Chapter 1

An Introduction to Computer Science and Problem Solving

What is in this Chapter?

This first chapter explains what computer science is all about. It will help you understand that the goal of a computer scientist is to solve problems using computers. You will see how problems are formulated by means of algorithms and how the process of abstraction can be used to break problems down to easily manageable pieces. Finally, we will discuss the notion of efficiency.

1.1 What is Computer Science ?

Computers are used just about everywhere in our society:

- Communications: internet, e-mail, cell phones
- Word Processing: typing/printing documents
- Business Applications: accounting, spreadsheets
- Entertainment: games, multimedia applications
- Database Management: police records, stock market
- Engineering Applications: scientific analysis, simulations
- Manufacturing: CAD/CAM, robotics, assembly
- ... many more ...

A computer is defined as follows (Wikipedia):

> *A* ***computer*** *is a programmable machine that receives input, stores and manipulates data, and provides output in a useful format.*

In regards to today's computers, the "*machine*" part of the computer is called the ***hardware***, while the "*programmable*" part is called the ***software***.

Since computers are used everywhere, you can get involved with computers from just about any field of study. However, there are specific fields that are more computer-related than others. For example. the fields of ***electrical engineering*** and ***computer systems engineering*** primarily focus on the design and manufacturing of computer *hardware*, while the fields of **software engineering** and **computer science** primarily focus on the design and implementation of *software*.

Software itself can be broken down into 3 main categories:

- **System Software:** is designed to operate the computer's hardware and to provide and maintain a platform for running applications. (e.g., Windows, MacOS, Linux, Unix, etc..)

- **Middleware:** is a set of services that allows multiple processes running on one or more machines to interact. Most often used to support and simplify complex distributed applications. It can also allow data contained in one database to be accessed through another. Middleware is sometimes called *plumbing* because it connects two applications and passes data between them. (e.g., web servers, application servers).

- **Application Software:** is designed to help the user perform one or more related specific tasks. Depending on the work for which it was designed, an application can manipulate text, numbers, graphics, or a combination of these elements. (e.g., office suites, web browsers, video games, media players, etc…)

The area of software design is huge. In this course, we will investigate the basics of creating some simple application software. If you continue your degree in computer science, you will take additional courses that touch upon the other areas of system software and middleware.

Software is usually written to fulfill some need that the general public, private industry or government needs. Ideally, software is meant to make it easier for the ***user*** (i.e., the person using the software) to accomplish some task, solve some problem or entertain him/herself. Regardless of the user's motivation for using the software, many problems will arise when trying to develop the software in a way that produces correct results, is efficient ad robust, easy to use and visually appealing. That is where *computer science* comes in:

> ***Computer science*** *is the study of the theoretical foundations of information and computation, and of practical techniques for their implementation and application in computer systems (Wikipedia).*

So, computer science is all about taking in information and then performing some computations & analysis to solve a particular problem or produce a desired result, which depends on the application at hand.

Computer science is similar to mathematics in that both are used as a means of defining and solving some problem. In fact, computer-based applications often use mathematical models as a basis for the manner in which they solve the problem at hand.

In mathematics, a solution is often expressed in terms of formulas and equations. In computer science, the solution is expressed in terms of a *program*:

> *A* ***program*** *is a sequence of instructions that can be executed by a computer to solve some problem or perform a specified task.*

However, computers do not understand arbitrary instructions written in English, French, Spanish, Chinese, Arabic, Hebrew, etc..

Instead, **computers have their own languages** that they understand. Each of these languages is known as a programming language.

> *A* ***programming language*** *is an artificial language designed to automate the task of organizing and manipulating information, and to express problem solutions precisely.*

A programming language "boils down to" a set of words, rules and tools that are used to explain (or define) what you are trying to accomplish. There are many different programming languages just as there are many different "spoken" languages.

Traditional programming languages were known as ***structural*** **programming** languages (e.g., C, Fortran, Pascal, Cobol, Basic). Since the late 80's however, ***object-oriented*** **programming** languages have become more popular (e.g., JAVA, C++, C#)

There are also other types of programming languages such as ***functional*** programming languages and ***logic*** programming languages. According to the **Tiobe** index (i.e., a good site for ranking the popularity of programming languages), as of February 2011 the 10 most actively used programming languages were (in order of popularity):

Java, **C**, **C++**, **PHP**, **Python**, **C#**, **VisualBasic**, **Objective-C**, **Perl**, **Ruby**

For many years, we used JAVA as the basis in this course, due to its popularity as well as its ease of use. However, JAVA does have some drawbacks for new programmers, pertaining to some overhead in getting started with the language.

We therefore recently adjusted this course to use a language called ***Processing*** (www.processing.org) which is a JAVA-based language with much less overhead in getting started in programming. In addition, the graphical nature of the Processing language allows for more visual applications to be developed quicker and easier. You will learn more about this language as the course goes on.

When thinking of jobs and careers, many people think that computer science covers anything related to computers (i.e., anything related to *Information Technology*). However, computer science is not an area of study that pertains to IT support, repairing computers, nor installing and configuring networks. Nor does it have anything to do with simply using a computer such as doing word-processing, browsing the web or playing games. The focus of computer science is on understanding what goes on behind the software and how software/programs can be made more efficiently.

The **Computer Sciences Accreditation Board** (CSAB) identifies four general areas that it considers crucial to the discipline of computer science:

- *theory of computation*
  - investigates how specific computational problems can be solved efficiently

- *algorithms and data structures*
  - investigates efficient ways of storing, organizing and using data

- *programming methodology and languages*
  - investigates different approaches to describing and expressing problem solutions

- *computer elements and architecture*
  - investigates the design and operation of computer systems

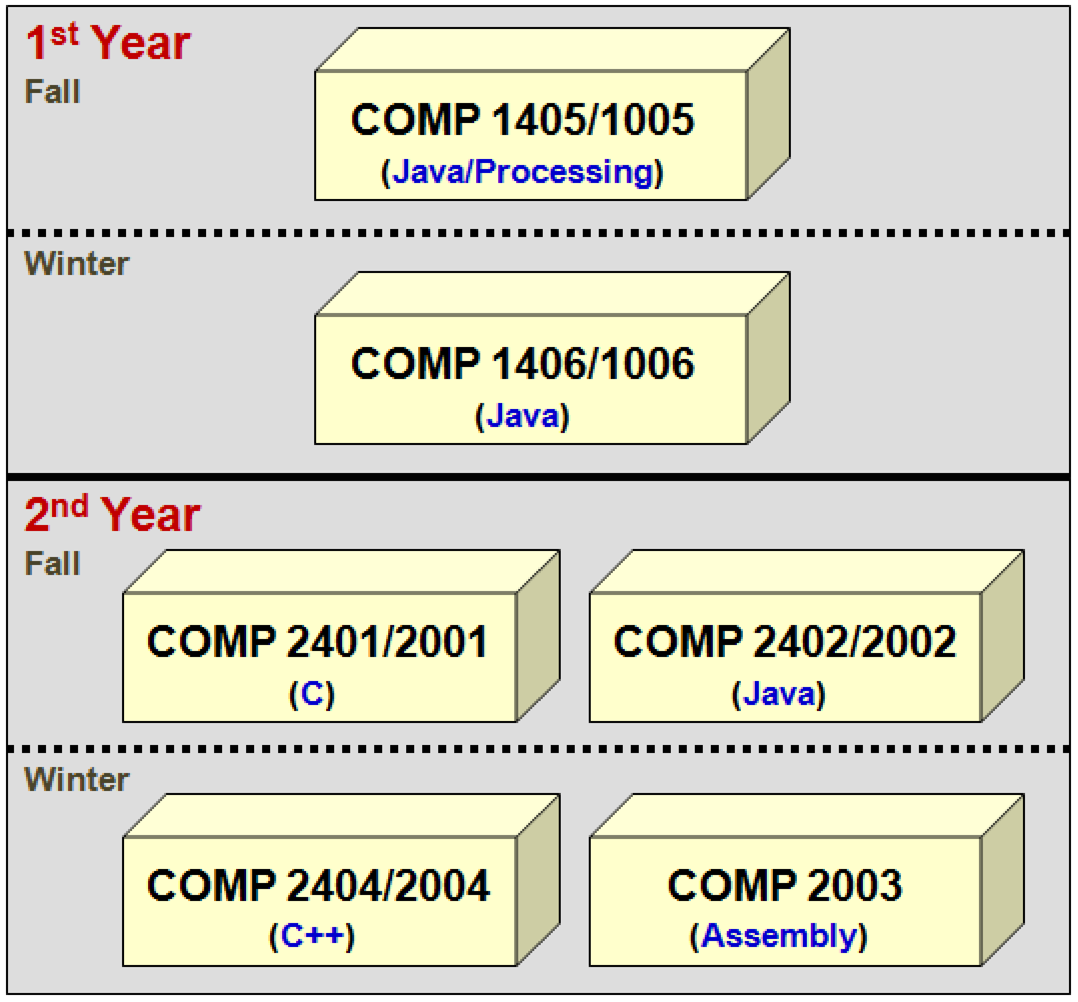
However, in addition, they also identify other important fields of computer science:

- software engineering
- artificial intelligence
- computer networking & communication
- database systems
- parallel computation
- distributed computation
- computer-human interaction
- computer graphics
- operating systems
- numerical & symbolic computation

There are aspects of each of the above fields can fall under the general areas mentioned previously. For example, within the field of **database systems** you can work on theoretical computations, algorithms & data structures, and programming methodology.

As you continue your studies in computer science, you will be able to specialize in one or more of these areas that interest you. This course, however, is meant to be an introduction to programming computers with an emphasis on problem solving.

This is your first programming course here in the School of Computer Science at Carleton. You have some more ***core*** programming courses coming up after this one. Here is a break-down of how this course fits in with your first 2 years of required programming courses:



Of course, there are other computer science courses as well. These are just the core courses that nearly everyone is required to take. After this course is over, you *should* understand how to write computer programs. In the winter term, you will take COMP1406/1006 which is a more detailed course focused on Object-Oriented programming in JAVA. Together, these two courses give you a solid programming background and you will be able to learn other computer languages easily afterwards … since they all have common features. If you want to do well in this course, attend all lectures and tutorials and do your assignments.

1.2 Writing Programs in *Processing*

It is now time to start writing simple programs to solve simple problems. As mentioned, we will be using the Processing language (available for free from **www.Processing.org** for your PC, MAC or Linux system).

Processing is a programming language and development environment all in one. It is an easy programming language to get started quickly in producing programs within a visual context. That means, it is a simple language that has powerful functionality for creating professional quality visual-based (i.e., graphical) applications and animations.

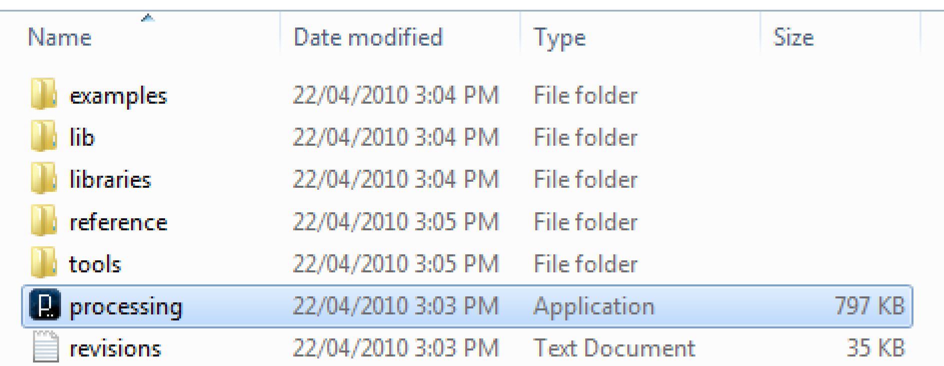
The Processing community has written over seventy libraries to help you produce applications that incorporate:

- computer vision
- data visualization
- music
- networking
- electronics

Tens of thousands of companies, artists, designers, architects, and researchers use Processing to create an incredibly diverse range of projects including:

- Motion graphics for TV commercials
- Animations for music videos
- Visualizations such as that of a coastal marine ecosystem

Processing allows you to export applets for use on the web or standalone applications for the PC, Mac or Linux operating systems. To start processing, just double-click on the processing application icon that is in the processing folder that you downloaded:

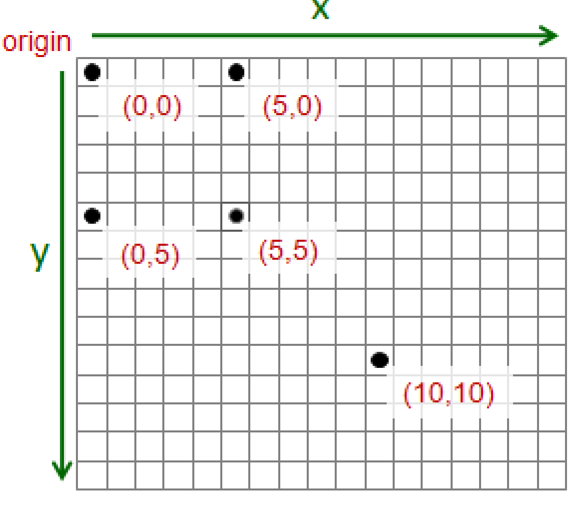


Here is what it looks like when you are working with Processing:

Each program is called a "sketch" in Processing. The top left play button starts your program which brings up a window (shown yellow here with a picture of a teddy bear). Since, many processing programs are meant to be animations, there is also a stop button beside the play button to stop the program. You should explore the Processing IDE (i.e., Integrated Development Environment) a little to get used to it.

Processing uses the same syntax as JAVA. That means, Processing code looks almost exactly like JAVA code. So when you are programming in Processing, you are actually learning JAVA as well. However, Processing has been designed in a way that makes it easier to get you started because some of the overhead in getting your first program working is hidden.

As you may recall, Processing is a graphics-based language and therefore we will spend a lot of time and effort drawing various things on the screen. When drawing anything, it is important to specify **where** you want to draw.



The output screen is organized as a 2-dimensional (2D) grid of pixels organized by the standard **x** and **y** coordinate system. That is, given an **(x,y)** pair, which we call a ***point***, the **x** specifies a number of horizontal pixels from the ***origin*** (or *start location* at the top-left of the screen) while **y** specifies the number of vertical pixels from the origin. So, point **(0,0)** is the origin and is at the top-left of the screen.

Lets write our first program. Lets draw a simple house like this one shown here. This involves drawing a square, a triangle, a rectangle and a dot.

Since Processing is a graphical-based language, there are pre-defined functions for drawing shapes. Each of these functions requires some ***parameters*** to specify further information about how to do the drawing such as locations and dimensions of what we are trying to draw.

Looking at the table of functions on the next page, it should be clear that we need to call the following functions in order to draw our house:

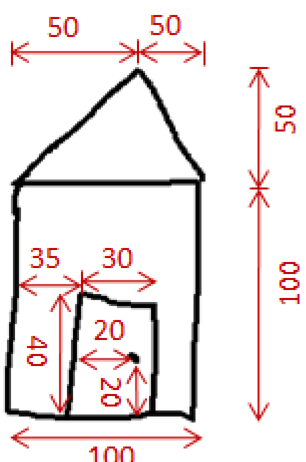
- **rect(x, y, w, h)** – for the main frame
- **triangle(x_1, y_1, x_2, y_2, x_3, y_3)** – for the roof
- **rect(x, y, w, h)** – for the door
- **point(x, y)** – for the door handle

All we need to do then is to figure out what the parameters should be.

Here are the ones that we can use to draw 2-dimensional shapes:

Function	Description	Example
point(x,y)	draws a single **dot** at the location specified by **x** and **y**.	(x,y)
line(x_1,y_1,x_2,y_2)	draws a **line** from location (**x_1, y_1**) to location (**x_2, y_2**).	(x_1,y_1) (x_2,y_2)
rect(x,y,w,h)	draws a **rectangle** with its top-left at location (**x, y**). The width and height of the rectangle are **w** and **h**. If **w** and **h** are equal, a **square** is drawn.	(x,y) h w

triangle(x_1,y_1,x_2,y_2,x_3,y_3)	draws 3 lines in order from (**x_1, y_1**) to (**x_2, y_2**) to (**x_3, y_3**) and back to (**x_1, y_1**) to form a **triangle**.	(x_2,y_2) (x_3,y_3) (x_1,y_1)
quad($x_1,y_1,x_2,y_2,x_3,y_3,x_4,y_4$)	draws 4 lines in order from (**x_1, y_1**) to (**x_2, y_2**) to (**x_3, y_3**) to (**x_4, y_4**) and back to (**x_1, y_1**) to form a **4-sided shape**.	(x_2,y_2) (x_1,y_1) (x_3,y_3) (x_4,y_4)
ellipse(x,y,w,h)	draws an **ellipse** (or **oval**) with its *center* at location (**x, y**). The width and height of the ellipse are **w** and **h**. If **w** and **h** are equal, a **circle** is drawn.	(x,y) h w
arc(x,y,w,h,start,stop)	draws an **arc** (i.e., a **portion of an ellipse or circle**) with its *center* at location (**x, y**). The width and height of the ellipse are **w** and **h**. If **w** and **h** are equal, a **circle** is drawn. The arc is draw from the **start** number of radians to the **stop** number of radians.	w h (x,y) start stop



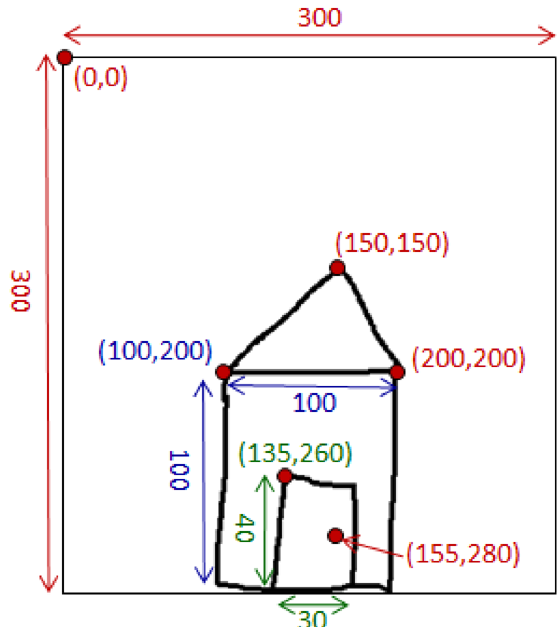
So the first step is to figure out the dimensions of the house in pixels. To do this, we begin with our hand-drawn sketch. Then, we add some dimensions to the house (in pixels).

Here to the right, is one possible set of dimensions for the house. Note that the drawing is quite rough and so the dimensions are not necessarily to scale.

Next, we'll need to know roughly how big the drawing area will be. Sometimes it is a good idea to know this even before we decide upon the dimensions so that our house is the "proper" size according to the context of the drawing area. Assume that we choose a drawing area of **300 x 300** pixels.

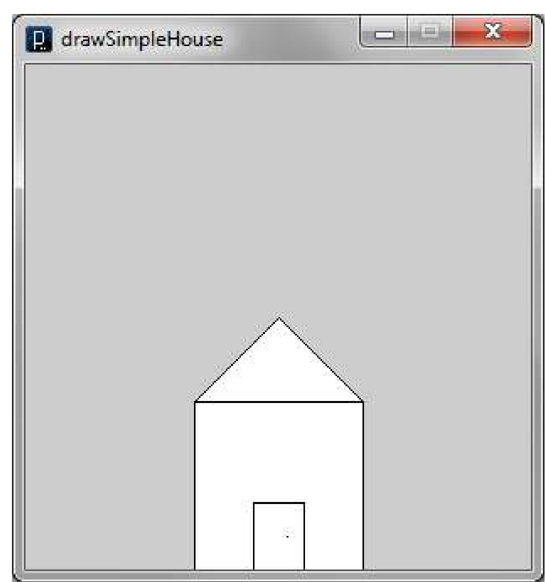
Now we need to decide *where* within the area the house will be located and assign point values to the corners of the house … remembering that (0,0) is at the top left corner of the drawing area.

Note that for rectangles, we just need to know the top left corner, along with the dimensions.



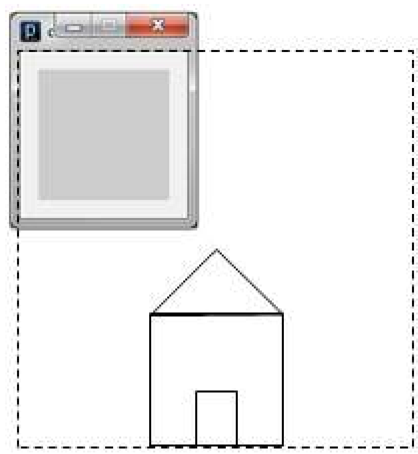
Here is the final program:

```
drawSimpleHouse §
// Set the size of the window
size(300,300);

// Draw the frame
rect(100,200,100,100);

// Draw the roof
triangle(100,200,150,150,200,200);

// Draw the door
rect(130,260,30,40);

// Draw the door handle
point(155,280);
```



The **size(w,h)** function in Processing allows you to specify the size of the drawing area. If you do not call this method, the default window size will be **100 x 100**.

In our case, if we did not call **size(300,300)**, then a small window would appear and the house would not be shown since it would be drawn off the window's visible area as shown here:



"Drawing things off the screen" is a common error that many students encounter when writing graphical programs.

So … we have just created our first Processing program. Notice that each function call ends with a **;** character. That's how you tell Processing (and JAVA) that you are done that step of the program. It is like having a period at the end of an English sentence to indicate that the sentence has ended and a new one is about to begin. Leaving off a semi-colon somewhere is considered a syntax error (i.e., like a spelling or grammar error) and will prevent your program from running:

```
// Draw the door handle

Syntax error, maybe a missing semicolon?
        at processing.app.Editor$DefaultRunHandler.run(Unknown
Source)
        at java.lang.Thread.run(Thread.java:619)

7
```

If this happens, Processing usually highlights the line in your program that appears just after the line that is missing the semicolon.

You may also notice that there are lines with double slashes **//**. These are called ***comments***. It tells the Processing interpreter that the rest of the line is to be ignored by the program. This allows you to put English-like explanations throughout your code so that anyone who read your program later (possibly you yourself) will understand what you are doing in that part of the program. It is always a good idea to use comments but not too many of them such that you code becomes too cluttered. Use just enough to make it clear what you are doing in that part of your program. Throughout the course, take notice of *where* and *how many* comments are being used in the example programs that we do together.

You can also make multiple-line comments by beginning with **/*** and ending with ***/**. For example, this could be a comment at the top of your program:

```
/* DrawSimpleHouse:
   ----------------
   This program draws a simple house that has
   A frame, a roof and a door with a door knob. */
```

Forgetting one / in your comment will produce this error:

```
// Draw the door handle

unexpected token: /
        at processing.app.Editor$DefaultRunHandler.run(Unknown
Source)
        at java.lang.Thread.run(Thread.java:619)

1
```

And forgetting to close a multi-line comment will produce this error:

```
// Draw the door handle

Missing the */ from the end of a /* comment */
        at processing.app.Editor$DefaultRunHandler.run(Unknown
Source)
        at java.lang.Thread.run(Thread.java:619)

1
```

As you learn to program, you will find many more errors. For example, a VERY common error is to spell one of the function names wrong. If you accidentally spell **rect** as **ret**, for example, this would be the error that you get:

```
The function ret(int, int, int, int) does not exist.
        at processing.app.Editor$DefaultRunHandler.run(Unknown
Source)
        at java.lang.Thread.run(Thread.java:619)

10
```

You will notice however, that when you spell a function name wrong, you usually notice it because it will not appear with the proper "orange" color that all the other functions have.

A less obvious error message occurs when you miss one of the parameters, or add too many, or pass the wrong type of values to the function. For example, assume that you called **rect(100, 200, 100)** instead of **rect(100, 200, 100, 100)**. This is the "less understandable" error that you would get:

```
The method rect(float, float, float, float) in the type PApplet is not applicable for the arguments (int, int, int)

10
```

There are many more functions that you can play around with.

When using computers, colors are often represented as:

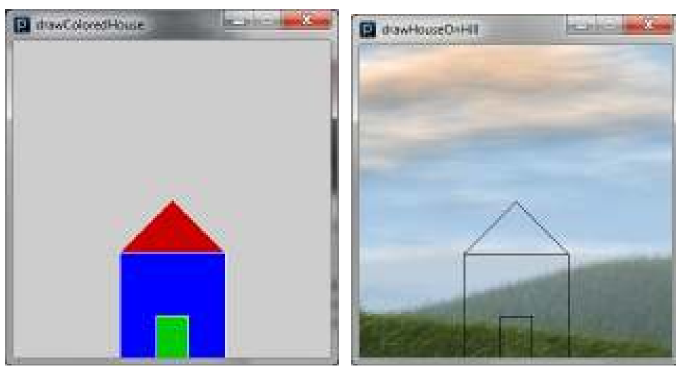
- ***Grayscale***: a shade of gray value (from **0 to 255**) where **0** represents **black**, **255** represents **white** and values in between represent various levels of gray.

- ***RGB***: 3 values (from **0 to 255**) representing the amount of red, green and blue in the color. Bright **red**, for example would be represented by parameters (**255, 0, 0**), bright **green** as (**0, 255, 0**) and royal **blue** as (**0, 0, 255**). By varying these values, you can obtain any color of the rainbow.

Here are a few color-related functions that you can make use of:

Function	**Description**
stroke(gray);	Sets the border-color for lines & shapes (e.g., points, lines, arcs, rectangles, triangles, ellipses) to a shade of gray specified by the value of the **gray** parameter.
stroke(r, g, b);	Sets the border-color for shapes to the color with the given amount of red, green and blue, specified by parameters **r**, **g** and **b**, respectively.
noStroke();	Sets the border-color to be transparent (i.e., glass).
fill(gray);	Sets the fill-in-color for enclosed shapes (e.g., rectangles, triangles, ellipses) to a shade of gray specified by the value of the **gray** parameter.
fill(r, g, b);	Sets the fill-in-color for enclosed shapes to the color with the given amount of red, green and blue, specified by parameters **r**, **g** and **b**, respectively.
noFill();	Sets the fill-in-color to be transparent (i.e., glass).
background(gray);	Sets the drawing area's background to a shade of gray specified by the value of the **gray** parameter.
background (r, g, b);	Sets the drawing area's background to the color with the given amount of red, green and blue, specified by parameters **r**, **g** and **b**, respectively.
background(**loadImage**("hills.png"));	Sets the drawing area's background to the image specified by the given image file name which may be a **png**, **gif** or **jpg** file.

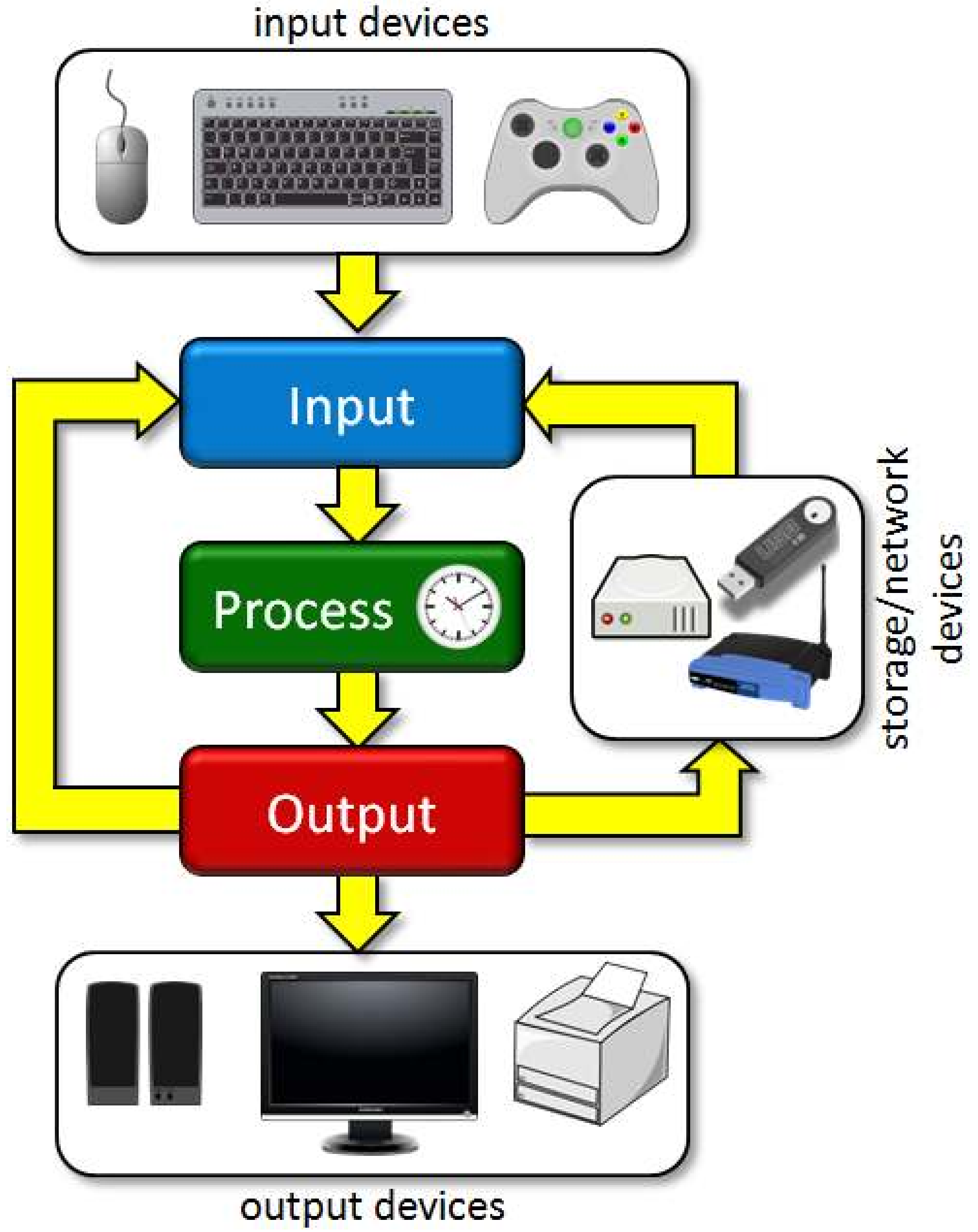
See if you can produce the following images (of course, you would need to supply your own 300 x 300 image for the example with the background hills):



1.3 Problem Solving

Regardless of the area of study, computer science is all about solving problems with computers. The problems that we want to solve can come from any real-world problem or perhaps even from the abstract world. We need to have a standard systematic approach to solving problems.

Since we will be using computers to solve problems, it is important to first understand the computer's information processing model. The model shown in the diagram below assumes a single CPU (Central Processing Unit). Many computers today have multiple CPUs, so you can imagine the above model duplicated multiple times within the computer.



A typical single CPU computer processes information as shown in the diagram. Problems are solved using a computer by obtaining some kind of user input (e.g., keyboard/mouse information or game controller movements), then processing the input and producing some kind of output (e.g., images, test, sound). Sometimes the incoming and outgoing data may be in the form of hard drives or network devices.

In regards to problem solving, we will apply the above model in that we will assume that we are given some kind of input information that we need to work with in order to produce some desired output solution. However, the above model is quite simplified. For larger and more complex problems, we need to iterate (i.e., repeat) the input/process/output stages multiple times in sequence, producing intermediate results along the way that solve part of our problem, but not necessarily the whole problem. For simple computations, the above model is sufficient.

It is the "problem solving" part of the process that is the interesting part, so we'll break this down a little. There are many definitions for "problem solving". Here is one:

> ***Problem Solving*** *is the sequential process of analyzing information related to a given situation and generating appropriate response options.*

There are 6 steps that you should follow in order to solve a problem:

1. Understand the Problem
2. Formulate a Model
3. Develop an Algorithm
4. Write the Program
5. Test the Program
6. Evaluate the Solution

Consider a simple example of how the input/process/output works on a simple problem:

Example: *Calculate the average grade for all students in a class.*

1. **Input**: get all the grades … perhaps by typing them in via the keyboard or by reading them from a USB flash drive or hard disk.

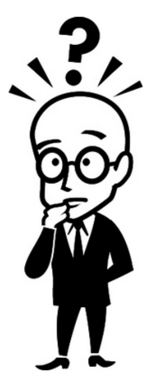
2. **Process**: add them all up and compute the average grade.

3. **Output**: output the answer to either the monitor, to the printer, to the USB flash drive or hard disk … or a combination of any of these devices.

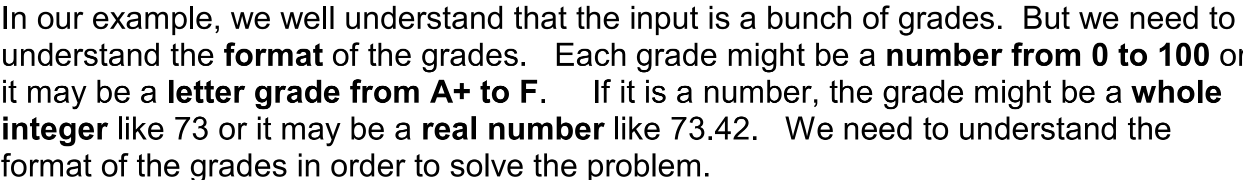
As you can see, the problem is easily solved by simply getting the input, computing something and producing the output. Let us now examine the 6 steps to problems solving within the context of the above example.

STEP 1: Understand the Problem:

It sounds strange, but the first step to solving any problem is to make sure that you understand the problem that you are trying to solve. You need to know:



- What input data/information is available ?
- What does it represent ?
- What format is it in ?
- Is anything missing ?
- Do I have everything that I need ?
- What output information am I trying to produce ?
- What do I want the result to look like … text, a picture, a graph … ?
- What am I going to have to compute ?



In our example, we well understand that the input is a bunch of grades. But we need to understand the **format** of the grades. Each grade might be a **number from 0 to 100** or it may be a **letter grade from A+ to F**. If it is a number, the grade might be a **whole integer** like 73 or it may be a **real number** like 73.42. We need to understand the format of the grades in order to solve the problem.

We also need to consider missing grades. What if we do not have the grade for **every** student (e.g., some were away during the test) ? Do we want to be able to **include** that person in our average (i.e., they received 0) or **ignore** them when computing the average ?

We also need to understand what the output should be. Again, there is a formatting issue. Should we output a **whole** or **real number** or a **letter grade** ? Maybe we want to display a pie chart with the average grade. It is our choice.

Finally, we should understand the kind of processing that needs to be performed on the data. This leads to the next step.

STEP 2: Formulate a Model:

Now we need to understand the processing part of the problem. Many problems break down into smaller problems that require some kind of simple mathematical computations in order to process the data. In our example, we are going to compute the average of the incoming grades. So, we need to know the model (or formula) for computing the average of a bunch of numbers. If there is no such "formula", we need to develop one. Often, however, the problem breaks down into simple computations that are well understood. Sometimes, we can look up certain formulas in a book or online if we get stuck.

In order to come up with a model, we need to fully understand the information available to us. Assuming that the input data is a bunch of integers or real numbers $\mathbf{x_1,x_2,\ldots,x_n}$ representing a grade percentage, we can use the following computational model:

$$\mathbf{Average1 = (x_1 + x_2 + x_3 + \ldots + x_n) / n}$$

where the result will be a number from 0 to 100.

That is very straight forward (assuming that we knew the formula for computing the average of a bunch of numbers). However, this approach will not work if the input data is a set of letter grades like **B-, C, A+, F, D-**, etc.. because we cannot perform addition and division on the letters. This problem solving step must figure out a way to produce an average from such letters. Thinking is required.

After some thought, we may decide to assign an integer number to the incoming letters as follows:

A$^+$ = 12	**B**$^+$ = 9	**C**$^+$ = 6	**D**$^+$ = 3	**F** = 0
A = 11	**B** = 8	**C** = 5	**D** = 2	
A$^-$ = 10	**B**$^-$ = 7	**C**$^-$ = 4	**D**$^-$ = 1	

If we assume that these newly assigned grade numbers are $\mathbf{y_1,y_2,\ldots,y_n}$, then we can use the following computational model:

$$\mathbf{Average2 = (y_1 + y_2 + y_3 + \ldots + y_n) / n}$$

where the result will be a number from 0 to 12.

As for the output, if we want it as a percentage, then we can use either **Average1** directly or use **(Average2 / 12)**, depending on the input that we had originally. If we wanted a letter grade as output, then we would have to use **(Average1/100*12)** or **(Average1*0.12)** or **Average2** and then map that to some kind of "lookup table" that allows us to look up a grade letter according to a number from 0 to 12.

Do you understand this step in the problems solving process ? It is all about figuring out how you will make use of the available data to compute an answer.

STEP 3: Develop an Algorithm:

Now that we understand the problem and have formulated a model, it is time to come up with a precise plan of what we want the computer to do.

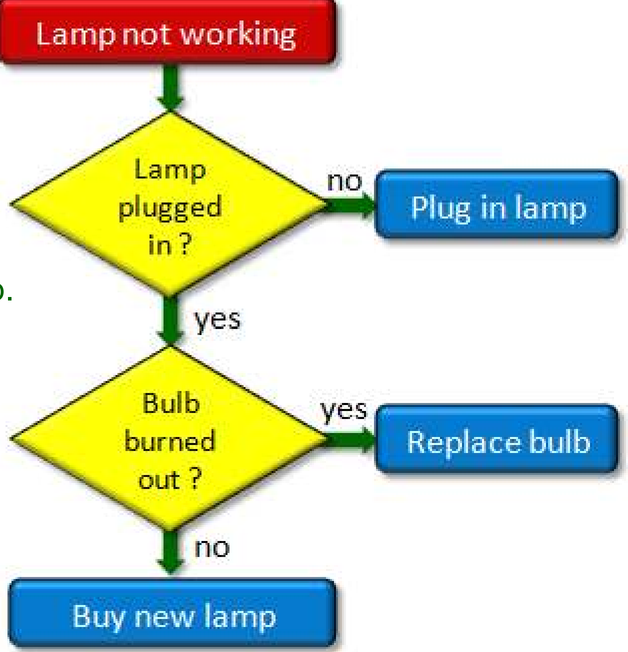
An algorithm is a precise sequence of instructions for solving a problem.

Some of the more complex algorithms may be considered "**randomized algorithms**" or "**non-deterministic algorithms**" where the instructions are not necessarily in sequence and in may not even have a finite number of instructions. However, the above definition will apply for all algorithms that we will discuss in this course.

To develop an algorithm, we need to represent the instructions in some way that is understandable to a person who is trying to figure out the steps involved. Two commonly used representations for an algorithm is by using (1) ***pseudo code***, or (2) ***flow charts***. Consider the following example (from Wikipedia) of solving the problem of a broken lamp. To the right is an example of a flow chart, while to the left is an example of pseudocode for solving the same problem:

Pseudo Code

1. IF lamp works, go to step 7.
2. Check if lamp is plugged in.
3. IF not plugged in, plug in lamp.
4. Check if bulb is burnt out.
5. IF blub is burnt, replace bulb.
6. IF lamp doesn't work buy new lamp.
7. Quit ... problem is solved.



Notice that:

> *pseudocode is a simple and concise sequence of English-like instructions to solve a problem.*

Pseudocode is often used as a way of describing a computer program to someone who doesn't understand how to program a computer. When learning to program, it is important to write pseudocode because it helps you clearly understand the problem that you are trying to solve. It also helps you avoid getting bogged down with syntax details (i.e., like spelling mistakes) when you write your program later (see step 4).

Although flowcharts can be visually appealing, pseudocode is often the preferred choice for algorithm development because:

- It can be difficult to draw a flowchart neatly, especially when mistakes are made.
- Pseudocode fits more easily on a page of paper.
- Pseudocode can be written in a way that is very close to real program code, making it easier later to write the program (i.e., in step 4).
- Pseudocode takes less time to write than drawing a flowchart.

Pseudocode will vary according to whoever writes it. That is, one person's pseudocode is often quite different from that of another person. However, there are some common control structures (i.e., features) that appear whenever we write pseudocode.

These are shown here along with some examples:

- **sequence**: listing instructions step by step in order (often numbered)

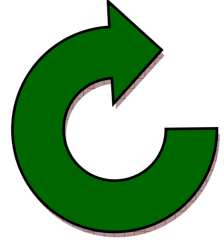
> **1.** Make sure switch is turned on
> **2.** Check if lamp is plugged in
> **3.** Check if bulb is burned out
> **4.** …

- **condition**: making a decision and doing one thing or something else depending on the outcome of the decision.

> **if** lamp is not plugged in
>   **then** plug it in

> **if** bulb is burned out
>   **then** replace bulb
> **otherwise** buy new lamp

- **repetition**: repeating something a fixed number of times or until some condition occurs.



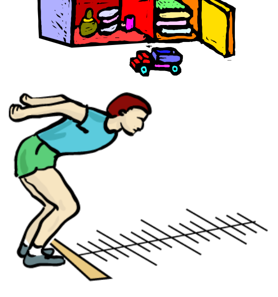
> **repeat**
>   get a new light bulb
>   put it in the lamp
> **until** lamp works or no more bulbs left

> **repeat** 3 **times**
>   unplug lamp
>   plug into different socket
> …

- **storage**: storing information for use in instructions further down the list

> x ← a new bulb
> count ← 8

- **jumping**: being able to jump to a specific step when needed



> if bulb works
>   then **goto step** 7

You will notice that the bold in the above examples highlights the specific control structure. You will notice that for the condition and repetition structures, the portion of the pseudocode that is part of the condition or the repeat loop are indented a bit so as to make it clear that

these are kinds “inner steps” that belong to that structure. Some people will use brackets to indicate what is in or out of a control structure as follows:

```
if (bulb is burned out) then {
  replace bulb
}
otherwise {
  buy new lamp
}
```

```
repeat {
   get a new light bulb
   put it in the lamp
} until (lamp works or no more bulbs left)
```

```
repeat 3 times {
   unplug lamp
   plug into different socket
}
```

The point is that there are a variety of ways to write pseudocode. The important thing to remember is that your algorithm should be clearly explained with no ambiguity as to what order your steps are performed in.

Whether using a flow chart or pseudocode, you should test your algorithm by manually going through the steps in your head to make sure that you did not forget a step or a special situation. Often, you will find a flaw in your algorithm because you forgot about a special situation that could arise. Only when you are convinced that your algorithm will solve your problem, should you go ahead to the next step.

Consider our previous example of finding the average of a set of **n** grades stored in a file. What would the pseudocode look like? Here is an example of what it might look like if we had the example of **n** numeric grades $x_1 \ldots x_n$ that were loaded from a file:

Algorithm: DisplayGrades

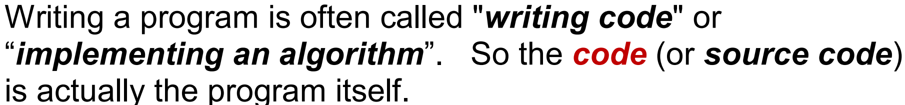
1. set the sum of the grade values to 0.
2. load all grades $x_1 \ldots x_n$ from file.
3. **repeat n times** {
4. get grade x_i
5. add x_i to the sum
   }
6. compute the average to be sum / **n**.
7. print the average.

It would be wise to run through the above algorithm with a real set of numbers. Each time we test an algorithm with a fixed set of input data, this is known as a ***test case***. You can create many test cases. Here are some to try:

> **n** = 5, x_1 = 92, x_2 = 37, x_3 = 43, x_4 = 12, x_5 = 71 … result should be **51**
> **n** = 3, x_1 = 1, x_2 = 1, x_3 = 1 .... result should be **1**
> **n** = 0 .... result should be **0**

STEP 4: Write the Program:

Now that we have a precise set of steps for solving the problem, most of the hard work has been done. We now have to transform the algorithm from step 3 into a set of instructions that can be understood by the computer.



Writing a program is often called "***writing code***" or "***implementing an algorithm***". So the ***code*** (or ***source code***) is actually the program itself.

Without much of an explanation, below is a program (written in processing) that implements our algorithm for finding the average of a set of grades. Notice that the code looks quite similar in structure, however, the processing code is less readable and seems somewhat more mathematical:

Pseudocode	Processing code (i.e., program)

1. set the sum of the grade values to 0.
2. load all grades x_1 **…** x_n from file.
3. **repeat n times** {
4.   get grade x_i
5.   add x_i to the sum
    }
6. compute the average to be sum / **n**.
7. print the average.
	`int sum = 0;`
	`byte[] x = loadBytes("numbers");`
	`for (int i=0; i<x.length; i++)`
	  `sum = sum + x[i];`
	
	`int avg = sum / x.length;`
	`print(avg);`

For now, we will not discuss the details of how to produce the above source code. In fact, the source code would vary depending on the programming language that was used. Learning a programming language may seem difficult at first, but it will become easier with practice.

The computer requires precise instructions in order to understand what you are asking it to do. For example, if you removed one of the semi-colon characters (**;**) from the program above, the computer would become confused as to what you are doing because the (;) is what it understands to be the end of an instruction. Leaving one of them off will cause your program to generate what is known as a ***compile error***.

> *Compiling is the process of converting a program into instructions that can be understood by the computer.*

The longer your program becomes, the more likely you will have multiple compile errors. You need to fix all such compile errors before continuing on to the next step.

STEP 5: Test the Program:

Once you have a program written that compiles, you need to make sure that it solves the problem that it was intended to solve and that the solutions are correct.

> *Running a program is the process of telling the computer to evaluate the compiled instructions.*

When you run your program, if all is well, you should see the correct output. It is possible however, that your program works correctly for some set of data input but not for all. If the output of your program is incorrect, it is possible that you did not convert your algorithm properly into a proper program. It is also possible that you did not produce a proper algorithm back in step 3 that handles all situations that could arise. Maybe you performed some instructions out of sequence. Whatever happened, such problems with your program are known as ***bugs***.

> *Bugs are problems/errors with a program that cause it to stop working or produce incorrect or undesirable results.*

You should fix as many bugs in your program as you can find. To find bugs effectively, you should test your program with many test cases (called a ***test suite***). It is also a good idea to have others test your program because they may think up situations or input data that you may never have thought of. The process of finding and fixing errors in your code is called ***debugging*** and it is often a very time-consuming "chore" when it comes to being a programmer. If you take your time to carefully follow problem solving steps 1 through 3, this should greatly reduce the amount of bugs in your programs and it should make debugging much easier.

STEP 6: Evaluate the Solution:

Once your program produces a result that seems correct, you need to re-consider the original problem and make sure that the answer is formatted into a proper solution to the problem. It is often the case that you realize that your program solution does not solve the problem the way that you wanted it to. You may realize that more steps are involved.

For example, if the result of your program is a long list of numbers, but your intent was to determine a pattern in the numbers or to identify some feature from the data, then simply producing a list of numbers may not suffice. There may be a need to display the information in a way that helps you visualize or interpret the results with respect to the problem. Perhaps a chart or graph is needed.

It is also possible that when you examine your results, you realize that you need additional data to fully solve the problem. Or, perhaps you need to adjust the results to solve the problem more efficiently (e.g., your game is too slow).

It is important to remember that the computer will only do what you told it to do. It is up to you to interpret the results in a meaningful way and determine whether or not it solves the original problem. It may be necessary to re-do some of the steps again, perhaps going as far back as step 1 again, if data was missing.

So there you have it. Those are the 6 steps that you should follow in order to solve problems using computers. Throughout the course, you should try to use this approach for all of your assignments. It is a good idea to practice problem solving to make sure that you understand the process. Below are some practice exercises that will help you practice the first 3 steps of the problem solving process. Later, you will gain experience with steps 4 through 6.

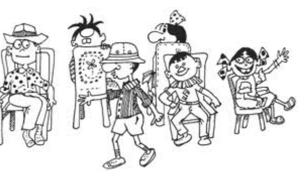
PRACTICE EXERCISES

Formulate a model and then develop an algorithm for each of the following problems. In each case, start with a simple algorithm and then try to think about situations that can realistically go wrong and make appropriate adjustments to the algorithm. Keep in mind that there is no "right" answer to these problems. Everyone will have a unique solution.

a. **Making a peanut butter and jam sandwich**

b. **Putting together a jigsaw puzzle**

c. **Playing the game of musical chairs**

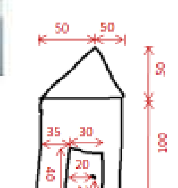


d. **Replacing a flat tire on your car**

e. **Getting home from school today**

f. **Emptying a case of drinks into your refrigerator**

g. **Drawing a house in processing**



1.4 Control Abstraction

When designing a program, it is not always clear as to how much detail should go into the algorithm. For example, consider that you are at home on the couch and you are thirsty. How do you solve this quench-thirsting problem ? Here is a simple algorithmic solution to your problem:

1. go to kitchen
2. open refrigerator
3. choose a drink
4. drink it

However, we could have come up with a more abstract (i.e., less detailed) algorithm:

1. get a drink
2. drink it.

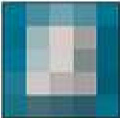
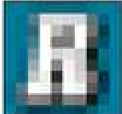
Or ... we could have been much more detailed:

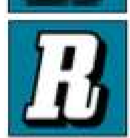
1. get off couch
2. walk to kitchen
3. open refrigerator
4. take carton of juice
5. close refrigerator
6. go to the cupboard
7. open cupboard
8. take a glass
9. close cupboard
10. pour juice into glass
11. go to refrigerator
12. open refrigerator
13. put carton in refrigerator
14. close refrigerator
15. drink glass of juice

Each of these algorithmic solutions solves the problem. So then, which one is best ? The answer is not always easy. As a rule of thumb, we want to produce the simplest possible algorithm that is easily understandable. While it is important to provide enough detail to be able to properly describe the problem solution, it is also important not to get hung up on too much detail so that the algorithm becomes cluttered and overly complicated.

So, likely, the first of the three algorithmic solutions here would suffice as an adequate solution to the problem. This idea of coming up with a clear algorithm without too many details is known as abstraction.

> *Abstraction is the process of reducing or factoring out details that are not necessary in order to describe an algorithm.*



Abstraction is important because it allows us to focus on a few concepts at a time. This allows us to get the "big picture" first in regards to the problem solution. We can then "fill in" the specific details at a later point in time. For example, in the above example, a statement like "**get a drink from the refrigerator**" would suffice when that step is part of a larger problem such as an algorithm that describes a person's daily routine around dinner time. However, if we needed to program such a step into a robot, then much more detail would be needed because the robot would require a more precise set of movements in order to carry out such an operation.

The analogy of image resolution can be used to describe abstraction. A low-resolution image has less detail and is more abstract than a high-resolution image. Abstraction allows us to lower the resolution of the image so as to save space (i.e., by hiding details) unless they are absolutely necessary.

We also see the idea of abstraction in 3D video game engines which hide details of objects further away as they are not necessary until the game character moves closer to those objects. It helps to speed up the game and reduce clutter while allowing the game player to focus on the more important objects nearby:

We have just been discussing a kind of abstraction that allows our algorithm steps to be either quite abstract (i.e., high level) or more detailed (i.e., low level). This kind of abstraction is known as ***control abstraction***.

When solving problems, it is often necessary to break the problem down into manageable steps. We do the same thing in real life so as to simplify the problem-solving process. For example, consider doing a jigsaw puzzle. If the jigsaw puzzle has many pieces (e.g., 1000), it can be an overwhelming problem to put it together. Some do not even enjoy such a task, yet others enjoy the challenge because they break it down into simpler, more easily-managed sub-problems.

A typical person may solve the puzzle as follows:

Algorithm: SolvePuzzle

1. pour out all pieces on the table
2. flip all pieces over to show the picture side
3. separate the edge pieces from the inside pieces
4. solve edge pieces
5. **repeat** until there are no more easily identifiable pieces **{**
6.   select pieces that form an easily identifiable part of the picture
7.   put these selected pieces together
8.   try to fit picture portion to border or to other picture portions

   }
9. **repeat** until no more pieces remain {
10.   pick piece up and try to fit it somewhere
11.   if its location is unclear, put piece back in pile

   }

Notice how the seemingly difficult problem is broken down into well-defined stages (or steps). By solving the edge pieces first, it gives the person a better idea as to the size of the whole image and it gives a feel for how the other parts of the puzzle will fit together.

In fact, steps 5 through 8 can be made into more specific steps if we had more information about the puzzle. For example, consider how you would solve this image:

Perhaps this would be your solution:

1. pour out all pieces on the table
2. flip all pieces over to show the picture side
3. separate the edge pieces from the inside pieces
4. solve edge pieces
5. solve the puppies
6. solve the sign
7. solve the grass part
8. solve the gate
9. solve the hay
10. solve the barn

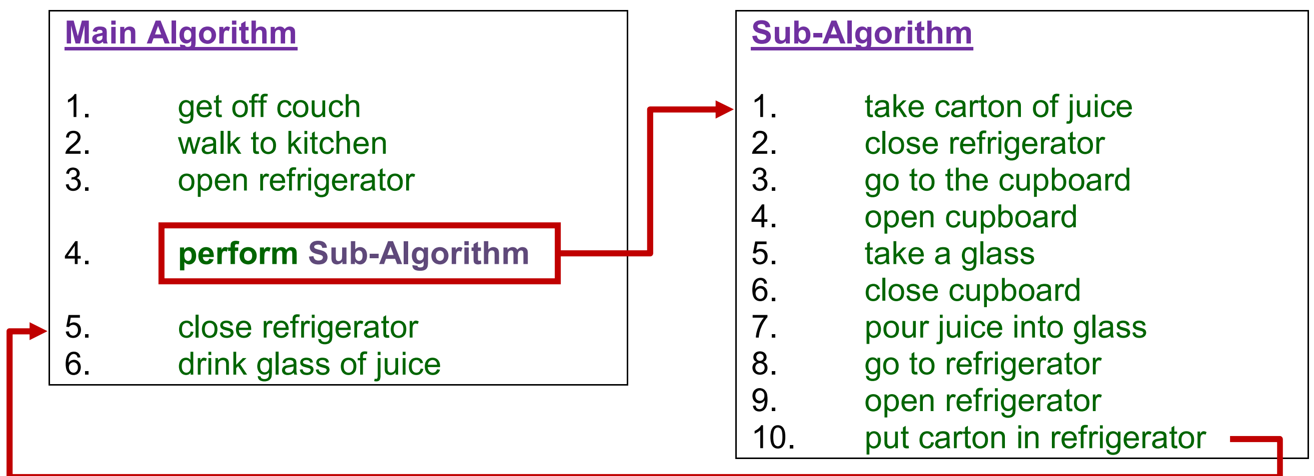
Notice how specific we can be now with our algorithm because we know the exact picture that we are trying to build. Of course, the order in which we solve the particular parts of the image is unimportant, but do you see how breaking a problem down into simpler, smaller procedures can make it easy ?

Programming computers should be done in the same way. Always break your problem down into simple pieces that you understand. Keep breaking it down until you have a simple problem that you can understand.

This strategy of breaking down the problem into smaller pieces is often called ***divide and conquer*** and it represents the fundamental principle for problem solving. It is not a difficult strategy. As mentioned, we do this every day naturally in real life. Even children can do this.

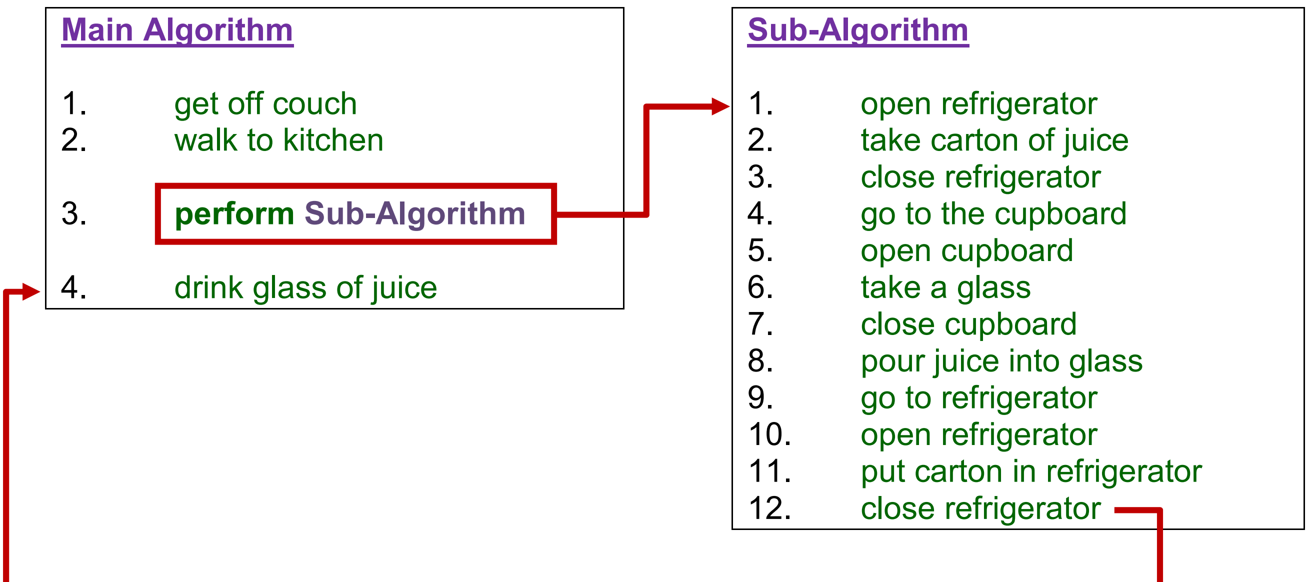
Often, in the cases where details are necessary, it is still possible to provide a higher-level algorithm, while allowing the more specific details to be described in a ***sub-algorithm*** or ***sub-program***. As a result, we are able to describe an algorithm at multiple layers of abstraction.

For example, in our thirst-quenching scenario, we could use the details of the most detailed algorithmic solution but hide the more detailed portions in a sub-algorithm as follows:

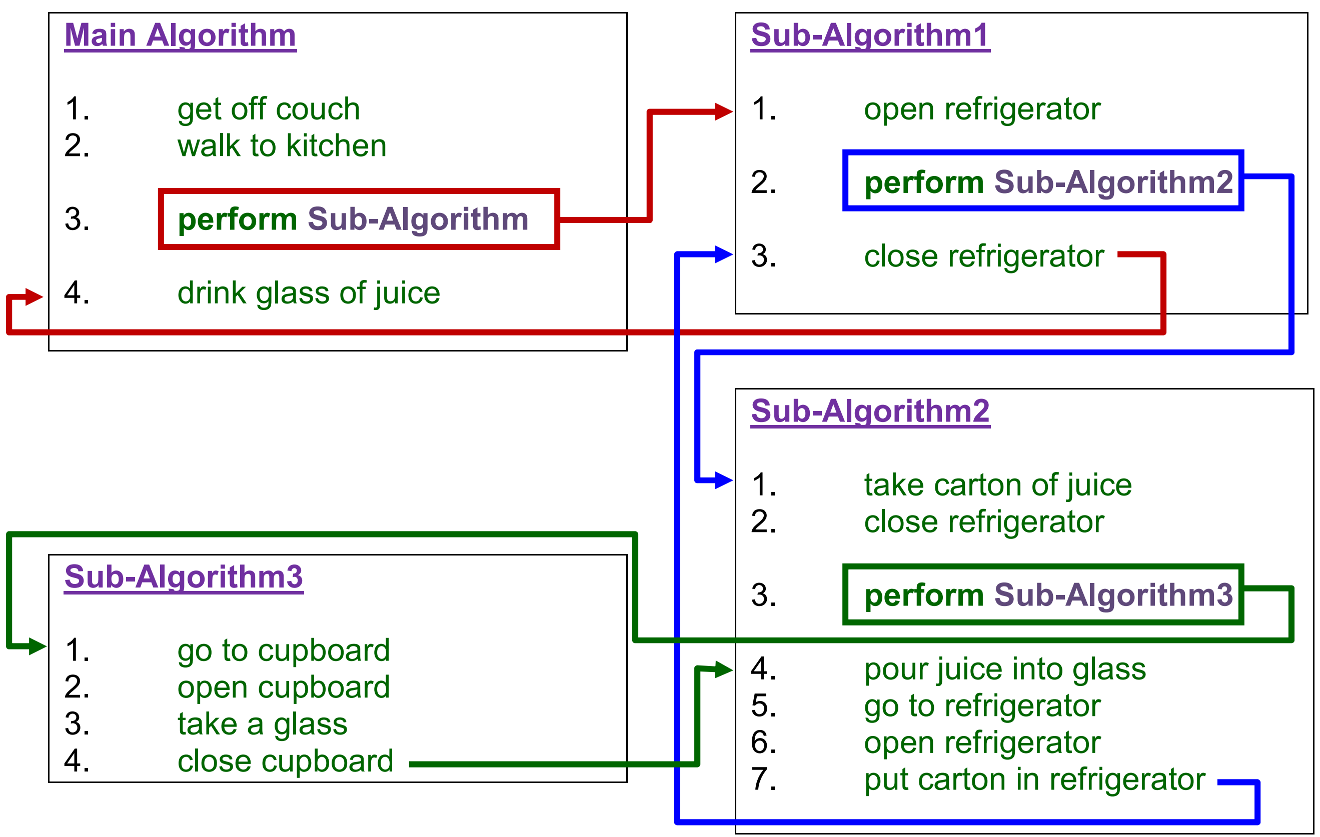


Notice how the main algorithm is quite simple now with much less detail. The **Sub-Algorithm** is not a ***stand-alone algorithm*** in that it cannot solve the problem on its own. It only solves part of the problem. For example, it assumes that the person is standing in front of the refrigerator and that the refrigerator door is open. We could also lessen these restrictions, for

example, by incorporating steps **3** and **5** of the **Main Algorithm** into the **Sub-Algorithm** and moving them out of the main algorithm as follows:

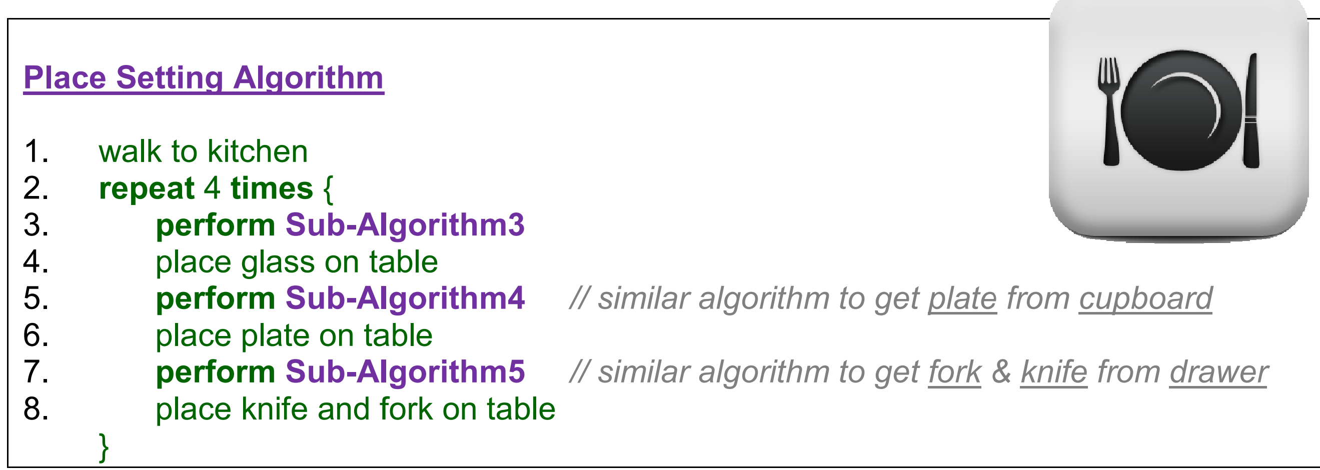


Now we have abstracted out a little further by hiding some details within the **Sub-Algorithm**. We could also perform additional layers of abstraction by abstracting within the **Sub-Algorithm**: to produce another **Sub-Algorithm2**. And we can even extract the code for getting a glass from the cupboard and place it in yet another **Sub-Algorithm3** as follows:



Notice how **Sub-Algorithm1** and **Sub-Algorithm2** became more abstract and easier to read. We can abstract out in this manner as often as we feel it to be necessary. When do we decide to stop this kind of abstraction ? Well, there is no fixed rule. However, when a sub-algorithm seems to be small enough to understand or when it relates to a well-defined real-life group of actions, then it is probably a good idea not to break it down any further.

Notice that **Sub-Algorithm3** is a nice simple function that gets a glass from the cupboard. We can use this smaller algorithm as a procedure (or module) in other algorithms, for example, if we wanted an algorithm to place a glass on the kitchen table as part of getting prepared for dinner guests:



Place Setting Algorithm

```
1.   walk to kitchen
2.   repeat 4 times {
3.        perform Sub-Algorithm3
4.        place glass on table
5.        perform Sub-Algorithm4     // similar algorithm to get plate from cupboard
6.        place plate on table
7.        perform Sub-Algorithm5     // similar algorithm to get fork & knife from drawer
8.        place knife and fork on table
     }
```

Hopefully you see now how practical and powerful abstraction can be in making algorithms simpler, more readable and ultimately more understandable.

In computer science we give a special name to the sub-algorithms. They are sometimes called ***modules***, ***functions*** or ***procedures***. In fact, it is not a good idea to simply number all the sub-algorithms but instead to give them meaningful names. As standard convention, when naming a function or procedure, you should use letters, numbers and underscore (i.e., **_**) characters but not any spaces or punctuation. Also, the first character in the name should be a lower case letter. If multiple words are used as the name, each word except the first should be capitalized. Lastly, we often use parentheses (i.e., **()**) after the function or procedure name to identify it as a sub-algorithm.

You should choose meaningful names that are not too long. Here are some meaningful names that we could use for our sub-algorithms:

SubAlgorithm1: **getADrink()** or **getADrinkFromRefrigerator()**
SubAlgorithm2: **pourDrink()** or **pourCartonDrink()**
SubAlgorithm3: **getGlass()** or **getGlassFromCupboard()**
SubAlgorithm4: **getPlate()** or **getPlateFromCupboard()**
SubAlgorithm5: **getUtensils()** or **getForkAndKnife()** or **getUtensilsFromDrawer()**

Interestingly, every step of the algorithm itself is a kind of function or procedure. That is, there are many hidden details in something as simple as getting off of a couch. It sounds straight forward, but imagine trying to program a robot to get off a couch ... there are many "ugly" details related to motor movements and balancing that make it a difficult problem in itself.

However, for the purposes of keeping our algorithm description at a high level, we can assume that these details are all hidden in a sophisticated module called **getOffCouch().** So, all of our code will become modular:

MainAlgorithm() {	getADrink() {	pourDrink() {	getAGlass() {

getOffCouch();
walkToKitchen();
getADrink();
drinkIt();
}
	openRefrigerator();
	pourDrink();
	closeRefrigerator();
	}
		takeCartonOfJuice();
		closeRefrigerator();
		getAGlass();
		pourJuiceIntoGlass();
		goToRefrigerator();
		openRefrigerator();
		putCartonInRefrigerator();
		}
			goToTheCupboard();
			openCupboard();
			takeGlass();
			closeCupboard();
			}

This is **close** to how Processing and JAVA organize code into modules. Notice that each module has its code encapsulated by the brace characters **{ }**. Also, each function call has a semi-colon after it. Although this is close, it is not quite JAVA code yet.

The place-setting algorithm would look like this:

```
MainAlgorithm() {
    walkToKitchen();
    repeat 4 times {                // This is not proper Processing or JAVA code
        getAGlass();
        placeGlassOnTable();
        getAPlate();
        placePlateOnTable();
        getUtensils();
        placeKnifeAndForkOnTable();
    }
}
```

Notice in the above code that we are getting a glass, a plate and a pair of utensils and we are placing these items on the table. Hence each of these sub-algorithms go off and come back (i.e., return) with some kind of object (i.e., drink, glass, plate or utensils). When a sub-algorithm comes back with some kind of object (such as the glass or plate) or value (such as a numerical result), we call the sub-algorithm a ***function***. If the sub-algorithm does not return any particular value, it is instead known as a ***procedure***.

An example of a procedure would be **walkToKitchen()** because although it does perform some actions, it does not come back with some kind of resulting object or value. It simply gets the person (or robot) to the kitchen and has no value to return.

1.5 Simple Procedures in Processing

As we just discussed, it is often a good idea to simplify our overall algorithm by making it higher-level. That means, we could hide details that are unnecessary. For example, if we wanted to create a home scenery, it may make sense to develop a high-level algorithm like this:

```
MainAlgorithm() {
    drawHouse();
    drawLaneway();
    drawCar();
    drawLawn();
    drawTrees();
}
```

This is an easily understood algorithm which hides all the unnecessary details of "how" to draw the various things. Recall our program for drawing a simple house:

```
size(300,300);

// Draw the house
rect(100,200,100,100);
triangle(100,200,150,150,200,200);
rect(135,260,30,40);
point(155,280);
```

How could we abstract out and make this a higher-level algorithm ? We could create a **drawHouse()** procedure that will draw the house. Then our program would like simpler like this:

```
size(300,300);
drawHouse();
```

What would the **drawHouse()** procedure look like ? Well in Processing and JAVA, this is the format for declaring a simple procedure:

```
void procedureName()  {
     // Write your procedure's code here
}
```

Notice that procedure has **void** at the front. This indicates that there is no value to be returned from the procedure.

The brace characters (i.e., **{ }**) indicate the code's body:

*The **body** of a function or procedure is the code that is evaluated each time that the function or procedure is called.*

So then, our **drawHouse()** procedure would look as follows:

```
size(300,300);
drawHouse();

void drawHouse() {
     rect(100,200,100,100);
     triangle(100,200,150,150,200,200);
     rect(135,260,30,40);
     point(155,280);
}
```

The code itself looks more complicated because nothing looks hidden at all! Actually, the above code, by itself, will not compile in Processing. In general, when writing a program in Processing, all of our program code must lie within a function or procedure of some kind. There are exceptions to this:

- we may declare "global variables" (discussed in the next chapter) outside a function at the top of our program
- when we are not creating any functions or procedures of our own, Processing will allow us to write a simple sequence of code that does not need to be inside a function or procedure.

So, Processing has provided a useful procedure called **setup()** into which we can place our code. The **setup()** procedure is called one time automatically by Processing whenever we start or restart the program. Here is code that will now compile and run in Processing:

```
void setup() {
     size(300,300);
     drawHouse();
}

void drawHouse() {
     rect(100,200,100,100);
     triangle(100,200,150,150,200,200);
     rect(135,260,30,40);
     point(155,280);
}
```

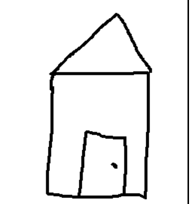
Notice that the code is simply made up of two procedures which perform various sub-procedures to do the actual drawing. Again, there seems to be more code, but notice how the main algorithm (i.e., the code in the **setup()** procedure) is much simpler now.

As a side point, in JAVA, C, C#, C++ and similar languages, there is a procedure similar to **setup()** called **main()** … which is called automatically upon program startup. You will see more of this in the follow-up course.

1.6 Algorithm Efficiency

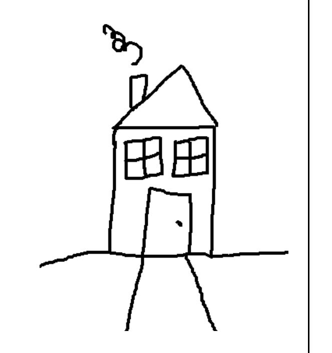
Consider an algorithm for drawing a house. Since the problems is a little vague, there are many potential solutions. Here is one of them:

Algorithm1: DrawSimpleHouse
1. draw a square frame
2. draw a triangular roof
3. draw a door



Obviously, we could have made a more elaborate house, but the solution above solves the original problem. What if this was our solution:

Algorithm2: DrawMoreComplexHouse
1. draw a square frame
2. draw a triangular roof
3. draw a door
4. draw windows
5. draw chimney
6. draw smoke
7. draw land
8. draw path to door



Which is a "**better**" solution ? That's not an easy question to answer. It depends on what "better" means. If time is of the essence (e.g., as in playing a game of Pictionary) then **Algorithm1** would be better because it can be drawn faster. The more elaborate house of **Algorithm2** would perhaps be "better" if visual appearance was the aim, as opposed to speed of drawing. This example brings up an important topic in computer science called *algorithm efficiency*.

> ***Algorithm efficiency*** *is used to describe properties of an algorithm relating to how much of various types of resources it consumes. (Wikipedia)*

Normally in computer science we are interested in algorithms that are **time** and **space** efficient, although there are also other ways (i.e., metrics) for measuring efficiency. For example, **Algorithm1** is more efficient in terms of **time** but it is also more efficient in terms of **ink or pencil usage** as well as the amount of **space that it takes** on the paper. **Algorithm2** may be more efficient in terms of **detailing** in that, depending on the context, it may take

longer for a person to guess what the drawing is (i.e., it could be confused with a barn, shed or dog house if this was drawn in a farm setting). In this case, the extra time taken to distinguish the house through the drawing of the windows and chimney may result in a quicker guess.

> *The **runtime complexity** (a.k.a. **running time**) of an algorithm is the amount of time that it takes to complete once it has begun.*

> *The **space complexity** of an algorithm is the amount of storage space that it requires while running from start to completion.*

Recall the following algorithm for setting a table:

```
MainAlgorithm() {
    walkToKitchen();
    repeat 4 times {              // This is not proper Processing or JAVA code
        getAGlass();
        placeGlassOnTable();
        getAPlate();
        placePlateOnTable();
        getUtensils();
        placeKnifeAndForkOnTable();
    }
}
```

Why is this not an efficient real-world solution ?

It is inefficient in that it requires a lot of unnecessary travelling back and forth to the cupboard and table because it gets one glass at a time.

While this may be a safer solution for a small child, an adult would likely grab all 4 glasses at once as well as the plates and utensils.

Here is a different (yet similar) algorithm:

```
MainAlgorithm() {
    walkToKitchen();
    getAllGlasses();
    placeGlassesOnTable();
    getAllPlates();
    placePlatesOnTable();
    getAllUtensils();
    placeUtensilsOnTable();
}
```

Notice that there is no longer a need for a “repeat loop” since we are getting all the glasses at once, all the plates at once and all the utensils at once.

We can actually generalize the algorithm to set the table for as many guests as we want by supplying some additional information in our functions.

A ***parameter*** *is a piece of data provided as input to a function or procedure*.

We can supply an arbitrary number in our algorithm to specify how many place settings to set as follows:

```
MainAlgorithm() {
    walkToKitchen();
    getGlasses(8);
    placeGlassesOnTable();
    getPlates(8);
    placePlatesOnTable();
    getUtensils(8);
    placeUtensilsOnTable();
}
```

Notice that we supplied a number **8** between the parentheses of our function. This is where we normally supply additional information (i.e., parameters) to our functions. Now our function is clear as to how many place settings will be made, whereas the previous algorithm was not clear.

But what would the **getGlasses()** function now look like ? Here was the 1-glass version:

```
getAGlass() {
    goToCupboard();
    openCupboard();
    takeAGlass();
    closeCupboard();
}
```

Now we need to specify the parameter for the function and use it within the function itself:

```
getGlasses(n) {
    goToCupboard();
    openCupboard();
    repeat n times {
        takeAGlass();
    }
    closeCupboard();
}
```

Notice how the parameter is now being used within the function to get the necessary glasses. The value of **n** will vary according to how we call the function. For example, if we use **getGlasses(8)**, then within the function, **n** will have the value of **8**. If we use **getGlasses(4)**, then within the function, **n** will have the value of **4**. So, the value for parameter **n** will always be the number that was passed in when the function was called.

For algorithms that are the most general, we often use the letter **n** as a kind of "placeholder" or "label" to indicate that we want the algorithm to work for any number from **0** to **n**. The "n" itself is not a special letter, it is just commonly used. So, the statement **getGlasses(n)** is indicating "get **n** glasses", where **n** may be any integer number that we want.

Obviously, there is a limit as to how many glasses a person could carry. However, to describe an algorithm in a very general way, we use **n** to indicate that our algorithm will work for any number from **0** to **n**. Using **n** instead of a fixed number, also allows us to compare two algorithms in regards to their efficiency. That is, we can often compare the number of steps that one algorithm requires with another algorithm.

For example, consider these two algorithms for setting the table for **n** people:

```
AlgorithmA() {
    walkToKitchen();
    repeat n times {
        getAGlass();
        placeGlassOnTable();
        getAPlate();
        placePlateOnTable();
        getUtensils();
        placeKnifeAndForkOnTable();
    }
}
```

```
AlgorithmB() {
    walkToKitchen();
    getGlasses(n);
    placeGlassesOnTable();
    getPlates(n);
    placePlatesOnTable();
    getUtensils(n);
    placeUtensilsOnTable();
}
```

What if we defined "efficiency" in this example to refer to the "***number of times we walked back and forth between the cupboard and the table***" ? Which algorithm is more efficient ? Well each time through the loop, **AlgorithmA** makes **3** trips between the cupboard and table. Since there are **n** place settings (i.e., **n** times through the loop), then the whole algorithm takes **n** x **3**, or **3n**, steps. What about **AlgorithmB** ? It takes only **3** trips between the cupboard and table altogether, regardless of how many place settings will be required.

So what can we conclude ? If we are setting a place for **1** person, either algorithm is good. If setting for **2** people, then **Algorithm B** is ***twice*** more efficient than **Algorithm A** since it requires ***half*** the travel between the cupboard and table. As **n** gets larger, the difference becomes more significant. For example, if we are setting the table for **8** people, then **Algorithm A** uses **8 times** (total of **24**) more trips than **Algorithm B** (which takes 3 trips).

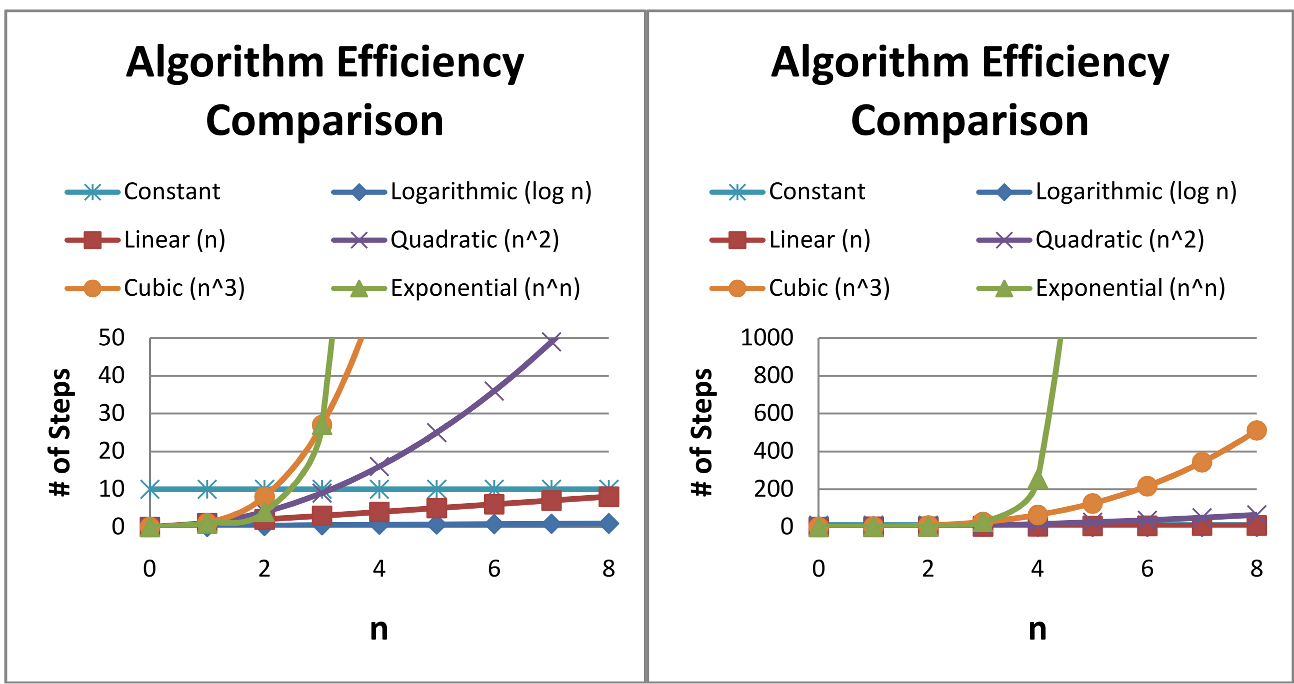
Regardless of the number of place settings, **Algorithm B** has a fixed cost of **3** (in regards to back and forth travels). Since this cost is fixed, we say that the algorithm has ***constant*** efficiency in terms of our particular cost metric.

In contrast, **Algorithm A** is said to be ***linear*** in that the efficiency grows equally with respect to the value of **n**. Sometimes an algorithm has a constant value times **n** (e.g., **3n**). Since the **3** is constant (i.e., fixed in our case, because we have exactly **3** kinds of items that we are placing), the algorithm is still considered to be linear.

If we were to vary the **3** items to be **n** items (e.g., place 8 items at each of the 8 people's place settings, or 12 items at each of the 12 person's place settings), then we would end up with an **n** x **n** (or n^2) algorithm which is called ***quadratic***.

Other common algorithm efficiency measures are ***logarithmic*** (i.e., $\log_2 n$), ***cubic*** (i.e., n^3) and ***exponential*** (i.e., n^n) … just to name a few.

Here are two graphs comparing various algorithm efficiencies as the value of **n** grows (graphs shown at two different scales):



Notice that the ***logarithmic***, ***constant*** and ***linear*** algorithms are insignificant when compared to the ***quadratic***, ***cubic*** and (especially) ***exponential*** algorithms. You may notice as well that the ***linear*** algorithm eventually passes the ***constant*** algorithm for larger values of **n**.

You may also notice that for very small values of **n**, the efficiency is generally not a big factor but that the efficiency can quickly become an issue for larger values. ***Exponential*** algorithms, for example, are usually unreasonable (i.e., useless) in practice except for very small values of **n**.

Logarithmic solutions are often preferred since they are significantly more efficient than even linear algorithms. For example, if **n** is **1,000,000** then a ***linear*** algorithm can take **1,000,000** steps whereas a ***logarithmic*** algorithm may take only **20** steps.

Sometimes it is hard to think in terms of an unknown number **n** because we are used to working with actual concrete numbers.

For example, assume that 16 players entered a one-on-one tennis tournament. If there can be no tie games how many games must be played if each player can be eliminated by one loss ?

We can figure this out with a simple table, doing a round-by-round count:

Round	Players Remaining	Games Played
1	**16** (nobody played yet)	**8**
2	**8** (winners from round 1)	**4**
3	**4** (winners from round 2)	**2**
4	**2** (winners from round 3)	**1**
		15 (total)

That was easy enough to figure out. But what if we had **n** players (assuming that **n** is an even number) ? What would the table look like ?

Round	Players Remaining	Games Played
1	**n**	**n/2**
2	**n/2**	**n/4**
3	**n/4**	**n/8**
4	**n/8**	**n/16**
5	**n/16**	**n/32**
…	**…**	**…**
…	**2**	**1**
		??? (total)

The answer seems to be (**n/2 + n/4 + n/8 + n/16 + n/32 + …**). But how far does that series of numbers go on ? We can try various values of **n** to find out. For example, if **n=100:**

(100/2 + 100/4 + 100/8 + 100/16 + 100/32 + …) = (50 + 25 + 12.5 + 6.25 + 3.125 + 1.5625 ...)

Hmmm…seems like the values approach 0 but never quite there. Try **n=128**:

(128/2 + 128/4 + 128/8 + 128/16 + 128/32 + …) = (64 + 32 + 16 + 8 + 4 + 2 + 1) = **127**

That one was easier. In fact, if you were to try various value of **n**, you would notice that the games placed would total **(n-1).** It works out evenly for games that are powers of 2 (e.g., n=2, 4, 8, 16, 32, 64, 128, 256, 510, 1024, etc). In fact, the number of rounds played is **$\log_2$(n)** (often written as just **log(n)** in computer science).

So, as you can see, trying to figure out efficiency with an unknown number is not always easy. Nevertheless, it is important for you as a computer scientist to understand how to write efficient algorithms that run faster, take up less computer memory or use less resources. We will briefly discuss efficiency at various times throughout this course. However, you will take further courses to investigate algorithmic efficiency much more thoroughly.