COSC1285/2123: Algorithms & Analysis
Laboratory 2

Topic

This lab introduces methods for benchmarking and comparing programs.

Objective

- Learn how to benchmark algorithms.

Benchmarking

Evaluation in computer science can be conducted theoretically or empirically. In this lab, we learn how to apply empirical analysis to a class of similar algorithms. The efficiency of an algorithm is often measured as the running time. Here we investigate two simple ways to benchmark the performance of a algorithm: the unix `time` command and the `nanotime()` method call.

Bubblesort

In this part of the lab we will compare the running time of two different *BubbleSort* implementations for different input sets. The exercise for this part of the lab is to **benchmark** the run time of the two given implementations **Bubblesort1.java** and **Bubblesort2.java** using the given sample data sets `test1.in` and `test2.in`. First, download and compile the two files **Bubblesort1** and **Bubblesort2** and perform a run time analysis on them using the input sets `test1.in` and `test2.in`. You are encouraged to test both implementations and compare their results. After you run your experiments you should be able to answer the following questions:

- Which of the two implementations is faster?
- Is it enough to perform an experiment once? Try running an experiment multiple times and compare the results?
- Is the `time` method accurate enough for your experiment?
- How do the two algorithms perform for different input sizes?

Download the code and compile both programs using the following commands:

```
javac BubbleSortUtils.java BubbleSort1.java
javac BubbleSortUtils.java BubbleSort2.java
```

Simple Run Time Evaluation

The easiest way to measure the running time of an algorithm is to use the unix `time` command on the program that implements the algorithm. For example, invoking the `time` command results in the following output:

```
$ time java BubbleSort1 < test1.in
<list of sorted integers>

real        12.0
user        11.0
sys         1.0
```

Note that the format of the timing information may be different depending on the machine you are using, but still to do with timing. The three different time values returned by the program `time` are (use the man pages to find out more details of what they mean):

- The elapsed (**real**) time between invocation of `bubblesort1`
- The **user** CPU time (the sum of the tms_utime and tms_cutime values in a struct `tms` (see `man -s2 times`)
- The **system** CPU time (the sum of the tms_stime and tms_cstime values in a struct `tms` (see `man -s2 times`)

Which of the values would you use in your experiments and why? Discuss with your lab demonstrator!

Accurate Run Time Evaluation

Using the `time` command gives you an easy way to quickly evaluate the run time of a program without the need to modify the actual source code. There are however multiple downsides to this method:

- You are **not** evaluating the run time of your *algorithm* but the *program* that implements it. This includes parts of your program like file I/O, program start up or memory allocation that you actually don't want to benchmark.
- The accuracy of the run time measurements are given in seconds. This is not very accurate in the area of run time analysis. Evaluating a fast algorithm on small input set will be very difficult using the `time` command.

A more accurate way of measuring the run time of an algorithm is to use the `System.nanoTime()` method call:

```
public static long nanoTime();
```

It returns the current time in nanoseconds precision. See javadocs for more details.

Consider the following example that measures the time required to count all prime numbers smaller than n:

```
public class CountPrimes
{
    public static int countPrimes(int n) {
        int count = 0;

        for (int i = n; i > 1; i--) {
            boolean np = false;
            for (int j = 2; j < i; j++) {
                // test if prime
                if (i % j == 0) {
                    np = true;
                    break;
                }

                // increment prime count
                if (!np) {
                    count++;
                }
            }
        }

        return count;
    }


    public static void main(String[] args) {
        int n = 40000;

        long startTime = System.nanoTime();
        CountPrimes.countPrimes(n);
        long endTime = System.nanoTime();

        double estimatedTime = ((double)(endTime - startTime)) / Math.pow(10, 9);

        System.out.println("time taken = " + estimatedTime + " sec");
    }
}
```

Figure 1: Sample time measurement using the `nanoTime()` system call.

The above code sample shows that you simply encapsulate your algorithm within two `nanoTime()` calls to measure its run time in nanoseconds. It is therefore easy to exclude the parts of your program like file I/O, program start up or memory allocation that you actually don't want to benchmark.

Modify both bubblesort implementations according to the sample code above and measure the running time using `nanoTime()`.

Considering the Enviroment

Execution times on a specific machine normally depend upon details of the machine and on the specific data used. Timings may vary from data set to data set and from machine to machine, so experiments from one machine and one data set may not be very helpful

in general. It is therefore important to **carefully consider the environment** in which you benchmark your algorithm. You will be running your experiments on the core teaching servers. These machines are used by many students at a time which in turn can affect the outcome of your experiments. It is common practice to run *multiple times* and use the **average mean** of these results during the evaluation of your experiments.

Data Set Generation

The performance of algorithms is also dependent on the data that it is applied on. Therefore when evaluating an algorithm, it is also desirable to consider test scenarios that the algorithm could face and test with representative data samples of these scenarios.

Sometimes it is not possible to obtain real data that covers the important data scenarios. Hence, it is useful to be able to generate data to test scenarios that we don't have real data for. In this part of the lab, we will study two simple ways to **generate testing data**.

There are many factors to consider, including what values to generate, how many to generate, do we allow duplicate values and how likely we generate a value. To demonstrate how to generate datasets, we generate data for the two Bubblesort algorithms. We consider the simple case of generating (non-negative, or 0 or greater) integers over a range of values that are equally likely to appear in our data samples. This can be implemented as **sampling** from a **uniform distribution**.

We consider two subcases, one where we allow duplicates, one where we do not. In the first subcase (duplicates are allowed), this is sampling (from a uniform distribution) **with replacement**. Without going into details, imagine we have a ball for each integer in the range. We place these balls into a bag, mix them, then draw/sample one ball. We note down the integer represented by the ball, then drop the ball back in (leading to the name "with replacement"). We mix them again, and draw again until we drawn the number of integers (balls) we need.

In the second subcase (only unique values, no duplicates allowed), this is sampling (from a uniform distribution) **without replacement**. Using the ball and integer analogy again, this time, when we draw/sample a ball, we note down the integer of the ball but do not drop the ball back in. We repeat this process until we have drawn the number of integers (balls) we need. Since the balls are not put back into the bag, then it the integer it represents can't be drawn again, so duplicates cannot happen.

Consider the file `DataGenerator.java`. We have implemented sampling without replacement. Your task is to implement sampling with replacement to generate data generation that allows duplicates. Complete the implementation for method `sampleWithReplacement()`. As a hint, consider

`java.util.Random.nextInt()`.

To run the program, type:

```
java DataGenerator <start of range for values> <end of range for values> <number
of values to generate> <with | without>
```

where:

- start of range for values: Start of integer range to generate values from (inclusive), must be 0 or greater.
- end of range for values: End of integer range to generate values from (inclusive), must be 0 or greater.

- number of values to generate: Number of values to generate.
- with | without: using sampling with (without) replacement.

As an example, the following command generates 20 integers in the range 2 to 100, using sampling without replacement (no duplicates):

```
java DataGenerator 2 100 20 without
```

