The Beauty and Joy of Computing[1]

Lab Exercise 6: Experimenting with Algorithms

Objectives

By completing this lab exercise, you should learn to

- Describe and implement basic searching and sorting algorithms;
- Use the BYOB timer to measure the running time of a script;
- Instrument code to count how many times specific operations occur; and
- Understand the difference between linear and quadratic time complexity, and its impact on the speed of a program.

Background (Pre-Lab Reading)

You should read this section before coming to the lab. It describes how various things work in BYOB and provides pictures to show what they look like in BYOB. You only need to read this material to get familiar with it. There is no need to actually do the actions described in this section, but you certainly can do them if it would help you follow along with the examples.

When computers deal with lots of data, the data should be organized in such a way that it can be worked with efficiently. In the the last lab, we introduced the idea of data structures, with the list being the simplest data structure. The two most common things we do using a list are *searching* (finding something in the list) and *sorting* (ordering the list by some rule), and while these might seem like simple things to do there are in fact entire books that have been written on just searching and sorting data in a computer.[2] In this lab we'll look at a few ways of searching and sorting data in a list, and in the process we will explore concepts of algorithms and algorithm time complexity.

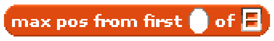
Searching in a List

First consider the problem of searching a list. Recall that *problems* are defined by input/output relationships, so when we talk about the problem of searching a list, we're not implying any particular *algorithm* for doing that. In the last lab, you actually wrote two different searching functions: one to search a list for a particular value (your case-insensitive "contains" predicate, for searching for a player's name), and one to search a list for the largest value (your high score reporter block). We are going to concentrate on the latter block in this lab, and modify it so it is useful in a sorting procedure. In particular, let's make it a general-purpose "max" block (in other words, don't refer to "high scores"), with two differences from a simple block from Lab 5: first, it will report the *position* of the maximum item rather than the value of the maximum item, and second it will only look at some *prefix of the list*, where a prefix is some number of elements at the beginning of the list, like the first 10 elements. The number of elements to consider will be a

[1] *Lab Exercises for "The Beauty and Joy of Computing" [This is the final version for the Fall 2012 class]*
Copyright © 2012 by Stephen R. Tate - Creative Commons License
See http://www.uncg.edu/cmp/faculty/srtate/csc100labs for more information

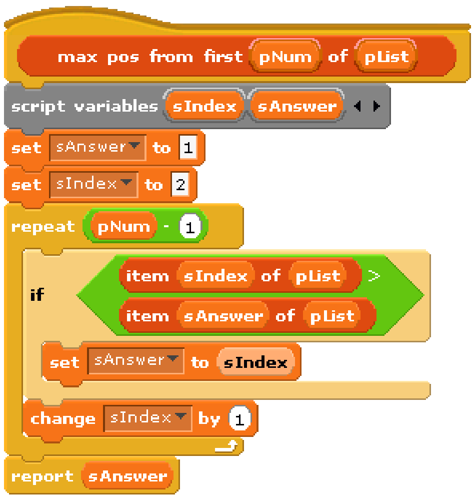
[2] See, for example, *The Art of Computer Programming, Volume 3: Sorting and Searching*, by Donald Knuth (Addison-Wesley, 1998), or *Sorting and Searching* by Kurt Melhorn (Springer-Verlag, 1984).

parameter along with the list, and we would like to be able to say “give me the position of the maximum element from the first *k* elements of this list,” resulting in a block that looks like this:



```
max pos from first ( ) of [ ]
```

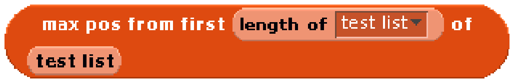
To implement this block, the algorithm that we will use simply scans through the part of the list that is indicated, and will look at each element comparing it with the largest it has seen so far. This is similar to the list index iterator pattern from the last lab, with a couple of changes: we will initialize the answer variable to the first element in the list rather than to a constant value, and we will repeat based on the prefix size parameter rather than the full length of the list. The result is the following definition:



```
max pos from first pNum of pList
script variables sIndex sAnswer
set sAnswer to 1
set sIndex to 2
repeat (pNum - 1)
    if (item sIndex of pList) > (item sAnswer of pList)
        set sAnswer to sIndex
    change sIndex by 1
report sAnswer
```

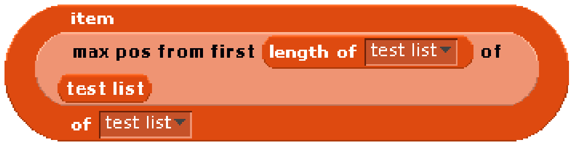
Note how skipping past the first item in the list before starting the loop requires us to subtract 1 from the number of loop iterations, but it means we didn’t have to make any assumptions about the data in the list (in the previous lab, we assumed that all scores were non-negative integers - you might want to look back at your solution and see how that assumption affected the algorithm, and consider what would happen if that assumption were violated).

This solution is a more general solution to finding the maximum value in a list than what we’ve done previously, and it is useful even in the more restricted cases that we considered before. For example, if we want to find the position of the maximum value in the entire list (rather than just a prefix) we could use:



```
max pos from first (length of test list) of test list
```

If we wanted to actual maximum value (rather than the position) we could use:



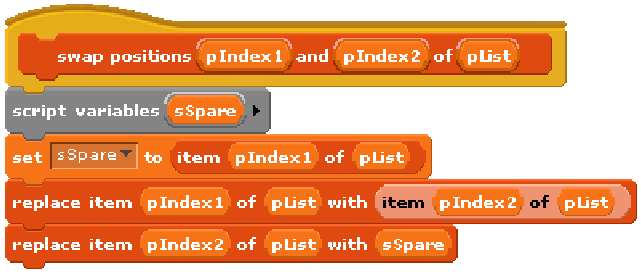
```
item (max pos from first (length of test list) of test list) of test list
```

These two additional problems (maximum position in the entire list and maximum value in the entire list) were implemented using just a few operations in addition to our “max pos from...” block, so using the terminology we introduced in Lab 3 we could say that we reduced the problem of finding the maximum value to the “max pos from...” problem. Notice that reductions are between problems, not between algorithms or programs or scripts - it relates problems in a fundamental way, and applies no matter how those problems are solved with particular algorithms.

Let's consider the time complexity of our "max pos from..." block. The first thing to notice is that the repeat loop iterates roughly pNum times[3] and pNum can be as large as the length of the list. Let's consider just that longest case right now, and use a variable *n* to denote the length of the list as is common practice in computer science. Looking at the definition, everything is constant time except for the fact that we use a repeat loop, which repeats some constant time operations roughly *n* times. As a result, the time is no worse than a constant times *n*, which we call *linear time*. The important take-away lessons from this section of the reading is that we have designed a very general-purpose maximum finding function, which has linear time complexity.

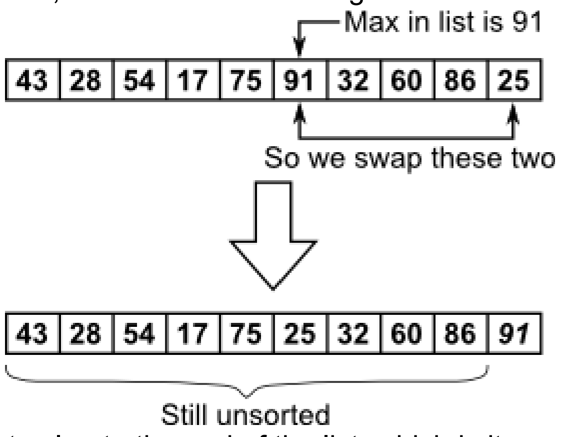
Sorting

In this section, we will use the "max pos from..." block that we described in the previous section to make a function that sorts a list, meaning that it rearranges the list so that it has the same elements but they are ordered from smallest to largest. Let's think about how we might use our "max pos from..." block to sort a list. If we used this to find the largest value in the list, we know where it should go in the final sorted list: at the end. We can't just replace the item at the end of the list with the largest item, because that would overwrite and destroy that item - remember, we just want to rearrange the list, while keeping all of the data. So rather than replacing the item at the end of the list we swap it with the maximum one that we found. We can use the "swap" block that we wrote in the last lab, and a solution for that problem looks like this:



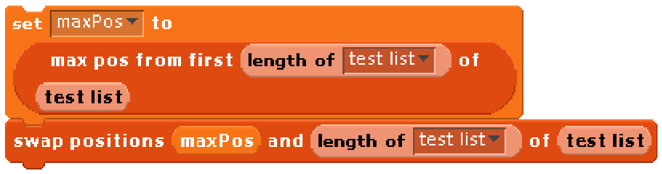
```
swap positions (pIndex1) and (pIndex2) of (pList)
script variables (sSpare)
set sSpare to item (pIndex1) of (pList)
replace item (pIndex1) of (pList) with item (pIndex2) of (pList)
replace item (pIndex2) of (pList) with (sSpare)
```

Looking at an example of a list, this transformation might look something like this:

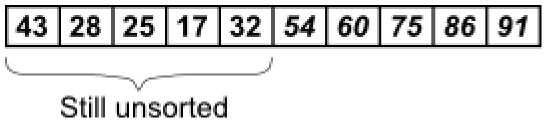


While we moved the largest value to the end of the list, which is its correct final position, we still don't know anything about the rest of the list, so everything except the last item is still labeled as "still unsorted." If we were to make BYOB blocks for just this sequence of operations, we first find the maximum of the entire list, and then swap it into the last position, giving:

[3] "roughly" because it's really pNum-1, but that doesn't make a significant difference in time complexity.

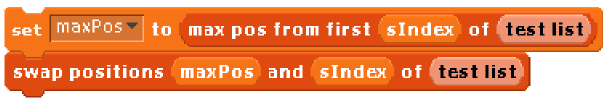

```
set maxPos to (max pos from first (length of test list) of test list)
swap positions maxPos and (length of test list) of test list
```

Next, consider repeating this process. Forget about the very last item in the list, since it's in the right place, so we'll look at the *n*-1 remaining unsorted items. We find the max in the first *n*-1 items, then swap that to place *n*-1. In the case of our example, the max of the remaining items is 86 - it's already in the right place, so now what? The interesting thing is that we don't need to worry about that at all. If you look at the way the "swap" block works, we can swap a position with itself without any harm, so we just do the same thing as above. Then we repeat this for position *n*-2, and then position *n*-3, and so on. After five rounds of this our example list looks like this:

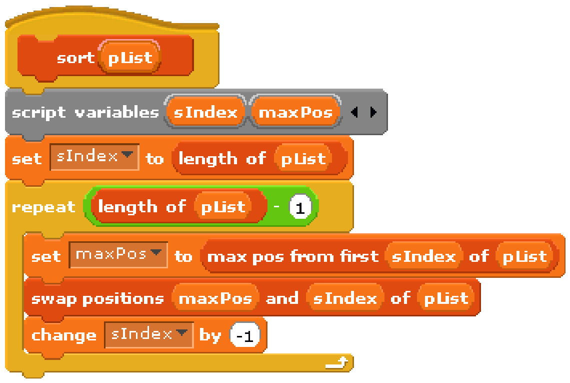


43	28	25	17	32	54	60	75	86	91

Generalizing what we're doing to an arbitrary position, indicated by variable sIndex, we have the following two block sequence that we want to evaluate for values of sIndex that start at *n*, then *n*-1, *n*-2, and so on:


```
set maxPos to max pos from first sIndex of test list
swap positions maxPos and sIndex of test list
```

When can we stop? It turns out the last sIndex value we have to consider is 2, since once the second item is in place the first one is the only one left, and it must be in the right place. You should recognize this pattern: We're counting down indices, so this is just the list index iterator pattern, with a decreasing index. Pulling all the pieces together, we get the following definition for a "sort" block:

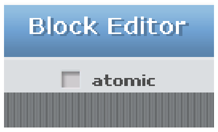

```
sort pList
script variables sIndex maxPos
set sIndex to length of pList
repeat ((length of pList) - 1)
    set maxPos to max pos from first sIndex of pList
    swap positions maxPos and sIndex of pList
    change sIndex by -1
```

This is the most complex script we've done so far - make sure you study this and are comfortable with how and why it works, because you'll be working with this in the lab activities.

Considering the time complexity of this sorting algorithm, the main thing to observe is that the sorting algorithm repeats roughly *n* times, and inside the loop it does a "max" operation which can take as many as *n* steps. The total number of operations is therefore something like *n* times *n*, so the total time is some constant times n^2, which is what we call *quadratic time*. A careful analysis of this is in fact a little more complex, but the end result is still the same: this is a quadratic time complexity.

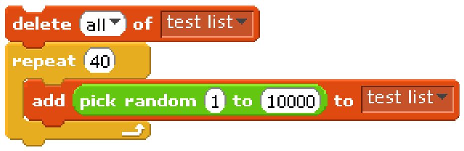
Timing BYOB Scripts

We're interested in how fast our algorithms and scripts are, so we'll next look at how to experiment with BYOB scripts in order to determine various measures of time and speed. Before we get into the timing details, we need to address one point in the "Block Editor" - when you were first exposed to the process for defining blocks in Lab 3, the "atomic" checkbox at the top of the Block Editor was pointed out and you were told that you could safely ignore that for the time being. However, this checkbox *does* have a very big effect on timing, as we'll describe in the "Discussion" section below. For now, it is very important that you make sure that all blocks that you define do _not_ have the "atomic" option checked. So when you build these blocks for the activities below, go through each definition - for "sort", "max pos from...", and "swap positions..." - and make sure the "atomic" option is unchecked, like this:



Without fully describing what this does now, just be aware that this will make your scripts slower (*much* slower), but that's OK for this lab.

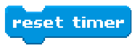
In order to experiment with the efficiency of your scripts, you will need to be able to generate _test data_ to use. For sorting, the simplest form of test data would be a list of a certain size that contains random values in some range. What I would recommend for this lab activity is that you create a list named "test list" and build the following small script in the scripts tab of your current sprite:



```
delete all of test list
repeat 40
    add pick random 1 to 10000 to test list
```

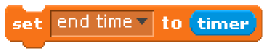
By having this sitting in the scripts area, you can easily set the list size to whatever you want (it is 40 in the example above) and click the script, which will create a list that you can use for testing. This is just a slightly more involved version of what we did in Lab 2, where we left a "go to …" block in the scripts area to easily reset Alonzo's position on the stage - in this case we're "resetting" our test list to a random list of values.

To actually time scripts, BYOB has a built-in _timer_ which can be used. This timer is constantly running, and the timer can be read or reset to zero using blocks that are in the "sensing" category. To reset the counter to zero, use the following command block:



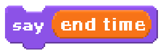
```
reset timer
```

There is no way to stop the timer, but you can copy the current value out to a variable, which will save the timer value at a certain time (similar to taking a lap time on a stopwatch). Therefore, to time a script you can put the "reset timer" block at the top of the script, and a block like



```
set end time to timer
```

at the bottom of the script. You can set up the "end time" variable as a watch variable so you can see what the value is, although you get more digits of precision if you have your sprite "say" the end time, like this:



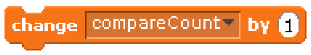
Make sure you enclose only the operations you want to time between the “reset timer” and “set end time ..” blocks - in particular, if you have any set-up operations that initialize an input array or something similar, do not put those in between the timing start/end measurements or else your times will be larger than they should.

Counting Operations

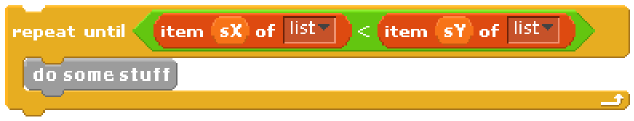
Using the BYOB timer to measure the speed of programs and scripts can be interesting, but if you’re really trying to understand the efficiency of an algorithm then this does not provide the information that you need. Why? Because there are many things that go into a measured time that are not properties of the algorithm - things like speed of the computer, other operations taking place on the computer, programming language or environment, etc.

In general, when we analyze the time complexity of an algorithm, we consider the number of *steps* that the algorithm takes, where a step is some basic operation like an addition, multiplication, comparison, etc. It has become standard practice in searching and sorting algorithms to count the number of *comparisons* that are made between data elements, since that is the core operation that drives all data movement and algorithm decisions. To get the best understanding of how many steps an algorithm takes, it is best to do this by analyzing the algorithm mathematically. This way, you can consider the efficiency of an algorithm before taking the effort to implement and test the algorithm, and you know you are looking at the inherent complexity of the algorithm rather than some side-effect of the way your implementation is running. Consider a situation in which you have to write a complicated program, and you think of three different algorithms to solve one of the core problems in this program: a little time up front analyzing the algorithms can give you a good indication of which one will be best and you can spend your time implementing that one algorithm rather than having to implement and test all three. This kind of analysis is beyond the scope of this class, although you are expected to start recognizing certain “patterns” of algorithms and understanding the corresponding time complexity of algorithms that fit each pattern. If you take further computer science courses, you will encounter more advanced analysis as a recurring topic, and you should get pretty good at it after a few courses! For this particular lab we’ll take an experimental approach, where you *instrument the code* that you are testing - instrumenting code refers to adding additional program statements or blocks that give you information about how the program is running, even though those code additions do not help in computing the answer that you’re after. In this case, our *instrumentation* will count how many operations the code performs while it is running.

To instrument a searching or sorting algorithm to count the number of comparisons, we define a global variable named “compareCount”, set it to zero before we start the script that we are testing, and add the block

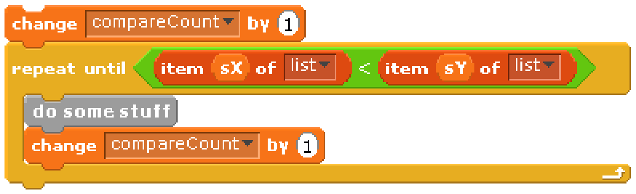


to be executed every time a comparison is made. For the most part, deciding where to put these change blocks is easy: find every comparison that is made in your script, and add this block right on top of the block that makes the comparison. This works fine for sequences of blocks and for comparisons that are made in “if” blocks, but there is one tricky case: a repeat block in which there is a condition that makes a comparison:



```
repeat until < item sX of list < item sY of list >
    do some stuff
```

In this case a comparison is made for every iteration of the loop (actually, if the loop iterates *n* times there will be *n*+1 comparisons - do you see why?). If we put the “change” block right before the “repeat” block that includes the comparison then it will only get executed once, not the *n*+1 times needed to reflect the number of comparisons made by this loop. While the basic idea of incrementing the counter before each comparison is good, the position in the script that is executed before the comparison isn’t as clear as in other cases - in this case, the first time we enter the loop we execute whatever is on top of the “repeat” block right before the comparison, and in the case of the loop causing an iteration, the block that is executed immediately before the comparison is the block at the *bottom* of the script inside the repeat C-block! If we put a change block in both of those places, like



```
change compareCount by 1
repeat until < item sX of list < item sY of list >
    do some stuff
    change compareCount by 1
```

then we are correctly counting the number of comparisons.

The main concepts you should have gotten out of this pre-lab reading are the following: You should understand how the simple searching and sorting scripts given above work. You should understand how to use the BYOB timer to time scripts. And finally, you should understand how to instrument code to count operations - in this case, we are counting comparisons.

Activities (In-Lab Work)

Activity 1: For the first activity, build and test the scripts described in the pre-lab reading: the “max pos of …” block, the “swap positions …” block, and the “sort …” block. To test, create the random list builder script described in the section “Timing BYOB Scripts”, and generate a random list of size 10. Set “test list” as a watch variable so you can see what values it contains, and then pull out and test the “max pos in …” block with various values for the prefix size parameter. Assuming that works correctly (if not, fix it before moving on!), pull out the “sort” block and run it with “test list” as the argument. You should be able to see the “test list” changing in its watch box, and at the end everything should be in sorted order. When you have these scripts working correctly, save your work as “Lab6-Activity1”.

Activity 2: [*Very important: Before starting this activity, double-check all of your block definitions and make sure that the “atomic” option is unchecked in each of your definitions.*] For this activity, use the BYOB timer (as described in the pre-lab reading) to time the “max pos of …” and “sort” functions. You should uncheck the “test list” list so that it is not a watch variable - there’s no need to take up computer time updating the display of the list when you’re just interested in is how long it takes to do the computation. You should start by using the list creation script that you made for testing in the previous activity to create a list of 500 random numbers, and time how long it takes to compute the position of the maximum value in this list. Then run the same test again! The reason for duplicating the test is that timing can vary if you run the same test multiple times, so you want to make sure you didn't get an inaccurate reading the first time. If your two times are roughly the same, then we will call that "good enough" and you should record the average of your two times. If they are not close (if they differ by more than 2%), then keep running the test until you get enough consistent times to be confident in your timing. You will then repeat this process with lists of length 1000 and 2000, so that in the end you can make a table that looks something like this:

List Size	Time (seconds)
500	20.50
1000	41.06
2000	81.98

You should type these numbers up into a table - use Microsoft Word or put them in an Excel spreadsheet, whichever you are more comfortable using, and save them in a filed named Lab6-Times.doc or Lab6-Times.xls (you will be submitting this file along with your code).

Next, do the same thing for the "sort" block - this is substantially slower, however, so you need to use smaller list sizes. For "sort", test your code with lists of size 20, 40, and 80. One warning: you have to create a new list for every test of the "sort" block. The reason for this is that running "sort" on your test list rearranges it into a sorted list, so if you just clicked "sort" again on the same list it would be "sorting" an already-sorted list! Sorting an already-sorted list might have a different time than sorting a randomly ordered list, which is what you're really trying to measure. Type up these times in a separate table in the same file as your first table of results, and make sure you label each table to indicate what it represents.

When you have finished all of this, save your BYOB code (including your scripts that do the timing) as Lab6-Activity2, and save your tables in a file as described above.

Activity 3: For this activity, you are to instrument your code so that you can count how many comparisons are made when it runs, as described in the pre-lab reading. The first step is to find everywhere in the code that it makes a comparison of two items from a list (*hint*: if you implement exactly the code given in the pre-lab reading there is only one place where a comparison is made!). Create a variable to count comparisons, and insert a change block to increment that variable at the appropriate place(s). Finally, make a small test script that sets the comparison counter to zero and then calls the block you are testing - this will be practically identical to the timing scripts from Activity 2, but instead of resetting the timer you are setting the comparison counter to zero, and instead of reading and saying the timer value your sprite should be saying the comparison count. Repeat your tests on both the "max pos of …" block and the "sort" block, using the same list sizes that you used in Activity 2, recording the number of comparisons made in each case in tables as you did with the times in Activity 2. *Helpful hint*: In this part we're not measuring time, so the "atomic" option in the block definitions does not make a difference for the results we're after - because of that, if you go into each definition and check the "atomic" option, you will get your results much faster!

Save your instrumented BYOB scripts with testing scripts as Lab6-Activity3, and save your tables as Lab6-Comparisons.

Activity 4: This activity doesn't actually require you to write any new code or turn anything in. It's what we call a *sanity check* - making sure the values you got in Activities 2 and 3 make sense. The reason this is given as an in-lab activity is that you should consider this during the lab time, so that if your answers don't make sense, you can go back and double-check those activities to see if you made a mistake somewhere. Then you can correct your files for those activities before you submit anything!

First consider the "max pos of …" block: This is a linear time algorithm, so if you double the size of the input, the time should roughly double - that's what linear time means! The sizes you used for tests doubled from test to test, so the times and number of comparisons should have doubled each time as well. Calculate how much the time (and number of comparisons) went up when you increased the input size from 500 to 1000 items, and also from 1000 to 2000 items. Was it roughly doubling each time? If so, that's what it's supposed to do! If not, then you should probably re-check your code.

Next, consider the "sort" block: This is a quadratic time algorithm, so what should happen when the size doubles? Try a few values and see what happens to n^2 when you double n. For example, if n=2, then n^2 is 4. If we double n to 4, then n^2 becomes 16. Let's try again: if n=3, then n^2 is 9. If we double n to 6, then n^2 becomes 36. Do you see the pattern? If not, try a few more values and you should see a consistent pattern in how much n^2 increases every time n is doubled. You can also derive this pretty easily as a general mathematical property: what is $(2n)^2$, and how is it related to n^2? Use this observation to check your time and comparison count tables for the "sort" block. Are the values going up as expected? On this one, don't expect the times to work out exactly: if you're within 10% of the expected increase, you are close enough.

Discussion (Post-Lab Follow-up)

Timing Code in Different Environments: There are many factors that determine how fast a particular implementation of an algorithm will run. Most obvious is the speed of the processor running the program, but the programming language and environment can make a huge difference as well. There are thousands of programming languages to choose from - in the late 1960's, Jean Sammet did a survey and found about 3,000 programming languages - and we've had over 40 years of inventing new languages since then! Why are there so many programming languages? Wouldn't one good language be sufficient? Maybe some day there will be "one language to rule them all," but for now the best way to think about programming languages is that they serve as an intermediate step between how people think and express algorithms and how computers work and execute programs. Some languages are closer to the way people think (we call these *high level languages*), and some are closer to the way computers operate (we call these *low level languages*). The job of the programmer is to turn his or her thoughts into statements in the programming language, and the job of computer tools like a compiler or execution environment is to turn the programming language into something the computer can execute. The first step - human thoughts and ideas translated to a programming language - can be viewed many different ways because different people think in different ways. Alan Perlis, a famous computer scientist who won the *Turing Award* (the "Nobel Prize of Computer Science") once said "A language that doesn't affect the way you think about programming, is not worth knowing." That's a good way to think about programming languages - they should affect how people approach problem solving. But as a result, some languages more efficiently translate to what computers do than other languages.

BYOB is a good example of this. It hides a lot of details that are present in other languages, taking care of things behind the scenes so you don't have to worry about them. The "Atomic" check box controls some of those things. A script is "atomic" if it takes complete control and can't be broken up into smaller executable pieces. Atomic scripts can be a little unfriendly to other scripts since it won't allow them to run, and some of you have already noticed in lab that if there is a bug in a script that is atomic it can be very hard to break out of that script and do other things. A script that is not atomic is checking after every step to see if there are other scripts that need some time to run, or if there are events to respond to, or if watch variables need to be updated, or... basically, lots of "housekeeping" tasks. As a result, just setting the "atomic" option will make a script run much faster - that hasn't changed the algorithm at all, but has just changed the environment in which it runs. Unfortunately, there are some strange things in BYOB with the way scripts are run and overhead for certain operations, and when the "atomic" flag is checked the performance of algorithms in BYOB doesn't behave as reliably and predictably as in most other programming languages and environments. That's why you were told to make everything non-atomic, because while it slows everything down, things perform in a more predictable way. BYOB is not a language for high-performance computing, so most people don't worry about this - it does make it a little more difficult to teach about algorithms in that setting, however!

To finish our discussion of the effect of programming languages and environments on running time, let's see how long the same algorithm takes to run when implemented using various languages/environments. To do this, the sorting algorithm given in the pre-lab reading was implemented in multiple languages in addition to BYOB: C, Java, and Python. In each of these languages, a test list of 1,000 random numbers was sorted (all on my laptop - a recent but not particularly high performance system). The time required is for each language is listed below:

Language	***Time (seconds)***
BYOB non-atomic	20,710 (that's over 5.5 hours)
BYOB atomic	462
Python	0.05
C	0.000844
Java	0.000588

To put this into perspective, a lot of people like to use Python because it has some nice and powerful features, but for this particular algorithm it is 85 times slower than Java (not all algorithms are this much slower in Python - this is a particularly bad case). But BYOB, even in the "fast" atomic setting, is over 786,000 times slower than Java. When the "atomic" option is not set, BYOB is over 35 million times slower than Java. Yes, 35 million times. That's not just slower, that's "what-in-the-world-are-they-possibly-doing slower." While BYOB is easy to use for beginning programmers, it's clearly not something you would want to use for any serious computational problems.

Data Structures for Fast Searching: Many search problems can be solved more efficiently if the data is ordered or structured in a particular way. This is a deep and sometimes complex area of computer science known as the study of data structures, and we describe briefly some examples of things you would learn about by studying data structures. As a simple example, if we are interested in finding the largest item in a list, if the list is stored in sorted order then we can easily do this: it's always the last item in the list, so we can find it in constant time! We will also see in the lectures that sorted data can be searched for any particular value much faster than is otherwise possible, using an algorithm known as binary search. The problem with both of these statements is that the data must be stored in sorted order, and if data changes dynamically (new items are inserted, some might be deleted, some changed during the execution of the program), maintaining the data in sorted order is actually quite inefficient. There are data structures known as heaps that allow fast searches for the maximum value, while allowing efficient changes to the data as well. There are data structures known as balanced search trees and hash tables that support searching for particular values even when the data can change dynamically. Learning about how these work is a standard part of any data structures class, such as UNCG's CSC 330.

Sorting: The sorting algorithm that we described in the pre-lab reading is an algorithm known as "selection sort" (the name comes from the process in which the maximum value is selected, then the maximum of the remaining *n*-1 is selected, and so on). As we discussed, this is a quadratic time algorithm. There are faster sorting algorithms - although it is impossible for a sort algorithm based on comparisons to be linear time (and you would typically see a proof of this in an algorithms class, like CSC 555 at UNCG), there are algorithms that are significantly faster than quadratic. For example, sorting an array of 1,000,000 random integers using selection sort (implemented in C++ on my laptop) takes almost 15 minutes, whereas sorting that same array using an algorithm called "quick sort" takes 0.08 seconds - so for that input, quicksort is approximately 10,000 times faster. It's not at all obvious how to make such a fast algorithm, and this is an example of the deep and very useful results that are studied in later computer science classes!

Terminology

The following new words and phrases were used in this lab:

- *algorithm*: a well-defined computational procedure that takes some value or set of values as input and produces some value or set of values as output - the focus is on *how* something is produced.
- *BYOB timer*: a BYOB functionality that is constantly advancing and measuring time, so that the "reset timer" and "timer" blocks can be used to measure how long any script takes to run.

- *comparison*: when used in regard to searching and sorting algorithms, a comparison means comparing values from the list being sorted - so if you are sorting a list of names, comparing names from the list counts as a "comparison", but comparing two indices does not count as a "comparison" in this sense.
- *high level language*: a programming language that allows the programmer to work closer to the way people think about algorithms than the low-level details of how computers operate.
- *instrumentation*: code that is put into a program to monitor how various aspects of the program operate or perform (such as counting the number of comparisons in a sorting function), rather than the code that is solving the problem of interest.
- *instrumenting code*: the process of putting instrumentation in a program.
- *low level language*: a programming language that closely resembles the low-level details of how computers operate.
- *linear time*: an algorithm runs in linear time if the number of steps that are required is proportional to some constant times n, where n is the size of the input.
- *prefix of a list*: the first values in a list - for example, in a list of 1000 items we might talk about the prefix of size 100, which is the first 100 items in the list.
- *problem*: a computational task that is specified only in terms of how correct outputs are related to inputs - the focus is on *what* is produced, and not *how*.
- *quadratic time*: an algorithm runs in quadratic time if the number of steps that are required is proportional to some constant times n^2, where n is the size of the input.
- *sanity check*: looking at results of a test to see if they are sensible.
- *searching*: the problem of looking through a data set for some item that meets some criterion (such as the largest element, an element that matches a search term, etc.)
- *sorting*: the problem or rearranging a list according to some rule, such as putting items in non-decreasing order.
- *step*: a basic operation which forms the basis of a time complexity analysis - simple operations such as plus, times, and comparisons are usually considered a single step.
- *test data*: a data set that is created for the purpose of testing some algorithm implementation (where testing can be for testing correctness or efficiency).
- *Turing Award*: the top prize in the field of computer science, much like the Nobel prize is the top prize in the field of physics.

Submission

In this lab, you should have saved BYOB files Lab6-Activity1, Lab6-Activity2, Lab6-Activity3, and Word or Excel files named Lab6-Times and Lab6-Comparisons. *Note: if BYOB file Lab6-Activity3 contains all of the code from activities 1 and 2 as well, you only need to submit that BYOB file (along with your tables).* Turn these files in using whatever submission mechanism your school has set up for you.