ALGORITHMS

Computers are very good at performing mathematical operations, comparing values, and storing vast amounts of data. Beyond that, however, a computer only performs exactly what it is told to do, even if that is not what a programmer has 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 C, New York, NY, p. 9.

We use algorithms every day. Take, for example, the following algorithm for making scrambled eggs.
(Your method of making scrambled eggs may be different from the one below.)

- Turn on the heat on the stove.
- Place a pan on the burner.
- Put a spoonful of butter in the pan.
- Crack an egg into a bowl.
- Whisk the egg until it is scrambled.
- Pour the scrambled egg into the heated pan.
- Stir the egg until it is cooked.
- Turn off the heat.
- Remove the pan from the burner.
- Transfer the egg to a plate.
- Enjoy breakfast.

Each step is clearly defined, and well-ordered. There is a result (tasty egg) produced in a finite amount of time (5 minutes). All recipes are analogous to algorithms, in this sense.

A slightly more complicated algorithm might be one for tying your shoes. Again, each statement must be unambiguous and precise. Most of us have probably learned how to tie our shoes so long ago that it is difficult to describe the process, since we are adept at doing it without much conscious effort; however, an algorithm might look something like the following.

- Take the left shoelace (henceforth called SL1) in your left hand, and the right shoelace (SL2) in your right hand.
- Cross SL1 over SL2, then let SL1 drop.
- Grab SL2 with your left hand, then release your right hand and use it to grab SL1, making an “X” with the two laces.
- Tuck the end of SL1 into the opening beneath the intersection of SL1 and SL2, then release it.
- Pick up SL1 in your right hand again, then pull both ends of SL1 and SL2 in opposite directions.
- Fold SL2 into a loop, then wrap SL1 around the loop and your thumb.
- Remove your thumb from the laces, then push SL1 through the resulting hole.
- Pull the two loops tight.

PSEUDOCODE

While there are no official rules concerning pseudocode, here are some guidelines:

- Write one statement per line. This ensures each task is clearly identifiable at a glance.
- Use a verb as a keyword, and capitalize it for emphasis. This makes it easy to spot the actionsinvolved in the task.
- Use indentation to group related tasks, such as decisions or repetitive processes. This makes pseudocode more readable.
- Ensure that pseudocode is language-independent.

An individual should be able to follow your pseudocode without requiring knowledge of a specific programming language.

Here is an example of pseudocode for making buttered toast.

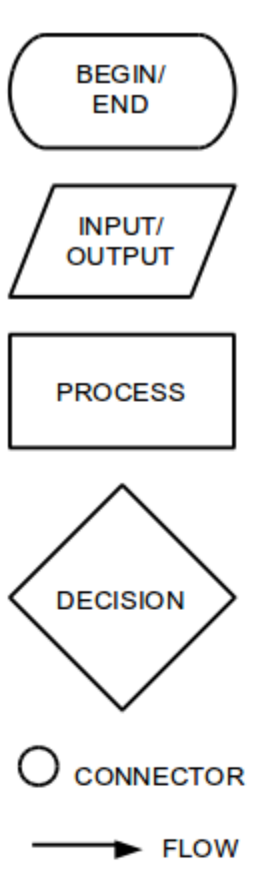
```
PUT bread in toaster
PRESS lever down so toaster is on
WHILE bread is not toasted
    WAIT for bread to turn golden brown
POP toast up
PUT toast on plate
SPREAD butter onto toast
```

Computational algorithms are often described using pseudocode. In fact, if pseudocode is well-written, it is usually fairly easy to translate it into a computer program, assuming that the programmer is familiar with a language’s syntax and commands.

FLOWCHARTS

A **flowchart** is a picture of an **algorithm**. That is, it is a visual representation of the step-by-step order to solve a problem. They use standard symbols to represent different actions, and connect each step in some logical sequence. In programming, we use flowcharts to visualize the order of the instructions in a program and the various directions that are taken to solve a problem or complete its task.

DRAWING A SIMPLE FLOWCHART



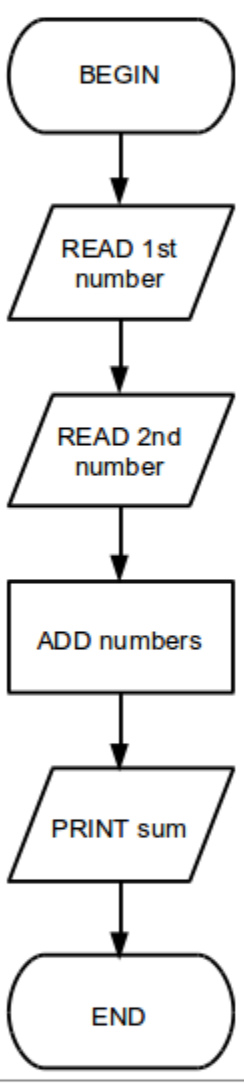
Pseudocode is a way to organize the steps of an algorithm, using a human-readable syntax. Each step is clearly identified using a set of rules, such as the one used above. Some people prefer to organize the steps *graphically*, so that they can visualize program flow better. For example, a loop in pseudocode might be indented to show that the steps within the loop are related. The loop itself would be indicated using a verb like WHILE, FOR, REPEAT, or DO … UNTIL. Using a visual representation, however, a loop might be easier to identify, since there would be some symbol or construct that would make it stand out.

Flowcharts are graphical representations of algorithms. They use standard symbols to represent different actions, and connect each step in some logical sequence. The graphic to the left shows some of the more common symbols used. There are many more, including those for reading and writing files or accessing certain types of storage, but this tutorial only covers a small subset. We will introduce other symbols as necessary.

Each flowchart begins and ends with a *terminator*. This indicates where the algorithm starts, and where it finishes. Between the terminators, each step is connected by an arrow indicating which step follows another. If two paths must connect, a *connector* is used to indicate that the two paths join. This is for clarity — sometimes it is necessary to cross over an arrow to reach another step. In this case, the lack of connector indicates this cross-over.

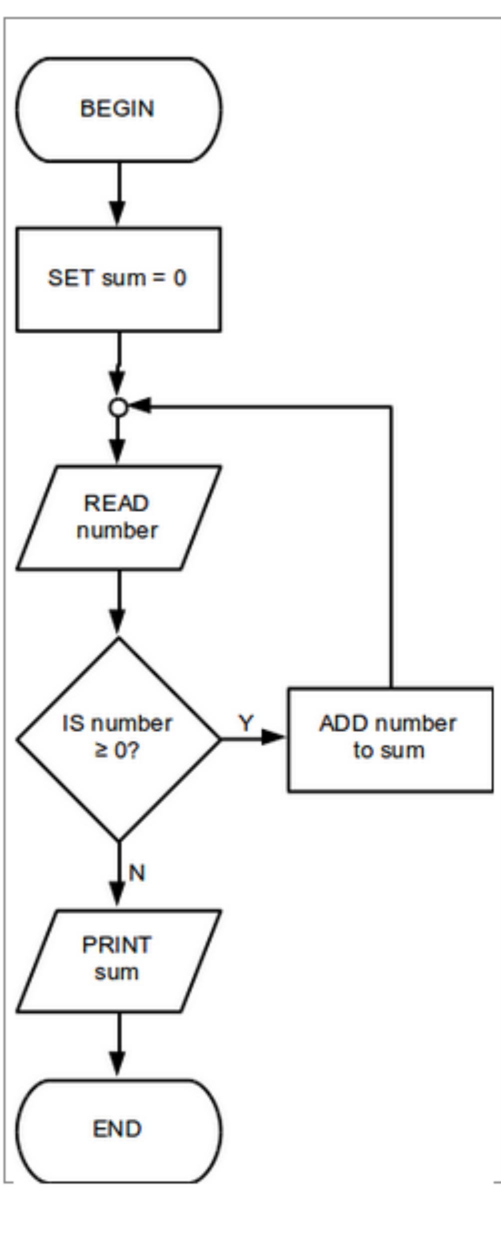
Some other common symbols used are:

- Parallelogram: indicates *input or output*, such as reading data from the user or displaying information to the screen.
- Rectangle: indicates a *process*, such as assigning a value to a variable or performing some mathematical operation.
- Diamond: indicates a *decision*, where the answer is typically “yes/no” or “true/false”. Each arrow leading out of a diamond should be labeled with the result of the decision.



To the left is a flowchart that illustrates how to add two values together. The algorithm is entirely sequential, with no loops or decisions in it. For the algorithm to work, the user must supply both numbers that are to be added. The flowchart uses parallelograms to indicate user input. After this, the algorithm processes the two numbers by adding them together. A rectangle indicates this processing. Then, the sum of the two numbers is displayed to the screen, as indicated by the final parallelogram.

Algorithms often include decision-making or repetition. A slightly more complicated example that uses both is the one below, which adds all values until a negative value is entered.



Before the user enters any values, the sum of all entered values is zero. The user then enters a value. If this value is non-negative (zero or greater), then we must update the sum accordingly. After this, the algorithm loops back to read another value. Note the connector that indicates that the arrow leaving the summation process rejoins the path just before the next value is entered. This arrow literally makes a loop, making it very easy to identify the repetition involved in the algorithm. It is very important that it rejoins the path *after* the sum is initialized to zero. Can you explain why?

At some point, the user will enter a negative value. Once this occurs, the algorithm displays the sum of the non-negative values and ends. The decision has exactly two outputs: either the number is greater than or equal to zero (non-negative), or it is below zero (negative). For any decision, there should always be two outputs to cover both possibilities.

Finally, consider the problem of determining the largest of three numbers. The flowchart below uses only three comparisons to do this.