An algorithm is a clearly-defined series of steps and decisions that are used to solve a specific problem. Algorithms are used in many different areas, not just computers.

We use algorithms every day in real life, but are not usually aware of them. Some examples:
- Adding 2 numbers together
- Following driving directions
- Think of some others.
Why are Algorithms Important?

- Allow us to clearly define and understand a problem.
- Provide us with a repeatable, precise way to solve the problem.

Algorithm Basics

- Must have a clear beginning and a clear ending.
- To get from the beginning to the end, there are a precise number of steps and decisions to follow.
- Operates exactly the same way any time it is used.

Algorithms and Computers

- Any problem we expect a computer to solve must first be broken down into an algorithm.
- This is because the computer can only make yes/no decisions and follow exact instructions.
- A problem that cannot be represented as an algorithm cannot be solved by a computer.
Computers and Intuition

- A lot of human thought involves intuition.
- Computers are not intuitive.
- Therefore, a computer cannot solve problems like
  - Is a person trustworthy?
  - Does a person dislike me?
  - Is this web site REALLY useful to me?
  - Did he mean $1.00 or $100?

Computers and Knowledge

- Humans may make decisions based on prior knowledge and experience.
- For example, if you are driving to school, and there is road construction, you may know another way to get here.
- Unless the computer is also aware of the construction, it cannot give you an alternate way of getting to school.

Computers and Errors

- Because a computer lacks intuition and certain knowledge, it has no way of knowing if the results it produces are “right”.
- It only knows how to follow exact instructions.
- Keep this in mind when we discuss the three types of programming errors later on.
Before we can design an algorithm, we must have a complete understanding of a problem and how it is solved. Then the problem must be broken down into a series of steps and decisions. The resulting algorithm must be precise and account for any possibility.

When we solve problems, we usually do not think in terms of small steps. Rather, we usually use intuition and experience in the course of solving problems. Because of this, it is not easy to design algorithms, even for the simplest of problems.

Let's take a closer look at an algorithm we all learned in grade school: Adding two whole numbers together. How do we do it?
Adding 2 Whole Numbers (Base 10)

- We start at the rightmost column (column 0).
- We add the bottom digit to the top digit.
- If the result < 10, we put in the corresponding result column.
- If result not < 10, we put \([\text{result} - 10]\) into the corresponding result column, and carry a 1 to the next column.
- Move to next column and repeat until all the columns are accounted for.

Adding – Some things to notice

- The steps are performed in a certain order.
- A decision is made and different action taken based on the decision (i.e. when do we carry?)
- Certain steps are repeated over and over again until we run out of columns.

Do we think this way?

- What if we wanted to add 100 + 50?
- What if we wanted to add 34789 + 0?
- What if we wanted to add 5286 + 1?
- Would we go through all of the steps of the algorithm to get the solution?
- A computer would have to do this.
Improving the algorithm

- Notice that the algorithm as stated only works for base-10 addition.
- How can we improve the algorithm to account for different bases?
- Can you think of any other limitations of the algorithm?
  - What about fractional numbers?

Scope of a problem

- As you can see an algorithm is only effective within a very specific SCOPE.
- When designing algorithms we need to clearly specify the scope of the problem.
  - What types of problems the algorithm can solve
  - More importantly, what types of problems the algorithm will NOT solve!

Controlling the Flow

- Our algorithm for adding two whole base-10 numbers together illustrates the 3 ways in which the flow of the algorithm is controlled.
- These are called control structures.
The 3 Control Structures

- **Sequence** – Do things in order
- **Selection (AKA Choice)** – Answer a yes/no question and perform some action based on the answer.
- **Iteration (AKA Repetition)** – Keep doing the same thing over and over again until a certain condition is met (or no longer met). In programming this is commonly called a “loop.”

Flow Charts

- To make algorithms easier to understand, we can use diagrams called **flow charts** to illustrate the flow of events.

Example Flow Chart
Sequence Illustrated

Start

- Do Step A
- Do Step B

Is some condition met?

Yes

No

- Do Step C
- End

These steps are done in order

Selection Illustrated

Start

- Do Step A
- Do Step B

Is some condition met?

Yes

No

- Do Step C
- End

Iteration Illustrated

Start

- Do Step A
- Do Step B

Is some condition met?

Yes

No

- Do Step C
- End

Step B repeats until the condition is no longer met.
Review

- Know what types of problems can be solved algorithmically.
- Understand the limitations and scope of an algorithm given its application.
- Understand the three control structures.
- Know how to read and understand a flow chart.

What Is A Program?

- A Program is a series of instructions that tell a computer what to do (or how to solve a problem).
- A program is therefore a software implementation of an algorithm.
- To put it another way, a program is an algorithm in a form that the computer can understand (as you already know, programs are either compiled or interpreted, based on the language they are written in).

What Is A Programming Language?

- A programming language is a tool that allows a person to tell a computer what to do.
- All programming languages must allow a person to explain an algorithm it in a way that the computer can later understand.
- Why do we need programming languages? (hint: do people speak in ones and zeros?)
Some example Programming Languages
- Java
- JavaScript
- C++
- Visual Basic
- VB Script
- BASIC
- COBOL
- SQL

Features of all Programming Languages
- Syntax (the grammar of the language)
- Variables (place to store values)
- Operators (do something with the stored values)
- Commands (direct instructions to the computer)
- Algorithmic Control (sequence, selection, iteration)

Syntax
- There are many different programming languages, and each has its own syntax.
- The Syntax of a program is a set of rules about how the instructions can be written.
Know Your Grammar

- The Syntax of a programming language is analogous to grammar in normal spoken/written languages.
- Example: In English, a predicate (verb) generally follows an object (noun). You wouldn’t say “Drove I to the store,” you would say “I drove to the store.”

Differences In Grammar

- Different languages may use different grammar rules. This also applies to programming languages.
- Example: In English, an adjective comes before a noun, as in “red car.”
- In Spanish, it is just the opposite. The above phrase would be translated as “car red”

Don’t Forget Vocabulary

- Any language also has a set of words, or vocabulary, that are “legal” in the language.
- Example: In English, there is no such word as “yplught.”
- Computer languages also have a list of legal words (usually much smaller than in human languages).
Programming Syntax Differences

- Some are case-sensitive, others are not.
- Some require specific symbols to mean “end of the statement,” others do not.
- There may be different “legal” commands in one language versus another.
- There may be different ways to express the selection and iteration control structures.
- There may be different ways to represent variables and operators (we will discuss these later).

Variables

- A variable is place (or “container”) to store a piece of information whose value may change.
- A variable is really a “pointer” to a specific memory address that reserve enough space for a certain amount of bits (depending on the type of variable we need).
- In modern programming languages, we do not specify what the address is specifically; the operating system will handle that behind the scenes.

Why do we need Variables?

- Let’s say we wanted to tell the computer to add any 2 numbers together.
- The computer needs a place to hold the first addend.
- The computer needs a place to hold the second addend.
- The computer needs need a place to store the sum after it has done the arithmetic.
What Makes A Variable?

- The **NAME** of a variable tells us what we will call the container. Every variable must have a name.
- The **TYPE** of a variable tells us what is allowed to be stored in it.
- The **CONTENTS** of the variable changes depending on what is actually stored in the container.

Variable in real life

- Suppose you have a jar at home you call "My Spare Change"
- Every day, you empty your pocket change into the jar.
- Also, you may take change out of the jar as you need to use it.
- Thus, the total amount of the money in the jar changes. But the jar itself does not.
- Also, you do not put anything into the jar that is not pocket change (like buttons or toothpicks).

The Jar as a Variable.

- If you can imagine your jar as a variable, and coins as what is allowed to be put into the variable, you will discover 2 things:
- The **NAME** of the variable is "My Spare Change"
- The **TYPE** of the variable is "coins".
- The **CONTENTS** of the variable depend on what you have inside the jar (how many nickels, pennies, dimes, etc.).
Variable Names

- Variable NAMES point to a location in memory where the variable is stored.
- Think of memory as a large collection of bins where you can store things.
- The computer understands the variable name as the location of the bin where the variable is kept.

Variable Types

- Variable TYPES tell the computer about what the information in the variable means, and what the computer is allowed to do with the information.
- Consider 2 types of variables – numeric (numbers) and character (words or letters).
- Would a computer attempt to multiply a number by a character?
- NOTE: You should recognize that this concept is very similar to field types when we discussed databases. The same principles apply as far as storage space in memory, etc.

Operators

- An operator represents an action that can be performed against one or more variables.
- Operators help the computer calculate values and make decisions.
- You already know all about operators from everyday life (but may not realize it).
3 Types of Operators

- Mathematical Operators
- Comparison Operators
- Logical Operators

Mathematical Operators

These are very familiar to most people. Some examples include:

- Add (+)
- Subtract (-)
- Multiply (*)
- Divide (/)
- Modulus (remainder from division)

Comparison Operators

Like the name implies, these operators compare the values of two variables or expressions. Some examples:

- GREATER THAN (>)  Example 5 > 3
- LESS THAN (<)  Example 3 < 5
- EQUAL TO (=)  Example 1 = 1
Processing of Comparison Operators

- When the computer compares two things using a comparison operator, it evaluates the comparison as either TRUE or FALSE (or YES/NO or 1/0).
- Therefore the result of a comparison operation is actually a boolean value.

Evaluating Comparisons

<table>
<thead>
<tr>
<th>Expression</th>
<th>What We Think</th>
<th>What the Computer Thinks</th>
</tr>
</thead>
<tbody>
<tr>
<td>4 &lt; 2</td>
<td>Is 4 less than 2?</td>
<td>False</td>
</tr>
<tr>
<td>5 = 6</td>
<td>Is 5 equal to 6?</td>
<td>False</td>
</tr>
<tr>
<td>5 = 5</td>
<td>Is 5 equal to 5?</td>
<td>True</td>
</tr>
<tr>
<td>100 &gt; 4</td>
<td>Is 100 greater than 4?</td>
<td>True</td>
</tr>
<tr>
<td>4 &lt; 10</td>
<td>Is 4 less than 10?</td>
<td>True</td>
</tr>
<tr>
<td>6 &gt; 200</td>
<td>Is 6 greater than 200?</td>
<td>False</td>
</tr>
</tbody>
</table>

Logical Operators

- Logical (boolean) operators are the key to helping the computer to make decisions and answer questions.
- You should remember these from early in the term:
  - NOT (condition must not be true)
  - AND (all of the conditions must be true)
  - OR (either of the conditions must be true)
Compound Operators

- These are in reality combinations of comparison and logical operators, but commonly written in this “shorthand” form:
  - GREATER THAN OR EQUAL TO (> =)
  - LESS THAN OR EQUAL TO (< =)
  - NOT EQUAL TO (! =, <>)

Commands

- Commands are special words or phrases that directly tell the computer to do something.
- Different programming languages can have different command words and phrases.
- What types of commands do you think we may want to give to the computer?

Some Types of Commands

- Begin the program
- End the program
- Take input from the user
- Send output to the screen or printer
- Make a new variable
- Put a value into variable
- Do a procedure or run another program
- Open a file
- etc.
- Each programming language would have a special way of telling the computer to do these things.
Every programming language must have a way to understand the three control structures:
- Sequence
- Selection
- Iteration.
Different languages have different ways of doing this.

A Fictitious Example (can you figure out what it does?)

BEGIN PROGRAM
MAKE VARIABLE ADDEND_1, TYPE IS NUMERIC
MAKE VARIABLE ADDEND_2, TYPE IS NUMERIC
MAKE VARIABLE SUM, TYPE IS NUMERIC
SEND OUTPUT "PLEASE ENTER 2 NUMBERS" TO SCREEN
TAKE INPUT FROM KEYBOARD INTO ADDEND_1
TAKE INPUT FROM KEYBOARD INTO ADDEND_2
PUT (ADDEND_1 + ADDEND_2) INTO SUM
SEND OUTPUT "HERE IS YOUR ANSWER" TO SCREEN
SEND OUTPUT SUM TO SCREEN
END PROGRAM

To Review
Every programming language must have a way to express:
- Syntax
- Variables
- Operators
- Commands
- Algorithmic Control
As you are well aware, programs can contain errors (colloquially called “bugs”). There are three main types of errors. It is important that you know what these are and how to identify them.

- Syntax Errors
- Run-Time Errors
- Logic Errors

If the way the program is written does not make sense to the computer, the program cannot be used. This is called a “syntax error.” If the syntax is correct, the computer is allowed to translate the written program into its own language (1s and 0s) and run the program. If not, will either (a) NOT compile, or (b) not be interpreted. Either way, the complete program can never run. Therefore, syntax errors are usually fairly easy to identify and correct.

IF (a > b) THEN LET c = 0;
IF (a < b THEN LET c = 1;

In the second line, notice that there is an unclosed parenthesis.
Run-Time Errors

- This is an error that causes a computer program that is already running successfully to "crash."
- It may work fine in some cases, but crash in other cases. Usually this is because some particular cases were not thought out well enough in design (i.e. a number gets too big to hold in memory the right way), or the program does not validate user input well enough (i.e. lets a user try to divide by zero).
- Run time errors are harder to identify and correct than syntax errors. It is very important to make sure that a program is rigorously tested and to deliberately try to "break" the program during testing. This can help identify run-time errors.

Run-Time error Example

- ENTER a;
- ENTER b;
- LET c = (a / b);
- DISPLAY c;
- What if the value of b is zero?
- This will result in a run-time error, unless care is taken to prevent this. For instance, we can keep asking to enter a value for b until the value entered is NOT zero (using iteration and selection).

Logic Errors

- The program executes properly in all cases, and never stops running because of a syntax or run-time error.
- HOWEVER, it does not solve the problem correctly!
- **Logic errors are the most dangerous type**, because the computer does not "know" that anything is wrong.
- Logic errors are extremely difficult to identify and correct.
- Imagine a mission critical system (medical, financial, navigation, defense, etc.) that has a logic error. **VERY BAD THINGS CAN HAPPEN!!!**
Logic Error Example
- LET X=0;
- WHILE (X < 10)
- DISPLAY "Hello";
- END–WHILE

This is called an endless loop. Since the condition to exit the loop will never be met, it will keep repeating indefinitely.

Another Logic Error Example
- ...
- LET carry = 0;
- LET result = addend1 + addend2;
- IF (result > base) THEN
  - LET result = result – base;
  - LET carry = 1
- END–IF
- ...

Based on the context of the problem, you should be able to identify the logic error in this piece of a larger program. Can you?

Errors – Review
- Know how to identify the three types of errors.
  - Syntax
  - Run–Time
  - Logic
The examples I gave were only to illustrate the concepts.
These were not examples of any real programming language.
The important thing to understand is HOW programming languages are constructed and what they are capable of doing. In general, if you do become proficient in a particular programming language, you can adapt that knowledge to any other language.