Part I

Computer Science and Programming

Unit 1

Computing Science Basics

Learning Outcomes

- Define the basic terms of computing science.
- Explain simple algorithms using pseudocode.

Learning Activities

- Read this unit and do the "Check-Up Questions."
- Browse through the links for this unit on the course web site.
- Read Chapter 1 in *How to Think Like a Computer Scientist*.

Before we start programming, we need to know a little about what *computing science* really is.

Topic 1.1 What is an Algorithm?

The concept of an "algorithm" is fundamental to all of computing science and programming. Stated simply, an algorithm is a set of instructions that can be used to solve a problem.

Figure 1.1 contains one simple algorithm that you might use in everyday life. This algorithm is used in baking and it is written in a way that most people can understand and follow. It is used to make cookies, cakes, muffins, and many other baked goods.

Of course, we aren't going to spend this whole course talking about cooking. (It might be more fun, but the University would get cranky if we started

1. Combine the room-temperature butter and the sugar. Mix until light and fluffy.
2. Add the eggs to the creamed butter and mix to combine.
3. In another bowl, combine the liquid ingredients and mix to combine.
4. Sift together the flour and other dry ingredients.
5. Alternately add the dry and liquid ingredients to the butter-egg mixture. Mix just enough to combine.

Figure 1.1: The "creaming method": an everyday algorithm.

giving cooking lessons in CMPT courses.) Still, the algorithm in Figure 1.1 has a lot in common with the algorithms we will be looking at during this course.

We are more interested in the kinds of algorithms that can be completed by computers. We will spend a lot of time in this course designing algorithms and having the computer complete them for us.

Here's a definition of "algorithm" that most computer scientists can live with: [Anany Levitin, *Introduction to The Design & Analysis of Algorithms*, p. 3]

> An *algorithm* is a sequence of unambiguous instructions for solving a problem, i.e., for obtaining a required output for any legitimate input in a finite amount of time.

There are a few words you should notice about the definition:

- **unambiguous:** When you read an algorithm, there should be no question about what should be done. Is this the case in Figure 1.1?

  If you understand cooking terms like "light and fluffy" and "sift together", then you can probably follow most of this recipe. You might have some problem with the last step: you're supposed to "alternately" add the dry and wet ingredients. Does that mean you should do dry-wet-dry? Dry-wet-dry-wet-dry-wet? How many additions should you make?

  Recipes in cookbooks are often written with small ambiguities like this either because it doesn't matter what you do or the author is assuming

that the reader will know what to do. For the record, the right thing in this case is probably dry-wet-dry-wet-dry.

- **problem:** An algorithm should always present a solution to a particular problem. Each algorithm is designed with a particular group of problems in mind.

  In Figure 1.1, the problem must have been something like "Using these ingredients, make muffins."

- **legitimate input:** An algorithm might need some kind of input to do its job. In the example problem, the inputs are the ingredients; you have to have the correct ingredients before you can start the algorithm.

  In addition to having the inputs, they have to be "legitimate". Suppose we start the instructions in Figure 1.1 with these ingredients: 1 can of baby corn, 1 cup orange juice; 1 telephone. We aren't going to get very far. In this example, "legitimate" ingredients include sugar, eggs, flour and butter.

  If you put the wrong inputs into the algorithm, it might not be able to deal with them.

- **finite amount of time:** This means that if we start the algorithm, we had better finish it eventually.

  A recipe that leaves us in the kitchen until the end of time isn't much good. Suppose we added this step to Figure 1.1:

  6. Stir with a fork until the mixture turns into Beef Wellington.

  No amount of stirring is going to make that happen. If you followed the recipe literally, you'd be standing there stirring forever. Not good.

Many later computing science courses cover algorithms for various problems. For example, CMPT 354 (Databases) discusses algorithms for efficiently storing database information.

Data Structures

When discussing algorithms, it also becomes necessary to talk about *data structures*. A data structure describes how a program stores the data it's working with.

To carry on with the cooking example, suppose you're trying to find a recipe for muffins. Most people have their recipes in cookbooks on a shelf. To find the recipe, you'd probably select a likely looking book or two and check the index of each one for the recipe you want—that's an algorithm for finding a recipe.

On the other hand, if you have recipes on index cards in a box (because you've just copied the good recipes out of all of your books), you might have to shuffle through the whole pile to find the one you want. If you keep the pile in some kind of order, e.g. alphabetical by the name of the dish it makes, you might be able to find the recipe much faster.

The point? The way you choose to store information can have a big effect on the algorithm you need to work with it. There are many data structures that represent different ways of storing information. We will explore a variety of data structures later in the course.

Courses that discuss algorithms for particular problems generally the corresponding data structures too.

Topic 1.2 What is Computing Science?

Why all this talk of algorithms? This is supposed to be a computing science course: we should be talking about computers. Consider this quote: [Anany Levitin, *Computing Research News*, January 1993, p. 7]

> Computer science is no more about computers than astronomy is about telescopes, biology is about microscopes or chemistry is about beakers and test tubes. Science is not about tools, it is about how we use them and what we find out when we do.

Computing science (also known as *computer science*) isn't all about computers. Still, there are certainly a lot of computers around. You will be using computers in this course when you program; most computing science courses involve using computers in one way or another.

Computing science is often defined as: [G. Michael Schneider and Judith L. Gersting, *An Invitation to Computer Science*]

> The study of algorithms, including
>
> 1. Their formal and mathematical properties.

2. Their hardware realizations.
3. Their linguistic realizations.
4. Their applications.

So, computing science is really about algorithms. We will spend a lot of time in this course talking about algorithms. We will look at how to create them, how to implement them, and how to use them to solve problems.

Here is a little more on those four aspects:

1. Their formal and mathematical properties: This includes asking questions like "what problems can be solved with algorithms," "for what problems can we find solutions in a reasonable amount of time," and "is it possible to build computers with different properties that would be able to solve more problems?"
2. Their hardware realizations: One of the goals when building computers is to make them fast. That is, they should be able to execute algorithms specified by the programmer quickly. They should also make good use of their memory and be able to access other systems (disks, networks, printers, and so on). There are many choices that are made when designing a computer; all of the choices have some effect on the capabilities of the final product.
3. Their linguistic realizations: There are many ways to express algorithms so a computer can understand them. These descriptions must be written by a person and then followed by a computer. This requires some "language" that can be understood by both people and computers. Again, there are many choices here that affect how easily both the person and computer can work with the description.
4. Their applications: Finally, there are questions of what actual useful things can be done algorithmically. Is it possible for a computer to understand a conversation? Can it drive a car? Can the small computers in cell phones be made more useful? If the answer to any of these is "yes," then how?

Most of our algorithms won't look much like Figure 1.1. We will focus on algorithms that computers can follow. See Figure 1.2 for an algorithm that is more relevant to a computing science course.

1. Tell the user to pick a secret number between 1 and 100.
2. The smallest possible number is 1; the largest possible is 100.
3. Make a guess that is halfway between the smallest and largest (round down if necessary).
4. Ask the user if your guess is too large, too small or correct.
5. If they say you're correct, the game is over.
6. If they say your guess is too small, the smallest possible number is now the guess plus one.
7. If they say your guess is too large, the largest possible number is now the guess minus one.
8. Unless you guessed correctly, go back to step 3.

Figure 1.2: An algorithm that guesses a secret number between 1 and 100.

The algorithm in Figure 1.2 is designed to solve the problem "guess a secret number between 1 and 100." It meets all of the criteria of the definition of "algorithm" from Topic 1.1.

You may have to spend a few minutes to convince yourself that this algorithm will always eventually guess the correct number, thus finishing in a "finite amount of time". It does. Try a few examples.

This algorithm works by keeping track of the smallest and largest possibilities for the user's secret number. At the start of the algorithm, the number could be anywhere from 1 to 100. If you guess 50 and are told that it's too large, you can now limit yourself to the numbers from 1 to 49—if 50 if too large then the numbers from 51 to 100 must also be too large. This process continues until you guess the right number.

By the end of this course, you should be able to create algorithms like this (and more complicated ones too). You will also be able to implement them so they can be completed by a computer.

Check-Up Questions

- Can you think of a number where the algorithm in Figure 1.2 will make 7 guesses? 8?
- What is "legitimate input" for the algorithm in Figure 1.2? What happens if the user enters something else?

Topic 1.3 What is Programming?

Much of this course will focus on *computer programming*. What is programming?

A *computer program* is an algorithm that a computer can understand. This program is often referred to as an *implementation*. Not all algorithms can be implemented with a computer: Figure 1.1 can't. We're interested in the ones that can.

A *programming language* is a particular way of expressing algorithms to a computer. There are many programming languages and they all have different methods of specifying the parts of an algorithm. What you type in a particular programming language to specify an algorithm is often referred to as *code*.

Each programming language is designed for different reasons and they all have strengths and weaknesses, but they share many of the same concepts. Because of this, once you have learned one or two programming languages, learning others becomes much easier.

Why Python?

In this course, we will be using the *Python* programming language. Python is an excellent programming language for people who are learning to program.

You may be wondering why this course doesn't teach programming in C++ or Java. These are the languages that you probably hear about most often.

In this course, we want you to focus on the basic concepts of programming. This is much harder to do in Java and C++: there are too many other things to worry about when programming in those languages. Students often get overwhelmed by the details of the language and can't concentrate on the concepts behind the programs they are writing.

write "Think of a number between 1 and 100."
set *smallest* to 1
set *largest* to 100
until the user answers "equal", do this:
    set *guess* to $\lfloor (smallest + largest)/2 \rfloor$
    write "Is your number more, less or equal to *guess*?"
    read *answer*
    if *answer* is "more", **then**
        set *smallest* to $guess + 1$
    if *answer* is "less", **then**
        set *largest* to $guess - 1$

Figure 1.3: Figure 1.2 written in pseudocode.

C++ and Java are very useful for creating desktop applications and other big projects. You aren't doing that in this course. Languages like Python are a lot easier to work with and are well suited for smaller projects and for learning to program.

Topic 1.4 Pseudocode

Before you start writing programs, you need a way to describe the algorithms that you are going to implement.

This is often done with *pseudocode*. The prefix "pseudo-" means "almost" or "nearly". Pseudocode is almost code. It's close enough to being a real program in a programming language that it's easy to translate, but not so close that you have to worry about the technical details. The natural language (English) algorithm descriptions in Figures 1.1 and 1.2 might be accurate, but they aren't generally written in a way that's easy to transform to a program.

Figure 1.3 is an example of the way we'll write pseudocode in this course. It is a translation of Figure 1.2.

Figure 1.3 uses the notation $\lfloor x \rfloor$, which you might not have seen before. It means "round down", so $\lfloor 3.8 \rfloor = 3$ and $\lfloor -3.8 \rfloor = -4$. It is usually pronounced "*floor*".

```
set hour to 0
set minute to 0
set second to 0
repeat forever:
    set second to second + 1
    if second is more than 59, then
        set second to 0
        set minute to minute + 1
    if minute is more than 59, then
        set minute to 0
        set hour to hour + 1
    if hour is more than 23, then
        set hour to 0
    write "hour:minute:second"
    wait for 1 second
```

Figure 1.4: Another pseudocode example: a digital clock

It is usually helpful to express an algorithm in pseudocode before you start programming. Especially as you're starting to program, just expressing yourself in a programming language is challenging. You don't want to be worrying about what you're trying to say and how to say it at the same time.

Writing good pseudocode will get you to the point that you at least know *what you're trying to do* with your program. Then you can worry about *how to say it* in Python.

There are several computing science courses where no programming language is used and you don't write any code at all. If you know how to program, it is assumed that you know how to convert pseudocode to a program. So, the courses concentrate on pseudocode and algorithms. The rest is easy (once you learn how to program).

There is another example of pseudocode in Figure 1.4. This algorithm could be used to manage the display of a digital clock. It keeps track of the current hour, minute, and second (starting at exactly midnight).

Summary

This unit introduced you to some of the fundamental ideas in computing science. The ideas here are key to all of computing science.

If you're a little fuzzy on what exactly a data structure or pseudocode is, you don't need to panic (yet). After you've written a few programs in later units, come back to these terms and see if they make a little more sense then.

Key Terms

- algorithm
- data structure
- computing science
- computer programming
- programming language
- Python
- pseudocode