

The Beauty and Joy of Computing¹

Lab Exercise 6: Exploring Programming Languages

Note for non-UNCG users of these exercises: This exercise is not as generic as the other Lab exercises, and depends on the particular setup of the computers in UNCG labs as far as versions of Python and Netbeans that are installed. If you use this lab exercise, you might have to adjust things to be appropriate for your school's computers.

Note for UNCG students: This may still be a little raw and require some proof-reading... if anything is unclear just ask for clarification.

Objectives

By completing this lab exercise, you should

- Be able to recognize some fundamental computing patterns in different languages;
- Understand the difference between interpreted and compiled languages;
- Gain some exposure to programming tools from the simple to the advanced;
- Start to understand performance characteristics of different languages and systems.

Background (Pre-Lab Reading)

In this lab we will look at solutions to two different programming problems, experimenting with solutions in three different programming languages: BYOB, Python, and Java. These three different languages and systems represent three different approaches to how programming works, but a key point of this lab is to see that the same fundamental ideas and programming patterns are present in all three languages. For example, you'll see the counter pattern that we introduced in Lab 3 in all three languages. One of the goals of this lab is to prod you into thinking beyond details and see the "big picture" regarding programming and computing constructs. The first five labs used BYOB, and it wasn't because we felt that BYOB is a great programming system that you'll use a lot (or ever) after this class – it was because you can get up and running, and make a program that does something interesting in the very first lab. And while you are getting an easy introduction, you are getting exposed to fundamental concepts that transfer over to other languages that you might use in the future to do useful things.

Programming languages and systems for creating and debugging programs can be big and complex. In this lab, you should focus on recognizing similarities between solutions in the different languages, and get a feel for what it's like to create a program and run it with different systems. All of the basic code will be provided for you, so you'll either construct it from a picture (for BYOB code) or just download it (for Python and Java), load it into the relevant programming system, and experiment with it. You will have to make minor modifications to programs, but the skill you will be mostly using here is reading code, looking for patterns, and adjusting code when

¹ 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

requirements change. You don't have to actually create a new program in Python or Java – in fact, I'd be very impressed if you *could* do that with just the little exposure you get from this lab.

Problem 1: Mathematical Explorations

Computers are basically mathematical machines, and so it shouldn't surprise you that they excel at exploring mathematical areas. This is the first problem that you'll see solutions for in the lab:

Math Problem 1: If we list all the natural numbers below 10 that are multiples of 3 or 5, we get 3, 5, 6 and 9. The sum of these multiples is 23.

Find the sum of all the multiples of 3 or 5 below 1000.

You could find this sum with some thought and reasoning about the mathematics, but it's also something that can very quickly and easily be solved by writing a program. Many math problems are like this, and some problems are beyond the reach of solving without the help of a computer. This problem is in fact the first problem from the **Project Euler** web site (www.projecteuler.net). Project Euler is designed for people who like solving math problems like this, and want to learn to program. There are 481 problems on the web site, that get progressively harder (the problem above is the easiest one!). You earn points and recognitions for each problem solved, so a lot of people treat this like a computer game, and want to see how high they can get on the "high scores" list. As of the time I write this, 392,924 people had solved "Math Problem 1" and submitted the answer on the web site (and 51 people have solved over 450 problems!).

Problem 2: Bioinformatics

Maybe numbers and exploring mathematical properties isn't what sparks your interest, so what about problems in biology? The discovery of DNA and its structure and role in biology has changed a lot of biological questions from questions about chemistry to questions about information. A DNA sequence is almost like a sequence of instructions – a program! – for constructing a living being. Knowing the DNA sequence for some organism, and being able to process and interpret the information embedded in that sequence, provides biological information at a level that people couldn't even dream about a century ago. Extracting the DNA sequence from a biological sample is a process known as "sequencing," and in 2003 a research project known as "The Human Genome Project" announced the first complete sequencing of human DNA sample – a massive and undertaking that took almost 20 years from the planning stages to completion, and cost roughly \$3 billion. There are in fact many organisms (and now thousands of humans) that have been sequenced, and the sequences are stored in computer files and databases for processing. The science of processing this biological information is a field known as bioinformatics, and is a combination of both computer science and biology, with some statistics and other disciplines mixed in. Here is an example of a simple bioinformatics problem, which is the second problem that we'll see solutions for in this lab:

Bioinformatics Problem 1: A string is simply an ordered collection of symbols selected from some alphabet and formed into a word; the length of a string is the number of symbols that it contains.

An example of a length 21 DNA string (whose alphabet contains the symbols 'A', 'C', 'G', and 'T') is "ATGCTTCAGAAAGGTCTTACG."

Given: A DNA string s of length at most 1000 nt.

Compute: Four integers counting the respective number of times that the symbols 'A', 'C', 'G', and 'T' occur in s . For the example string above, there are 6 A's, 4 C's, 5 G's, and 6 T's.

Like our math problem above, you could do this by hand for the example or for a length 1000 DNA string, but it could be done much faster, and with less chance of error, by a computer. And when you get beyond the length 1000 “toy problems,” it gets much harder indeed – imagine doing this for the human genome, which has a length of 3 billion symbols. This problem is the first problem from **Rosalind** (www.rosalind.info), a problem-challenge site that was inspired by Project Euler but oriented around bioinformatics problems rather than mathematical problems. There are currently 228 problems on Rosalind, and it keeps track of the same kind of “scoring” information as Project Euler – currently 12,975 people have submitted programs that solve the problem above, and 53 people have solved more than 128 of the problems.

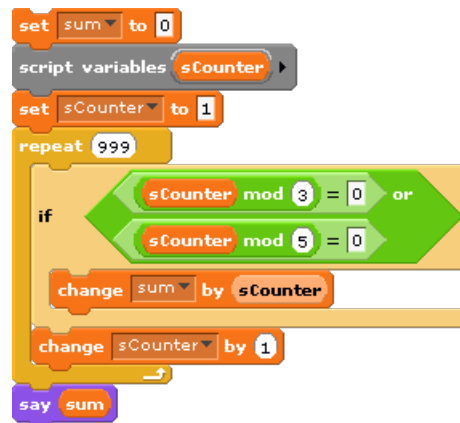
Programming Challenges for Self-Improvement

A lot of people find challenge sites like Project Euler and Rosalind to be a fun and motivating way of learning programming. You're not just learning a new programming technique because it's the next thing in a textbook, but you're learning it and using it because it helps you solve a problem that you're interested in solving. The two sites mentioned above are great for people who enjoy solving mathematical problems or bioinformatics problems, but there are also sites that are focused around other areas. For a general computer science focus, CodingBat (<http://codingbat.com/>) has lots of basic programming challenges (using Java and Python), as does Code Abbey (<http://www.codeabbey.com/>). There are even sites where the challenges look more like playing a video game (you are essentially coding sprites to solve challenges): CodeCombat (<http://codecombat.com/>) features programming in JavaScript, Python, Coffeescript, Clojure, Lua, and Io. As you get more advanced in your programming skills, you can consider challenge sites such as TopCoder (<http://www.topcoder.com/>) and Kaggle (<http://www.kaggle.com/>) that provide advanced challenges, and often give cash prizes. Aside from pure programming challenges, there are also a lot of sites that offer security challenges, such as Vortex Wargames (<http://overthewire.org/wargames/>) which offers “hacking challenges” in basic system security, web application security, cryptography, and more.

If you want to learn more about programming and you take the time to work through challenges on these sites, you will almost certainly see a big improvement in your programming skills. Imagine if you learned useful skills every time you sat down to play a game, rather than just blowing some computer generated character away for no real reason or benefit. If you get hooked on these learning-based challenges and spend time on them like some people do on video games, you'll be a master programmer in no time!

Activities (In-Lab Work)

Activity 1 (BYOB solutions): This activity involves creating solutions to both problems from the pre-lab reading using BYOB. First think about how you would approach solving the math problem in general terms: Count from 1 up to 999 (stay below 1000!), and for each number test if it is a multiple of 3 or 5 and add it to a running sum if it is. The “Count from 1 up to 999” statement is a clear indication that we will use the counter pattern from Lab 3. Inside the loop we need to check if the counter value is a multiple of 3, which we can tell by testing whether the counter mod 3 is 0, or if the counter is a multiple of 5. To keep the running total, we will use a variable named “sum.” Putting these idea together in BYOB, and adding a “say” block at the end to tell the user what the answer is, we get the following script:



Trace that through and think about how it works, and then build it in BYOB in Alonzo’s Scripts pane. It is always good to test your code, but to make testing quicker try using a smaller value in the “repeat” block. For example, try 9 rather than 999, which will give you the sum of all multiples of 3 or 5 below 10. Look back at the problem statement in the pre-lab reading, and it will tell you what the answer should be in this case – is your code giving the right answer?

Once you’re convinced that your code is computing the right value, let’s see how fast it is by using the built-in BYOB “timer” variable. The timer is simply a variable that measures how many seconds have elapsed since the last time it was reset. Your program interacts with the timer using the following two blocks:



The block on the left is what resets the timer to 0, while the reporter block on the right will give you the “current elapsed time” at any point in your program. The typical way this is used is with the following pattern:

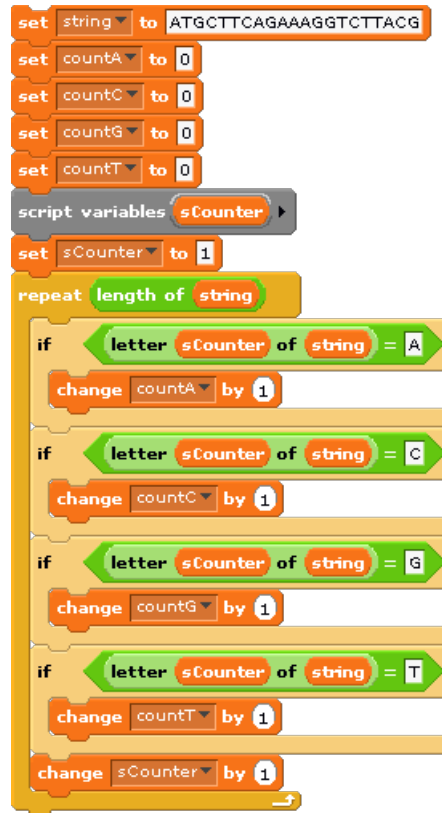


So the timer is first reset, then we run some code, and then after the code is run we save the elapsed time to a variable. You can then check the value of this variable later to see how long it took for your code to run. It is important to use a variable at the end to take a “snapshot” of the current elapsed time, because there is no way to stop the timer!

Add the timer code to your BYOB script, and change the repeat count back to 999 if you haven’t done that already. Now run the script, and make a note of both the resulting sum (which Alonzo says at the end) and the time it took for your program to run. You will need to report these values later in the lab quiz, so make sure you and your lab partner have both written this information down!

Now we turn to the Bioinformatics problem. What’s the general algorithm description for this one? We will create a variable for each of the counts – how many A’s have you seen, how many C’s have you seen, etc. We’ll initialize them all to zero, and then go through the string character by character and increase the appropriate count for each character. Going through the string is just the iterator pattern that we introduced in Lab 5 – note that we iterated through items in a list for Lab 5, but iterating through characters in a string is no different except for the use of string operations (from BYOB’s “Operators” category) rather than list operations.

Our next goal is to build this solution in BYOB. Use the same BYOB project that you used for the math solution, and just create a new sprite (it doesn’t matter what the costume is) and use the script pane of your new sprite as a blank area to work. The solution is given below – again, think carefully about why this works, and then build it and test it. It is a little repetitive, so think about building the “if” construction once and using “duplicate” to make 4 copies that you can modify appropriately.



Keep the countA, countC, countG, and countT variables checked so you can see their final values, and make sure you get the right answers for this string (check against the answer given in the pre-lab reading). Next, add timing code like you did for the math problem, and see how long this takes. Make sure you and your lab partner both record the amount of time that it took!

When you are finished with this, and understand the BYOB solutions, save your project as Lab6-Activity1 (it should contain both problem solutions, with the timing code additions, as scripts under different sprites).

Activity 2 (Python): Next we will look at Python solutions for these two problems, and see how you write and run Python programs. As you do this activity, keep BYOB up on your computer with your previous solutions. As Python solutions are described, look back to the BYOB solution and look particularly for how similar concepts are represented.

Python is a high-level language, with programs typed out in text rather than assembled from graphical pieces like in BYOB. Python is very dynamic and interactive, so working in Python can be very experimental. It is easy to simply type a line or two of code and see what it does, much like how easy it is to pull out blocks and click on them to experiment in BYOB. Like BYOB again, variables in Python are **untyped**, which means that after you create a variable it can store any type of data – the same variable could hold, at different times, a number, a string, a list, or any other type of data that Python supports. While it may not be clear at this point why this is worth mentioning, since untyped variables are all you've seen, when we see Java below you'll see a different approach. While there are different ways of executing Python code, in the basic form

Python is an **interpreted language**. That means that there is a program, called an **interpreter**, that reads your Python code and figures out – as the code is running – how to execute those statements. Again, since BYOB and Python are both interpreted, it may not be clear why this is important, but you will see a different approach with Java. While BYOB and Python are the same with respect to those two characteristics – they both have untyped variables and are interpreted – Python is useful for writing much more complex programs than BYOB. Consider that while BYOB has around 125 different built-in blocks you can use, Python has thousands of such programming constructs and functions you can use in your program. There is also a large community of people who write useful code for Python and distribute it for others to use freely, and if you count that then there are probably over a hundred thousand different Python functions that you can use in writing your own programs. That’s a lot of power!

While BYOB is both a programming language and a system for creating those programs, there are many different systems you can use for writing Python programs. The one we’ll use is called IDLE² – you start IDLE on the UNCG lab systems by looking under “All Programs” in the Windows Start Menu and looking in the “Python” folder for “IDLE (Python GUI)”. Start it now!

The window that opens is the “Python Shell” that you can use to interact with the Python interpreter directly. There’s a “prompt” consisting of three greater than signs “>>>” and you can type Python programming statements directly at the prompt. For example, type “4*9” and it will print out “36” in response:

```
>>> 4*9
36
>>>
```

In Python you save a value in a variable by using the variable name followed by an equals sign, and the value you want to set it to – this is the equivalent of the “set” block in BYOB. For example, you can set the variable “x” to the value 12, and then we can use x in expressions just like in regular algebra. Note that you don’t have to do anything to create the variable x first – you can just go ahead and use it, and it will be created for you. For example, consider:

```
>>> x=12
>>> 3*x+9
45
>>>
```

Take a little time to experiment with the Python Shell in IDLE. Set some variables to different values and use them in different calculations. Once you have a feel for how Python interaction works, you should download a file with the solutions to both of the problems from the pre-lab reading. Go to the Lab Exercises page of the class web site, and right click on the link to “Python Code” under “Lab 6”, and select “Save link as...”. In the file dialog window, select a location to save this file (in a folder on the “S:” drive is recommended at UNCG).

² There’s actually a little pun here. While the serious meaning of “IDLE” is the “Integrated DeveLopment Environment,” it is named this way partially because Eric Idle was a member of Monty Python – get it?

Next, load this program into IDLE: In the “File” menu select “Open...” and then find the file you just saved to open. It should now open a new window, and you should see about a dozen lines of code in that window. The code that loads consists of two function definitions, which we’d say are two “custom block definitions” in BYOB terminology. Let’s look at the first one of these:

```
def sum_of_multiples(limit):
    sum = 0
    for i in range(1, limit):
        if (i%3 == 0) or (i%5 == 0):
            sum += i
    return sum
```

Let’s examine this a little bit at a time. The first line tells Python that you are defining a function named “sum_of_multiples.” Unlike BYOB, you cannot put spaces in a Python function name, and the standard naming convention for Python instructs a programmer to use underscores between words in a name³. Parameters are included and named on this line, in parentheses after the name. Unlike BYOB, you cannot put parameters between different parts of the function name (like we did when we defined a block “GCD of ... and ...”) – all parameters have to go after the function name. This line ends with a colon, to indicate that the lines that follow it are controlled by this line, much like a “C-block” in BYOB controls the blocks that it contains.

Here’s one of the most important things about Python: If one line “controls” other lines, the lines that it controls are indented underneath it. The first line underneath the controlling line that lines up with it at the same indentation level marks the end of the controlled lines. Consider the line that starts with “for” above. The first line that lines up with this is the one that starts with “return”, so everything between the “for” and the “return” are controlled by the “for” statement. In BYOB terms, all the lines between the “for” and the “return” would be inside the C-block for iteration (like “repeat”). Grouping/controlling lines by indentation level is a fairly unique characteristic of Python, and not one used by any other major programming language. Some people love it and some people hate it, but very few programmers are indifferent about this unique feature.

Following the function definition line, the next line initializes the variable “sum” to 0. Then there’s a line that starts with “for” – this is the counter pattern all in a single line! The counter variable is named “i” (we used “sCounter” for this in BYOB – in most programming languages, the basic counter pattern will use single-letter variables named “i”, “j”, or “k”), and takes values in the range 1..limit-1. Did you catch that? It goes through limit-1, not limit. For a variety of reasons, the “range” function in Python stops one short of the endpoint of the range. So if we call this function with an argument of 1000 (so “limit” is set to 1000), the last value of i that is used inside the for loop is actually 999. That’s confusing to many people at first, but Python programmers get used to it quickly.

Inside the for loop is an if statement. Compare the condition that is checked in the “if” with the condition in the BYOB solution. Can you figure out what “i%3 == 0” means? It turns out that the

³ See <http://legacy.python.org/dev/peps/pep-0008/> for complete Python style guidelines – naming conventions are just one part of the larger style guide.

percent sign is commonly used in many programming languages for the “mod” operator, and the “double equals” sign is a comparison (like the BYOB predicate “=” block). Two equal signs are used to make it different from setting a variable, which uses a single equals sign. Using these two observations, “i%3 == 0” is a formula that evaluates to true or false, just like the BYOB predicate:

The only other part of the “if” condition is the word “or”, which has the obvious meaning. One last thing to describe here is the “sum += line”. In Python, “+=” does the exact same thing as the “change” block in BYOB – it adds to the current value of a variable, so this is exactly like the “change sum by sCounter” block in the BYOB solution.

The last thing in the function definition is the “return” line, which is exactly like the “report” block in BYOB. Read through this code and discuss it with your lab partner. Talk through what would happen, step by step, if this were executed with an argument of 6.

And now finally, let’s run it and see what happens! Unlike BYOB, these definitions can’t be used just because you can see them and they are in IDLE. To make them available, you need to have Python process the “Lab6.py” file, which you can do by selecting “Run Module” in the “Run” menu (note that you can also do this by pressing F5). When you run the module, you’ll see a message in the Python Shell window that says “RESTART” and it will give you a new Python Shell prompt. Try typing `sum_of_multiples(10)` – you should see this:

```
>>> sum_of_multiples(10)
23
>>>
```

And 23 is the correct answer! Now try `sum_of_multiples(1000)`. Does it give you the same answer that your BYOB solution gave? It should! Did you notice a speed difference between BYOB and Python? You should have!

Let’s see how to time a function in Python. What we’re about to do is not the most accurate way of timing, but it is probably the easiest way, and easy is good for now.

Python has thousands of functions that you can use, but not all of them are available all the time. They are organized in modules, and the ones we want are in the “time” module, which you can load by typing the following in to the Python Shell:

```
>>> import time
```

Now try typing “`time.time()`” at the Python prompt, which should give you a very large number. This actually the same exact concept as the “timer” in BYOB – it’s a number of seconds, but it’s the number of seconds since January 1, 1970, and you can’t reset it! So now we need one last Python trick: you can put multiple Python statements on a single line if you separate them by semicolons. We will use this to do the same basic thing we did to time BYOB code: First we will

save the time when we start (rather than resetting the timer), then we will run our function, and then finally we'll record the elapsed time. Here's what that looks like, timing how long our math problem solution takes to run:

```
>>> starttime=time.time(); sum_of_multiples(1000); elapsed=time.time()-starttime
```

After you do this, you can simply type "elapsed" at the prompt to get the value of the "elapsed" variable and see how long the function took to run. Make sure you and your lab partner write down the elapsed time, because you'll need it in the lab quiz! Finally, since that was so fast, try the same thing with the parameter changed to a hundred million (100000000), and see how long that takes (remember to write down that time as well!).

There's one last thing for you to do with this solution. If you understood the description above about how this Python program works, change it so that it adds up all values that are multiple of 3, 5, or 7. When you change the function definition, you will need to save it and run it again before you can use your modified definition. If you do this correctly, then it should calculate for you that the sum of all multiples of 3, 5, or 7 below 1000 is 271066.

Finally, let's look at the solution to the bioinformatics problem, which is in the "countDNA" function defined in the same file you have been working with:

```
def count_DNA(string):  
    counts = { 'A': 0, 'C': 0, 'G': 0, 'T': 0 }  
    for c in string:  
        counts[c] += 1  
    return counts
```

You should understand parts of this, but there are also some new features. First, the "counts" variable is initialized with a data type that Python calls a "dictionary." We recently talked about the "Dictionary ADT" in class, and that's exactly what this is. It's a collection of key/value pairs, and this particular statement sets up four dictionary entries with keys 'A', 'C', 'G', and 'T', all of which have an initial value of zero. These are our counters!

The version of the "for" loop in this solution is just the iterator pattern from Lab 5, all in one simple statement. This statement will go through the string variable one character at a time, where each character is assigned to the variable c. For example, if we were to execute this for loop with "CAT" stored in variable string, it would execute the body of the for loop 3 times, with c having the value 'C' the first time it was executed, the value 'A' the second time it was executed, and the value 'T' the third time it was executed. Isn't that a simple way to use the iterator pattern!

Finally, inside the "for" loop, you see a "+=" line like in our previous solution, so the variable on the left side is being changed (increased) by one. What is the variable on the left side? It is the dictionary entry corresponding to the key given in variable c. So if the current character we are looking at is 'C', it would increase the counter for 'C' by one. Compare this code with the BYOB

solution. See how much simpler this solution is? That's primarily because of the power of the dictionary data type – it's amazing how many common programming tasks are simplified by using a dictionary!

OK, enough talk – let's run this function. In the Python Shell, call the function with a string argument, remembering that strings are enclosed in quotation marks. Use the same string that was in the example in the pre-lab reading, and you should see this:

```
>>> count_DNA('ATGCTTCAGAAAGGTCTTACG')
{'A': 6, 'C': 4, 'T': 6, 'G': 5}
>>>
```

Use the same technique that was described above to time how long `count_DNA` takes, and record your time for reporting in the quiz.

Finally, make a change to the `count_DNA` function: what if the DNA sequence could also include the character 'X', to indicate that that particular base pair is unknown. Change the `count_DNA` function so that it will count the number of X's as well as A's, C's, G's, and T's. It turns out that you only need to change the line that initializes the "counts" variable, and you should be able to figure that out given just the information about Python described above. When you've done that and tested it, make sure you've got everything saved as `Lab6.py`. You're finished with Python now, so you can shut down IDLE by using "Exit" in the File menu.

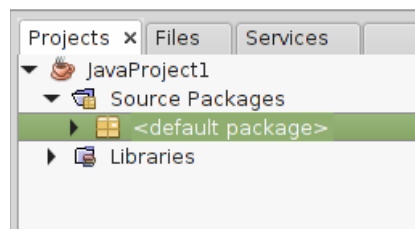
Activity 3 (Java): Next we will look at Java solutions for these two problems, and see how you write and run Java programs. Java is another high-level language, but is different from Python (and BYOB) in several important ways. First, in Java all variables are **typed**, which means you must create them with a **declaration** statement that indicates what type of data will be stored in the variable. Once you do that, the variable is restricted to holding only that type of data. For example, to create a variable named "x" that will hold an integer value you would use the Java statement "int x". When created this way, the variable "x" can never hold any other type of data, such as strings or lists. Java is also a **compiled language**, which means the program you write in Java must first be translated by a program called a **compiler** from Java into a language that the computer can understand more directly. Among other things, this means that Java does not have an "interactive mode" like the Python Shell. Both typed variables and requiring a running the code through a compiler first make Java somewhat less flexible and experimental than Python. It's more difficult to just try things out and see what happens. However, these two properties also have some advantages. By making variables typed, the programmer must indicate how she intends to use a variable. If you start off saying you want to store integer values in a variable, and try to store a string in that variable at some later part of your program, then you have probably made a mistake. The Java compiler would catch this error and tell you about it (and refuse to produce the program that you can actually run), while Python would happily assign the string to the variable without indicating that something is probably wrong. By being a compiled language, the compiler can catch mistakes like this. The compiler can also spend some time looking for the best possible translation of the program to a runnable form, so compiled languages tend to be faster than interpreted languages.

Let's try Java! We won't go into the same detail that we did with Python, as the purpose of this activity is just to give you a feel for what it's like to work with a compiled language. First, we need to download the Java solution from the class web page, similar to the way we did for the Python solution, however in this case it is important to store the Java solution in a folder created just for that code. In other words, right click on the "Lab6.java" link and select "Save link as to bring up the file save dialog. Now navigate into your working folder(on the "S:" drive at UNCG), but before saving right click in this working folder and select "New Folder" – name it "JavaWork" and navigate into it before saving the Lab6.java file.

We will use a program called "Netbeans" to try out our Java solutions. You should find this program in the Windows Start menu and run it. Netbeans is a much more complex program than IDLE, reflecting the more advanced program development that is often done in Java. While this can be confusing, just stick with the basic commands described in this lab write-up and you shouldn't have any problems.

Once Netbeans is started we need to load the solution that we downloaded. Even this simple task takes several steps: First, select "New Project" from the "File" menu. This will pop up a window that shows you several different categories of project you can create, and several project types for each category. Select "Java" for the category, and "Java Project with Existing Sources" as the project type, and then click "Next". On the next screen it will give you some default entries for the name of the project, where to store it, etc. It's OK just to accept the defaults here, although you can pick a different name if you'd like. Click "Next" to move on from this screen. On the next screen you select the "Source Package Folders" – this is why you had to save the solution from the web page in its own folder! Click on "Add Folder", and navigate to the folder where you saved the Java solution from the class web page, and then click "Open". When you have done that, you can click "Finish" at the bottom, and your Java project will be created.

You should now see a "Projects" pane in the upper left of the Netbeans window that looks like this – depending on the version and system you are using, this might look slightly different (for example, there might be "+" icons to the left of items rather than triangles):



You should click on the triangle (or plus) to the left of "<default package>" and it will open up that package and show you the Lab6.java file that contains the Java solution. Double-click on the "Lab6.java" entry, and it should pop up in Netbeans to the right in what is called the editor pane.

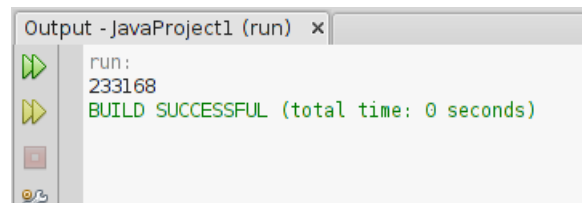
You can see the Java solution now, with two functions defined for the two problems. The function for the math problem is named “sumOfMultiples” and the name for the bioinformatics problem is named “countDNA”. In some ways this code looks similar to Python (look at the “if” statement inside sumOfMultiples, for example), and in some ways looks very different (look inside the “for” loop in countDNA). Take a few minutes to look through this code and see if you can make any sense out of it. We’re not going to go through it like we did with the Java code, but it’s good just to see what you can understand.

In Java you typically have a “main” function that is where the program starts (think of this as a “When green flag clicked” hat block in BYOB). Find this function in the Lab6.java file. The first line in this function says “int testToRun = 1;” which is a declaration statement (as described above) that says we want a variable named “testToRun”, that it will hold an integer value (an “int”), and that we are initializing it to the value 1. This solution file will run either the math solution or the bioinformatics solution, depending on the value of this variable. The value of 1 means to run the math solution.

To run the program, find the button at the top of Netbeans that looks like a green arrow:



The first time you click that, it will ask you to select the main class – just click OK. What should happen at this point is that the program should run, with the output shown in a new pane below the editor pane. The output will look something like this:



See the number 233168? That’s the answer to the math problem!

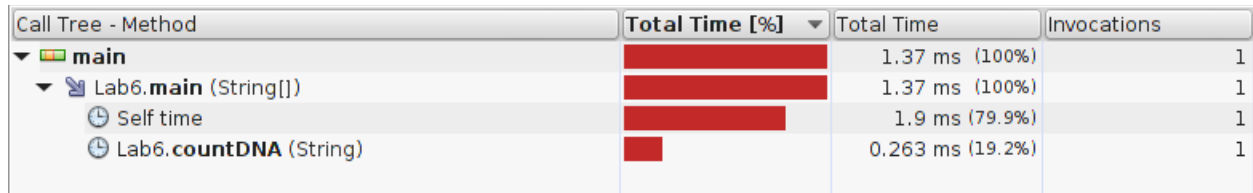
Next, go into the editor window and change the initialization value of testToRun from 1 to 2, and then click the green triangle to run the program again. Now you’ll see the output to the bioinformatics problem.

Finally, let’s time how long these Java programs take to run. Built in to Netbeans is the ability to “profile” your program, or to time all the different parts of the program so you can see how much time it is spending in different parts of your code. This is very helpful if you are writing a big program and want to see where it is running slowly! To profile your code, run it by clicking on the icon that looks like a green arrow on top of a stopwatch:



This will pop up a window asking for profiling options. Select “Advanced (instrumented)” and then click “Run”. The first time you perform profiling, it will probably say it needs to calibrate the profiler. Just click through these dialog boxes to allow the system to calibrate itself.

When your program finishes running, a window should pop up that says “Application Finished”, and it will ask you if you want to take a snapshot of the collected results – click “Yes”, and you will see something that looks like this:



Call Tree - Method	Total Time [%]	Total Time	Invocations
main		1.37 ms (100%)	1
Lab6.main (String[])		1.37 ms (100%)	1
Self time		1.9 ms (79.9%)	1
Lab6.countDNA (String)		0.263 ms (19.2%)	1

If you look at the “Total Time” column to the right of “Lab6.countDNA”, you’ll see that the program spent 0.263 ms (or 0.000263 seconds) to run the countDNA function.

For your final task in this lab, close this profile snapshot (click the “x” on the tab at the top of the profile) and you’ll get back to the editor pane. Change the testToRun value back to 1 (so you are running the math solution), and figure out how to make it do the sum through 100 million, like you did in Python (hint: look below the testToRun initialization and find the call to sumOfMultiples and find the argument). Profile this code and see how long it takes sumOfMultiples to run. Record this time so you can report it in the lab quiz, make sure your updated file is saved, and then you can exit Netbeans.

Submission

You should submit the final versions of all three files: Lab6-Activity1.ypr, Lab6.py, and Lab6.java.

Discussion (Post-Lab Follow-up)

There’s some really cool stuff to discuss about different languages, but that will have to wait....

Terminology

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

- ***compiled language***: a language in which the program is first transformed into a different language or form by a compiler, and then the program that is actually executed is the transformed version.
- ***compiler***: a program which translates a program written in a high-level language into a language or form that is easier for the computer to execute.
- ***declaration***: a statement that creates a typed variable, declaring that it will hold data of a particular type.
- ***interpreted language***: a language which is executed by running programs through an interpreter.
- ***interpreter***: a program which executes an interpreted language, interpreting each statement of the programming language as it runs.

- *Project Euler*: A web site that has a large collection of mathematics-themed programming challenges.
- *Rosalind*: A web site that has a large collection of bioinformatics-themed programming challenges.
- *typed variable*: a variable (in a language like Java) that is declared to hold data of a particular type (like an integer or a string), and then can only hold data of that type.
- *untyped variable*: a variable in a language like Python or BYOB that can hold data of any type.