Introduction to Problem Solving

chapter 4

In This Chapter

4.1 Introduction

There is a famous quote by Steve Jobs, which says, *"Everyone in this country should learn to program a computer, because it teaches you to think"*.

This quote itself is proof-enough to say that in order to program a solution for a problem, you should think in a specific way. And this is what we are going to talk about in this chapter, *i.e.*, problem solving, *i.e.*, analysing a problem, designing algorithms using tools like flowcharts and pseudocode etc.. So, let us begin.

4.2 Problem Solving Cycle

Programs are not quick creations. In order to create efficient and effective programs, you should adopt a proper problem solving methodology and use appropriate techniques.

Broadly problem solving requires *four* main steps :

1. Identify and analyse the problem.
2. Find its solution and Develop algorithm of the solution.
3. Code the solution in a programming language.
4. Test and Debug the coded solution.

And finally **implement and maintain** it.

The above mentioned steps are four major steps in a problem solving cycle. Each step contains many sub-steps. The aim of a problem solving cycle is to create a working program for the solution.

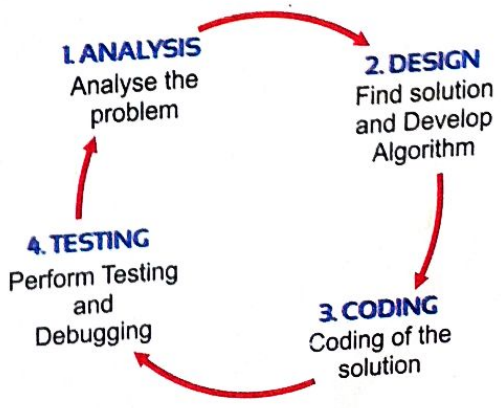


Figure 4.1 Steps in problem solving cycle.

The sub-steps of a problem solving cycle to create a working program are :

Analyze the problem	1.	*Understand the problem well*	It is very important to understand the problem minutely because sometimes the problem is not that appears but something else that is causing it.
	2.	*Analyze the problem*	Problem analysis is very important as the outcome of this will convert to the success of final solution. While analyzing, ❖ identify processing components ❖ identify the relationships among processing components.
Developing the Algorithm	3.	*Think of possible solutions*	As per the problem analysis, think of possible solutions by trying this : ❖ think of different ideas for solutions ❖ check your ideas for aptness ❖ finally, zero on most appropriate solution.
	4.	*Follow Modular approach while designing most appropriate solution*	Many small logically related modules or functions must be preferred over a big problem. It is called *Modular approach* wherein we divide a big program into many smaller and more manageable and understandable modules. Design the final program by ❖ deciding step by step solution ❖ breaking down solution into simple steps.
	5.	*Identify operations for solution*	Program coding involves identification of constituent operations required to get the desired output. One must decide about the minimum but simple operations required. Identify : ❖ minimum number of inputs required ❖ arithmetic and logical operations required for solutions ❖ simple and efficient data structures suiting the need.
Coding	6.	*Code program using appropriate control structures*	The next step is to code the program as per finding of previous step. Coding is the technical word for writing the program. This step is to translate the algorithm into a programming language. You should use most appropriate control structures out of available options. Thus, it is important to know the working of different control structures and their suitability in different situations. ❖ Use appropriate control structures such as conditional or looping control structures. ❖ Think program's efficiency in terms of speed, performance and effectiveness.
Testing and Debugging	7.	*Test and Debug your program*	Testing is the process of finding errors in a program and debugging is the process of correcting errors found during the testing process. Thus, this phase involves : ❖ finding errors in it. ❖ rectifying the errors.
	8.	*Complete your documentation*	Documentation is intended to allow another person or the programmer at later date, to understand the program. Documentation might also consist of a detailed description of what the program does and how to use the program.
Implement and Maintain	9.	*Implement your code*	After testing and documentation, implement your program for actual use on site. Now, the real users can use your programs.
	10.	*Maintain your program*	Maintaining programs involves modifying the programs to remove previously undetected errors, to enhance the program with different features or functionality, or keep the program up-to-date as government regulations or company policies change.

4.2.1 Problem Solving using Decomposition

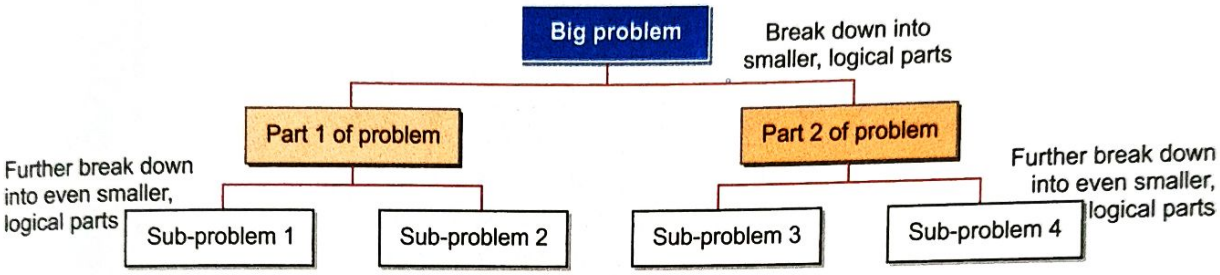
The first five steps of problem solving cycle involve understanding the problem, analyzing it, and thinking of possible solution with sub-modules and operations in it. All this is effectively carried out using decomposition. Let us know what decomposition means.

Decomposition is the process of breaking down a big or complex problem into a set of smaller sub-processes to allow us to describe, understand, or execute the problem better. Decomposition involves :

- Dividing a task into a sequence of subtasks.
- Identifying elements or parts of a complex system.

DECOMPOSITION

The process of breaking down a big or complex problem into a set of smaller sub-processes in order to understand a problem or situation better, is known as **decomposition.**



Consider some examples of decomposition :

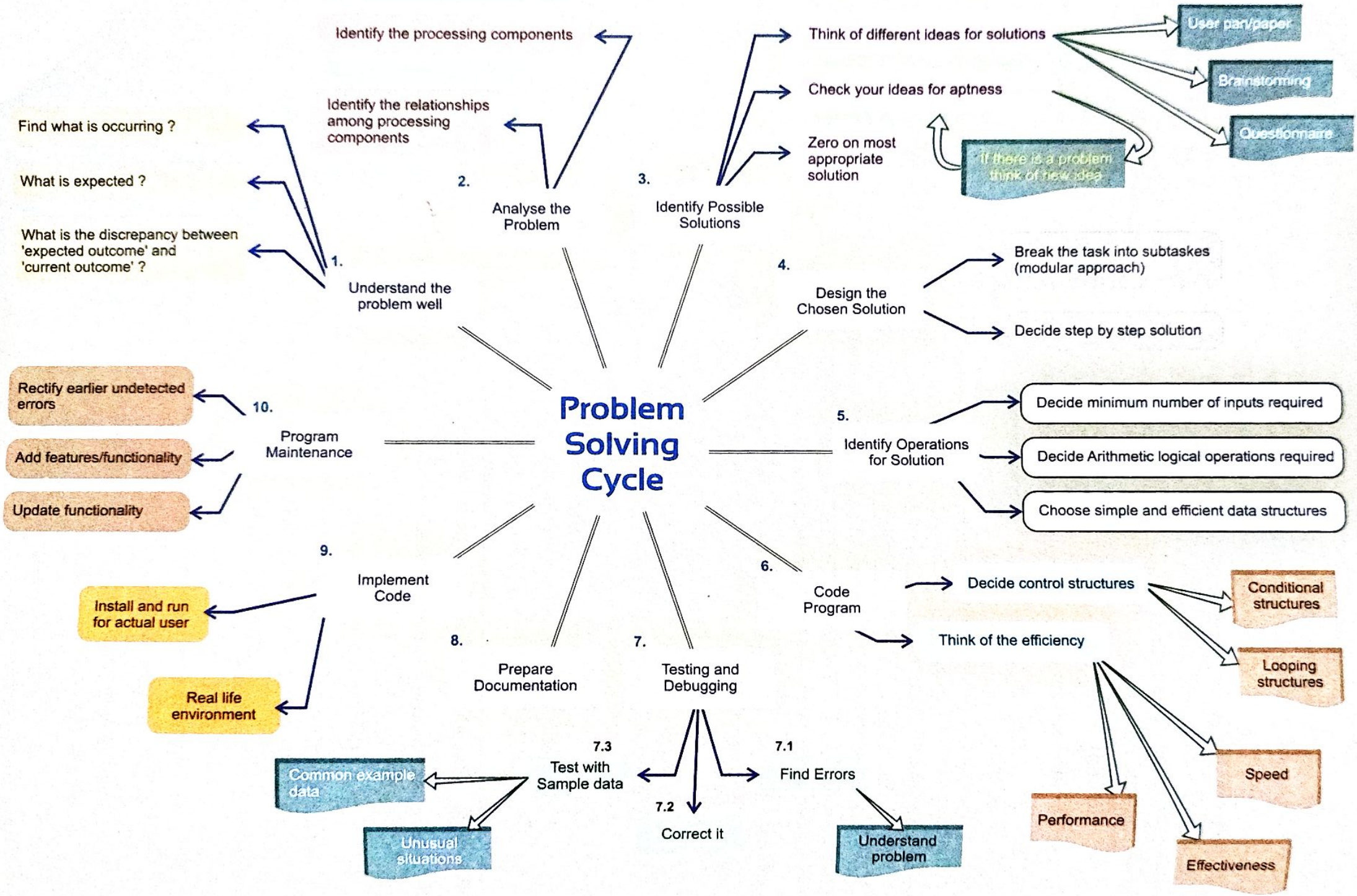
- **Everyday example.** Making cookies is a complex task that can be broken down into smaller, simpler tasks such as dough kneading, forming into shapes *via.* cookie cutters, and baking.
- **Academic example.** Writing an essay is a complex task that can be broken down into smaller tasks such as developing a thesis, gathering evidence, and creating a bibliography page.
- **Engineering example.** Designing a solution to construct a bridge by considering site conditions, technology available, technical capability of the contractor, foundation, etc.
- **Computer Science example.** Writing a computer program/software by determining a well-defined series of smaller steps (mostly in the form of modules and functions) to solve the problem or achieve a desired outcome.

EXAMPLE 1 *Decompose the task of creating mobile app.*

SOLUTION To decompose the task of creating a mobile app, we would need to know the answer to a series of smaller problems :

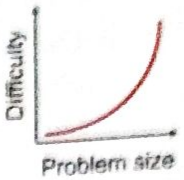
- what kind of app is to be created.
- who the target audience for the app is.
- what the user interface of the app will be (what all screens, type of input etc.).
- what the app's graphics will look like.
- what audio will be included.
- what software/platform will be used to build the app.
- how the user will navigate the app.
- what additional services will be required for the app, *e.g.*, database etc.
- how the app will be tested.

This list has broken down the complex problem of creating an app into much simpler problems that can now be worked out.


Problem Solving Cycle
1. Understand the problem well
Find what is occurring ?
What is expected ?
What is the discrepancy between 'expected outcome' and 'current outcome' ?
2. Analyse the Problem
Identify the processing components
Identify the relationships among processing components
3. Identify Possible Solutions
Think of different ideas for solutions
User pan/paper
Brainstorming
Questionnaire
Check your ideas for aptness
If there is a problem think of new idea
Zero on most appropriate solution
4. Design the Chosen Solution
Break the task into subtaskes (modular approach)
Decide step by step solution
5. Identify Operations for Solution
Decide minimum number of inputs required
Decide Arithmetic logical operations required
Choose simple and efficient data structures
6. Code Program
Decide control structures
Conditional structures
Looping structures
Think of the efficiency
Speed
Performance
Effectiveness
7. Testing and Debugging
7.1 Find Errors
Understand problem
7.2 Correct it
7.3 Test with Sample data
Common example data
Unusual situations
8. Prepare Documentation
9. Implement Code
Install and run for actual user
Real life environment
10. Program Maintenance
Rectify earlier undetected errors
Add features/functionality
Update functionality

Need for Decomposition

Decomposition is the process of breaking a large problem into more manageable sub-problems. It is very important to decompose a problem into smaller sub-problems, reason being that large problems are disproportionately harder to solve than smaller problems. It's much easier to write two 500-line programs than a single 1000-line program. Larger a problem is, harder and more difficult it is to program as compared to a smaller problem (*see* figure).



Once you know how you can decompose a problem in smaller steps, you can create its solution by designing algorithms for it.

NOTE

Decomposing a problem into smaller sub-problems is important because large problems are disproportionately harder to solve than smaller problems.

4.3 Designing Algorithms

The set of rules that define how a particular problem can be solved in finite number of steps is known as algorithm. An algorithm is composed of a finite set of steps, each of which may require one or more operations. But there are certain constraints to be placed on the type of operations as algorithm can include. These are :

Each operation must be definite	*i.e.*, it must be clearly defined what should be done. For instance, '$x = 6/0$', 'add 3 or 8 to a' are not permitted as these operations are not clearly defined.
Each operation must be effective	*i.e.*, each step must be such that it can be done using pencil and paper in a finite amount of time. *For example*, arithmetic on integers is effective operation, whereas arithmetic on real numbers is not since some values may be expressible only by an infinitely long decimal expansion.
Each operation must be finite	*i.e.*, the algorithm should terminate after a finite number of operations.

Thus, we can say that a good algorithm must have characteristics as listed in the following sub-section.

Characteristics of a Good Algorithm

In order to be an effective algorithm, an algorithm must have the following characteristics :

1. **Precision.** An algorithm should be precise, *i.e.*, its steps should be precisely defined.
2. **Uniqueness.** The result of a step is unique and it is only dependent on the input and the result of the preceding steps.
3. **Finiteness.** An algorithm must be finite. It must not repeat the steps endlessly. The algorithm must stop after a finite number of instructions have been executed.
4. **Input.** An algorithm requires a specific type of input to work upon.
5. **Output.** An algorithm produces output as per the stated objectives and the input it has received.

Components of an Algorithm

An algorithm clearly identifies the following components :

- **Input** – the inputs and the type of inputs required by the algorithm. The inputs can be provided by a user or can be self-obtained too, such as reading from a file.

- **Processing** – what and how the processing would use inputs to produce the desired output.
- **Output** – the output expected from the algorithm ; the objective of the algorithm.

A program is the expression of an algorithm in a programming language.

NOTE

One must be able to identify where inputs, processing and outputs are taking place within an algorithm.

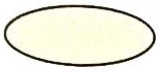
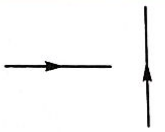
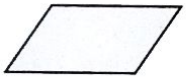
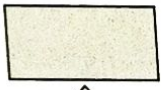
4.3.1 Flowcharts

FLOWCHART

A **flowchart** is a pictorial representation of step by step solution of a problem.

A flowchart is a pictorial representation of step by step solution of a problem. A flowchart not only pictorially depicts the sequence in which instructions are carried out in an algorithm but also is used as an aid in developing algorithms.

There are various flowchart symbols that carry different messages and are used for different purposes. These symbols are shown below :

Symbol	Purpose	Symbol	Purpose
	Start/Stop		Flow of control symbol
	Input/Output		Annotation
	Processing		
	Decision Box		Connector

Following section illustrates the working and use of flow charts along with algorithm development.

Writing Algorithms

We have chosen a language for algorithms that resembles our programming language Python. Let us discuss certain rules for writing algorithms.

Identifiers

Identifiers are the names given to various components of a program by the programmer *e.g.*, to variables that hold values, to functions, modules etc.

While choosing names for identifiers, make sure that these are meaningful and not unnecessarily long or short names.

Assignment

The assignment of values to variables is done through assignment statement as :

`variable ← <expression>` *e.g.*, `A ← 10`

In an algorithm, the left arrow (←) denotes the act of assigning the value of its right-hand side to the variable on its left. Some people take liberty and use assignment operator to assign values to variables in algorithms.

Sequence

The steps of an algorithm are executed in the sequence of top to bottom. One after another so steps must be listed in the correct order.

Selection/Conditional Statements

A conditional statement in an algorithm takes the following form :

```
if condition :                          if condition :
    statement     # block 1      OR        statement          # block 1
                                        else
                                            S2                 # block 2
```

There may be one or more if-then-else statements embedded in another if-then-else statement.

Figure 4.2 depicts conditional statements pictorially.

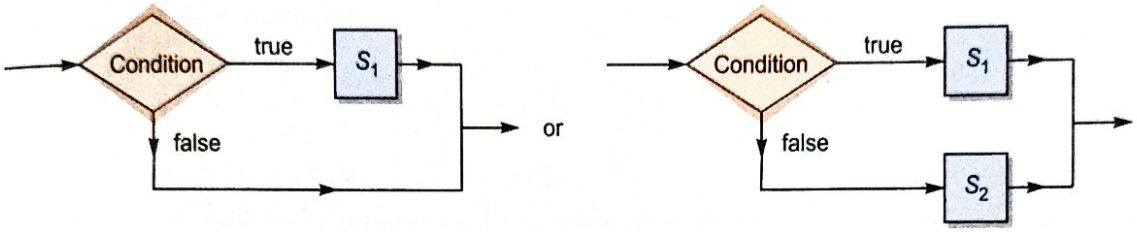


Figure 4.2 The conditional **If** statements.

Repetition – Looping Statement

A looping statement (also called iteration statement) lets you repeat a set of statements depending upon a condition. To accomplish iteration, the **for loop** and **while loop** are used.

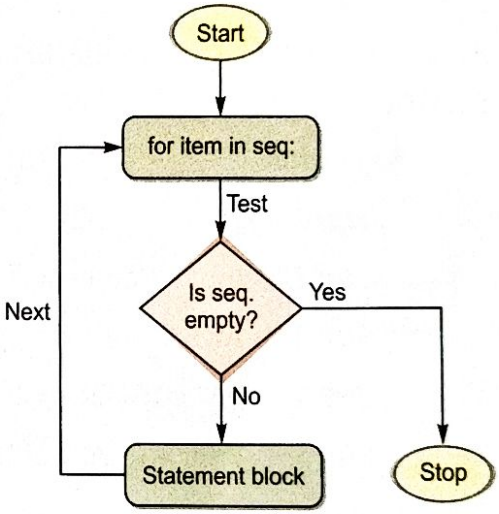


Figure 4.3 (*a*) for loop process.

for looping statement

```
for item in sequence :
    : St              # block of statements
:
```

St represents the set of statements to be repeated for each item in the sequence. When the statements block gets executed for each item in the sequence, the for-loop stops.

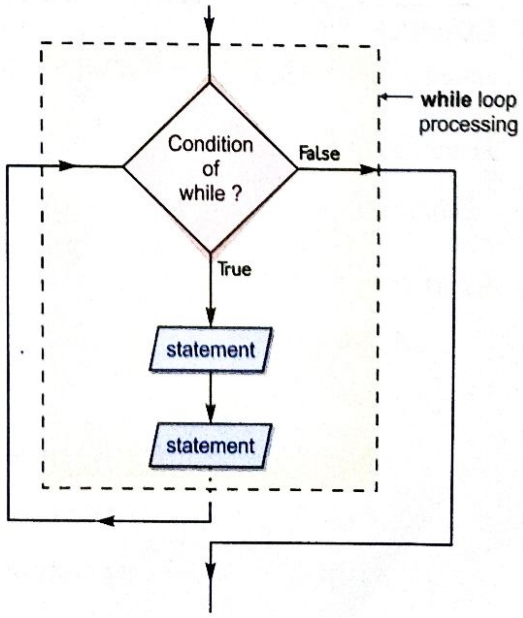


Figure 4.3 (*b*) while loop processing.

while looping statement

```
while condition :
    :   St            # block of statements
:
```

St represents the set of statements to be repeated. The *while* iteration statement tests the condition before entering into the loop. Thus, if the condition is false even before entering into the loop, the while-loop will never get executed.

Let us now consider some examples of algorithms and flowcharting.

EXAMPLE 2 *Draw a flowchart to add two numbers, along with algorithm in simple English.*

SOLUTION

Algorithm

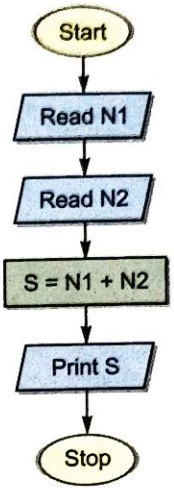
Step 1. Start

Step 2. Get two numbers N1 and N2

Step 3. S ← N1 + N2

Step 4. Print the result S

Step 5. Stop



EXAMPLE 3 *Draw a Flowchart to find Area and Perimeter of Rectangle, along with algorithm in simple English.*

L : Length of Rectangle, ***B*** *: Breadth of Rectangle*

AREA *: Area of Rectangle,* ***PERIMETER*** *: Perimeter of Rectangle*

SOLUTION

Algorithm

Step 1. Start

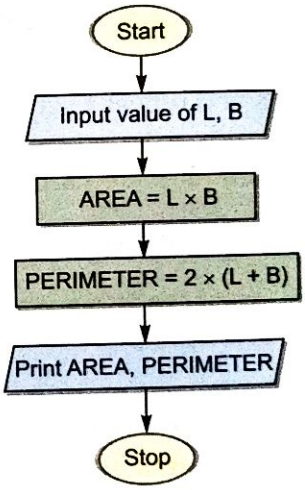
Step 2. Input Side-Length & Breadth say *L, B*

Step 3. AREA ← $L \times B$

Step 4. PERIMETER ← $2 \times (L + B)$

Step 5. Print AREA, PERIMETER

Step 6. Stop



EXAMPLE 4 *Draw a flowchart to convert temperature from* ***Celsius*** *to* ***Fahrenheit*** *along with algorithm in simple English.* ***C*** *: temperature in Celsius,* ***F*** *: temperature Fahrenheit*

Formulas to be used :

Fahrenheit to Celsius : $C = (F - 32).\frac{5}{9}$; *Celsius to Fahrenheit :* $F = C.\frac{9}{5} + 32$

SOLUTION

Algorithm

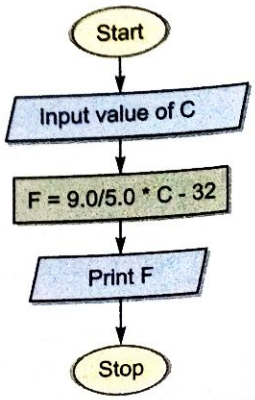
Step 1. Start

Step 2. Input temperature in Celsius in variable C

Step 3. F ← $(9.0 / 5.0 \times C) + 32$

Step 4. Print Temperature in Fahrenheit F

Step 5. Stop



EXAMPLE 5 *Draw a flowchart to determine the larger of two numbers, along with algorithm in simple English.*

SOLUTION

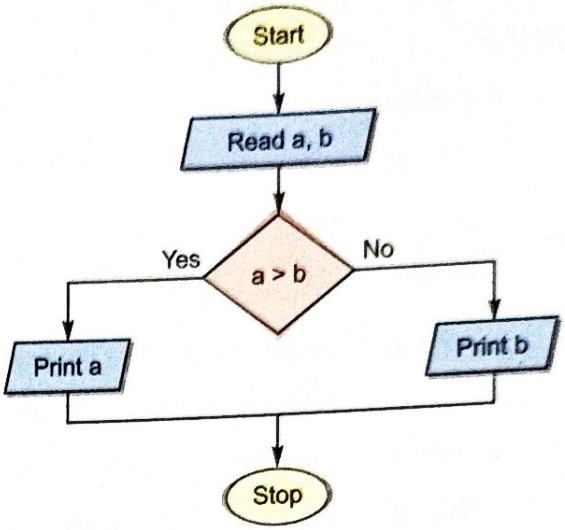
Algorithm

Step 1. Start

Step 2. Get two numbers *a* and *b*

Step 3. If $a > b$ then print *a* else print *b*

Step 4. Stop



EXAMPLE 6 *Draw a flow chart to print even numbers from 2 to 10^4 using a loop approach, along with algorithm in simple English.*

SOLUTION

Algorithm

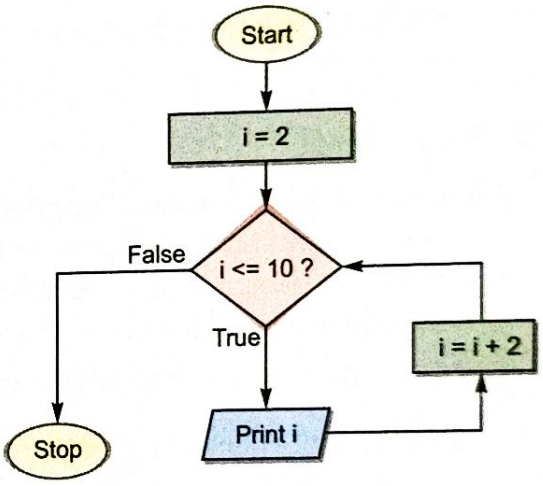
Step 1. $i \leftarrow 2$

Step 2. While $i < 10$
Repeat steps 3 through 4

Step 3. Print *i*

Step 4. $i \leftarrow i + 2$

Step 5. Stop



EXAMPLE 7 *Draw a flow chart to calculate 2^4, along with algorithm in simple English.*

SOLUTION

Algorithm

Step 1. Input Base (2), Power (4)

Step 2. Product ← Base

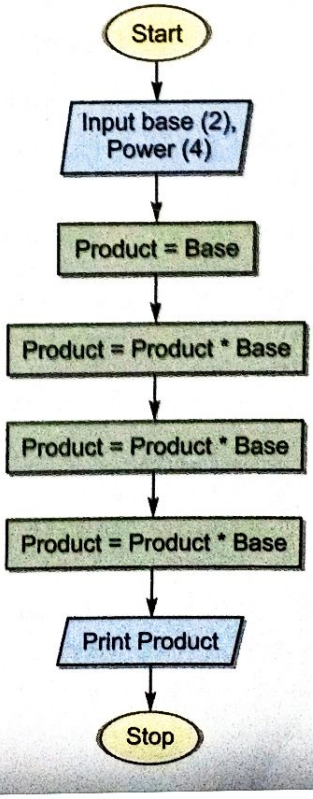
Step 3. Product ← Product * Base

Step 4. Product ← Product * Base

Step 5. Product ← Product * Base

Step 6. Print Product

Step 7. Stop



EXAMPLE 8 *Draw a flow chart to calculate 2^4 using a loop approach, along with algorithm in simple English.*

SOLUTION

Algorithm

Step 1. *Input* Base (2), Power (4)

Step 2. Product ← Base

Step 3. Counter ← 1

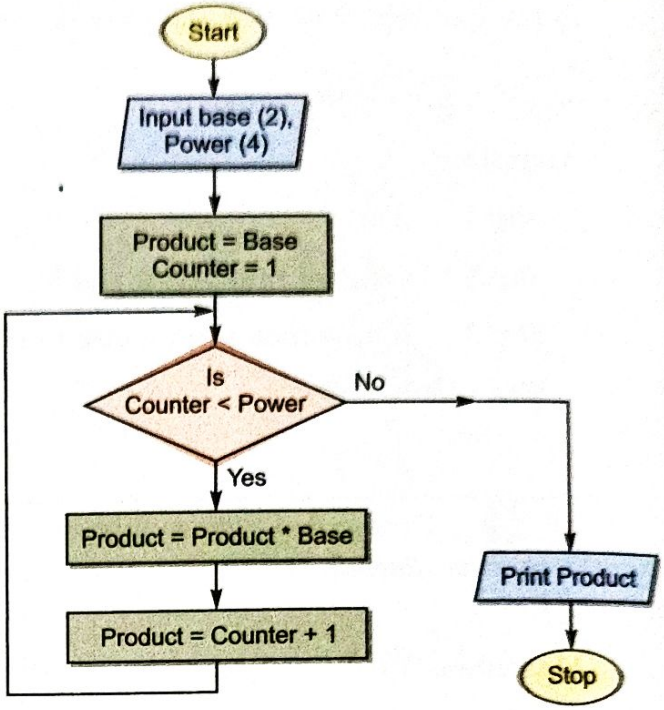
Step 4. *While* (Counter < Power)
Repeat steps 4 through 6

Step 5. Product ← Product * Base

Step 6. Counter ← Counter + 1

Step 7. Print Product

Step 8. Stop



4.3.2 Pseudocode

Pseudocode is an informal language that helps programmers describe steps of a program's solution without using any programming language syntax. Pseudocode is a "text-based" detail (algorithmic) design tool. A pseudocode creates an outline or a rough draft of a program that gives the idea of how the algorithm works and how the control flows from one step to another.

For example, consider the following pseudocode :

```
If student's marks is greater than or equal to 50
    display "passed"
else
    display "failed"
```

Consider the same algorithm that we talked in example 5 (***determine if the first number is greater than the second number or not***).

The pseudocode of this algorithm can be like :

```
Input first number in variable firstnum.
Input second number in variable secondnum

if the firstnum is > the secondnum
    display 'the first number IS greater than second number'.
else
    display 'the first number IS greater than second number'.
```

Please note there are no standard rules to write a pseudocode as it is totally informal way of representing an algorithm.

NOTE

Pseudocode is an informal way of describing the steps of a program's solution without using any strict programming language syntax or underlying technology considerations.

NOTE

In pseudocode, usually the instructions are written in upper-case, variables in lowercase and messages in sentence case.

Advantages and Disadvantages of Pseudocode

The pseudocode offers many advantages and it has some disadvantages too.

Pseudo-code Advantages

- It uses a language similar to everyday English, thus is easy to understand.
- It highlights the sequence of instructions.
- The structure of an algorithm is clearly visible through pseudocode, *e.g.*, selection and repetition blocks.
- Pseudocode can be easily modified without worrying about any dependencies.

Pseudo-code Disadvantages

- It is not a visual tool like flowcharts.
- Since there is no accepted standard for pseudocode, so it varies from programmer to programmer.
- It can only be tested on paper, there is no program available to test it.
- It is an informal representation of an algorithm.

EXAMPLE 9 *Write pseudocode that converts from Fahrenheit to Celsius or from Celsius to Fahrenheit, depending on the user's choice.*

SOLUTION **Pseudocode :**

```
x = INPUT "Press 1 to convert from Fahrenheit
   to Celsius or Press 2 to convert from
   Celsius to Fahrenheit."
y = INPUT ask what number?
IF 1 is pressed
   # do f to c conversion
   C = 5/9 * (F - 32)
   PRINT output C
ELSE IF 2 is pressed
   # do c to f conversion
   F = 9/5 * (C + 32)
   PRINT output F
ELSE
   PRINT "Please enter either 1 or 2"
```

EXAMPLE 10 *Write pseudocode that lets the user type words and when they press 'x', it prints how many words the user inputted then quits program.*

SOLUTION **Pseudocode :**

```
WHILE true
   INPUT "Enter a Word"
   PRINT word
   IF word = 'x'
      PRINT number of words entered
      BREAK OUT from the loop
   ELSE
      add 1 to count
```

EXAMPLE 11 *Write pseudocode that asks how many numbers (say n), and then print all those numbers after n numbers have been entered.*

SOLUTION **Pseudocode :**

```
INPUT value of n              # how many numbers
Initialise numbercount = 0
Initialise listnum as an empty list
WHILE the numbercount <= n
   ask for a number
   add one to number count
   add number to listnum
# now the numbers have been entered, loop is over
# take listnum's numbers in item one by one
FOR item in listnum
   PRINT item
   # item now takes next number in listnum
```

4.3.3 Verifying an Algorithm

Verification of an algorithm means to ensure that the algorithm is working as intended. For example, if you have written a program to add two numbers but the algorithm is giving you the product of the two input numbers. In this case, the algorithm is not working as intended.

To verify an algorithm, we should test it with a sample inputs for which we know the output; if the expected output matched with the output produced by the algorithm, the algorithm is verified.

This is really helpful, but for larger algorithms, we may want to verify the intermediate results too of various steps of the algorithm. And for such requirements, a useful mechanism of verifying algorithm called **Dry-Run** is used.

NOTE

The word **'algorithm'** comes from the ninth-century Arab mathematician, *Al-Khwarizmi*, who worked on 'written processes to achieve some goal.' The term 'algebra' also comes from the term *'al-jabr,'* which he introduced.

4.3.3A Dry Run

A dry run is the process of a programmer manually working through their code to trace the value of variables. There is no software involved in this process.

Traditionally, a dry run would involve a print out of the code. The programmer would sit down with a pen and paper and manually follow the value of a variable to check that it was used and updated as expected.

If a programmer found that the value is not what it should be, they are able to identify the section of code that resulted in the error.

Characteristics of a dry run are :

- It is carried out during design, implementation, testing or maintenance.
- It is used to identify the logic errors in a code.
- It cannot find the execution errors in a code.

DRY RUN

A **dry run** is the process of a programmer manually working through their code to trace the value of variables. There is no software involved in this process.

Trace Tables

Dry running an algorithm is carried out using trace tables where the impact of each line of the code is seen on the values of **variables** of the algorithm. As per the line of the code, the processing that takes place is applied on the variables' values.

Check Point 4.1

1. Name some useful tools for program development.
2. What is the difference between an algorithm and pseudocode ?
3. Which of the following is graphical ?
   (*a*) algorithm (*b*) flowchart
   (*c*) pseudocode (*d*) dry run
4. Which of the following useful for tracing a pseudo code ?
   (*a*) algorithm (*b*) flowchart
   (*c*) pseudocode (*d*) dry run
5. What is the use of trace table ?

For example, we want to calculate the double of the numbers in sequence 1, 2, 3 and print the value as 1 added to the double value :

```
1.  FOR number ← 1 TO 3
2.      Val ← number * 2
3.      PRINT Val + 1
```

To see Dry Run in action

Scan QR Code

The trace table for the above code will record the code movement and its impact on the variables, line by line :

Line number	Number	Val ← number * 2	Print Val + 1
1	1		
2			
3		2	
			3
Loop's next pass starts			
1	2		
2			
3		4	
			5
Loop's next pass starts			
1	3		
2			
3		6	
			7

Number assigned value 1

Val is = number *2

Val+1 is printed

So the given code's trace table will be as shown above, when the code is dry-run. The above given trace table has documented the impact of each line of the code separately.

However, you can skip this style and simply show the impact of code on variables too, *e.g.*, as shown below :

Line numbers	Number	Val ← number * 2	Print Val + 1
1-3	1	2	3
1-3	2	4	5
1-3	3	6	7

You can choose to skip line numbers too, if you want as we have done in the example below.

NOTE

Trace tables enable the variable values in an algorithm to be recorded as the algorithm is dry run.

EXAMPLE 12 *Dry-run the code given below and show its trace table.*

```
num ← USER INPUT
count ← 0
WHILE num < 500 THEN
    num ← num * 2
    count ← count + 1
# end while
PRINT num
PRINT count
```

SOLUTION

num	count	num < 500	OUTPUT (Values printed)
16	0	TRUE	
32	1	TRUE	
64	2	TRUE	
128	3	TRUE	
256	4	TRUE	
512	5	FALSE	
			512
			5

4.3.4 Comparing Algorithms

To solve a problem, many a times, in fact, mostly multiple solutions are available. In order to decide which algorithm to choose over another, their efficiency will be the deciding factor.

Major factors that govern the efficiency of an algorithm are :

- the time it takes to find the solution and
- the resources which are consumed in the process.

For instance while buying a car, how do you choose a car from available choice? Yup, you are absolutely right; you might consider both the speed and the fuel consumption. The factor that matters the most to you, will win ☺.

Similarly, for algorithms, the *two* deciding factors are :

- **Space-wise efficiency.** How much amount of (memory) space the algorithm will take up before it terminates. For this, we consider the amount of data the algorithm uses while running.
- **Time-wise efficiency.** How much time the algorithm takes to complete. This factor is dependent on so many issues like, RAM available, programming language chosen, the hardware platform and many others.

When you have hardware platform fixed with a specific RAM size, then space efficiency becomes the deciding factor and the algorithms that takes up less space is chosen.

LET US REVISE

- *A Flowchart is a pictorial representation of step by step solution of a problem.*
- *The logical sequence of precise steps that solve a given problem, is called Algorithm.*
- *An algorithm must be a finite series of steps; each step must be precise ; it must be efficient i.e., terminate in a reasonable amount of time, and must produce the correct results i.e., it must be effective.*
- *The process of breaking down a big or complex problem into a set of smaller sub-processes in order to understand a problem or situation better, is known as* ***decomposition****.*
- *Decomposing a problem into smaller sub-problems is important because large problems are disproportionately harder to solve than smaller problems.*
- *Pseudocode is an informal way of describing the steps of a program's solution without using any strict programming language syntax or underlying technology considerations.*

OBJECTIVE TYPE QUESTIONS OTQs

MULTIPLE CHOICE QUESTIONS

1. What is an algorithm ?
   (*a*) A set of steps to solve a problem (*b*) Software that analyses data
   (*c*) A hardware device that stores data (*d*) All of these
2. What is pseudo-code ?
   (*a*) A diagrammatic representation of a set of instructions
   (*b*) An algorithm written out in words
   (*c*) A very specific programming language used by all computers (*d*) All of these
3. Which of the following would assign user input to a variable?
   (*a*) INPUT = myVar (*b*) myVar ← INPUT (*c*) INPUT ← myVar (*d*) All of these
4. What shape represents a decision in a flowchart ?
   (*a*) A diamond (*b*) A rectangle (*c*) An oval (*d*) None of these
5. What is decomposition ?
   (*a*) Breaking code down once it has been run
   (*b*) Breaking a problem down into smaller, more manageable sections
   (*c*) Breaking a problem into subroutines (*d*) Breaking big data into small data