# 1 Introduction

**Learning Outcome**

Algorithmic Representation
- Use appropriate techniques or tools such as pseudo-code and flowchart to show program flow
- Use standard flowchart symbols
- Use a combination of various control structures
- Use decision tables to explore the actions for combinations of different input conditions
- Use modular design to decompose a problem into smaller problems

Data Validation and Program Testing
- Explain the difference between data validation and data verification
- Understand data validation techniques such as range check, format check, length check, presence check, check digit
- Identify, explain and correct syntax, logic and runtime errors
- Design appropriate test cases using normal, abnormal and extreme data for testing and debugging programs

Programming is a creative experience to apply computational thinking to design the solution, and implement it in a programming language that the computer can understand. With the development of hardware, computers can help to solve numerous problems in an efficient way.

Computational thinking is a high level problem solving skills and it comprises four key techniques:

- Decomposition: breaking down a complex problem or system into smaller, more manageable parts
- Pattern Recognition: looking for similarities among and within problems
- Abstraction: focusing on the important information only, ignoring irrelevant details
- Algorithms: developing a step-by-step solution to the problem, or the rules to follow to solve the problem

Furthermore, we also need to learn communication skills to understand the problem and explain the solution clearly. Let's start with a simple example as an illustration of the complete procedure to solve a real-life problem.

We will follow the six steps below to solve the problem:
1. Define the problem
2. Analyze the problem
3. Design the solution
4. Implement the solution
5. Test the solution
6. Document the solution

**Example 1**
Write a program to indicate the grade of any given integer mark based on the following grading scheme:

| Mark Range | Grade |
|---|---|
| 0 – 44 | Fail |
| 45 – 100 | Pass |

## 1.1 The First Step: Define the Problem

Here, we write a statement of the objectives to be accomplished – the problem we are trying to solve. If necessary, we also specify the users of our program.

## 1.2 The Second Step: Analyze the Problem

Here, we identify the following:

(a) values that must be supplied from outside the program (input data)

(b) values that are given on the problem (constant data)

(c) values that must be produced as a result of solving the problem (output)

Effectively, the first two steps focus on ***what*** the program must achieve.

For Example 1, we have input data of a positive integer *Mark*, constant data of 0, 44, 45, 100, and the output data is *'Fail'* or '*Pass'*.

## 1.3 The Third Step: Design the Solution

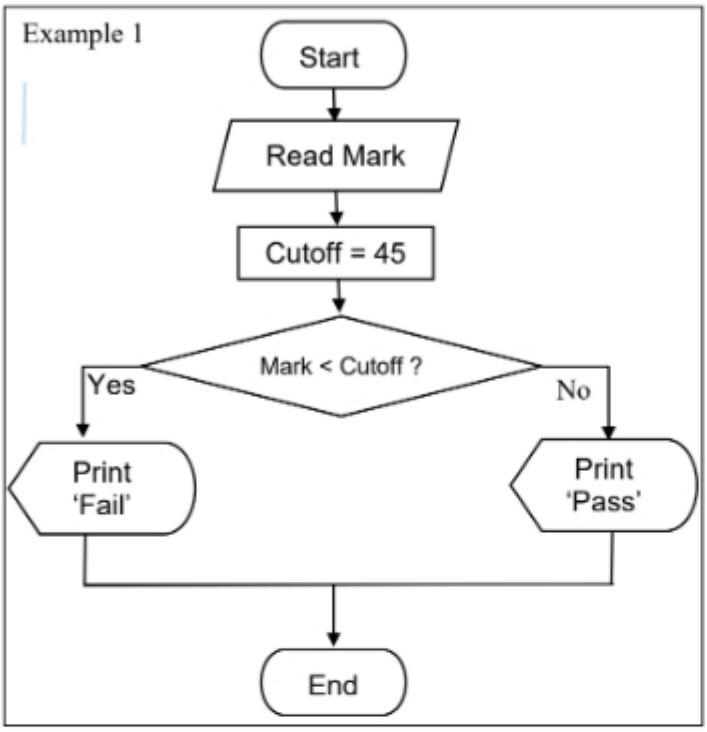
Here, we are concerned with the question of ***how*** to solve the problem in a systematic manner. We write an algorithm to take the given input data and constants and produce the desired output values.

**Algorithm** – A step-by-step process or sequence of instructions that can be used to solve a problem in a finite amount of time.

All steps will specify action taken (a verb) and the data that action has taken on. Generally, we can use either flowchart or pseudocode to describe an algorithm.

### 1.3.1 Algorithms in Flowcharts

A flowchart is a formalized graphic representation of a logic sequence, work process, organization chart or similar structure. The purpose of a flow chart is to provide people with a common language or reference point when dealing with a project or process.




**Common Flowchart symbols**

| Symbol | Symbol Name (alias) | Symbol Description |
|---|---|---|
| **Process / Operation Symbols** | | |
|  | Process | Show a Process or action step. This is the most common symbol in both process flowcharts and business process maps. |
| **Branching and Control of Flow Symbols** | | |
|  | Flow Line (Arrow, Connector) | Flow line connectors show the direction that the process flows. |
|  | Terminator (Terminal Point, Oval) | Terminators show the start and stop points in a process. When used as a Start symbol, terminators depict a *trigger action* that sets the process flow into motion. |
|  | Decision | Indicates a question or branch in the process flow. Typically, a Decision flowchart shape is used when there are 2 options (Yes/No etc.) |
| **Input and Output Symbols** | | |
|  | Data (I/O) | The Data flowchart shape indicates inputs to and outputs from a process. As such, the shape is more often referred to as an I/O shape than a Data shape. |
|  | Display | Indicates a process step where information is displayed to a person (e.g., PC user, machine operator). |

### 1.3.2 Algorithms in Pseudocode

Pseudocode – essentially English with some defined rules of structure and some keywords similar to program code.

Main advantage – allows us to focus on the logical details of our program solution, without getting bogged down with the syntax rules of the programming languages.

Primary disadvantage – has no single form. Its format varies depending on the author. In this view, we shall standardize a format for pseudocode.

**Pseudocode Guidelines**

We shall use the following keywords:

| | |
|---|---|
| For Start: | Begin |
| For Finish: | End |
| For Input: | Read |
| For Output: | Print |
| For Selection: | If, then, else, end-if |
| For multi-way selection: | Case-where, otherwise, end-case |
| For pre-test repetition: | While, end-while |
| For post-test repetition: | Repeat Until |

Note the following guidelines as well:

1. Structural elements come in pairs e.g. Begin-End, If-Endif
2. Indenting is used to show structure in the algorithm.

Example 1 Pseudocode:

```
BEGIN

        Read Mark

        Cutoff = 45

        if Mark < Cutoff
                PRINT 'Fail'
        else
                PRINT 'Pass'
        endIf

END
```

**Principles of a Good Algorithm**

A good algorithm is one that satisfies the following:

(a) It is easily understood by another person.
(b) It solves the problem efficiently and cleanly.

Some guidelines for writing a good algorithm include:

(a) Ask yourself: "Can it be implemented by someone else?"
This means that in your algorithm, you should
- Use unambiguous expressions and be concise and precise.
- State all necessary information, including assumptions.

(b) Use only the three basic flow of execution constructs (we will learn in subsequent chapters):
- Sequential
- Selection
- Repetition

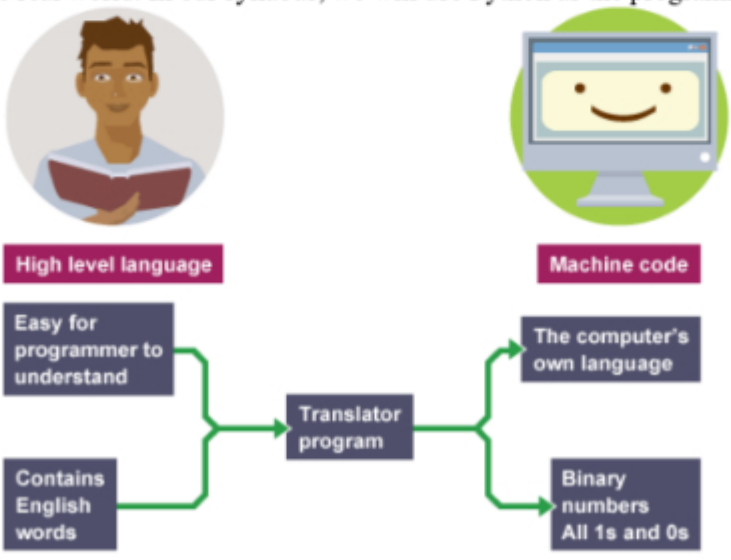
(c) Sub-divide the problem solution into modular (self-contained and logical), easy-to-manage chunks, through step-wise refinements.
(d) Have only one start and one termination point.
(e) Use indentation to show structure and improve readability.

## 1.4 The Fourth Step: Implement the Solution

Here, we ***write*** (code) the program using an appropriate programming language, and following its rules (syntax) exactly.

Computers are built to understand machine language only, which are long strings of ones and zeros. Human languages are error-prone for computers. As a tradeoff for both sides, we use high level languages to communicate with computer and give instructions.

Programming languages are well developed and still developing to suit our needs for various problems in the real world. In our syllabus, we will use Python as the programming language.



## 1.5 The Fifth Step: Test the Solution

Here, we check that the program does as required by running ***test plans*** with known solutions. We ***debug*** the program if it fails to meet our expectations. We also use ***data validation*** and ***verification*** techniques to prevent the wrong input data.

**Test plan** – set of possible inputs along with their associated expected output.

Design test plans to check a computer program for correctness during and after its development. As you design the solution, you should complete a fully-developed test plan. If created later, likely that you may miss out some boundary cases or "tricky" input. A test plan must contain enough sets of possible inputs to thoroughly test the program and its algorithm. We can consider the following types of input test data:

- Typical, valid values (normal data)
- Boundary values (extreme data)
- Invalid values (abnormal data)

The test plan for Example 1:

| **Possible inputs** | | **Expected output** | **Reason for test item** |
|---|---|---|---|
| **Item #** | **Mark** | | |
| 1 | 30 | Fail | Normal data |
| 2 | 70 | Pass | Normal data |
| 3 | 0 | Fail | Extreme data |
| 4 | 44 | Fail | Extreme data |
| 5 | 45 | Pass | Extreme data |
| 6 | 100 | Pass | Extreme data |
| 7 | −1 | Error: mark cannot be negative | Abnormal data |
| 8 | 7.5 | Error: mark must be integer | Abnormal data |

Note that the algorithm designed cannot give the error message for abnormal data in test plan 7 and 8. So there is a need to refine the algorithm to take into account these user input scenarios.

**Data Validation** – Process to ensure the correct input data

| Type | How it works | Example |
|---|---|---|
| Range Check | Check if the data value is within certain range | In the above Example 1, the mark must be from 0 to 100 |
| Format Check | Check if the data is in the right format | Date in the format dd/mm/yyyy |
| Length Check | Check if the length of the data is correct | Length of password |
| Presence Check | Check if data is entered into a field | Username cannot be left blank |
| Check Digit* | The last one or two digits in data are used to check whether the other digits are correct | ISBN of a book |

**We will learn check digit with programming practice questions in subsequent chapters.*

**Data Verification** – Process to ensure the input data matches the original resource

Even when data is entered in the correct format/range/length, the user may still make a mistake when inputting data. For example, we are often being asked to enter the password twice in order to change it, this double entry of data is a form of data verification.

**Debug** – detect, locate and remove all errors in a computer program.

Error-free is ideal but most of the time, we need to learn to debug our program to find the errors.

1. *Syntax error* occurs when the syntax rules in the programming language is not adhered to (e.g. error in the syntax of a sequence of characters of tokens). Program will not compile until all syntax errors are corrected.
2. *Logical error* is caused by wrong program design. This could happen when the algorithm for solving the program in incorrect. The program may be compiled and executed but does not give the expected output.
3. *Run-time error* is detected at run-time e.g. stack overflow, division by 0, open a file that does not exist.

Techniques for debugging:

- Specifying a point in the source code and have the program **execute up to a point** (break point) to evaluate different portions of the code.
- **Displaying** the procedure stack to list the sequence of the procedures called during the executing of the program (trace facility)
- **Marking out** as comments and procedures or program statements to be excluded in the testing of the intended portion of the code.

## 1.6 The Sixth Step: Document the Solution

Here, we ***document*** the solution by writing a description of its purpose and process. Such a description would allow other programmers to understand the solution, and make changes easily when necessary.

Example 2

(i) Problem: How to make a sandwich?
Objective: To make a sandwich.

(ii) Input: Mayonnaise, bread, cheese, lettuce, ham.
Output: Sandwich

(iii) Solution:
(a) Gather all given items needed for the sandwich
(b) Apply desired condiments (mayonnaise, mustard, etc.) to 2 slices of bread.
(c) Apply main sandwich ingredients (ham, cheese, lettuce, etc) to one of the bread slices.
(d) Finish by placing the second slice of bread on top of the sandwich ingredients, effectively "sandwiching" the ingredients between the two slices of bread.
(e) Clean up the mess.

Of course, you might make sandwiches differently, that's okay – various programmers solve problems differently, too.

## 1.7 Structured Programming

Structured programming is a methodology for developing programs. It helps us to manage complexity and to develop programs which are easy to read, test and maintain. It emphasizes the use of modules in program design and implementation.

### Top-Down Design

Top-down design is a method of designing a solution to a problem by repeatedly breaking a problem down into simpler problems until the problems can be solved easily.

These sub-problems are solved separately and are then liked together by means of simple control structures. It is a useful technique for deriving solutions to large problems using the divide and conquer or top-down approach to problem solving. Top-down design is sometimes known as step-wise refinement.

### Modularity

Modularity is the feature of designing problem solution as a collection of well-defined separate modules, each of which serves a specific purpose and has specific connections with the main program.

A module is a complete part-program that is used from within the main program.

Advantages

- Modules can be kept in a library and re-used in other solutions
- Many programmers can work in the same problem as each can be given different modules to solve; Each module can be coded and test separately
- Easier to debug as modules are small
- Easier to maintain and modify as modules can be removed/added easily
- A large project becomes easier to monitor and control.

Disadvantages

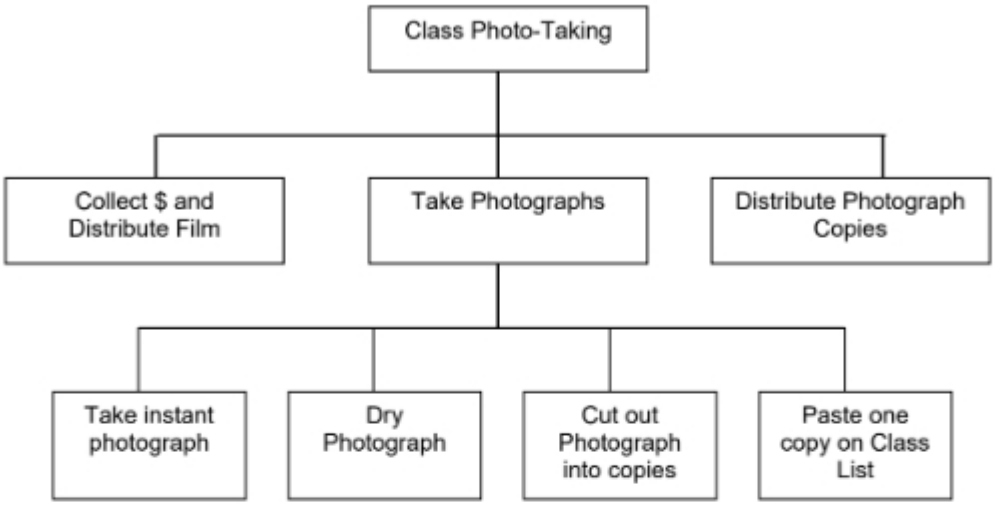
- Time needed to partition problem into modules.
- Larger memory space needed.
- More files need to be managed (lost of files)
- May not be able to see all code or refer to all the documentation.
- It is possible to over modularize.

### Structure Chart

A structure chart is a pictorial description of a top-down design. The chart consists of boxes – one for each module of the solution at each level in the hierarchical breakdown.

The top-box is the main module or main program. Boxes in the second level are called from the main module, each connected to their respective sub-tasks.

Example 3 To take individual pictures for a class




Tasks at level 1 and 2 are repeatedly executed until no more students, and the problem is solved.

**Tutorial 1A**

1.

| Mark Range | Grade |
|---|---|
| 70 – 100 | A |
| 60 – 69 | B |
| 55 – 59 | C |
| 50 – 54 | D |
| 45 – 49 | E |
| 40 – 44 | S |
| 0 – 39 | U |

The grading scheme in Example 1 is modified as above. Design the pseudocode.

2. John is requesting a program that computes his income tax. The rule for calculation is

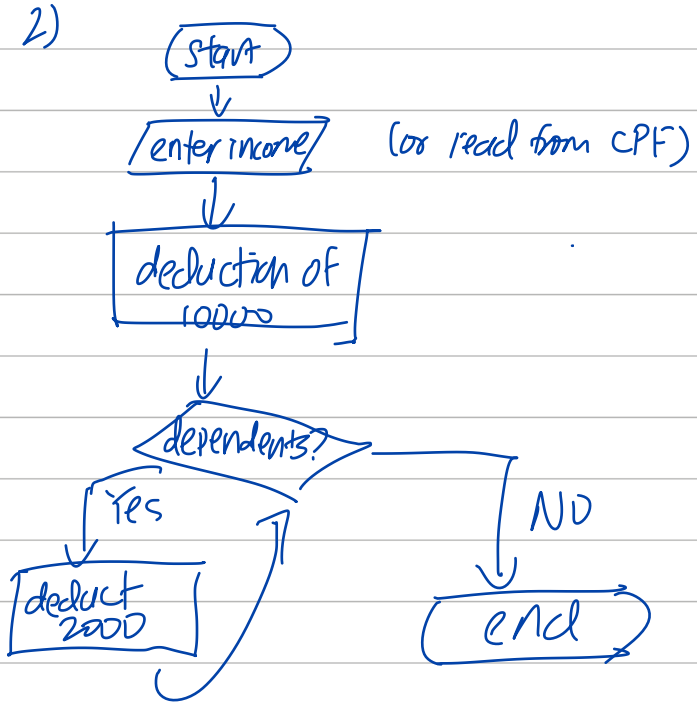
- Taxpayers must enter their gross income.
- Taxpayers are allowed $10,000 standard deduction.
- For each dependent, taxpayer is allowed additional $2000 deduction.
- Income tax is charged at a flat rate of 20% based on taxable income after all deductions.

Design the flowchart, pseudocode, and test plan.

1)

```
Begin
  Read mark
  if mark >= 70
      print 'A'
  else if mark >= 60
      print 'B'
  else if mark >= 55
      print 'C'
  else if mark >= 50
      print 'D'
  else if mark >= 45
      print 'E'
          else if mark >= 40
              print 'S'
              else
                  print 'U'
```

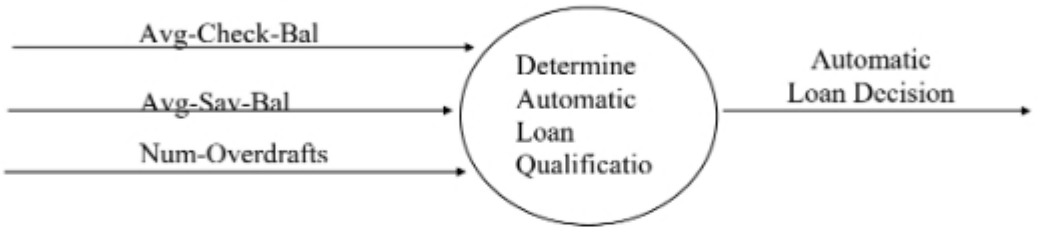
2)



## 1.8 Decision Tables

Decision tables provide an alternative for specifying conditions or processing alternatives. All possible conditions and outcomes are listed in a two-dimensional table that shows the outcome results from each combination of conditions.

The customer is approved if he has maintained monthly checking account balance of at least $1000 for each of the last three months and has averaged no more than two overdrafts per month. Customers meeting only one of these conditions but maintaining an average savings account balance of at least $500 for each of the last three months receive conditional approval with an automatic loan limit of $500.




- 3 conditions can be identified
  i. checking account balance: value >= 1000
  ii. number of overdrafts: <= 2
  iii. average savings balance: values >= 500

- 3 possible outcomes
  i. approval (no limit)
  ii. conditional approval ($500 limit)
  iii. rejection

Table is divided into four quadrants

- Upper left quadrant – one row for each condition
- Lower left quadrant – one row for each outcome
- Upper right quadrant – values associated with each of the conditions. You may use Y/N or T/F.
  *We may count the total number of columns needed by multiplying the number of values each condition can assume. For the above example, 3 conditions each can assume 2 values. So total of 2 x 2 x 2 = 8 possible combinations or columns.*
- Lower right quadrant – x mark is used to designate each outcome.

| AVG-CHK-BAL >= 1000 | Y | Y | Y | Y | N | N | N | N |
|---|---|---|---|---|---|---|---|---|
| NUM-OVERDRAFTS <= 2 | Y | Y | N | N | Y | Y | N | N |
| AVG-SAV-BAL >= 500 | Y | N | Y | N | Y | N | Y | N |
| APPROVE | X | X | | | | | | |
| COND-APPROVE | | | X | | X | | | |
| REJECT | | | | X | | X | X | X |

Upon constructing the table, verify the policy and simplify the table.

To verify the policy, the completed decision table is reviewed with the end-users. Resolve any rules for which the actions are not specific. Verify that rules you think are impossible or cannot in actuality occur. Resolve apparent contradictions, such as one rule with two contradictory actions. Finally, verify that each rule's actions are correct.

To simplify the table, we combine rules with indifferent conditions. An indifferent condition is one whose values do not affect the decision and always result in the same action.

To determine indifferent conditions, first look for rules with exactly the same actions. From these, find those whose condition values are the same except for one and only one condition (called the indifferent condition). This latter set of rules has the potential for being collapsed into a single rule with the indifferent condition value replaced with a dash. Note that all possible values of the indifferent condition must be present among the rules to be combined before they can be collapsed.

| AVG-CHK-BAL >= 1000 | Y | Y | - | N | N | N |
|---|---|---|---|---|---|---|
| NUM-OVERDRAFTS <= 2 | Y | N | N | Y | Y | N |
| AVG-SAV-BAL >= 500 | - | Y | N | Y | N | Y |
| APPROVE | X | | | | | |
| COND-APPROVE | | X | | X | | |
| REJECT | | | X | | X | X |

(column 1 & 2, 4 & 8 have been combined)

**Tutorial 1B**

1. The company has a factory in China which manufactures different parts of a vehicle for its own consumption.

   A part will be deemed fit for use if it passes the following three tests:
   - All dimensions are correct;
   - Strength tests are passed;
   - Paint tests are passed.

   If the first test is passed but exactly one of the other two fails, the part is sent for repair. Otherwise the part is rejected.

   (i) Create a decision table showing all the possible outcomes and results.
   (ii) Simplify your decision table by removing redundancies.

2. The rules that are used when deciding whether to offer insurance to customers and whether to offer discounts are as follows:

   - If the customer has been refused insurance by another company and their car is over 10 years old then insurance is refused.
   - If the customer has been refused insurance by another company and their car is not more than 10 years old then insurance without any discount is available.
   - If the customer has not been refused insurance by another company and their car is over 10 years old then insurance without any discount is available.
   - If the customer has not been refused insurance by another company and their car is not over 10 years old and they have made not more than three claims previously then insurance with a discount is available.

   (i) Create a decision table showing all the possible outcomes and results.
   (ii) Simplify your decision table by removing redundancies.

1)

| Test | Dimension | Paint/strength | |
|---|---|---|---|
| 1 | ok | Both pass | OK |
| | | One pass | Repair |
| | | No pass | Reject |
| 2 | Fail | any result | reject |

2)

| Test | Refused insurance? | Age of car | No. of claims |
|---|---|---|---|
| 1 | Yes | >10 | ignore |
| | | <10 | |
| 2 | NO | >10 | |
| | | <10 | <3 |

rejected

no discount

discount