Lab 5 - Evaluating Algorithm Performance
Due (via Bitbucket) Thursday, 9 March 2017
50 points

Lab Goals

- Practice timing the performance speed of algorithms on a variety of input sizes
- Evaluate the Big-O performance of said algorithms
- Answer a few questions to test your knowledge about the class content to-date

Assignment Details

In the past few classes, we have been discussing the performance of algorithms. We saw that there are three primary ways to evaluate the performance of an algorithm: measuring the running time of a section of code, counting the number of primitive operations with respect to input size, and giving a general estimate of the number of primitive operations by examining code structure. In this lab, we will write a few short functions and measure the performance of the code by these methods.

String vs StringBuilder (15 points)

Java supports a few different methods for storing strings of characters. The first is the String class, which is the one we have most commonly used so far. One downside to Strings is that they are immutable objects – once created, they cannot be changed. That means that when concatenating two Strings, we are in fact creating a brand new String. In other words, the following code segment:

```java
String str1 = "My String";
str1 = str1 + " is awesome!";
```

actually replaces the original str1 reference with a new String that is declared and initialized implicitly to the phrase “My String is awesome!”. Concatenating and modifying Strings is thus an expensive process in Java. For strings that will be modified frequently, StringBuilder is a better choice. StringBuilder objects are mutable objects. By calling the append() method of a StringBuilder object, the internal character array will be altered, which is a much more efficient process than creating a whole new object.

In this section of the lab, we will test just how quickly we can append strings with both String and StringBuilder objects. You should create a class (the name is up to you) to compare the
running time of these classes. You should use `System.nanoTime()` to measure this performance. In your code, you will be running a loop of various lengths, appending a single character onto one object from each class. Your `String` loop should look something like:

```java
String myString = new String();
for (int i = 0; i < N; i++) {
    myString = myString + "a";
} //for
```

and your `StringBuilder` loop should look something like:

```java
StringBuilder mySB = new StringBuilder();
for (int i = 0; i < N; i++) {
    mySB.append("a");
} //for
```

You should add your timing code around these loops, to measure how long it takes to run through the loops. You should time runs with loop sizes (values of N) of 1000, 10000, 50000, and 100000. For each of these loop sizes, you should measure 5 running times, to ensure that one outlying datapoint isn’t confusing your analysis.

You should then take the average of these timings for each loop size, and then plot them on a chart. The x-axis of your chart should measure loop size, and the y-axis of your chart should note the running time.

When complete, you should see average timing results somewhat similar to what is reported in the textbook on pages 152-153.

**Arrays vs LinkedList vs LL vs IteratingGenericLL** (25 points)

Since we spent some time earlier in the class discussing both arrays and linked lists, it would also be interesting to evaluate their performance with respect to each other. Since we also created our own LL and `IteratingGenericLL` classes, we can include these in the analysis as well.

For this section of the lab, you should test both insertion and retrieval from arrays, Java’s `LinkedList`, our LL class, and our `IteratingGenericLL` class. For the insertion component, you can add 1000, 10000, 50000, and 100000 items (values of N) to an integer array, `LinkedList`, LL, and `IteratingGenericLL` data structures. For retrieval, you can do something similar, using the `get()` function on the created lists (and an Iterator on the `IteratingGenericLL`), and using the index in the array. See the code on the next page.

As with the last section, you should measure 5 running times for each of the array and list sizes, to ensure that one outlying datapoint isn’t confusing your analysis. Additionally, you should take the average of these timings for each array/list size and plot them on a chart. The x-axis of the chart should measure the array/list size, and the y-axis of the chart should note the running time.
// Add timing code around each of these code chunks and measure them separately
// Add to array
int[] myArray = new int[N];
for (int i = 0; i < N; i++) {
    myArray[i] = i;
} //for

// Add to Java’s LinkedList
LinkedList<Integer> myList = new LinkedList<Integer>();
for (int i = 0; i < N; i++) {
    myList.add(i);
} //for

// Add to our LinkedList
LL mySList = new LL();
for (int i = 0; i < N; i++) {
    mySList.add(i);
} //for

// Add to our IteratingGenericLL
IteratingGenericLL<Integer> myIList = new IteratingGenericLL<Integer>();
for (int i = 0; i < N; i++) {
    myIList.add(i);
} //for

// Get from array
for (int i = 0; i < N; i++) {
    int something = myArray[i];
} //for

// Get from Java’s LinkedList
for (int i = 0; i < N; i++) {
    int variable = myList.get(i);
} //for

// Get from our LinkedList
for (int i = 0; i < N; i++) {
    int somethingElse = mySList.get(i);
} //for

// Get from our IteratingGenericLL
Iterator<Integer> iter = myIList.iterator();
while (iter.hasNext()) {
    int anotherThing = iter.next();
} //while
Here is what one trial run of the output (in nanoseconds) looks like on my old and slow laptop for N=10000:

One run results:
Adding to array: 193213
Adding to LinkedList: 3168166
Adding to LL: 1493947
Adding to IteratingLL: 2834840
Get from array: 198122
Get from LinkedList: 96927147
Get from LL: 101330452
Get from IteratingLL: 43786288

Additional Questions (10 points)

Please answer the following questions thoroughly:

1. Evaluate the Big-O order of growth of the following loops and explain why:

   ```
   for (int j = 0; j < N; j++) {
       for (int k = 0; k <= j; k++) {
           total += arr[j];
       } //for
   } //for
   ```

2. Evaluate the Big-O order of growth of the following loop and explain why:

   ```
   for (int j = 0; j < N; j=j+2) {
       total += arr[j];
   } //for
   ```

3. Evaluate the Big-O order of growth of the following loops and explain why:

   ```
   for (int i = 0; i < N; i++) {
       for (int j = 0; j < N; j++) {
           for (int k = 0; k <= j; k++) {
               total += arr[k];
           } //for
       } //for
   } //for
   ```
4. Evaluate the Big-O order of growth of the following loops and explain why:

```java
for (int j = 0; j < N; j++) {
    for (int k = 1; k < N; k = k * 2) {
        total += arr[j];
    }
}
```

**Submission Details**

For this assignment, please submit the following to your `cs112s2017-<your user name>` repository (and ensure that the instructor has access to your repository):

1. Your commented source code for the timing algorithms that you wrote
2. Tables and charts showing the runtimes for the first two sections of the lab
3. The answers to the questions from the “Additional Questions” section