

# The Beauty and Joy of Computing<sup>1</sup>

## Lab Exercise 7: Experimenting with Algorithms

### Objectives

By completing this lab exercise, you should learn to

- Describe basic searching and sorting algorithms;
- Understand and modify implementations of searching and sorting algorithms in Python;
- Use the Python “timeit” module 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 some of the basic ideas that will be central to the lab activities, along with sample code that you should read and understand. There is no need to actually implement or experiment with any of the code 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 class and in the the last lab, we discussed 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.<sup>2</sup> 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.

In this lab you will also start working more with Python, which you were introduced to in the last lab. To transition you into using Python, initial examples with code will be shown in BYOB and in Python. While the algorithms are the same in all cases, the code varies somewhat due to different naming conventions in BYOB and Python, and different operators that are available in Python. For example, while the basic iterator pattern take four different blocks in BYOB (declaring the index script variable, initializing this index variable, doing a repeat block, and changing the index), it is a single line in Python.

Note that some of the Python code is not written the way an experienced Python programmer would write it. In particular, I am keeping the basic structure the same so that you can recognize

---

<sup>1</sup> Lab Exercises for “The Beauty and Joy of Computing”

Copyright © 2012-2014 by Stephen R. Tate – Creative Commons License


See <http://www.uncg.edu/cmp/faculty/srtate/csc100labs> for more information

<sup>2</sup> 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).

the equivalence between the code in BYOB and the code in Python, and I am avoiding some of the more unique and powerful features of Python that have nothing equivalent in BYOB.

## 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 Lab 5 (the list lab), you 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 function in this lab, and modify it so it is useful in a sorting procedure. In particular, let’s make it a general-purpose “max” function (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 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.” We would like to be able to use this function as shown below – on the left is a BYOB reporter block, like we’ve defined before in this class, and on the right is what it looks like using an equivalent Python function:

	<code>max_pos_from_first(testList, 6)</code>
--	--

Note that the parameter order is reversed in these two definitions, but don’t let that confuse you. In BYOB, when parameters can be put in the middle of the title, they are typically ordered in a way that makes a sensible English statement. In Python and other languages that put all the parameters after the function name, the typical style is to put larger parameters that are operated on (like the list) first, and smaller parameters that control the operation (like the size of the prefix) after.

To implement this function, the algorithm will use the iterator pattern to look 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. There are a couple of small changes to the basic iterator pattern to notice: 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. Study the BYOB implementation below, and make sure you understand how it works. Then look at the Python code to the right, and make sure you understand how it does the same thing – pay attention to differences as well as similarities, which will be discussed below the code.

<p>The Scratch code consists of the following blocks:</p> <ul style="list-style-type: none"> <li><b>max pos from first</b> <code>pNum</code> of <code>pList</code></li> <li><b>script variables</b> <code>sIndex</code> <code>sAnswer</code></li> <li><b>set</b> <code>sAnswer</code> to <code>1</code></li> <li><b>set</b> <code>sIndex</code> to <code>2</code></li> <li><b>repeat</b> <code>pNum - 1</code> times: <ul style="list-style-type: none"> <li><b>if</b> <code>item sIndex of pList</code> &gt; <code>item sAnswer of pList</code>: <ul style="list-style-type: none"> <li><b>set</b> <code>sAnswer</code> to <code>sIndex</code></li> </ul> </li> <li><b>change</b> <code>sIndex</code> by <code>1</code></li> </ul> </li> <li><b>report</b> <code>sAnswer</code></li> </ul>	<pre>def max_pos_from_first(data, howmany):     maxSeen = 0     for i in range(1, howmany):         if data[i] &gt; data[maxSeen]:             maxSeen = i     return maxSeen</pre>
---	---

First, note that there is nothing equivalent to the “script variables” block in the Python code. All variables that are used in a Python function definition are assumed to be local to that function unless specifically marked as global (we’ll see how to do that a little later), so “maxSeen” and “i” are automatically local to the function, just like a script variable. In a previous lab, I made a big point of how variables should be declared with the smallest scope possible (as script variables if possible), calling this the “Principle of Least Scope” – in Python this is the default behavior, so you don’t have to remember to do it! This also means that you can’t accidentally mix up local and global variables like you could in BYOB, so the “naming convention” we used to help distinguish between global variables, script variables, and parameters isn’t necessary here. But you still need to give variables meaningful names!

Next, pay attention to index values. The position of the first item in a BYOB list is 1, while the position of the first item in a Python list is 0. Therefore, if we initialize the position of the “largest item we have seen so far” to the first position in the list, BYOB would use 1 (which we do in the “set sAnswer to 1” block) and in Python we would use 0 (which we do in the “maxSeen = 0” statement).

Also notice how simple the iterator pattern is in Python: just a single “for” line, which declares the index variable (which we call “i”), sets the range of values (beginning and end), and does the repetition (like the BYOB “repeat” block). You’ll notice that there’s nothing like the “change sIndex by 1” block in the Python code, so how does the index variable get increased each time through the loop? That’s built-in to the “range( )” function – you can actually have it change the index by different amounts each time through the loop, and we’ll see how to make it count down later in this lab. Here’s a trick to experiment with what range( ) does: put it inside the built-in Python list( ) function and it will give you a list of all the values that i would take on in that range. Here is a snapshot from the Python Shell showing a few examples of this:

```
>>> list(range(10))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> list(range(1,6))
[1, 2, 3, 4, 5]
>>> list(range(4,9))
[4, 5, 6, 7, 8]
>>>
```

Notice that if you only give one argument to `range()` it is the upper limit of the range, and the list of values stops before you hit the upper limit (the upper limit value is never included – in mathematical terms, the range is open at the upper limit). If you give two arguments, then the first value is the first value in the range, which would otherwise default to 0.

Finally, notice a few other small things with Python. The code to get an item out of a list is less wordy, where you just have to say “`data[i]`” rather than “item *i* of *data*” (or “item *sIndex* of *pList*” using our longer BYOB names). You also have to be extra-careful with how Python code looks on the screen, since indentation controls a lot of how your code executes. While in BYOB there’s a clear place for each piece to snap in, you can position things almost anywhere you want in Python, and changing the indentation level of lines of code can radically change how or when they are executed!

Now consider the algorithm in general. 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 the fact that the built-in “`len`” function in Python gives the length of a list as follows:

```
max_pos_from_first(testList, len(testList))
```

If we wanted to actual maximum value (rather than the position) we could use the output of this function (which is a position) to index into the list and get the value:

```
testList[max_pos_from_first(testList, len(testList))]
```

While `max_pos_from_first()` is a powerful function, the previous expression is awkward to type every time you want to get the maximum value from a list. The right “programmer’s mindset” here is to look at that and think that it’s a perfect situation to define your own function, leading to the following definition:

```
def max_in_list(data):
    return data[max_pos_from_first(data, len(data))]
```

See how clear it is when you build things up from smaller pieces like that? Now the reality is that there is a built-in “`max`” operation in Python, so if you really want to know the maximum value in a list, you’d just use `max(testList)` and not write any of your own functions at all. But that

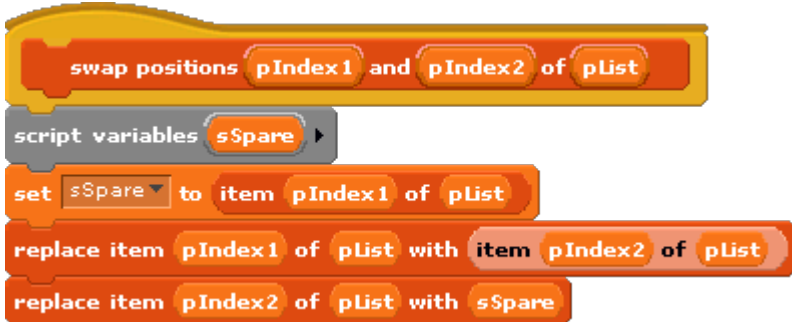
wouldn't be a very good example for learning, would it? Plus, it wouldn't tell you the position of the maximum value, which is what we really want.

These two additional problems (maximum *position* in the entire list and maximum *value* in the entire list) were solved by using just a few operations in addition to our `max_pos_from_first()` function, 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_first()` 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_first()` function. The first thing to notice is that the iterator loop executes roughly howmany (or `pNum` in BYOB) times<sup>3</sup> and howmany 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 loop through the list, 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.

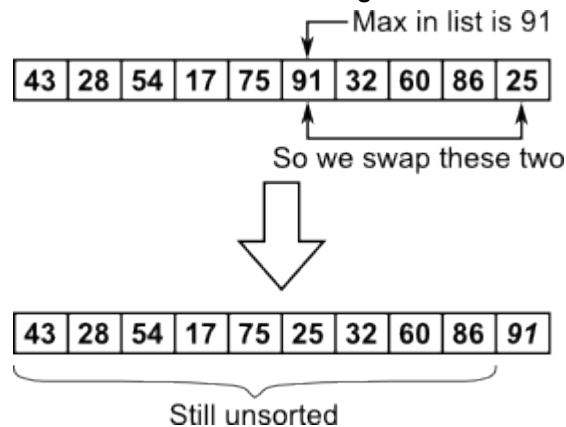
## Sorting

In this section, we will use the `max_pos_from_first()` function 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_first()` function to sort a list. If we use 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. Recall the “swap” block from Lab 5, and equivalent function in Python:

	<pre>def swap(data, index1, index2):     spare = data[index1]     data[index1] = data[index2]     data[index2] = spare</pre>
---	--

<sup>3</sup> “roughly” because it's really `howmany-1`, but that doesn't make a significant difference in time complexity.

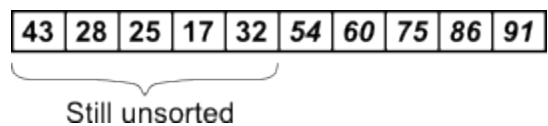
Looking at an example of a list, this transformation might look something like this:



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." The following snapshot of interaction with the Python Shell shows experimenting with this list in actual Python code. First the `testList` is displayed, then the maximum position is found and displayed (verify that this is correct by looking at the list!), and then we use the `swap` function and display the resulting list. See how it shows the same operations as in the picture above?

```
>>> testList
[43, 28, 54, 17, 75, 91, 32, 60, 86, 25]
>>> maxPos = max_pos_from_first(testList, len(testList))
>>> maxPos
5
>>> swap(testList, maxPos, len(testList)-1)
>>> testList
[43, 28, 54, 17, 75, 25, 32, 60, 86, 91]
>>>
```

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 into position  $n-1$ . 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()` function 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:



To put this into code, let's think about how many items remain unsorted in the list. At first the whole list is unsorted, so there are  $n$  unsorted items in the list. After the first swap, there are only  $n-1$  unsorted values, and after the second swap there are  $n-2$  unsorted values, and so on. After  $n-1$  swaps, we are down to  $n-(n-1)=1$  unsorted items, which means the whole list is sorted. To keep track of how many items are left unsorted, we need to count down, while all our previous range examples were counting up. It turns out that we can use a third argument to

`range()` to accomplish this. The third argument is the “step” value, or the amount the index changes each time through the loop – this is exactly the value that was in the “change `sIndex`” block in the BYOB version of the iterator loop, and when we wanted to count down in BYOB (like in the GCD function from Lab 3) we made the “change” block change the index by `-1`. Here’s what it looks like when we experiment with the `range()` function with `-1` for the step value:

```
>>> list(range(10,0,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]
>>> list(range(9,4,-1))
[9, 8, 7, 6, 5]
>>>
```

Notice that these ranges count down, just like we want, and the final limit is not included in the range just like before. Let’s use a variable named “left” (for the number of items left unsorted) to count down, and then we can apply our `max_pos_from_first()` and `swap()` functions to finish the sorting function. The solution is given below:

```
def sort(data):
    for left in range(len(data), 1, -1):
        maxPos = max_pos_from_first(data, left)
        swap(data, maxPos, left-1)
```

This is a deceptively simple looking script, and there’s a lot going on despite it just being a few lines of code. Study this code carefully, and make sure you understand how it works. Pay particular attention to the `left-1` in the call to `swap()` – do you see why that is the right value to use?

Considering the time complexity of this sorting algorithm, the main thing to observe is that the sorting algorithm consists of a loop that executes 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 the time complexity is in fact a little more involved, but the end result is still the same: this is a quadratic time algorithm.

## Timing Python Functions

We’re interested in how fast our algorithms and scripts are, so we’ll next look at how to experiment using Python in order to determine various measures of time and speed. One challenge in timing simple programs like the ones you’re writing here is that they run fast. Very fast. In our example above, the test list has 10 items. Running the `max_pos_from_first()` function on a 10-element list takes around 1 microsecond, or one millionth of a second. You are obviously not going to measure that by pressing start and stop on a stopwatch – in fact, unless you are very careful it is difficult even for computers to accurately measure something that quick. To be able to time these fast functions, we need to be able to do two things: generate larger test data (just typing in a list with 10 elements isn’t large in any sense), and running some

test code multiple times so that the amount of time taken can be accurately measured. We'll look at both of these issues below.

Creating *test data* for list operations is not difficult – just make a loop and add a number of random values to an initially-empty list. If we were doing this in BYOB, we could make a 40-item list like this:



In Python we can use a programming expression that is called a “list comprehension” that is not available in BYOB. A list comprehension is just a programming version of standard “set-builder” mathematical notation. Here’s an example of the mathematical set notation:

$$E = \{ 2*x \mid 0 \leq x < 7 \} = \{ 0, 2, 4, 6, 8, 10, 12 \}$$

You read the mathematics by saying that E is the set of values 2\*x where x ranges over the range [0,7). The following Python shell interaction shows how we say this same thing in Python. Compare it closely to the mathematical expression above and think about how these match up:

```
>>> E = [ 2*x for x in range(7) ]
>>> E
[0, 2, 4, 6, 8, 10, 12]
>>>
```

The value that goes into the list does not have to be a simple function of x, like the 2\*x that we used in this example. In particular, we can use a random number generator to make a list of random values. In typical Python style, you only load in the functions that you are actually going to use, and so if you want to generate random numbers you’ll need to do an “import random” operation first. This would typically be done at the top of the program file you are working on (in the program window of IDLE), but in the following example it is shown in a Python shell interaction. The function `random.randint(a,b)` returns a random integer value r chosen from the range  $a \leq r \leq b$ , giving us a way of creating a random test list:

```
>>> import random
>>> testList = [ random.randint(1,10000) for i in range(10) ]
>>> testList
[6662, 3146, 1346, 192, 3098, 6720, 3682, 4456, 2658, 3824]
>>>
```

By changing `range(10)` to `range(10000)` you can easily make a `testList` with 10,000 randomly chosen items. You probably wouldn’t want to print this out though!

Now we discuss how you can time your code in Python using the standard Python module named `timeit`. Using this module can be a little tricky for a beginner because of the way some things work in Python, but if you carefully follow the instructions given here everything should work fine. The idea with `timeit` is that there is some operation you want to time, and you may



need to repeat the operation a number of times in order for it to take long enough to get an accurate measurement of the time. For example, if you are timing a function that takes 83 milliseconds to run, you might want to run it 100 times in a row so that the total time is 8.3 seconds – long enough to be measured accurately. You need to remember to divide the resulting time by the number of repetitions to get the time for your function ( $8.3 / 100 = 0.083$  seconds), but if you shoot for a total running time of around 10 seconds then this is a very accurate way to time your function.

Let's move some of the experimentation and ideas that we illustrated previously with the Python Shell into the program window so that we can save and reload helpful functions easily. First off, we can import the modules that we need by putting this at the top of the program (in the activities for this lab that would mean putting this at the top of the Lab7.py file):

```
import random
import timeit
```

Next, let's make a function that we can use to set up a test data list of some size, where the size is sent in as a parameter. We would like to store our test data in a global variable so that it can be accessed by other functions, but recall that above we said that all variables in a Python function default to local scope. If we set up test data in a list that was a local variable, then as soon as the function finished and returned, the list would cease to exist! We solve this problem by telling Python that a particular variable is a global name, by just saying "global" followed by the variable name on a line at the beginning of the function definition. Putting this together, here is our setup function:

```
def setup(size):
    global dataSource
    dataSource = [ random.randint(1,10000) for i in range(size) ]
```

Next, we will make a function called `test_max()` that uses this global `dataSource` variable as a parameter to `max_pos_from_first()`, this function would look like this:

```
def test_max():
    global dataSource
    max_pos_from_first(dataSource, len(dataSource))
```

The reason for having a separate function and using a global variable for the data probably isn't completely clear, and in fact it seems to go against good programming practices that we've discussed in here (you should in general avoid using global variables and use parameters for input data). The reasons for doing it this way are a little obscure, but just trust me that this makes using the `timeit` module simpler.

Now with those two functions defined, we're ready to try timing our code. If all of the above code is saved in your program, and you have reloaded this into Python (press F5!), then you enter the

following two commands into the Python shell to set up the test with a length 5000 list and then time it:

```
>>> setup(5000)
>>> timeit.timeit('test_max()', 'from __main__ import test_max',
number=1)
0.0010412340052425861
>>>
```

The call to `timeit.timeit()` is one of the ugly parts of this: in the second parameter, “main” is preceded by two underscores and followed by two underscores (and spaces), and the reason you have to do this probably isn’t clear. For now, don’t worry about it – just type it like this, and remember to change the `test_max` part of that argument if you are testing other functions. If you’re curious about what this actually does, you can find explanations on-line, but you don’t need to understand the reasons just to use it here.

The result returned from `timeit.timeit()` tells you that one call to `test_max()` took about 1 millisecond. The next thing you should do is figure out the right number of repetitions so that the running time is close to 10 seconds (say, between 5 and 15 seconds). Since this took only 1 millisecond, 10,000 times that would be about 10 seconds, so let’s try that:

```
>>> timeit.timeit('test_max()', 'from __main__ import test_max',
number=10000)
4.009385789278895
>>>
```

So that didn’t take quite long enough, and it illustrates why we do testing this way: if the timing were perfect, doing `test_max()` 10,000 times should have taken 10,000 times as long as a single execution, but it actually took only about 3,855 times as long. Why? Because that first time, with a single iteration, includes some basic overhead that is present no matter how many times the function is repeated, and with such a small time this overhead is a significant portion of the running time. We try one more adjustment, changing the “number” argument to 25,000 and get a time of 10.0989 seconds – this is what we want! So how much time did the execution of `test_max()` take? Just divide  $10.0989 / 25000$  to get approximately 0.000404 seconds, or about 0.4 milliseconds. This is a much more accurate time measurement that we got from our single execution time of around 1 millisecond.

The last issue to discuss here is what I’ll call the *data restoration issue*. When we ran `max_pos_of_first()` on our test data, it simply read the data, and so it was unchanged when we repeated that test run a second (and third and ten thousandth) time. However, if we want to time `sort()`, it actually changes the list by re-arranging it. It is conceivable that a function will take a different amount of time if it is given an already-sorted list, so we need to make sure to restore the original test data before each run. Once you realize what the problem is, the solution is simple: just make a local copy of the test data, and call `sort()` on that local copy. If this is part of your test function, then it will restore the original test data for each run of `sort()`:

```
def test_sort():
    global dataSource
    testcopy = dataSource.copy()
    sort(testcopy)
```

As a final note, if you wanted to be very precise in your timing, you would notice that making a copy of the test data does take some time, and it's not part of the time for `sort()`. Measuring this overhead and subtracting it from the time for `test_sort()` would give you a better measurement of the time for `sort()`, although for this lab it is unnecessary: the time for `sort()` will be so much larger than the test data copying overhead, that it amounts to a fraction of a percent error in your results.

## Counting Operations

Using the a 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 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 comparisons 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 function that we are

testing, and add the statement “compareCount += 1” so that it is executed every time a comparison between data items is made. Let’s do an example by considering a function that takes a sorted list that may contain duplicate values as an argument, and returns the number of distinct values in the list. So for example, if called with the list [1, 2, 2, 2, 3, 3, 7, 9, 11] it would return 6 (there are six distinct values: 1, 2, 3, 7, 9, and 11). Here’s the function:

```
def count_unique(data):
    items = 1
    for i in range(len(data)-1):
        if data[i] != data[i+1]:
            items += 1
    return items
```

Now we instrument this function to count the number of comparisons, as described above. There is only one place where we compare data values, so we put in the line to increment compareCount at that point. Don’t forget that we need to let Python know that compareCount is a global variable! The final instrumented version of count\_unique() is then as follows:

```
def inst_count_unique(data):
    global compareCount
    items = 1
    for i in range(len(data)-1):
        compareCount += 1
        if data[i] != data[i+1]:
            items += 1
    return items
```

Now we run this as follows – first set compareCount to 0 in Python Shell window, then call inst\_count\_unique(), and finally check the value of compareCount at the end. Here’s a picture showing that interaction:

```
>>> compareCount = 0
>>> inst_count_unique([1,2,2,2,3,3,7,9,11])
6
>>> compareCount
8
>>>
```

The conclusion from this? count\_unique() made 8 comparisons when processing that test list of length 9.

That example was pretty simple, and for the most part, deciding where to put these statements in any function is easy: find every comparison that is made in your program, and add this counter increment statement right on top of the statement that makes the comparison. This works fine for all the code in this lab, but be aware that when a comparison is made in a loop condition then it can get more complicated. Just make sure that you always add one to the compareCount variable the exact same number of times that you make a comparison.

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 Python `timeit` module to time functions. And finally, you should understand how to instrument code to count operations – in this case, we are counting comparisons.

### Self-Assessment Questions

Use these questions to test your understanding of the Background reading. Answers can be found at the end of the lab exercise.

1. Consider a variable `testList` that contains `[144, 7104, 5160, 2918, 7636, 450]`. Using our definition from the pre-lab reading, what does the call `max_pos_from_first(testList,3)` return?
2. If we are using the `max_pos_from_first()` function inside a sorting algorithm to help us rearrange the list, why is it important that `max_pos_from_first()` return an index in the list rather than the maximum value?
3. Consider the following two function calls, where `list2000` is a list that contains 2000 items.

```
max_pos_from_first(list2000, 1000)      max_pos_from_first(list2000,
400)
```

If the call on the left takes 15 seconds to run, how long would you expect the call on the right to take?

4. Consider the following two function calls, where `list2000` is a list that contains 2000 items and `list3000` is a list that contains 3000 items.

```
max_pos_from_first(list2000, len(list2000))
max_pos_from_first(list3000, len(list3000))
```

If the first call takes 12 seconds to run, how long would you expect the second call to take?

5. Explain step-by-step what happens if the following function is called (assuming `testList` has more than 5 items in it):

```
swap(testList, 4, 4)
```

6. If we use  $n$  to represent the length of the list sent to the `sort()` function. How many times does the for loop iterate? Give your answer as a formula in terms of  $n$ ?

## Activities (In-Lab Work)

To make your work easier, you can save the code for all activities in a single file, Lab7.py. Just add new functions as needed for Activities 2 and 3, without changing or deleting any of the code you have written for earlier activities. This way you can just work with and submit a single file.

**Activity 1:** For the first activity, you will gain some experience working with IDLE, the Python development environment that you saw in the last lab, by building and testing the functions described in the pre-lab reading: the `max_pos_from_first()` function, the `swap()` function, and the `sort()` function.

To do this, bring up the Python 3.4 IDLE environment, click on “New File” in the Python Shell “File” menu, and then immediately do a “Save As...” in the new (blank) Python program window using the name “Lab7.py” and saving in a convenient location. Now you can type the three function definitions given in the pre-lab reading into this window, saving your work as you go.

To test everything, create the random list builder script described in the section “Timing Python Scripts”, generate a random list of size 10, and store it in variable `testList`. Next, run `max_pos_from_first()` from the Python Shell window, using `testList` as the first argument and experimenting with different values for the second argument. Make sure you are getting the right answers (figuring out the correct answer is similar to Self-Assessment question 1 above – hopefully you did that as part of your pre-lab reading!). Assuming that works correctly (if not, fix it before moving on!), try using `swap()` to swap some positions in `testList` to see if that works correctly. And finally, try sending `testList` to the `sort()` function, and check the value of `testList` afterwards – everything should be rearranged into sorted order. When you have these scripts working correctly, make sure your final versions are saved in “Lab7.py”.

**Activity 2:** For this activity, use the Python `timeit` module (as described in the pre-lab reading) to time the `max_pos_from_first()` and `sort()` functions. You should start by setting up functions to initialize a test list and do some sample timing runs as described in the pre-lab reading to determine an appropriate number of repetitions for the `timeit` module to perform. When you test `max_pos_from_first()`, make sure that the second argument is set to the length of your test list so that you are searching through the **whole** list, not just part of it! Start your testing with a list of size 5,000, and adjust the number argument to `timeit` so that the full execution time takes around 10 seconds. Then run the same test again on a different random list with 5,000 items and see if your measurements are consistent! 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. Repeat this process with lists of length 10,000 and 20,000, so that in the end you can make a table that looks something like this (note that the units on the last column are milliseconds, not seconds):

List Size	Total Time	Number of	Time per
-----------	------------	-----------	----------

	(seconds)	Repetitions	max_pos_from_first() execution (milliseconds)
5,000	10.0	25,000	0.4
10,000	7.99	10,000	0.799
20,000	9.47	6,000	1.58

You should type the numbers you obtain from your code and tests into a similar table – use Microsoft Word or put them in an Excel spreadsheet, whichever you are more comfortable using, and save them in a file named Lab7-Times.doc or Lab7-Times.xls (you will be submitting this file along with your code).

Next, do the same thing for the `sort()` function – this function is substantially slower, so you need to use much smaller repetition counts. In fact, for a 20,000 item list even doing a single iteration (`number=1`) will take longer than 10 seconds (but not too long to run – on the lab systems at UNCG, expect a single repetition to take around 30 seconds). Make sure you take care of the “input restoration” issue that is described in the pre-lab reading, since `sort()` will change the test data. 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, make sure your Python code is saved, 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 every place in the code where it makes a comparison of two data items (*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 the increment statements (“`countCompare += 1`”) at the appropriate place(s). Finally, try running this from the Python Shell, resetting the counter at the beginning and examining the counter at the end. Once you’re confident that everything is working correctly, re-run the test cases from Activity 2 on `max_pos_from_first()` and `sort()`, creating tables of the number of comparisons made for each function and each input size (5000, 10000, and 20000). Of course, you only have to run the function once here – there’s no such thing as an inaccuracy in the comparison count, like there is for the `timeit` timer, so there is no reason to run repeatedly.

Make sure your instrumented functions are saved with Lab7.py, and save your tables as Lab7-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_from_first()` function: 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 (in the last column of the time table) 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 5000 to 10,000 items, and also from 10,000 to 20,000 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()` function: 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()` function. 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.

## Submission

In this lab, you should have saved Python file `Lab7.py`, and Word or Excel files named `Lab7-Times` and `Lab7-Comparisons`. Turn these files in using whatever submission mechanism your school has set up for you.

## 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 translate more efficiently into the actual computer operations than do 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. BYOB is not a language for high-performance computing, and executes slowly. However, it is good at what it does: provide an easy-to use scripting environment where people can do interesting animation scripting with no previous programming experience.

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: 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). For BYOB, performance is radically affected by the “Atomic” checkbox in the block definition window, so it was tested both with thich checked and without. The time required is for each test is shown below:

<i>Language</i>	<i>Time (seconds)</i>
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!

It's also worth pointing out that Python has a built-in "sort" function that is an implementation of one of these better algorithms, so if all you're interested in is getting a list `testData` in sorted order, then all you have to put in your Python program is a statement that says "`testData.sort()`". For a list with 1000 random integer values, this statement takes about 0.00021 seconds on my laptop – compare to the times in the table just above for different languages and the selection sort algorithm.

## 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.
- ***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 of 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).
- ***timeit***: a Python module that provides a way to time how long Python code takes to run.
- ***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.

## Answers to Pre-Lab Self-Assessment Questions

1. Calling `max_pos_from_first(testList,3)` returns 1. If you said it would return 7104, then you need to remember that it reports an index (position), not a value. If you said it returns 2, then you need to remember that the first item in a Python list is at position 0. If you said it returns 4 (or 5), then you need to remember that it is only searching through the first 3 items since the second argument to the function is 3. And if you said it reports 7636, then you confused both of these issues!
2. Operations that change a list, such as our swap function or BYOB blocks “insert”, “delete”, and “replace”, all require an index into this list as an argument. If all we have is the maximum value, we’d have to find it in the list before we could change anything – and we do not want to have to search through the list again just to find this value!
3. The call on the right should take about 6 seconds. This is because the running time is linear in the number of items searched, and since the call on the right searches 0.4 times as many elements (400/1000) it should take time 0.4 times as long, or  $0.4 * 15 = 6$  seconds. Note that both of these observations are important: linear time and 0.4 times

the input size. If this were a quadratic time algorithm (like “sort”), then this would no longer be the correct estimate (if it were a quadratic time algorithm, we would expect the time to be around 2.4 seconds – can you figure out why?).

4. These calls search through the entire list in both cases, so the first call searches through 2000 items and the second one searches through 3000 items. Since this is a linear time function, time scales up linearly with the number of items searched, so the running time should be approximately  $3000/2000 * 12 = 18$  seconds.
5. First, the item at index 4 (let’s call this “item 4”) of `testList` is copied to the variable `spare`. Then the second statement replaces item 4 of `testList` with the contents of item 4 of `testList` – in other words, it doesn’t change anything! Finally, the value from `spare` (which is the same as item 4) is used to replace item 4 in the list, which again doesn’t change the list at all. So the end effect is that the list isn’t changed – we did 3 different copy operations, so it took a few steps of computation, but in the end it is as if nothing happened.
6. The for loop iterates  $n-1$  times. Since it uses `range(len(data), 1, -1)`, where `len(data)` is  $n$ , as we showed in other examples with a  $-1$  step value, this will iterate over values  $n, n-1, n-2, \dots, 3, 2$  and stop (note that it stops *before* it gets to the ending value 1. If we had gone all the way down to 1, then there would be  $n$  values in this range (it’s just counting 1 to  $n$  backwards), but since we stop at 2 the number of iterations is  $n-1$ ).