

2. ALGORITHMS, FLOWCHARTS, DATA TYPES AND PSEUDOCODE

2.1 ALGORITHMS

The term **algorithm** originally referred to any computation performed via a set of rules applied to numbers written in decimal form. The word is derived from the phonetic pronunciation of the last name of *Abu Ja'far Mohammed ibn Musa al-Khowarizmi*, who was an Arabic mathematician who invented a set of rules for performing the four basic arithmetic operations (addition, subtraction, multiplication and division) on decimal numbers.

An algorithm is a representation of a solution to a problem. If a problem can be defined as a difference between a desired situation and the current situation in which one is, then a problem solution is a procedure, or method, for transforming the current situation to the desired one. We solve many such trivial problems every day without even thinking about it, for example making breakfast, travelling to the workplace etc. But the solution to such problems requires little intellectual effort and is relatively unimportant. However, the solution of a more interesting problem of more importance usually involves stating the problem in an understandable form and communicating the solution to others. In the case where a computer is part of the means of solving the problem, a procedure, explicitly stating the steps leading to the solution, must be transmitted to the computer. This concept of problem solution and communication makes the study of algorithms important to computer science.

Throughout history, man has thought of ever more elegant ways of reducing the amount of labour needed to do things. A computer has immense potential for saving time/energy, as most (computational) tasks that are repetitive or can be generalised can be done by a computer. For a computer to perform a desired task, a method for carrying out some sequence of events, resulting in accomplishing the task, must somehow be described to the computer. The algorithm can be described on many levels because the algorithm is just the procedure of steps to take and get the result. The language used to describe an algorithm to other people will be quite different from that which is used by the computer, however the actual algorithm will in essence be the same. An example of an algorithm people use would be a recipe to make a cake.

- "4 extra large eggs, beaten
- 1&1/2 C. stock
- 1/2 teaspoon salt
- 1 scallion, minced
- 1 C. small shrimp or lobster flakes
- 1 t. soy sauce
- 1 Tablespoon oil
- 1. Mix all the ingredients, except the oil, in a deep bowl.

2. Put 1" water in wide pot, then place deep bowl of batter inside.
3. Cover pot tightly and steam 15 min.
4. Heat oil very hot and pour over custard.
5. Steam 5 more min. Serves 4 people"

This breaks down 'Making Chinese egg custard' into smaller steps. To make the product one still needs to know how to execute each of the steps in the procedure and understand all of the terms.

Definition:

A **procedure** is a finite sequence of well-defined instructions, each of which can be mechanically carried out in a finite amount of time.

The procedure must break up the problem solution into parts that the recipient party can understand and execute. In the case of a computer, the problem solution is usually in the form of a program that encompasses the algorithm and explains to the computer a clearly defined procedure for achieving the solution. The procedure must consist of smaller steps each of which the computers understand. There may be no ambiguities in the translation of the procedure into the necessary action to be taken. A program is then just a specific realisation of an algorithm, which may be executed on a physical device.

A computer is essentially a physical device designed to carry out a collection of primitive actions. A procedure is a sequence of instructions written in terms of which evoke a proper operation. To make effective use of an algorithm on a computer one must not only find and understand a solution to the problem but also convey the algorithm to the computer, giving the correct sequence of understood commands that represent the same algorithm.

Definition:

An **algorithm** is procedure consisting of a finite set of unambiguous rules (instructions) which specify a finite sequence of operations that provides the solution to a problem, or to a specific class of problems for any allowable set of input quantities (if there are inputs). In other word, an **algorithm** is a step-by-step procedure to solve a given problem

Alternatively, we can define an algorithm as a set or list of instructions for carrying out some process step by step. A recipe in a cookbook is an excellent example of an algorithm. The recipe includes the requirements for the cooking or ingredients and the method of cooking them until you end up with a nice cooked dish.

In the same way, algorithms executed by a computer can combine millions of elementary steps, such as additions and subtractions, into a complicated mathematical calculation. Also by means of algorithms, a computer can control a manufacturing process or co-

ordinate the reservations of an airline as they are received from the ticket offices all over the country. Algorithms for such large-scale processes are, of course, very complex, but they are built up from pieces.

One of the obstacles to overcome in using a computer to solve your problems is that of translating the idea of the algorithm to computer code (program). People cannot normally understand the actual machine code that the computer needs to run a program, so programs are written in a programming language such as C or Pascal, which is then converted into machine code for the computer to run.

In the problem-solving phase of computer programming, you will be designing algorithms. This means that you will have to be conscious of the strategies you use to solve problems in order to apply them to programming problems. These algorithms can be designed through the use of **flowcharts** or **pseudocode**.

2.2 FLOWCHARTS

Flowcharting is a tool developed in the computer industry, for showing the steps involved in a process. A flowchart is a diagram made up of *boxes*, *diamonds* and other shapes, *connected by arrows* - each shape represents a step in the process, and the arrows show the order in which they occur. Flowcharting combines symbols and flowlines, to show figuratively the operation of an algorithm.

In computing, there are dozens of different symbols used in flowcharting (there are even national and international flowcharting symbol standards). In business process analysis, a couple of symbols are sufficient. A box with text inside indicates a step in the process, while a diamond with text represents a decision point. See the figure for an example.

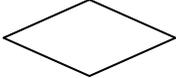
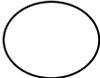
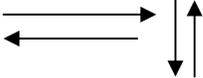
If the flowchart is too messy to draw, try starting again, but leaving out all of the decision points and concentrating on the simplest possible course. Then the session can go back and add the decision points later. It may also be useful to start by drawing a high-level flowchart for the whole organisation, with each box being a complete process that has to be filled out later.

From this common understanding can come a number of things - process improvement ideas will often arise spontaneously during a flowcharting session. And after the session, the facilitator can also draw up a written procedure - a flowcharting session is a good way of documenting a process.

Process improvement starts with an understanding of the process, and flowcharting is the first step towards process understanding.

Flowcharting Symbols

There are 6 basic symbols commonly used in flowcharting of assembly language programs: Terminal, Process, input/output, Decision, Connector and Predefined Process. This is not a complete list of all the possible flowcharting symbols, it is the ones used most often in the structure of Assembly language programming.

Symbol	Name	Function
	Process	Indicates any type of internal operation inside the Processor or Memory
	input/output	Used for any Input / Output (I/O) operation. Indicates that the computer is to obtain data or output results
	Decision	Used to ask a question that can be answered in a binary format (Yes/No, True/False)
	Connector	Allows the flowchart to be drawn without intersecting lines or without a reverse flow.
	Predefined Process	Used to invoke a subroutine or an interrupt program.
	Terminal	Indicates the starting or ending of the program, process, or interrupt program.
	Flow Lines	Shows direction of flow.

Generally, there are many standard flowcharting symbols.

General Rules for flowcharting

1. All boxes of the flowchart are connected with Arrows. (Not lines)
2. Flowchart symbols have an entry point on the top of the symbol with no other entry points. The exit point for all flowchart symbols is on the bottom except for the Decision symbol.
3. The Decision symbol has two exit points; these can be on the sides or the bottom and one side.

4. Generally a flowchart will flow from top to bottom. However, an upward flow can be shown as long as it does not exceed 3 symbols.
5. Connectors are used to connect breaks in the flowchart. Examples are:
 - From one page to another page.
 - From the bottom of the page to the top of the same page.
 - An upward flow of more than 3 symbols
6. Subroutines and Interrupt programs have their own and independent flowcharts.
7. All flow charts start with a Terminal or Predefined Process (for interrupt programs or subroutines) symbol.
8. All flowcharts end with a terminal or a contentious loop.

Flowcharting uses symbols that have been in use for a number of years to represent the type of operations and/or processes being performed. The standardised format provides a common method for people to visualise problems together in the same manner. The use of standardised symbols makes the flow charts easier to interpret, however, standardising symbols is not as important as the sequence of activities that make up the process.

Flowcharting Tips

- Chart the process the way it is really occurring. Do not document the way a written process or a manager thinks the process happens.
- People typically modify existing processes to enable a more efficient process. If the desired or theoretical process is charted, problems with the existing process will not be recognised and no improvements can be made.

Note all circumstances actually dealt with.

- Test the flow chart by trying to follow the chart to perform the process charted. If there is a problem performing the operation as charted, note any differences and modify the chart to correct. A better approach would be to have someone unfamiliar with the process try to follow the flow chart and note questions or problems found.
- Include mental steps in the process such as decisions. These steps are sometimes left out because of familiarity with the process, however, represent sources of problems due to a possible lack of information used to make the decision can be inadequate or incorrect if performed by a different person.

Examples of Algorithms and Flowcharts

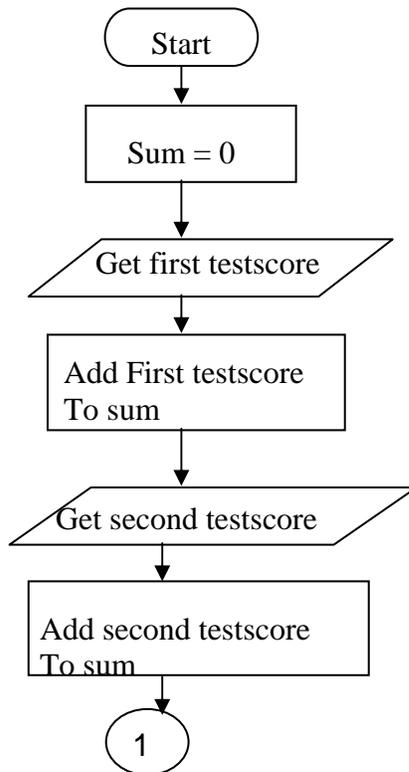
Example 1. Design an algorithm and the corresponding flowchart for adding the test scores as given below:

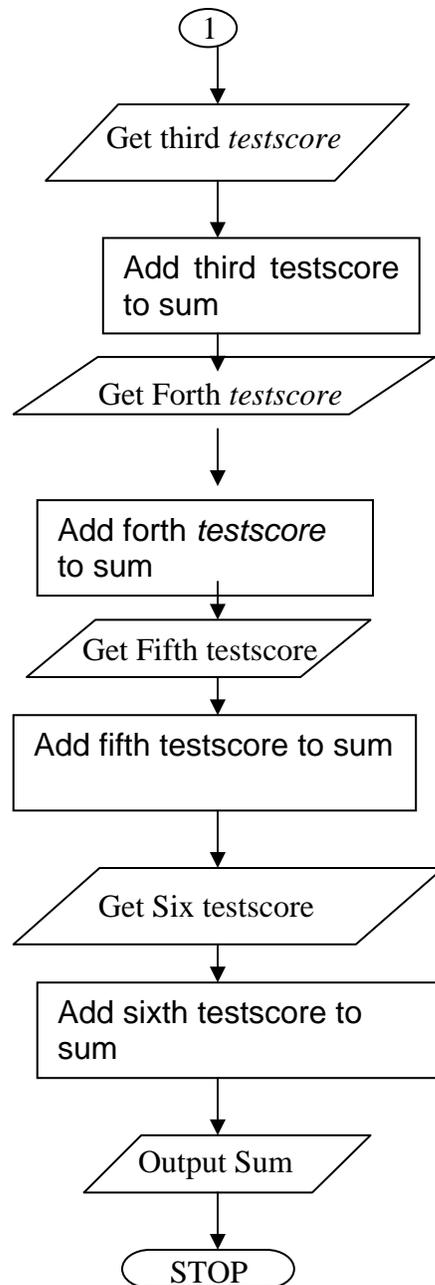
26, 49, 98, 87, 62, 75

a) Algorithm

1. Start
2. Sum = 0
3. Get the first test score
4. Add first test score to sum
5. Get the second test score
6. Add to sum
7. Get the third test score
8. Add to sum
9. Get the fourth test score
10. Add to sum
11. Get the fifth test score
12. Add to sum
13. Get the sixth test score
14. Add to sum
15. Output the sum
16. Stop

b) The corresponding flowchart is as follows:

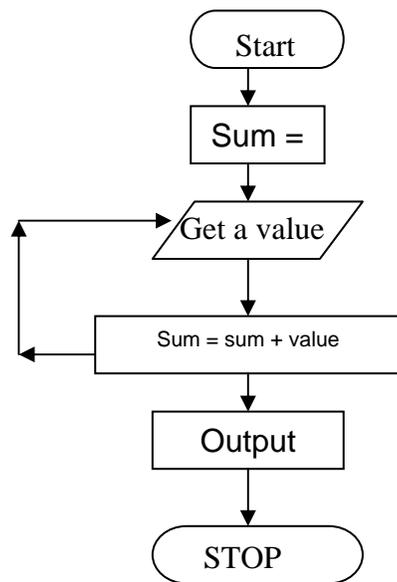




The algorithm and the flowchart above illustrate the steps for solving the problem of adding six test scores. Where one test score is added to sum at a time. Both the algorithm and flowchart should always have a **Start** step at the beginning of the algorithm or flowchart and at least one **stop** step at the end, or anywhere in the algorithm or flowchart. Since we want the sum of six test score, then we should have a container for the resulting sum. In this example, the container is called **sum** and we make sure that sum should start with a zero value by step 2.

Example 2: The problem with this algorithm is that, some of the steps appear more than once, i.e. step 5 get second number, step 7, get third number, etc. One could shorten the algorithm or flowchart as follows:

1. Start
2. Sum = 0
3. Get a value
4. sum = sum + value
5. Go to step 3 to get next Value
6. Output the sum
7. Stop



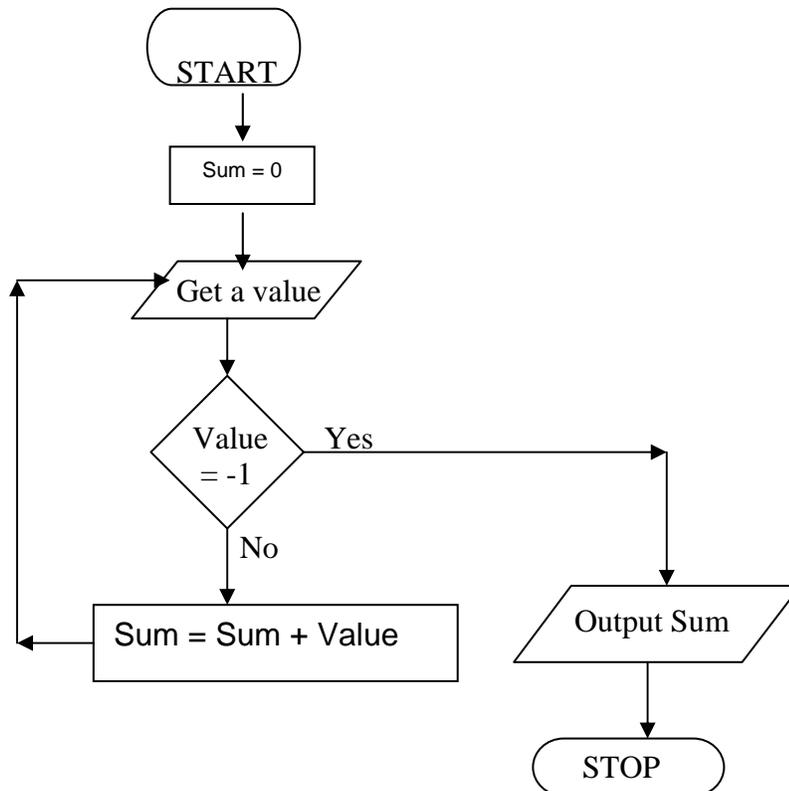
This algorithm and its corresponding flowchart are a bit shorter than the first one. In this algorithm, step 3 to 5 will be repeated, where a number is obtained and added to sum. Similarly the flowchart indicates a flowline being drawn back to the previous step indicating that the portion of the flowchart is being repeated. One problem indicates that these steps will be repeated endlessly, resulting in an **endless** algorithm or flowchart. The algorithm needs to be improved to eliminate this problem. In order to solve this problem, we need to add a last value to the list of numbers given. This value should be unique so that, each time we get a value, we test the value to see if we have reached the last value. In this way our algorithm will be a finite algorithm which ends in a finite number of steps as shown below. There are many ways of making the algorithm finite.

The new list of numbers will be 26, 49, 498, 9387, 48962, 1, -1. The value -1 is a unique number since all other numbers are positive.

1. Start
2. Sum = 0
3. Get a value

4. If the value is equal to -1, go to step 7
5. Add to sum ($\text{sum} = \text{sum} + \text{value}$)
6. Go to step 3 to get next Value
7. Output the sum
8. Stop

Corresponding flowchart



3. DATA TYPES

Although some contemporary languages allow programmers to invent his own data types, and define their related operations, there are a number of traditional data types found in most languages:

Integer

Integers are numeric data items, which are either positive or negative including zero, i.e. 1, 488, -22, 0, 456. Some programming languages put restrictions on the magnitude of integers which may be used in program instructions. These restrictions are usually dependent on the size of the memory location of the computer in which the language may run.

Real Numbers

There are two types of real numbers, Fixed-Point and Floating Point.

Fixed Point

Fixed point data items are numbers which have embedded decimal point i.e. 1.5, 458.4589, -0.569.

Floating Point

Floating point data items are numbers, which are, held as binary fractions by a computer. The numbers are expressed in a form where you have a mantissa and an exponent, for example

Number		Mantissa	Exponent
12.3	= $0.123 * 10^2$	0.123	2
123000	= $0.123 * 10^6$	0.123	6
0.000123	= $0.123 * 10^{-3}$	0.123	-3

Floating point representation of data is used to overcome the restrictions placed on the magnitude of numbers by the size of computer's memory locations.

Character

Character data, sometimes referred to as "string" data, may consist of any digits, letters of the alphabet or symbols which, the internal coding system of the computer is capable of representing. Many programming languages require character data to be enclosed by quotation marks when used in program instructions, for example PRINT "HAPPY NEW YEAR".

Boolean

Boolean data items are used as status indicators and may contain only one of two possible values: True or False.

DATA ITEM

There are two basic methods of using data items in a program:

a) Constants

Data items are sometimes required to keep their values throughout the program, hence the term constant. A constant is a specific value or character string used explicitly in an operation. Consider the constant values 47.5, 1, and 13 in the example below.

```
Multiply ... by 47.5
Add 1 to ...
If ... = 13
Print "PLEASE INPUT TODAY'S DATE"
```

b) *Variables and Variable names*

A variable is a symbolic name assigned to a data item by the programmer. At any particular time, a variable will stand for one particular data, called the value of a variable, which may change from time to time during a computing process. The value of a variable may change many times during the execution of a program. A variable is usually given a name by the programmer.

Assignment

The assignment operation has the form:

variable := expression. (Pascal)

variable = expression. (C/C++/Java)

The assignment operation is used to assign a name to a value. Thus it is used whenever you need to keep track of a value that is needed later. Some typical uses include:

- initialize a variable (count = 0)
- increment/decrement a counter (count = count + 1)
- accumulate values (sum = sum + item)
- capture the result of a computation (y = 3*x + 4)
- swap two values (t = x; x = y; y = t)

The assignment operator is not commutative i.e. $x = e$ is not the same as $e = x$. The variable must be declared. Variables used in the expression must be defined (have values). The type of the expression must be compatible with the type of the variable.

The order in which assignments are performed is important for example, if the first and second assignments in the swap sequence were interchanged, x and y would end up assigned to the same value. The input operation and the output operation share some of the same constraints.

2.5 PSEUDOCODE

Pseudocode is one of the tools that can be used to write a preliminary plan that can be developed into a computer program. **Pseudocode** is a generic way of describing an algorithm without use of any specific programming language syntax. It is, as the name suggests, *pseudo* code —it cannot be executed on a real computer, but it models and resembles real programming code, and is written at roughly the same level of detail.

Pseudocode, by nature, exists in various forms, although most borrow syntax from popular programming languages (like **C**, **Lisp**, or **FORTRAN**). Natural language is used whenever details are unimportant or distracting.

Computer science textbooks often use pseudocode in their examples so that all programmers can understand them, even if they do not all know the same programming languages. Since pseudocode style varies from author to author, there is usually an accompanying introduction explaining the syntax used.

In the algorithm design, the steps of the algorithm are written in free English text and, although brevity is desired, they may be as long as needed to describe the particular operation. The steps of an algorithm are said to be written in pseudocode.

Many languages, such as Pascal, have a syntax that is almost identical to pseudocode and hence make the transition from design to coding extremely easy.

The following section deal with the control structures (control constructs) Sequence, Selection and Iteration or Repetition.

CONTROL STRUCTURES OR LOGICAL STRUCTURES

The key to better algorithm design and thus to programming lies in limiting the control structure to only three constructs. These are illustrated below:

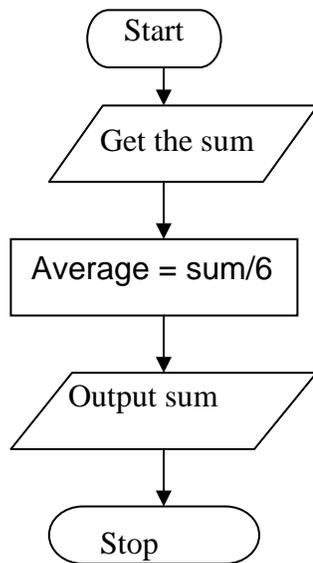
The sequence structure

The first type of control structures is called the sequence structure. This structure is the most elementary structure. The sequence structure is a case where the steps in an algorithm are constructed in such a way that, no condition step is required. The sequence structure is the logical equivalent of a straight line.

For example, suppose you are required to design an algorithm for finding the average of six numbers, and the sum of the numbers is given. The pseudocode will be as follows

Start
Get the sum
Average = sum / 6
Output the average
Stop

The corresponding flowchart will appear as follows:



Example 3: This is the pseudo-code required to input three numbers from the keyboard and output the result.

Use variables: sum, number1, number2, number3 of type integer
Accept number1, number2, number3
 $Sum = number1 + number2 + number3$
Print sum
End program

Example 4: The following pseudo-code describes an algorithm which will accept two numbers from the keyboard and calculate the sum and product displaying the answer on the monitor screen.

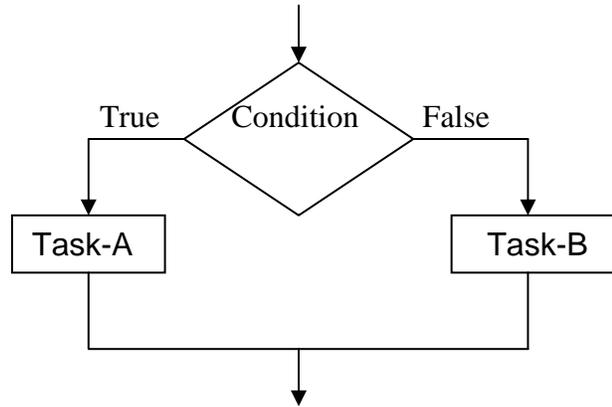
Use variables sum, product, number1, number2 of type real
display "Input two numbers"
accept number1, number2
 $sum = number1 + number2$
print "The sum is ", sum
 $product = number1 * number2$
print "The Product is ", product
end program

Decision Structure or Selection Structure

The decision structure or mostly commonly known as a selection structure, is case where in the algorithm, one has to make a choice of two alternatives by making decision depending on a given condition.

Selection structures are also called **case** selection structures when there are two or more alternatives to choose from.

This structure can be illustrated in a flowchart as follows:

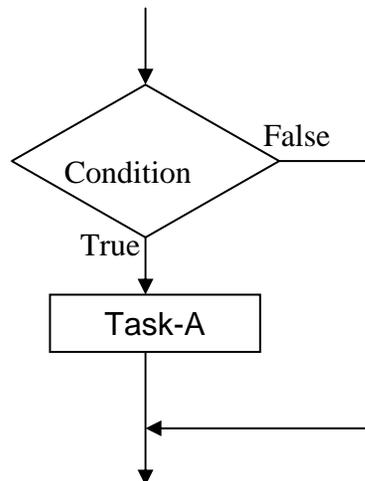


In pseudocode form we get

```
If condition is true  
Then do task A  
else  
Do Task-B
```

In this example, the condition is evaluated, if the condition is true Task-A is evaluated and if it is false, then Task-B is executed.

A variation of the construct of the above figure is shown below



From the above structure, we have the following

If condition is true then

Do Task-A

In this case, if condition is false, nothing happens. Otherwise Task-A is executed.

The selection requires the following

- Choose alternative actions as a result of testing a logical condition
- Produce code to test a sequence of logical tests

Making Choices

There are many occasions where a program is required to take alternative actions. For example, there are occasions where we need to take action according to the user choice. All computer languages provide a means of selection. Usually it is in the form of **If** statement and our pseudo-code is no exception to this.

We will use the if statement together with logical operators to test for true or false as shown below.

```
If a = b  
print "a = b"
```

The action is only taken when the test is true.

The logical operators used in our pseudo-code are

```
=      is equal to  
>     is greater than  
<     is less than  
>=    is greater than or equal  
<=    is less than or equal  
<>    is not equal to
```

Example 5: The following shows how the selection control structure is used in a program where a user chooses the options for multiplying the numbers or adding them or subtracting.

```
Use variables: choice, of the type character  
ans, number1, number2, of type integer  
display "choose one of the following"  
display "m for multiply"  
display "a for add"  
display "s for subtract"  
accept choice  
display "input two numbers you want to use"  
accept number1, number2
```

```
if choice = m then ans = number1 * number2
if choice = a then ans = number1 + number2
if choice = s then ans = number1 - number2
display ans
```

Compound Logical Operators

There are many occasions when we need to extend the conditions that are to be tested. Often there are conditions to be linked.

In everyday language we say things like *If I had the time and the money I would go on holiday*. The **and** means that *both conditions must be true* before we take an action. We might also say *I am happy to go to the theatre or the cinema*. The logical link this time is **or**. Conditions in **if** statements are linked in the same way. Conditions linked with **and** only result in an action when all conditions are true. For example, if $a > b$ and $a > c$ then display “a is the largest”. Conditions linked with an **or** lead to an action when either or both are true.

Example 6: The program is to input a examination mark and test it for the award of a grade. The mark is a whole number between 1 and 100. Grades are awarded according to the following criteria:

```
>= 80 Distinction
>= 60 Merit
>= 40 Pass
< 40 fail
```

The pseudo-code is

```
Use variables: mark of type integer
If mark >= 80 display “distinction”
If mark >= 60 and mark < 80 display “merit”
If mark >= 40 and mark < 60 display “pass”
If mark < 40 display “fail”
```

An **if** statement on its own is often not the best way of solving problems. A more elegant set of conditions can be created by adding an **else** statement to the if statement. The **else** statement is used to deal with situations as shown in the following examples.

Example 7: A person is paid at top for category 1 work otherwise pay is at normal rate.

```
If the work is category 1
    pay-rate is top
Else
    pay-rate is normal
```

The else statement provides a neat way of dealing with alternative condition. In pseudo-code we write

```
If work = cat1 then p-rate: = top
Else p-rate = normal
Or
If work = cat1 then
    p-rate: = top
Else
    p-rate = normal
```

The following example illustrate the use of if ... else statements in implementing double alternative conditions.

```
If salary < 50000 then
    Tax = 0
Else
    If salary > 50000 AND salary < 100000 then
        Tax = 50000 * 0.05
    Else
        Tax = 100000 * 0.30
```

The case statement

Repeating the if ... else statements a number of times can be somewhat confusing. An alternative method provided in a number of languages is to use a selector determined by the alternative conditions that are needed. In our pseudo-code, this will called a **case statement**.

Example 8: The following program segment outputs a message to the monitor screen describing the insurance available according to a category input by the user.

```
Use variables: category of type character
Display "input category"
Accept category
If category = U
    Display "insurance is not available"
Else
    If category = A then
        Display "insurance is double"
    Else
        If category = B then
            Display "insurance is normal"
        Else
            If category = M then
                Display "insurance is medically dependent"
            Else
                Display "entry invalid"
```

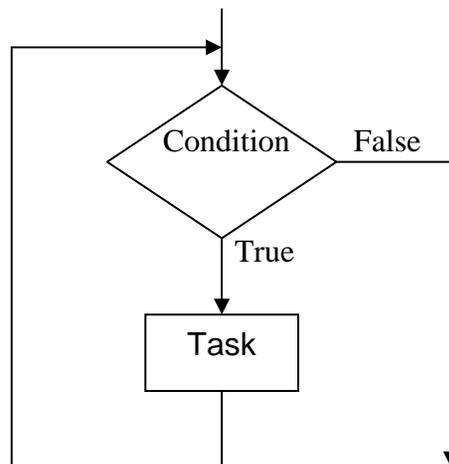
This can be expressed in a case statement as follows:

```
Use variables: category of type character
Display "input category"
Accept category
DO case of category
CASE category = U
    Display "insurance not available"
CASE category = A
    Display "insurance is double"
CASE category = B
    Display "insurance is normal"
CASE category = M
    Display "insurance is medically dependent"
OTHERWISE
    Display "entry is invalid"
ENDCASE
```

Instead of using the word *otherwise*, one can use *else*.

Repetition or Iteration Structure

A third structure causes the certain steps to be repeated.



The Repetition structure can be implemented using

- Repeat Until Loop
- The While Loop
- The For Loop

Any program instruction that repeats some statement or sequence of statements a number of times is called an **iteration** or a **loop**. The commands used to create iterations or loops

are all based on logical tests. There three constructs for iterations or loops in our pseudo-code.

The Repeat Until loop.

The syntax is

```
REPEAT  
    A statement or block of statements  
UNTIL a true condition
```

Example 9: A program segment repeatedly asks for entry of a number in the range 1 to 100 until a valid number is entered.

```
REPEAT  
DISPLAY "Enter a number between 1 and 100"  
ACCEPT number  
UNTIL number < 1 OR number > 100
```

Example 10. A survey has been carried out to discover the most popular sport. The results will be typed into the computer for analysis. Write a program to accomplish this.

```
REPEAT  
    DISPLAY "Type in the letter chosen or Q to finish"  
    DISPLAY "A: Athletics"  
    DISPLAY "S: Swimming"  
    DISPLAY "F: Football"  
    DISPLAY "B: Badminton"  
    DISPLAY "Enter data"  
    ACCEPT letter  
    If letter = 'A' then  
        Athletics = athletics + 1  
    If letter = 'S' then  
        Swimming = Swimming + 1  
    If letter = 'F' then  
        Football = Football + 1  
    If letter = 'B' then  
        Badminton = Badminton + 1  
    UNTIL letter = 'Q'  
    DISLAY "Athletics scored", athletics, "votes"  
    DISLAY "Swimming scored", swimming, "votes"  
    DISLAY "Football scored", football, "votes"  
    DISLAY "Badminton scored", Badminton, "votes"
```

The WHILE loop

The second type of iteration we will look at is the while iteration. This type of conditional loop tests for terminating condition at the beginning of the loop. In this case no action is performed at all if the first test causes the terminating condition to evaluate as false.

The syntax is

```
WHILE (a condition is true)
    A statement or block of statements
ENDWHILE
```

Example 11: A program segment to print out each character typed at a keyboard until the character 'q' is entered.

```
WHILE letter <> 'q'
    ACCEPT letter
    DISPLAY "The character you typed is", letter
ENDWHILE
```

Example 12: Write a program that will output the square root of any number input until the number input is zero.

In some cases, a variable has to be initialised before execution of the loop as shown in the following example.

```
Use variable: number of type real
DISPLAY "Type in a number or zero to stop"
ACCEPT number
WHILE number <> 0
    Square = number * number
    DISPLAY "The square of the number is", square
    DISPLAY "Type in a number or zero to stop"
    ACCEPT number
ENDWHILE
```

The FOR Loop

The third type of iteration, which we shall use when the number of iterations is known in advance, is a **for loop**. This, in its simplest form, uses an initialisation of the variable as a starting point, a stop condition depending on the value of the variable. The variable is incremented on each iteration until it reaches the required value.

The pseudo-code syntax will be:

```
FOR (starting state, stopping condition, increment)
    Statements
ENDFOR
```

Example 13.

```
FOR (n = 1, n <= 4, n + 1)
    DISPLAY "loop", n
ENDFOR
```

The fragment of code will produce the output

```
Loop 1
Loop 2
Loop 3
Loop 4
```

In the example, n is usually referred as the loop variable, or counting variable, or index of the loop. The loop variable can be used in any statement of the loop. The variable should not be assigned a new value within the loop, which may change the behaviour of the loop.

Example 14: Write a program to calculate the sum and average of a series of numbers. The pseudo-code solution is:

```
Use variables:  $n$ , count of the type integer
               Sum, number, average of the type real
DISPLAY "How many numbers do you want to input"
ACCEPT count
SUM = 0
FOR ( $n = 1, n \leq \text{count}, n + 1$ )
    DISPLAY "Input the number from your list"
    ACCEPT number
    SUM = sum + number
ENDFOR
Average = sum / count
DISPLAY "The sum of the numbers is ", sum
DISPLAY "Average of the numbers is ", average
```

Flowcharts have been used in this section to illustrate the nature of the three control structures. These three are the basic control structures out of which all programs are built. Beyond this, flowcharts serve the programmer in two distinct ways: as problem solving tools and as tools for documenting a program.

Example

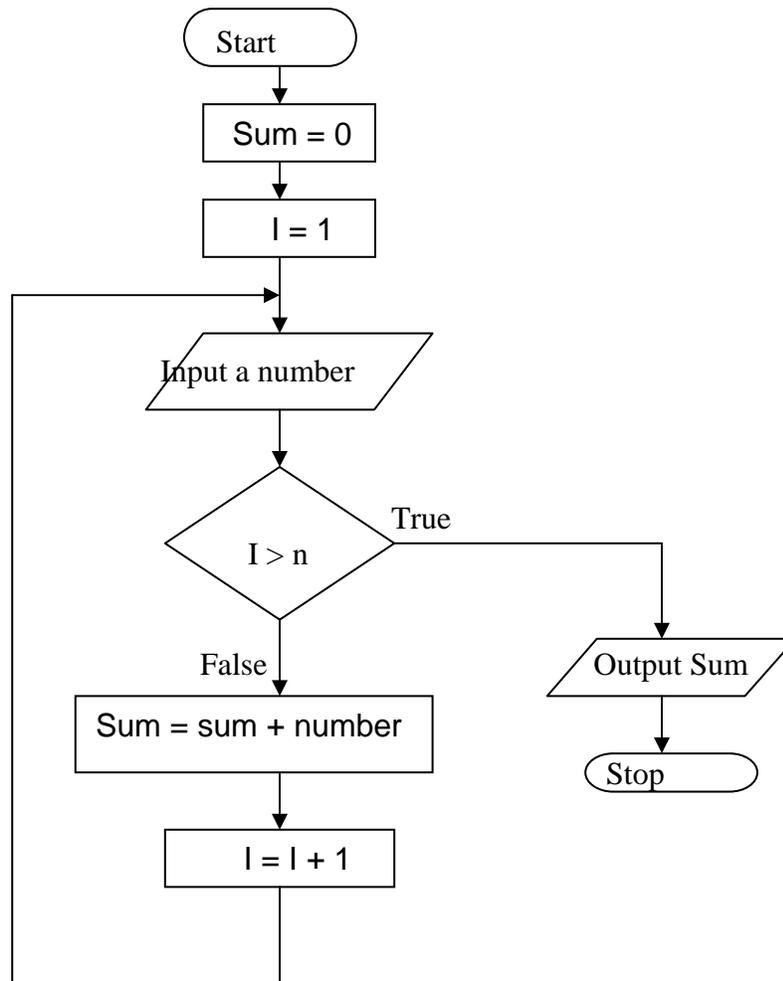
Design an algorithm and the corresponding flowchart for finding the sum of n numbers.

Pseudocode Program

```
Start
Sum = 0
Display "Input value n"
Input n
For( $I = 1, n, 5$ )
    Input a value
    Sum = sum + value
ENDFOR
Output sum
Stop
```

In this example, we have used I to allow us to count the numbers, which we get for the addition. We compare I with n to check whether we have exhausted the numbers or not in order to stop the computation of the sum (or to stop the iteration structure). In such a case, I is referred to as a **counter**.

The corresponding flowchart will be as follows:



Exercise

1. Design an algorithm and the corresponding flowchart for finding the sum of the numbers 2, 4, 6, 8, ..., n
2. Using flowcharts, write an algorithm to read 100 numbers and then display the sum.
3. Write an algorithm to read two numbers then display the largest.
4. Write an algorithm to read two numbers then display the smallest
5. Write an algorithm to read three numbers then display the largest.
6. Write an algorithm to read 100 numbers then display the largest.