CMPT 110 Study Guide

Unit 1: What is programming about?

Adapted from Tony Dixon’s Distance Education Version of CMPT 110 at SFU

Objectives

- To understand the importance of knowing how to describe how to solve a problem.
- To learn what is required of a description for solving a problem that qualifies it as an algorithm.
- To become familiar with three important ways of documenting a program: flowcharting, hierarchy diagrams, and pseudo-coding.
- To become familiar with what you will be expected to submit to complete each assignment.

To Do

1. All this material is based on PPTs given in lecture.
2. Familiarize yourself with the use of the Notepad software, a utility available with all Windows environments. It can be used to create text files of data for testing your programs later in this course.
3. Read about using flowcharts, hierarchy charts, and pseudocode.
What is Programming About?

Before you begin to write programs, it is important to understand what this activity is about and what is expected of you as a developing programmer. In many disciplines and even in your own day-to-day activities, you are required to engage in the task of "problem solving." Historically, the term was associated only with mathematical disciplines because they recognized and treated the activity not just as a skill to be mastered but also as topic of research itself.

Today, this often overworked but still descriptive term is identified as an important skill in more and more disciplines. It describes the task of achieving a desired result as a consequence of finding and performing an appropriate sequence of steps. It frequently involves the use of information about circumstances that may affect what decisions are made as the steps are performed. Since we all employ problem solving in our daily lives, we all have at least an informal understanding of the meaning of the term even if we do not always recognize that we are actually engaged in it.

Of course, we don’t always know how to solve a problem! In such circumstances, we usually seek help, possibly by having someone else solve the problem for us. However, while this is likely to be the most expedient solution, it is not usually the best way to proceed, particularly if we are likely to encounter the same type of problem again. A more valuable approach is to have someone explain to us how to solve the problem so that we will then have a method for addressing future occurrences. The effectiveness of this approach, however, will depend on how good a technique we are provided with and how well the method of solution is explained.

A computer program is a description that explains how to solve a problem and it is expressed using a precise notation called a programming language: that is, one that can be understood by both humans and computers. Before you can write a program in Visual Basic or any other language, you must be able to do all three of the following:

1. Solve the problem. That is, can you already solve this problem if asked to do so?
2. Describe how to solve the problem. Can you express how you solve the problem in a way that is unambiguous and complete? Such an expression is called an "algorithm."
3. Express yourself in the programming language you will use. An algorithm written in a programming language is called a "program."

Programming is not really about problem solving. Rather it is mostly about using a programming language to express yourself. However, once you become familiar with the syntax of a programming language, most of your time as a programmer is actually spent problem solving rather than writing the program. In the textbook and in these Canvas pages, you will encounter examples that describe some of the steps that were taken in order to obtain a description of how to solve the problem posed in the example. These examples are useful because sometimes the reasoning that was used for one problem can be applied to another similar problem. So, by looking at examples, it is
possible to learn in an informal way some useful techniques that will enhance your own skill at finding and developing ways to solve problems.

Programming does require a proper description of how to solve a problem: that is, an algorithm. There are some ways of expressing algorithms that have proven effective not only in documenting how to solve a problem but also in assisting the programmer to write a program for the algorithm quickly and effectively. Before describing some of these "proven" techniques, it is important for you to know a little more about what makes a good algorithm.

**What is an Algorithm?**

An algorithm is a "properly formulated" description of how to perform a task. Many courses in computer science today are devoted to examining the different strategies that algorithms employ and measuring their efficiency with regard to the length of time they take to perform their tasks or how much memory they require. However for this analysis to be valid, as well as for a program that implements an algorithm to function correctly, certain requirements must be satisfied for a description properly to be called an algorithm.

Informally, and for this course, it is usually enough to view the term "algorithm" as simply meaning a description that explains how to solve a problem. Construction manuals are examples of such descriptions. Using a given collection of parts, they explain how to construct a product from them. Such instruction manuals and algorithms have the following important features in common:

- There are a finite number of steps provided.
- They consist of a sequence of instructions, with the sequence defining the order that the instructions should be performed.
- Decisions may be required that affect which instructions are followed subsequently.
- Some sequences of instructions may be repeated until certain conditions are satisfied.
- They terminate after a certain amount of time passes. Assuming that the instructions are correct and are performed in the correct order, on termination one will have constructed the expected product (i.e., produced the "correct" answer).

Since computer programs are simply algorithms expressed in a programming language, these features are observed in all computer programs. Learning to use a particular programming language such as Visual Basic is therefore a matter of learning how to express the features of an algorithm using only the notation of the programming language.

So, to summarize, there are two stages to writing computer programs:

1. Developing a suitable algorithm that describes how to perform a particular task.
2. Expressing that algorithm in the notation of a programming language, for example, Visual Basic.
Step 1 is by far the harder task, and success comes only from the experience gained by discovering or constructing algorithms that correctly describe how to obtain a solution. Even then, there is often more than one way to solve a problem, and so the novice programmer must develop some critical insight that will help him or her to decide on the "best" algorithm to use. To gain this insight and master the craft of programming, students must write programs.

While it is not possible to teach you how to solve problems, there are some practices that you can adopt with each algorithm that you design that will help you to construct your eventual algorithm from the initial specifications. These "tools of program design" have been developed by experienced programmers to assist them in writing algorithms that will be complete, correct, unambiguous, and capable of ready expression in a programming language. Throughout the text and on certain occasions in the Study Guide, you will find descriptions of techniques that can assist you in the efficient design of algorithms. You should take the time to understand these techniques, and we expect you to employ them in the development of solutions to the assignments.

Documenting Algorithms

When writing algorithms, natural language is not the best way to express them, particularly if they are to be interpreted by a computer. Natural language lacks the precision of expression required to avoid ambiguities of meaning and ensure that all possibilities are addressed explicitly.

Although contemporary programming languages are designed to avoid the pitfalls of natural language, they also do not provide a good way for describing algorithms to people. The reason is that writing an algorithm in a programming language requires an additional level of detail so it can be translated into binary sequences that provide the machine instructions to the computer. This translation is performed by software called compilers and interpreters, and it is necessary for a computer to be able to execute the steps of the algorithm.

If you look at a typical Visual Basic program, you will probably find much of it readable enough to be able to ascertain what the program is doing generally even if you have never seen a Visual Basic program before. This is typical of most high-level programming languages. However, it is also the case that you will see a number of statements in the program that are not related to describing the algorithm but rather are required to enable the compiler to interpret your intentions correctly: in other words, "to make it work!"

For example, if you examine the Visual Basic program provided in the textbook in Chapter 5, pages 214 to 217, you will observe a number of statements whose meaning you may grasp simply by reading them as if they were statements in English. These statements are the ones that are actually describing the algorithms. The remainder can be viewed as "programming overhead"—statements that begin with words like `Dim`, `Private`, `Function`, and `End`. All of this overhead makes a program listing
much longer than it would need to be if its only purpose was to make clear the method of the algorithm.

Therefore, because neither natural language nor programming languages are entirely satisfactory for describing algorithms, they are rarely used alone. As programming has evolved, techniques have been adopted and notation developed for describing algorithms in various ways. The purposes of such descriptions are twofold:

1. To identify the critical features of an algorithm before converting it into a program. Documenting the algorithm in this manner at this time reduces the number of oversights and other logical errors that may occur if a program is not carefully stated before it is written.
2. To summarize in a succinct way what a program was intended to do after it is written. In this respect, the description of the algorithm assists in its maintenance and revision, important considerations if the program is likely to be repaired or extended by another programmer who is not the original author.

Section 1.3 of the textbook describes three commonly used techniques for expressing algorithms: flowcharts, hierarchy charts, and pseudocode. All three are commonly employed in a large programming project to describe the role or organization of various parts of a program. Since you will be expected to employ one or more of these techniques in documenting your assignments, the primary purpose of each is provided here. For more details, you should read Section 1.3.

**Documentation Methods**

**Flowcharts**

Flowcharting is a method for documenting algorithms that is particularly effective for capturing the control activity of an algorithm. Most algorithms include steps where decisions must be made that will determine how to proceed to the next step. Flowcharting makes these decision steps and their consequences readily visible because it uses a graphical notation to represent the steps of the algorithms and, more importantly, the order in which the steps are taken.

Each step in a flowchart is a detailed description of some activity to be performed or some condition to be tested. Therefore, the text provided within each flowchart box should give enough detail for the reader to know not only what is happening but how.

For flowcharts to be understood and shared, it is necessary to adopt conventions for how the different flowcharting symbols ([Links to an external site.](http://example.com)) are to be interpreted and used. Page 8 of the *textbook* provides you with a summary of the representation and purpose of each of the commonly used flowcharting symbols. When you choose to represent an algorithm by a flowchart, it is important that you adopt the same conventions and interpretation.
For the assignments in this course, the following flowcharts symbols are the only ones you will most likely require (found at this website):

http://www.rff.com/flowchart_shapes.htm

Hierarchy Diagrams

Hierarchy diagrams (or hierarchy charts) are primarily a planning tool for partitioning an algorithm into major components and then displaying the relationships among the parts graphically.

Hierarchy diagrams are easiest to construct (and most appropriate) for algorithms that can be formulated from a problem solving technique called "divide and conquer."

The divide-and-conquer technique consists of taking a complex problem and dividing it into a number of more manageable, "simpler" problems to be addressed in a particular order. These problems may, in turn, be further subdivided until each of the components is determined to be readily amenable to solution. To obtain a solution to the original problem, one simply applies the solutions to the simpler problems in the order indicated.

An example of a hierarchy chart is provided on page 10 of the textbook (Figure 1.9 on the next page of this study guide).
As the example illustrates, hierarchy diagrams provide the most compact way of describing an algorithm but, of course, they have the least amount of detail. However, particularly for large programming problems, they are frequently the most constructive way to begin because they force the programmer to focus attention initially on the major tasks to be addressed before tackling the details.

![Hierarchy chart for the postage-stamp problem.](image)

**Pseudocode**

Pseudocode is probably what you would write if you tried to describe an algorithm using notation that mimicked the Visual Basic statements in the example on pages 214 through 217 of the textbook. If you have programmed before, then pseudocode is probably the easiest way to describe an algorithm, particularly for a new programming language with which you may be unfamiliar. In some respects, pseudocode might be considered similar to flowcharting without the boxes. However, good pseudocode is characterized by expressing tasks at a higher or more abstract level. While a step in a flowchart usually indicates not only what is being done but how, in a pseudocode statement, it is usually sufficient simply to identify what task is to be performed without necessarily providing the details about how. For example, you may require that a list be sorted without explaining what method of sorting will be used.

As a consequence, pseudocode descriptions are often shorter than the equivalent flowchart. Furthermore, pseudocode descriptions should model the organization of statements that will be observed in the eventual program but without the strict adherence to the rules of syntax that are a necessary part of a working program.

The primary disadvantage of a pseudocode description is that the control structures that were readily visible in a flowchart are less apparent. They may even be rendered invisible if they require a level of detail greater than that being used to write the pseudocode. For example, a pseudocode statement might simply state:
"Perform sort in ascending order by last name" when, in fact, the details of a sort often require a rather complex control structure.

You should examine carefully the two examples of algorithm documentation provided at the end of Section 1.3 of the textbook. Each example illustrates how the same algorithm can be described using a flowchart, pseudocode, and a hierarchy diagram.