CHAPTER
3

ALGORITHMIC PROBLEM SOLVING

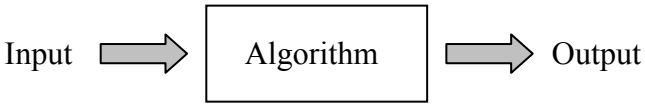
In this chapter you will learn about

- What an algorithm is.
- The relationship between data and algorithm.
- The characteristics of an algorithm.
- Using pseudo-codes and flowcharts to represent algorithms.

3.1 Algorithms

In Chapter 2, we expounded the working of problem solving from a general perspective. In computing, we focus on the type of problems categorically known as algorithmic problems, where their solutions are expressible in the form of algorithms.

An *algorithm* [1] is a well-defined computational procedure consisting of *a set of instructions*, that takes some value or set of values, as *input*, and produces some value or set of values, as *output*. In other word, an algorithm is a procedure that accepts data, manipulate them following the prescribed steps, so as to eventually fill the required unknown with the desired value(s).

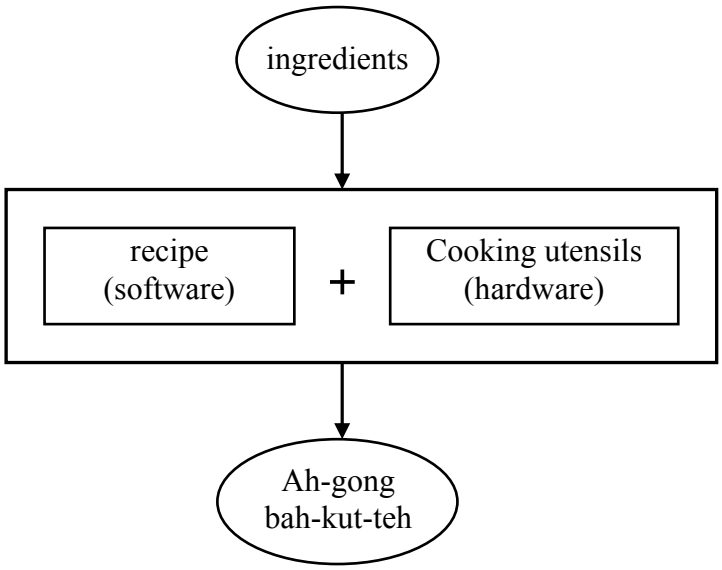


Tersely put, an algorithm, a jargon of computer specialists, is simply a procedure. People of different professions have their own form of procedure in their line of work, and they call it different names. A cook, for instance, follows a procedure commonly known as a recipe that converts the ingredients (input) into some culinary dish (output), after a certain number of steps.

An algorithm, whose characteristics will be discussed later, is a form that embeds the complete logic of the solution. Its formal written version is called a *program*, or *code*. Thus, algorithmic problem solving actually comes in two phases: derivation of an algorithm that solves the problem, and conversion of the algorithm into code. The latter, usually known as coding, is comparatively easier, since the logic is already present – it is just a matter of ensuring that the syntax rules of the programming language are adhered to. The first phase is what that stumbles most people, for two main reasons. Firstly, it challenges the mental faculties to search for the right solution, and secondly, it requires the ability to articulate the solution concisely into step-by-step instructions, a skill that is acquired only through lots of practice. Many people are able to make claims like "oh yes, I know how to solve it", but fall short when it comes to transferring the solution in their head onto paper.

Algorithms and their alter ego, programs, are the *software*. The machine that runs the programs is the *hardware*. Referring to the cook in our analogy again, his view can be depicted as follows:

[1] The term 'algorithm' was derived from the name of Mohammed al-Khowarizmi, a Persian mathematician in the ninth century. Al-Khowarizmi → Algorismus (in Latin) → Algorithm.



The first documented algorithm is the famous *Euclidean algorithm* written by the Greek mathematician Euclid in 300 B.C. in his Book VII of the *Elements*. It is rumoured that King Ptolemy, having looked through the *Elements*, hopefully asked Euclid if there were not a shorter way to geometry, to which Euclid severely answered: "In geometry there is no royal road!"

The modern Euclidean algorithm to compute gcd (greatest common divisor) of two integers, is often presented as followed.

1. Let A and B be integers with $A > B \geq 0$.
2. If $B = 0$, then the gcd is A and the algorithm ends.
3. Otherwise, find q and r such that

   $$A = qB + r \text{ where } 0 \leq r < B$$

   Note that we have $0 \leq r < B < A$ and $\gcd(A,B) = \gcd(B,r)$.
   Replace A by B, and B by r. Go to step 2.

Walk through this algorithm with some sets of values.

3.2 Data Types And Data Structures

In algorithmic problem solving, we deal with objects. Objects are data manipulated by the algorithm. To a cook, the objects are the various types of vegetables, meat and sauce. In algorithms, the data are numbers, words, lists, files, and so on. In solving a geometry problem, the data can be the length of a rectangle, the area of a circle, etc. Algorithm provides the logic; data provide the values. They go hand in hand. Hence, we have this great truth:

Program = Algorithm + Data Structures

Data structures refer to the types of data used and how the data are organised in the program. Data come in different forms and types. Most programming languages provides simple data types such as integers, real numbers and characters, and more complex data structures such as arrays, records and files which are collections of data.

Because algorithm manipulates data, we need to store the data objects into variables, and give these variables names for reference. For example, in mathematics, we call the area of a circle A, and express A in terms of the radius r. (In programming, we would use more telling variable names such as *area* and *radius* instead of A and r in general, for the sake of readability.) When the program is run, each variable occupies some memory location(s), whose size depends on the data type of the variable, to hold its value.

3.3 Characteristics Of An Algorithm

What makes an algorithm an algorithm? There are four essential properties of an algorithm.

1. Each step of an algorithm must be **exact**.

   This goes without saying. An algorithm must be precisely and unambiguously described, so that there remains no uncertainty. An instruction that says "shuffle the deck of card" may make sense to some of us, but the machine will not have a clue on how to execute it, unless the detail steps are described. An instruction that says "lift the restriction" will cause much puzzlement even to the human readers.

2. An algorithm must **terminate**.

   The ultimate purpose of an algorithm is to solve a problem. If the program does not stop when executed, we will not be able to get any result from it. Therefore, an algorithm must contain a finite number of steps in its execution. Note that an algorithm that merely contains a finite number of steps may not terminate during execution, due to the presence of 'infinite loop'.

3. An algorithm must be **effective**.

   Again, this goes without saying. An algorithm must provide the correct answer to the problem.

4. An algorithm must be **general**.

   This means that it must solve every instance of the problem. For example, a program that computes the area of a rectangle should work on all possible dimensions of the rectangle, within the limits of the programming language and the machine.

An algorithm should also emphasise on the *whats*, and not the *hows*, leaving the details for the program version. However, this point is more apparent in more complicated algorithms at advanced level, which we are unlikely to encounter yet.

3.4 Pseudo-Codes And Flowcharts

We usually present algorithms in the form of some *pseudo-code*, which is normally a mixture of English statements, some mathematical notations, and selected keywords from a programming language. There is no standard convention for writing pseudo-code; each author may have his own style, as long as clarity is ensured.

Below are two versions of the same algorithm, one is written mainly in English, and the other in pseudo-code. The problem concerned is to find the minimum, maximum, and average of a list of numbers. Make a comparison.

Algorithm version 1:

First, you initialise *sum* to zero, *min* to a very big number, and *max* to a very small number.

Then, you enter the numbers, one by one.

For each number that you have entered, assign it to *num* and add it to the *sum*.

At the same time, you compare *num* with *min*, if *num* is smaller than *min*, let *min* be *num* instead.

Similarly, you compare *num* with *max*, if *num* is larger than *max*, let *max* be *num* instead.

After all the numbers have been entered, you divide *sum* by the numbers of items entered, and let *ave* be this result.

End of algorithm.

Algorithm version 2:

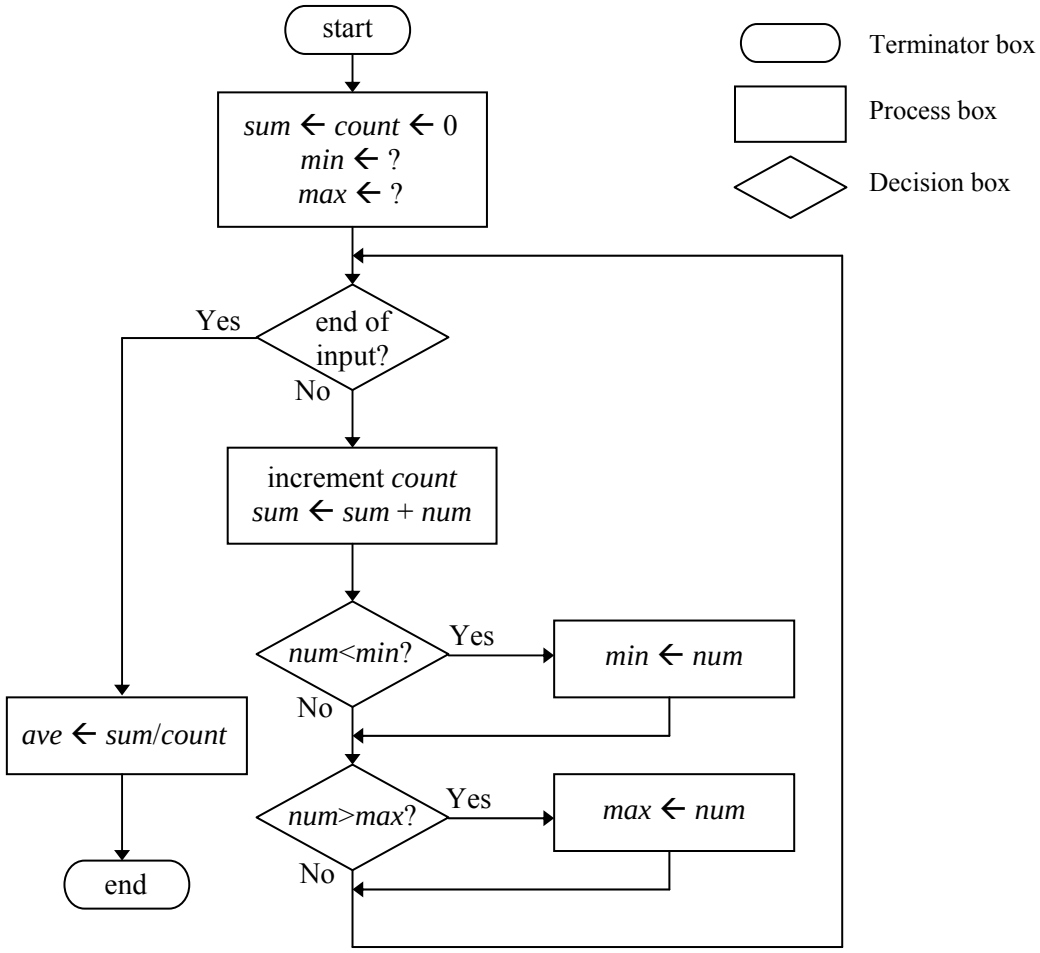
```
sum ← count ← 0        { sum = sum of numbers;
                         count = how many numbers are entered? }
min ← ?                { min to hold the smallest value eventually }
max ← ?                { max to hold the largest value eventually }
for each num entered,
      increment count
      sum ← sum + num
      if num < min then min ← num
      if num > max then max ← num

ave ← sum/count
```

Note the use of indentation and symbols in the second version. What should *min* and *max* be initialised with?

Algorithms may also be represented by diagrams. One popular diagrammatic method is the *flowchart*, which consists of terminator boxes, process boxes, and decision boxes, with flows of logic indicated by arrows.

The flowchart below depicts the same logic as the algorithms above.



Whether you use the pseudo-code or the flowchart to represent your algorithm, remember to walk through it with some sets of data to check that the algorithm works.

3.5 Control Structures

The pseudo-code and flowchart in the previous section illustrate the three types of *control structures*. They are:

1. Sequence
2. Branching (Selection)
3. Loop (Repetition)

These three control structures are sufficient for all purposes. The sequence is exemplified by sequence of statements place one after the other – the one above or before another gets executed first. In flowcharts, sequence of statements is usually contained in the rectangular process box.

The *branch* refers to a binary decision based on some condition. If the condition is true, one of the two branches is explored; if the condition is false, the other alternative is taken. This is usually represented by the 'if-then' construct in pseudo-codes and programs. In flowcharts, this is represented by the diamond-shaped decision box. This structure is also known as the *selection* structure.

The *loop* allows a statement or a sequence of statements to be repeatedly executed based on some loop condition. It is represented by the 'while' and 'for' constructs in most programming languages, for unbounded loops and bounded loops respectively. (Unbounded loops refer to those whose number of iterations depends on the eventuality that the termination condition is satisfied; bounded loops refer to those whose number of iterations is known before-hand.) In the flowcharts, a back arrow hints the presence of a loop. A trip around the loop is known as iteration. You must ensure that the condition for the termination of the looping must be satisfied after some finite number of iterations, otherwise it ends up as an infinite loop, a common mistake made by inexperienced programmers. The loop is also known as the *repetition* structure.

Combining the use of these control structures, for example, a loop within a loop (nested loops), a branch within another branch (nested if), a branch within a loop, a loop within a branch, and so forth, is not uncommon. Complex algorithms may have more complicated logic structure and deep level of nesting, in which case it is best to demarcate parts of the algorithm as separate smaller *modules*. Beginners must train themselves to be proficient in using and combining control structures appropriately, and go through the trouble of tracing through the algorithm before they convert it into code.

3.6 Summary

An algorithm is a set of instructions, and an algorithmic problem lends itself to a solution expressible in algorithmic form. Algorithms manipulate data, which are represented as variables of the appropriate data types in programs. Data structures are collections of data.

The characteristics of algorithms are presented in this chapter, so are the two forms of representation for algorithms, namely, pseudo-codes and flowcharts. The three control structures: sequence, branch, and loop, which provide the flow of control in an algorithm, are introduced.

Exercises

In the exercises below, you are asked to find the answers by hand, and jot down the steps involved for the generalized versions. You need not write programs for them yet – you will be doing that later – you need only to discover the steps and put them down into an algorithm.

1. Our coins come in denominations of 1 cent, 5 cents, 10 cents, 20 cents, 50 cents and 1 dollar. Assuming that there are unlimited number of coins, how would you devise a ***coin-change algorithm*** to compute the **minimum** number of coins required to make up a particular amount? For example, for 46 cents you need 4 coins: two 20-cent coins, one 5-cent coin, and one 1-cent coin.

2. How would you sort 10 numbers in increasing order? Do you know any of the basic sorting techniques?

3. A word is a **palindrome** if it reads the same when the order of its characters is reversed. For instance, "123321", "RADAR" and "step on no pets" are palindromes. Derive an algorithm to determine if a word is a palindrome.

4. Two words are **anagrams** of each other if they differ only by the order of their characters. For example, "HATE" and "HEAT", "NOW" and "WON", "reset" and "steer" are anagrams, but "sell" and "less" are not. Write an algorithm to determine if two given words are anagrams.

5. How do you test for **primality**? That is, given a positive integer, how do you determine that it is a prime number?

6. In a **mastermind** game, the code maker hides a secret code consisting of 4 digits, and the code breaker is to provide 4-digit guess codes until he gets the code right. The code maker gives feedback in this manner, a *sink* is awarded to a perfect match of a digit plus its position, and a *hit* refers to a right match of digit but in the wrong position. How do you devise an algorithm to provide the correct feedback? First, work on the special case that no digit in the code should be repeated. Then work on the general case where digits can be duplicated.