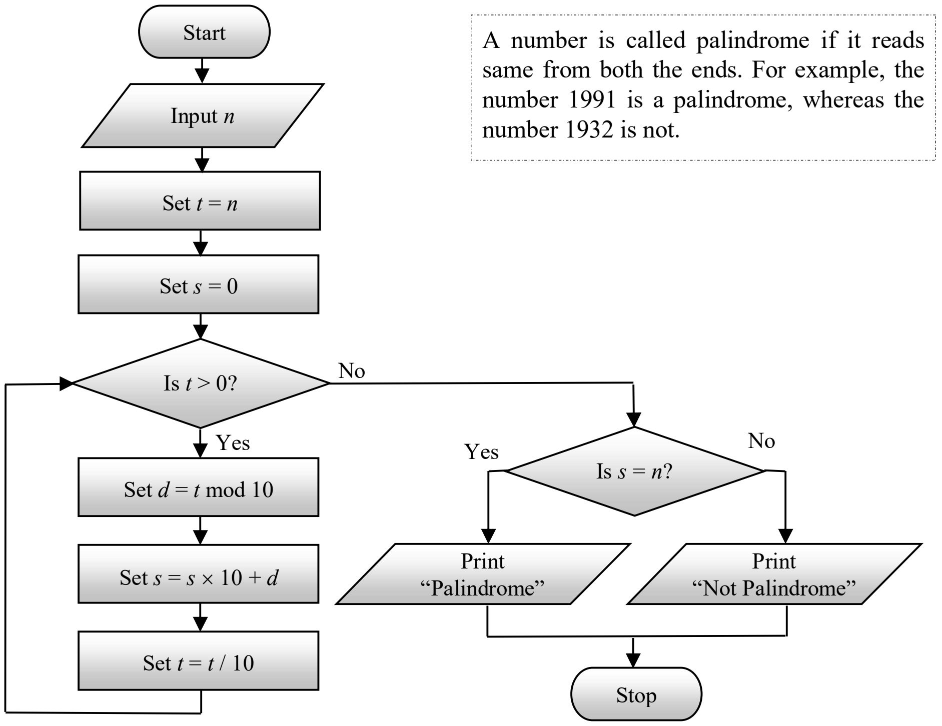
Figure 1.16: Flowchart to find the sum of digits of a number

Pseudocode 1.7

```
Begin
    Read: n
    Set s = 0
    While  ( n > 0 ) do
        Set d = n mod 10
        Set $s = s + d$
        Set $n = n / 10$
    Endwhile
    Print: “Sum = ”, $s$
End.
```

Example 1.7: Draw a flowchart and write pseudocode to check whether the given number n is palindrome or not.

Solution:



A number is called palindrome if it reads same from both the ends. For example, the number 1991 is a palindrome, whereas the number 1932 is not.

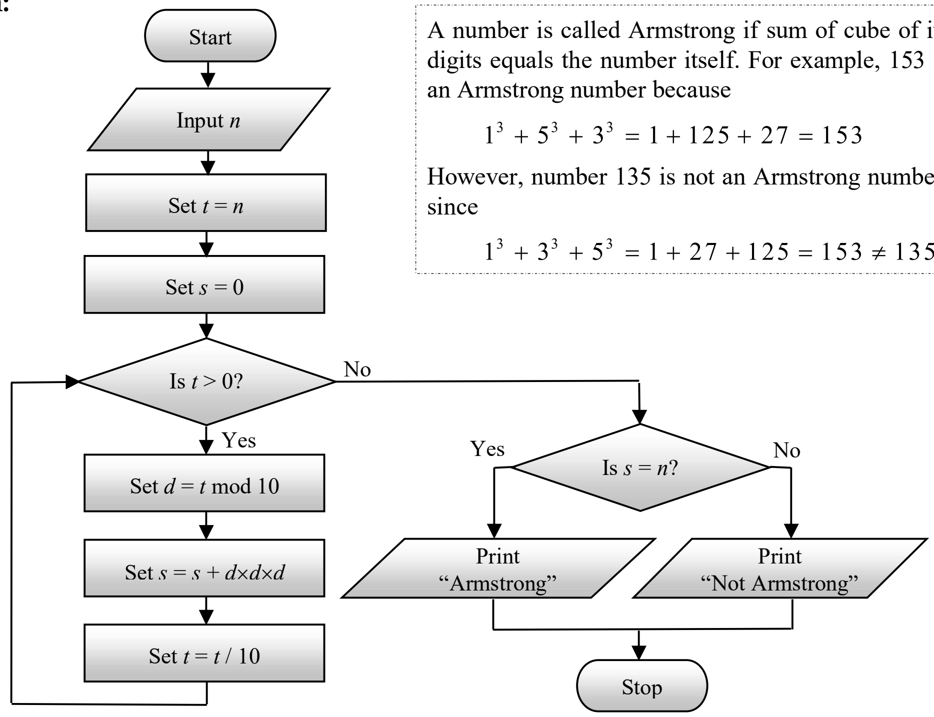
Figure 1.17: Flowchart to check whether the given number n is palindrome or not

Pseudocode 1.8

```
Begin
    Read: $n$
    Set $t = n,\ s = 0$
    While ( $t > 0$ ) do
        Set $d = t$ mod 10
        Set $s = s \times 10 + d$
        Set $t = t / 10$
    Endwhile
    If ( $s = n$ ) then
        Print: “Palindrome”
    Else
        Print: “Not a Palindrome”
    Endif
End.
```

Example 1.8: Draw a flowchart and write a pseudocode to check whether the given number n is an Armstrong number or not.

Solution:



A number is called Armstrong if sum of cube of its digits equals the number itself. For example, 153 is an Armstrong number because

$$1^3 + 5^3 + 3^3 = 1 + 125 + 27 = 153$$

However, number 135 is not an Armstrong number, since

$$1^3 + 3^3 + 5^3 = 1 + 27 + 125 = 153 \neq 135$$

Figure 1.18: Flowchart to check whether the given number n is an Armstrong number or not

Pseudocode 1.9

```
Begin
    Read: $n$
    Set $t = n,\ s = 0$
    While ( $t > 0$ ) do
        Set $d = t$ mod 10
        Set $s = s + d \times d \times d$
        Set $t = t / 10$
    Endwhile
    If ( $s = n$ ) then
        Print: “Armstrong”
    Else
        Print: "Not Armstrong"
    Endif
End.
```

Example 1.9: Draw a flowchart to find whether the given natural number *n* is a prime number or not.

Solution: A natural number is said to be prime if it is divisible by 1 and itself only, *i.e.*, it cannot be factorized. In addition, to this definition, an even number except 2 is not a prime number. Therefore, our test criteria becomes

1. If *n* is greater than 2 and is even then *n* is not a prime number.
2. If test at step 1 fails, then we try to divide number *n* by factors $k = 3, 5, 7, \ldots \sqrt{n}$. Therefore, if *n* is divisible by any value of *k*, number *n* is not a prime number.
3. If test at step 2 also fails, then *n* is a prime number.

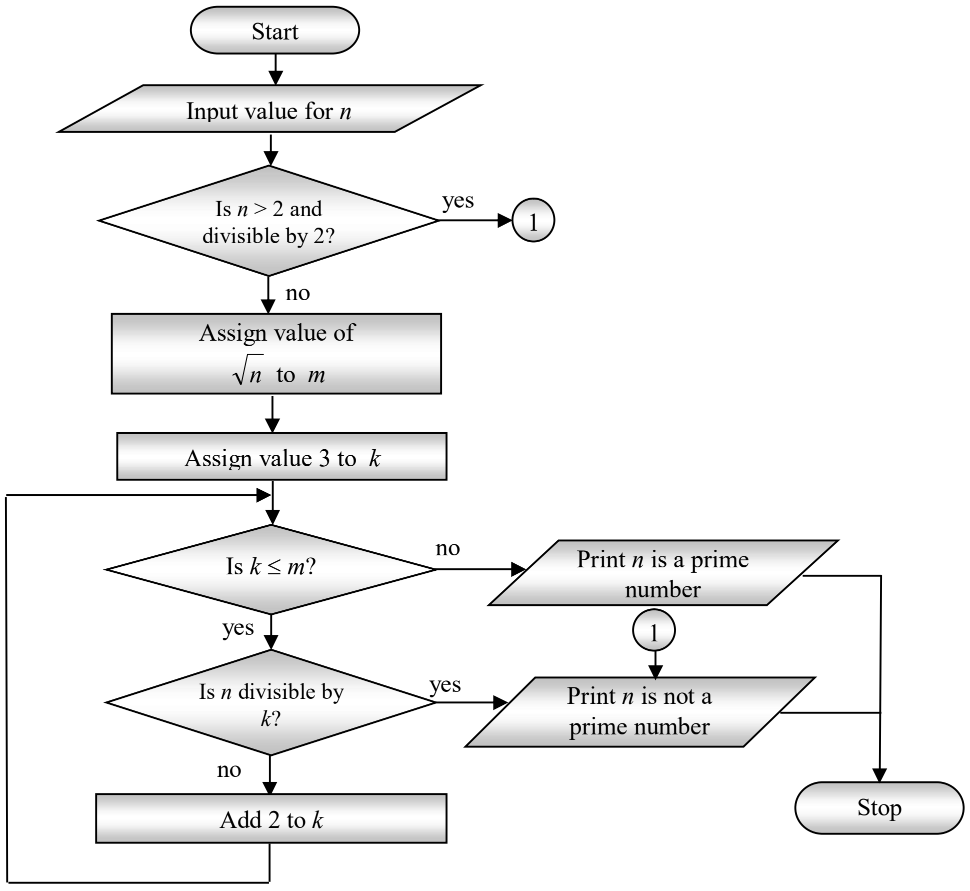


Figure 1.19: Flowchart to check whether number *n* is prime or not

The following is the pseudocode to find whether the given positive number *n* is a prime number or not.

Pseudocode 1.10

```
Begin
  Read: $n$
  If ( $n > 2$ and $n$ mod 2 == 0 ) then
     Print: $n$, " is not a prime number"
     Exit
  Else
     Set $m = \sqrt{n}$
     For $k = 3$ to $m$ by 2 do
        If ( $n$ mod $k$ == 0 ) then
           Print: $n$, " is not a prime number"
           Exit
        Endif
     Endfor
     Print: $n$, " is a prime number"
  Endif
End.
```

Example 1.10: To find the highest common factor (HCF), also known as the greatest common divisor (GCD), of two natural numbers *m* and *n*.

Solution:

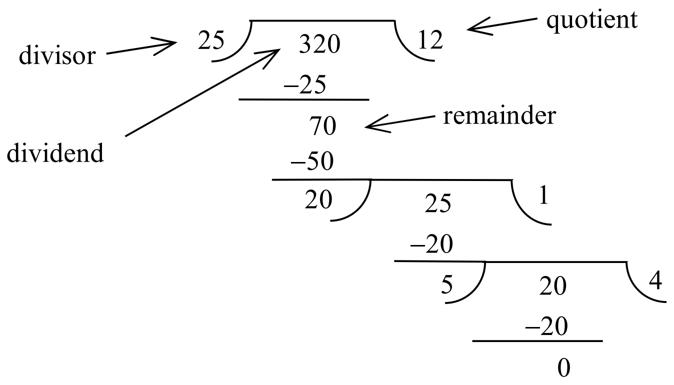


Figure 1.20: Illustration of computational procedure for HCF/GCD

Figure 1.20 demonstrates the long/continued division method to find the HCF/GCD of two natural numbers. You must have observed that in successive divisions, the divisor of the previous division becomes dividend, remainder of becomes divisor, and division is again carried out. This process is continued till the reminder becomes zero, and the current divisor is taken as HCF/GCD of the given natural numbers.

This process can be implemented by using the following steps

1. Perform division.
2. If remainder is zero, then stop and take the divisor as HCF/GCD.
3. Replace dividend by divisor.
4. Replace divisor by remainder.
5. Repeat from step 1.

Pseudocode 1.11

```
Begin
    Read: m, n
    Set r = m mod n
    While ( r ≠ 0 ) do
        Set m = n
        Set n = r
        Set r = m mod n
    Endwhile
    Print: "HCF/GCD = ", n
End.
```

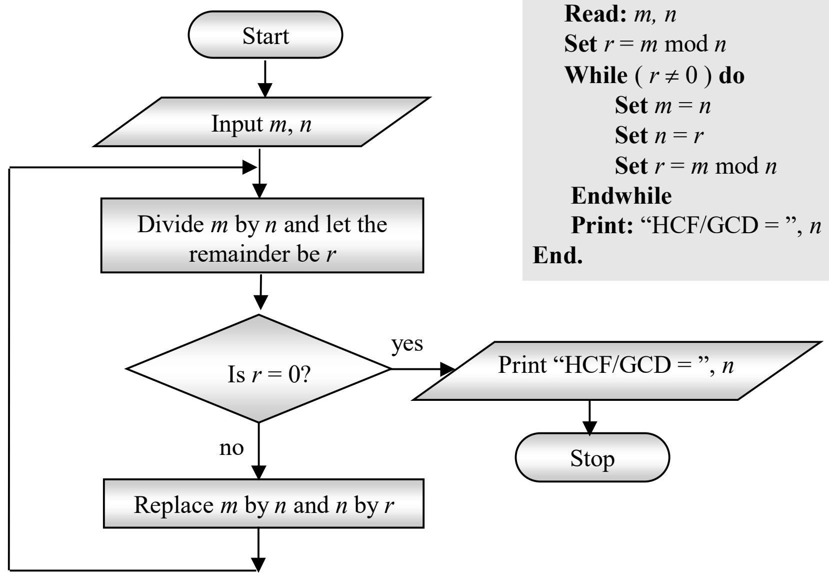

Start
Input *m*, *n*
Divide *m* by *n* and let the remainder be *r*
Is *r* = 0?
yes
no
Print "HCF/GCD = ", *n*
Stop
Replace *m* by *n* and *n* by *r*

Figure 1.21: Flowchart and pseudocode to compute HCF of two numbers

Example 1.11: Draw a flowchart and write a pseudocode to print first *n* terms of the Fibonacci sequence.

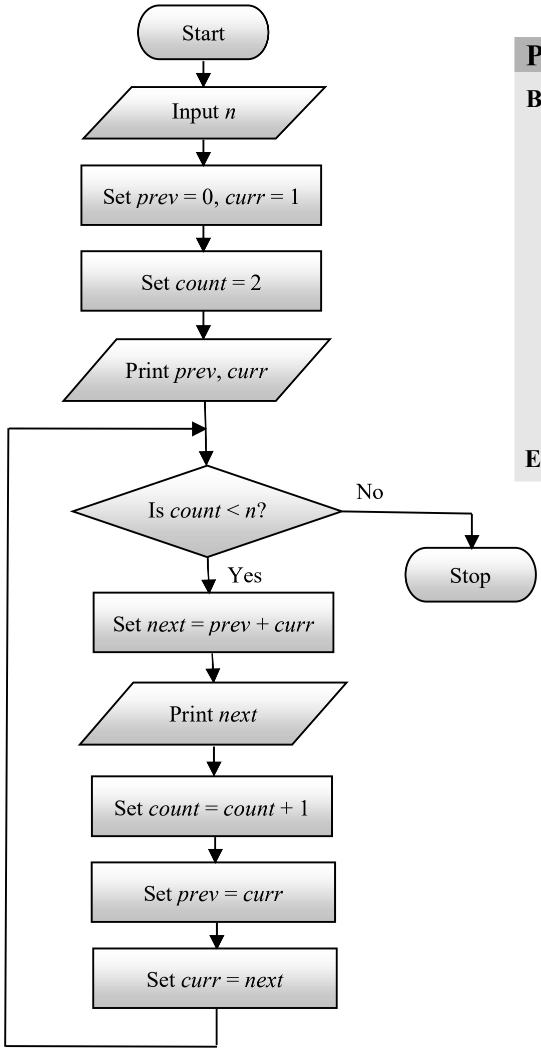
For example, if input value for *n* is 8, the output should be

```
0   1   1   2   3   5   8   13
```

Solution: Observe that, leaving first two terms, each term is obtained as the sum of the immediately preceding two terms.

If we use variable *prev* for previous term, *curr* for current term, *next* for next term, and setting *prev* and *curr* to values 0 and 1, respectively, *i.e.*, first two terms of the sequence, then the entire sequence can be generated by using the recurrence relation

next = *prev* + *curr*
replace *prev* by *curr*
replace *curr* by *next*



Pseudocode 1.12

```
Begin
   Read: n
   Set prev = 0, curr = 1
   Set count = 2
   Print: prev, curr
   While ( count < n ) do
       Set next = prev + curr
       Print: next
       Set count = count + 1
       Set prev = curr
       Set curr = next
   Endwhile
End.
```

Figure 1.22: Flowchart and pseudocode to print first n terms of the Fibonacci sequence

1.6 ALGORITHM TO PROGRAM

By now, you have learned that algorithms are a way to solve given problems using computer. The algorithm contains the logic to solve a given problem. This logic needs to be converted to a program using a programming language.

Python language is our programming language for this course.

Here is an example of a Python program to test whether the given natural number is palindrome or not.

Listing 1.1

```
/* Program to check whether given natural number is
   palindrome or not
*/
#include <stdio.h>
int main()
{
   int n, t, sum, d;
   printf("Enter any natural number : ");
   scanf("%d",&n);
   t = n;
   sum = 0;
   while ( t > 0 )
   {
      d = t % 10;
      sum = sum * 10 + d;
      t = t / 10;
   }
   if ( sum == n )
      printf("\n%d is a palindrome number.\n", n);
   else
      printf("\n%d is not a palindrome number.\n", n);
   return 0;
}
```

First Program Run

```
Enter any natural number : 1221

1221 is a palindrome number.
```

Second Program Run

```
Enter any natural number : 1205

1205 is not a palindrome number.
```

We will be learning about C language in subsequent chapters.

REVIEW EXERCISE

1. What do you understand by the term *programming*? Elaborate.
2. What is problem-solving? What is the need for problems-solving?
3. What is a process? Differentiate between *adhoc process* and *defined process.*
4. Name the various approaches for problem solving.
5. Describe the top-down approach of problem solving.

6. Describe the bottom-down approach of problem solving.
7. Differentiate between top-down approach and bottom-up approach to problem solving.
8. What do you mean by structured programming?
9. Describe the various control structures that meet the requirement of structured programming.
10. What is an algorithm? Describe the essential characteristics of an algorithm.
11. Given a choice to represent the solution of a problem by flowchart or pseudocode, which you will prefer and why? Elaborate.

NUMERICAL AND LOGICAL PROBLEMS

Solve following numerical and logical problems and express their solution using flowchart/pseudocode:

1. To find largest of the five numbers *a*, *b*, *c*, *d*, and *e*.
2. To find the day of the week on a given date.
3. To test whether a given year is a leap year or not. [**Hint:** A given year will be a leap year if it is divisible by 4 but not by 100. If a year is divisible by 4 and by 100, it is not a leap year unless it is also divisible by 400.]
4. To test whether the given date in format *dd*/*mm*/*yyyy* is valid or not.
5. To test whether a given number is a perfect number or not. [**Hint:** A perfect number is a number in which the sum of its proper divisors is equal to the number itself. Proper divisors of a number are all divisors of a number excluding itself.]
6. To test whether a given number is strong number or not. [**Hint:** A strong number is a number whose sum of the factorial of its each digit equals to the number itself.]
7. Suppose an amount *p* is deposited in a commercial bank, which pays compound interest at the rate of *r*% annually, for *n* years. Write a pseudocode that prints the amount in account after each year.
8. To find LCM and HCF of two natural number *m* and *n*.
9. To find factorial of a natural number *n*.
10. To print first *n* pairs of twin prime numbers. Note that two consecutive prime numbers are said to twin prime numbers if they differ by 2.
11. The monthly telephone bill is to be computed as follows:
    Minimum Rs. 200 for upto 100 calls
    plus Rs. 0.60 per call for next 50 calls
    plus Rs. 0.50 per call for next 50 calls
    plus Rs. 0.40 per call for any call beyond 200 calls.
    The input contains name of the customer and number of calls made and the desired output is the name and telephone bill to be paid by the customer.

12. State pollution control board has the following classification policy:

Pollution Index	**Classification**
< 30	Pleasant
30 – 60	Unpleasant
> 60	Hazardous

To prints the appropriate classification for given pollution index.

13. A department store places an order with a company for *n* pieces of miners, *m* pieces of toasters, and *p* number of fans. The cost of items are as follows:

Item Description	**Price per Unit (in Rupees)**
Miners	1,500
Toaster	200
Fan	450

The discount allowed for various items are 5% for miners, 15% for fan, and 10% for toaster. The company charge 10% as sales tax on all items on net value after deducting the discount. To compute the amount to be paid by the store for given value of *m*, *n*, and *p*.

14. To check whether a triangle can be formed or not from given three line segments whose measure is given as *a*, *b*, and *c*.
15. To check whether a triangle can be formed or not from given three angles whose measure is given as *a*, *b*, and *c*.
16. To find the type of the angle when a measure of one angle is given in degrees and in anti-clockwise direction.
17. Given three points $A(x_1, y_1)$, $B(x_2, y_2)$ and $C(x_3, y_3)$, to determine whether they are collinear, *i.e.*, lie on the same line.
18. Given points (x_1, y_1) & (x_2, y_2) on line *AB*, and points (x_3, y_3) & (x_4, y_4) on line *CD*, write a the steps to determine whether lines AB & CD intersect each other.
19. To find your age when your date-of-birth and today's date is given, both in the format *dd*/*mm*/*yyyy*.
20. To find convert a time in 12 hours system to 24 hours system.
21. To find convert a time in 24 hours system to 12 hours system.
22. To find difference in time when the start time and ending time is given, both in the format *hh*:*mm*:*ss*.

❑ ❑ ❑