ICS3U: Algorithms and Pseudocode

Algorithms

Computers are very good at performing mathematical operations, comparing values, and storing vast amounts of data. Beyond that, however, a computer only does *exactly* what it is told to do, even if it isn't what the programmer had in mind. Because of this, it is extremely important for the programmer to understand *exactly* what the problem is, so that any ambiguity or misunderstanding does not translate into incorrect code.

To successfully solve a problem, it is necessary to establish a set of rules that will allow us to find the solution. In computer science and in mathematics, the term for this is an **algorithm**. A more precise definition of an algorithm would be something like this.

> *An algorithm is a well-ordered collection of unambiguous and effectively computable operations that when executed produces a result and halts in a finite amount of time.*
>
> – Schneider, M. and J. Gersting (1995), *An Invitation to Computer Science*, West Publishing Company, New York, NY, p. 9.

We use algorithms every day. Take, for example, the following recipe for making scrambled eggs.

1. Place a frying pan on the burner.
2. Put a teaspoonful of butter in the pan.
3. Turn a burner on the stove to low heat.
4. Crack two eggs into a bowl.
5. Whisk the eggs until they are scrambled.
6. When the pan has heated up, pour in the scrambled egg mixture.
7. Gently stir the eggs until it has a soft but firm consistency.
8. Turn off the heat.
9. Remove the pan from the burner.
10. Transfer the eggs to a plate.

Each step is clearly defined, and well-ordered. There is a result (tasty egg) produced by an ordered sequence of steps (the recipe) in a finite amount of time (5 minutes or so). A recipe is analogous to an algorithm, in this sense.

Of course, most mathematical processes can be formalized as algorithms. Recall that $n!$ ("n factorial") is defined as $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$. In other words, beginning with an integer value n, multiply it by all integers less than n and greater than 0. An algorithm might look something like this.

```
1. Obtain an integer value $n$.
2. Set $f$ equal to the value of $n$.
3. Set $i$ equal to the integer value one less than $n$.
4. While $i$ is greater than zero:
   A. Multiply $f$ by the value of $i$.
   B. Subtract one from $i$.
5. Output the value of $f$.
```

Try this with $n=4$, for $4!=4\times3\times2\times1=24$, to convince yourself that this algorithm works. Then compare it to the Python code below.

```
n = int(input("n: "))
f = n
i = n-1
while i > 0:
   f *= i
   i -= 1
print(f)
```

The code looks nearly identical to the algorithm, only the code is expressed using a more mathematical notation and uses Python-specific operators and keywords. We call this an **implementation** of an algorithm. Since the algorithm itself uses general language, it can be applied to any other programming language to produce other implementations. In any case, we achieve our goal via a sequence of clear instructions, producing a result in a finite amount of time. All computer programs, whether large or small, perform algorithms as designed by the programmer. It is *your* responsibility to ensure that the algorithm is correct, if you are to guarantee that the program runs correctly.

Pseudocode

Computers can only understand **machine language**, a sequence of 0s and 1s that is processed by the CPU in order to control various electronic components. Humans, on the other hand, typically write code using a **programming language**, which is either compiled into machine language before it is run or interpreted by a piece of software on-the-fly.

There are many different programming languages out there, and it is not realistic to expect that any programmer is proficient in *all* of them. When describing an algorithm, then, it is not a very good idea to use a *specific* programming language to describe the steps involved. Doing so would require knowledge of that language's keywords, syntax and structure. Instead, programmers may use **pseudocode**, a natural-language description of an algorithm that is structured like a program.

The factorial example above was written using a form of pseudocode. There are no industry-standard rules describing how pseudocode should be written, so there is some amount of freedom in how an algorithm is presented; however, some general guidelines that are often followed are below.

- **Write one statement per line**. This ensures each task is clearly identifiable at a glance.
- **Use a verb as a keyword**. I like to capitalize the verb, to make it easy to spot the action.
- **Use indentation to group related tasks**, including decisions or repetitive processes. This makes pseudocode more readable.
- **Ensure that pseudocode is language-independent**. This allows for implementations of the algorithm in any language.

Here is an example of pseudocode for making toast.

```
1. PUT slice of bread in toaster.
2. PRESS toaster lever down.
3. WHILE bread is not toasted to desired consistency:
   A. WAIT for bread to darken.
4. POP toaster lever up.
5. REMOVE toast from toaster.
```

And here is the algorithm for calculating $n!$ using the guidelines described above.

```
1. GET $n$
2. SET $f \leftarrow n$
3. SET $i \leftarrow n$-1
4. WHILE $i > 0$:
   A. SET $f \leftarrow f \times i$
   B. SET $i \leftarrow i$-1
5. OUTPUT $f$
```

In the example above, the left arrows indicate that a value is being assigned to a variable. For instance, the statement "SET $f \leftarrow n$" means "assign the value of n to f".

Computational algorithms are often described using pseudocode, in addition to other formats that we will soon encounter. If pseudocode is well-written, it is generally a straightforward task to translate it into a computer program, assuming that the programmer is familiar with a given language's syntax and commands.

See if you can understand the pseudocode below. Can you think of ways in which you might create an implementation of this algorithm using Python?

```
1. GENERATE a random number $1 \leq n \leq 10$
2. GET guess
3. SET count $\leftarrow$ 1
4. WHILE guess $\neq n$
   A.  GET guess
   B.  SET count $\leftarrow$ count+1
5. OUTPUT count
```