Lab 1A

Experimental Analysis of Algorithms

Download Lab1A.zip archive and examine the classes given: **Unique1, Unique2, AlgAnalysis**.
In this lab you are going to experimentally measure the running time of algorithms. We first give you some direction on how to measure the running time of a method.

Measuring the running time of a method

Ideally, we would like to measure the CPU time of your method, not the elapsed clock time. The difference is that if your operating system is busy with other tasks which are sharing your CPU, the clock time is not going to be a precise measure of the time spent on your method. However, for simplicity you may use **System.nanoTime()** which records the clock time in nanoseconds. Using nanoseconds you can get meaningful values for fast methods that when using seconds would give you always 0.

To measure the time of a method, record the time **before** the call to the method and **after** the call to the method and calculate **after-before**.

Part 1) Experimenting with performance of java.utils.Arrays.sort() method

Method you will implement: **AlgAnalysis.arraySortRuntime(int)**

Java provides the method **java.utils.Arrays.sort(int[])**. From Oracle documentation for this method we get the following note:

> "Implementation note: The sorting algorithm is a Dual-Pivot Quicksort by Vladimir Yaroslavskiy, Jon Bentley, and Joshua Bloch. This algorithm offers O(n log(n)) performance on many data sets that cause other quicksorts to degrade to quadratic performance, and is typically faster than traditional (one-pivot) Quicksort implementations."

It is known that quicksort and its variations have a worst-case running time in O(n^2) while average-case running time in O(n log n). We suspect what the above quote is trying to convey is that this particular variation of Quicksort has a higher concentration around the average, behaving as O(n log n) for most inputs.
In this part of the lab, you will experiment computing the running time (CPU time) for **java.utils.Arrays.sort(int[])** for increasing values of array input size n.

You will design and perform an experiment to verify the following two hypothesis: whether the running time of this method behaves more closely as O(n^2) or more closely as O(n log n).

When running the code, select option 0 for Part 1).
The main program for method **AlgAnalysis** already prompts you for the following quantities: "number of arrays to test"(count) and "largest array size"(maxsize) which are used in the call of the method you must implement: **AlgAnalysisSol.arraySortRuntime(count, maxsize)**.
This means you should test "count" equally spaced array sizes where the largest has size "maxsize".

You will loop through array sizes as specified by the values above; for each array size n you will repeat the experiment x=10 times with randomly shuffled arrays and record the maximum time over the x=10 runs. For each n this "maximum over x=10 runs" is the time that should be reported as T(n) which we will take as the "worst case" running time of the algorithm for input of size n. A method to randomly shuffle an array is provided.
(**Note:** The value x=10 was chosen arbitrarily for not taking too long to run this experiment. Feel free to increase x and verify if the same results are observed.)

To test the two hypothesis above, we ask you print for each n, the following quantities:
n, T(n), T(n)/(n*n), T(n)/(n log n)
If the method indeed behaves as O(n log n), we expect that the last fraction will converge to a constant, while the second-to-last fraction should be decreasing.

Part 2) Comparing the running times of two methods for determining if an array has duplicate entries

Method you will implement: **unique1Runtime(int)** and **unique2Runtime(int)**.

We will be comparing the running times of the methods **Unique1.unique1(int[])** and **Unique2.unique2(int[])**. The two algorithms are theoretically analyzed in the textbook in pages 174-175 and is also discussed in your first DGD.

The worst-case running time of **unique1** is shown to be O(n^2). In your experiments, you must use an input array for each n that gives the worst-case case running time for **unique1** over all inputs of size n. This is not a difficult task if you inspect the method carefully and see how to force all iterations to run.

The worst-case running time of **unique2** is shown to be O(n log n) provided the sorting method it calls java.utils.Arrays.sort(int[]) is O(n log n). Because it is hard to construct a worst case running time for a sorting method we do not know well, you can run this using a randomly selected input array of the given size.

Experiments you should run for part 2):

- Run both methods with the same input size n. For example, you may try n=50, 200, 10000, 20000, 50000, 100000, 200000.
- For each of the methods **unique1** and **unique2**, try to find the answer to the following question: find the largest size of array for which the result can be returned within 1 minute. You may use trial and error to estimate your answer.
- You may organize your findings in a word or text document where you store annotations and outputs obtained.

Summary

- Download the `Lab1A.zip` archive and examine the following classes given
  - **Unique1, Unique2, AlgAnalysis**
- Follow the specifications in Part 1) to implement the method **arraySortRuntime(int count, int maxsize)**
- Choose a large maximum array size (maxsize) and enough number of intermediate sample

points(count) and experimentally verify if **java.utils.Arrays.sort(int[])** seems to behave as O(n^2) or O(n log n). You may try different combinations of count and maxsize.
- Follow the specifications in Part 2) to implement **unique1Runtime(int)** and **unique2Runtime(int)**
- Follow the "Experiments you should run for Part 2)" section.
- Meditate about the practical importance of theoretical analysis of algorithms. :-))
