PROBLEM SOLVING & PROGRAM DESIGN

Two phases involved in the design of any program:
(i) Problem Solving Phase
(ii) Implementation Phase

1. In the problem-solving phase the following steps are carried out:

- Define the problem
- Outline the solution
- Develop the outline into an algorithm
- Test the algorithm for correctness

2. The implementation phase comprises the following steps:

- Code the algorithm using a specific programming language
- Run the program on the computer
- Document and maintain the program

Structured Programming Concept

Structured programming techniques assist the programmer in writing effective error free programs.

The elements of structured of programming include:

- Top-down development
- Modular design.

The Structure Theorem:
It is possible to write any computer program by using only three (3) basic control structures, namely:

- Sequence
- Selection (if-then-else)
- Repetition (iteration, looping, DoWhile)

ALGORITHMS

An algorithm is a sequence of precise instructions for solving a problem in a finite amount of time.

Properties of an Algorithm:

- It must be precise and unambiguous
- It must give the correct solution in all cases
- It must eventually end.

Algorithms and Humans

Algorithms are not a natural way of stating a problem’s solution, because we do not normally state our plan of action.

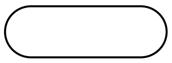
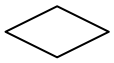
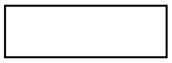
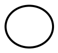
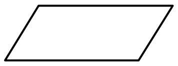
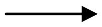
- We tend to *execute as we think* about the problem. Hence, there are inherent difficulties when writing an algorithm.
- We normally *tailor our plans of action to the particular problem at hand* and not to a general problem (i.e. a nearsighted approach to problem solving)
- We *usually do not write out our plan*, because we are usually unaware of the basic ideas we use to formulate the plan. We hardly think about it – we just do it.
- Computer programmers need to adopt a *scientific approach* to problem solving, i.e. writing algorithms that are comprehensive and precise.
- We need to be aware of the *assumptions* we make and of the initial conditions.
- Be careful not to *overlook a step* in the procedure just because it seems obvious.
- Remember, *machines* do not have judgment, intuition or common sense!

Developing an Algorithm

- Understand the problem (Do problem by hand. Note the steps)
- Devise a plan (look for familiarity and patterns)
- Carry out the plan (trace)
- Review the plan (refinement)

Understanding the Algorithm

Possibly the simplest and easiest method to understand the steps in an algorithm, is by using the **flowchart** method. This algorithm is composed of block symbols to represent each step in the solution process as well as the directed paths of each step. The most common block symbols are:

Symbol	**Representation**	**Symbol**	**Representation**
Start/Stop		Decision	
Process		Connector	
Input/Output		Flow Direction	

Problem Example
Find the average of a given set of numbers.

Solution Steps

1. Understanding the problem:

(i) Write down some numbers on paper and find the average manually, **noting each step carefully.**

e.g. Given a list say: 5, 3, 25, 0, 9

(ii) Count numbers | i.e. How many? 5

(iii) Add them up | i.e. 5 + 3 + 25 + 0 + 9 = 42

(iv) Divide result by numbers counted | i.e. 42/5 = 8.4

2. Devising a plan:
Make note of not what you did in steps (i) through (iv), but how you did it. In doing so, you will begin to develop the algorithm.

How do we count the numbers?

Starting at 0 i.e. set **COUNTER** to 0
Look at 1st number, add 1 to **COUNTER**
Look at 2nd number, add 1 to **COUNTER**
and so on, until you reach the end of the list

How do we add numbers?

Let **SUM** be the sum of numbers in list. Set **SUM** to 0
Look at 1st number, add number to **SUM**
Look at 2nd number, add number to **SUM**
and so on, until we reach end of list

How do we compute the average?

Let **AVG** be the average

$$\textbf{AVG} = \frac{\text{total sum of items}}{\text{number of items}} \quad \text{i.e.} \quad \frac{\textbf{SUM}}{\textbf{COUNTER}}$$

3. Identifying patterns, repetitions and familiar tasks.

Familiarity: Unknown number of items? i.e. **n** item

Patterns : look at each number in the list

Repetitions: Look at a number
Add number to sum
Add 1 to counter

4. Carrying out the plan

Check each step
Consider special cases
Check result
Check boundary conditions:
i.e. what if the list is empty?
Division by 0?
Are all numbers within the specified range?

In this example, no range is specified - No special cases.
Check result by tracing the algorithm with a list of numbers e.g. 7, 12, 1, 5,13.
If list is empty, we do not want to compute average.

Therefore, before calculating **AVG**, check if **COUNTER** = 0
i.e. If **COUNTER** = 0 then **AVG** = 0
else **AVG** = **SUM/COUNTER**

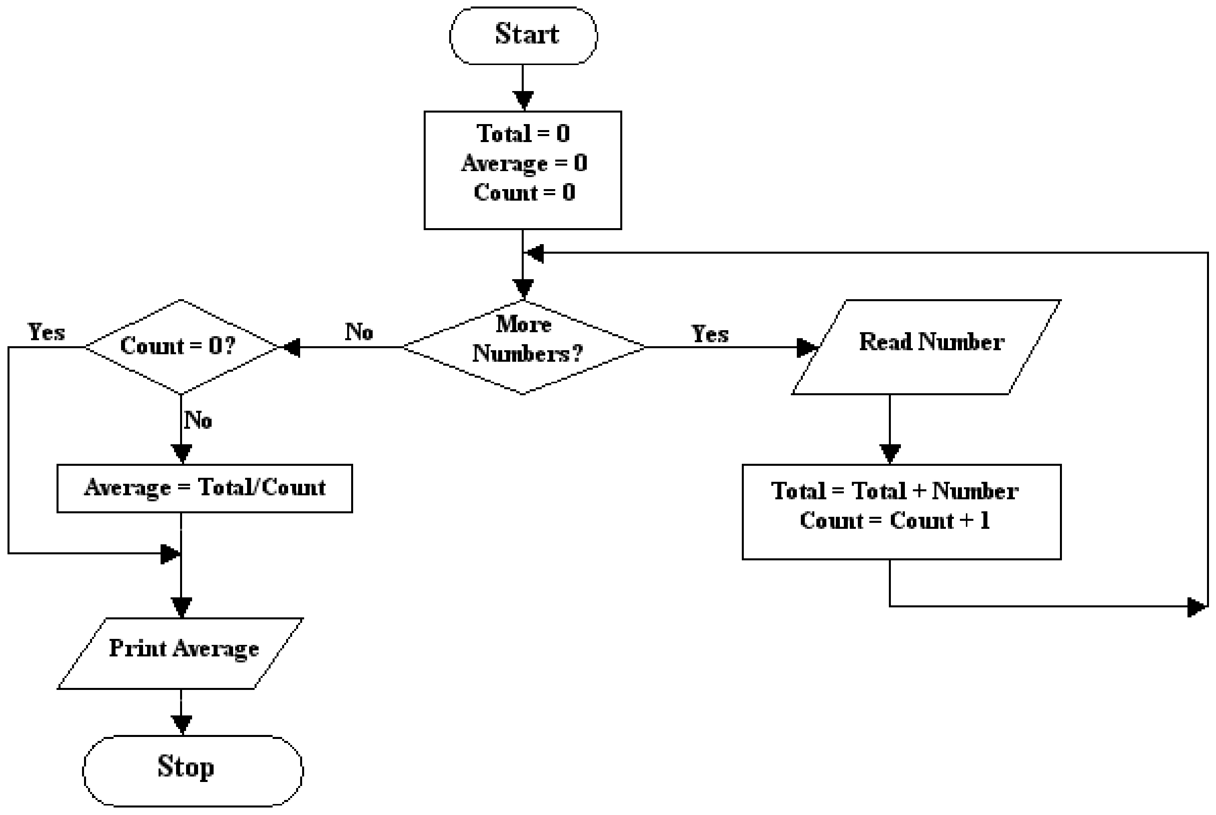
5. Review the plan:

- Can you derive the result differently?
- Can you make the solution more general?
- Can you use the solution or method for another problem?
  e.g. average temperature or average grades

Before we write out the algorithm in its proper form, let’s introduce the concept of a **VARIABLE.**

A variable is a symbolic name assigned to a memory cell which stores a particular value. e.g. COUNTER, SUM, AVG

A flowchart representation of the algorithm for the above problem can be as follows:



6. The Algorithmic Language
During development of an algorithm, the language gradually progresses from English towards a programming language notation. An intermediate notation called pseudocode is commonly used to express algorithms.

Algorithmic Structure

Every algorithm should have the following sections, in the stated order:

Header : Algorithm's name or title.

Declaration : A brief description of algorithm and variables.
i.e. a statement of the purpose.

Body : Sequence of steps

Terminator : End statement

How to write Pseudocode

An algorithm can be written in pseudocode using six (6) basic computer operations:

• **A computer can receive information.**
Typical pseudocode instructions to receive information are:

```
Read name
Get name
Read number1, number2
```

• **A computer can output (print) information.**
Typical pseudocode instructions are:

```
Print name
Write "The average is", avg
```

• **A computer can perform arithmetic operation**
Typical pseudocode instructions:

```
Add number to total, or
Total = Total + Number
Avg = sum/total
```

• **A computer can assign a value to a piece of data:**
e.g. to assign/give data an initial value:

```
Initialize total to zero
Set count to 0
To assign a computed value:
    Total = Price + Tax
```

• **A computer can compare two (2) pieces of information and select one of two actions.**

Typical pseudocode e.g.

```
IF number < 0 then
    add 1 to neg_number
ELSE
    add one to positive number
end-if
```

• **A computer can repeat a group of actions.**

Typical pseudocode e.g.

```
REPEAT until total = 50
    read number
    write number
    add 1 to total
end-repeat
```

OR

```
WHILE total < = 50 do:
    read number
    write number
end-while
```

Let’s review the plan and write out algorithm for the average problem in the specified format:

Algorithm **Average**

This algorithm reads a list of numbers and computes their average.

Let: **SUM** be the total of the numbers read
COUNTER be the number of items in the list
AVG be the average of all the numbers

```
Set SUM to 0,
Set COUNTER to 0. (i.e. initialize variables)

While there is data do:
    Read number
    COUNTER = COUNTER + 1
        (i.e. add 1 to COUNTER, storing result in COUNTER)
    SUM = SUM + number
        (i.e. add number to SUM, storing result in SUM)
end-while

if COUNTER = 0 then
    AVG = 0
else
    AVG = SUM/ COUNTER
end-if

Stop.
```

A Systematic Approach to Defining a Problem

Defining the problem is the first step towards a problem solution. A systematic approach to problem definition, leads to a good understanding of the problem. Here is a tried and tested method for defining (or specifying) any given problem:

Divide the problem into three (3) separate components:
- (a) input or source data provided
- (b) output or end result required
- (c) processing - a list of what actions are to be performed

EXAMPLE 1:
A program is required to read three (3) numbers, add them and print their total.
It is usually helpful to write down the three (3) components in a **defining diagram** as shown below:

INPUT	**PROCESSING**	**OUTPUT**
Read : Num1, Num2, Num3	Total = Num1 + Num2 + Num3	Print : Total

EXAMPLE 2:
Design a program to read in the max. and min. temperature on a particular day and calculate and print the average temp.

INPUT	**PROCESSING**	**OUTPUT**
Read : max_ temp, min_ temp	Calculate avg_temp	Print : avg_temp

Once the problem has been properly defined, making sure that no assumptions are made and every action required is stated in the Processing column, we can proceed to develop the algorithm. To do this, simply focus on the actions in the processing column, then write down the steps that specify how to perform each action and in what sequence.

Sample solution for EXAMPLE 1

```
Algorithm Add _Numbers
    Read Num1, Num2, Num3
    Total = Num1 + Num2 + Num3
    Print Total
Stop
```

Top-Down Design Approach (Modularization)

The modularization approach involves breaking a problem into a set of sub-problems, followed by breaking each sub-problem into a set of tasks, then breaking each task into a set of actions.

Problem 1

Add 23 and 35

- no further refinement is required.

Problem 2

Turn on a light bulb

Sub-problem 1: locate bulb (one task, one action)

Sub-problem 2: depress switch

Problem 3

Given a list of students' test scores, find the highest and lowest score and the average score.

Sub-problem 1: read students' scores

Sub-problem 2: find highest score

Sub-problem 3: find lowest score

Sub-problem 1 can be considered as one action and therefore needs no further refinement.

Sub-problems 2 and 3 however can be further divided into a group of actions.

Advantages of the Top-Down Design Method

• It is easier to comprehend the solution of a smaller and less complicated problem than to grasp the solution of a large and complex problem.

• It is easier to test segments of solutions, rather than the entire solution at once. This method allows one to test the solution of each sub-problem separately until the entire solution has been tested.

• It is often possible to simplify the logical steps of each sub-problem, so that when taken as a whole, the entire solution has less complex logic and hence easier to develop.

• A simplified solution takes less time to develop and will be more readable.

• The program will be easier to maintain.