Craig Lombardo

CS 150 Lab 2

Professor Liew

2/8/15

1. Introduction – This lab was created with intention of providing us with exposure to key testing features available through using the Java API. The goals of part a included creating and testing out the basics of JUnit to test a provided class. The goals of part b extended our knowledge of unit testing, while incorporating working with random numbers, measuring and evaluating execution times, and gaining our first exposure to generics. Another goal, which I found to be important however not stated, was recognizing the importance of maximizing a program. For this lab I, only at first, assumed that what I was doing was most efficient. In the very beginning I assumed that ArrayLists are the fastest for this lab, I know now this may not be the case. I also had to assume that the random, pseudorandom, numbers truly were pseudorandom and that the seed truly did provide limitless variations of pseudorandom numbers. I also had to assume that the method by which I am testing and running things is the fastest (all execution time programs were run through terminal).

2. The design of the program is relatively simple; however, it uses the simplicity to do thousands of actions. The first class created is my RandomIntContainer class, this class creates an Integer container that will hold random values in the form of an ArrayList. When an instance of this class is created a size integer is passed in which then gives the program a “hint” as to how large our ArrayList will be. Every instance of the RandomIntContainer class has their own ArrayList and the ability to add to the list from the front, end, or sorted, as well as return this ArrayList for testing purposes. The second class is Wheel, each instance of this class is created with two integer values, a seed and an upper bound. These two numbers are individual to each instance of Wheel and are used in the generation of pseudorandom numbers. With the use of these two in a class called ExperimentController we are able to calculate the execution time taken to add numbers from the front, end, or in a sorted application. This class has methods within it which allow me to get averages on execution time in relation to the ArrayList size, as created through implementation of RandomIntContainer.

3. The setup of this program was broken down into different parts. There were methods created that ran the experimental work. Each test was run multiple times, the first round for each of the tests (adding to the front, end and sorted) was a small sample size to see if it was enough to come to a valid conclusion as to the program’s complexity. I quickly found this was not nearly enough testing to come to a conclusive result. I then ran the tests again, this time with a much larger step size, increase in number of data points not number of tests per each individual test type. This was enough to come to a reasonably clear result; however, given computer usage affecting data points I quickly found averaging the time taken for tests was important, which led me to my final runs of 10 tests per size. After I got conclusive results I decided to test each one yet again, using the same final process but with a different seed to determine whether or not my answer held validity. Although the times were not identical, it further proved that the complexity matched, which is most important. For the tests, the only parameters I changed were which test was being examined, how many iterations of each to run, the step size from one test size to another, and the seed value. I decided to add the parameter option of which test to run, which others may not have done, because I wanted to maintain control of which test to run at a time, I did not want to run all three at the same time. I wanted to maintain efficiency but increase my control so I may easily retest or revamp parameters. I decided I would change the parameters based on my initial tests, for example, I quickly learned using small numbers, such as those in the thousands, was a complete waste for the test that adds numbers to the end of the ArrayList so I made the step size 1,000,000. I also learned that a step size like this for the sorted test or front test, was too much as the numbers caused huge delay times and made testing take much longer than it should have.

4. The data was somewhat unexpected; however, it makes sense in the end. At first glance of the graphs, I see that the graph for the add to front test and add sorted test have similar looking graphs; however, the add to end test has a different looking graph, I initially expected all three to have different complexities, thus different looking graphs. I came to the result that the complexity of the front and sorted test was O(n) while the complexity of the add to end test was O(1). My rationale for this was the first test moves n elements each time, which is guaranteed O(n); the sorted test can also, although not guaranteed, move n elements each time so it too has a complexity of O(n). The end test is O(1) because it only has to add one element to the end of the ArrayList, there is no need to move any other data points. As shown with the graphs, those for front and sorted data have execution times that increase as the size increases, where as execution per number stays the same for the different sizes; the overall only takes longer because we are adding more and more numbers.

This graph is a plot of the average execution time vs the number of numbers added to the front of the ArrayList. We can see here that the average time increases as the sample size increases; however, not in a linear distribution which concludes the point of O(n) complexity.

This graph is a plot of the average time vs the number of numbers added to the end of the ArrayList. We can see by this graph that there is a linear distribution between the number and time, which means the complexity is O(1) as it takes the same amount to add each element to the end.

This graph is a plot of average time vs number of numbers for adding sorted, which follows the same distribution as that of the add to front test; the complexity for this graph is also O(n).

5. Based on this data I can come to the conclusion that ArrayLists are much faster when trying to add elements to the end of the list; however, should you need to add to the beginning or in a sorted fashion they are slower but take roughly the same time. I am also fairly confident, based on my experimental date, that the complexity for adding to the end of an ArrayList is O(1) and to add either in the beginning or in a sorted fashion is complexity O(n).

6. <http://www.cs.lafayette.edu/~liew/courses/cs150/lab/labs/lab02f/>

<http://www.mycstutorials.com/articles/sorting/insertionsort>