Assignment overview: Comparing two algorithms

This week we are investigating the running times of two different algorithms for the same **(very simple) problem ("VSP")**. One algorithm is an $O(n^2)$ brute force algorithm, and the other is a transform-and-conquer (pre-sorting) algorithm that is O(nlogn).

You will:

- Write some code
- Run your code and copy some output values into some data tables

The VSP

Given an array of integers, return true iff the array contains any duplicated values.

Yes, this is the logical opposite of the example in the lecture notes (Week 5) for checking "element uniqueness" in an array. "All values are unique" means the opposite of "There exist duplicate values".

We are actually going to be using this algorithm like a subroutine to find an *estimated* answer to the following curious question: What is the probability that an array of random numbers contains any duplicates?

Program overview

Things you'll need to do:

- Write a method that takes an (unsorted) array and uses Brute Force to solve the VSP. This should be the basic "two nested for loops, check all possible pairs" algorithm. *By the way, this program uses plain arrays, not ArrayList.*
- Write a method that takes an (unsorted) array and uses Transform and Conquer (pre-sorting) to solve the VSP. The sorting is considered part of the T&C algorithm and should take place *inside* this method. You can use Arrays.sort() to sort the array. It is an O(NlogN) algorithm (a variant of Quicksort). After the sort, do a linear scan of the array to look for duplicates.
- Write a method to initialize an array with random integers. Repetition is allowed; every number is chosen independently from within the range mentioned below.
- Place some easy-to-find and clearly-labeled declarations of constants/variables somewhere near the top of your code in which the following three values can be set/changed:
    - N (the size of the array).
    - Upper bound of the random numbers. If this value is "10000", that means all the numbers generated and stored in the array will be between 0 and 9999 (inclusive). (I'm calling this value "RANGE" in this handout, but you can use whatever name feels right.)
    - Number of "trials" to be performed.
- Include a main program that performs a number of "trials" (more info below), while tracking and accumulating various results. After all of the trials have been performed, the program will report some of this information.

What are these "trials"?

During *one trial*, this is what your program will do:

- Fill/refill the entire array with new random numbers from the specified range.
- Call the BF algorithm to test the array for duplicates.
- Call the T&C algorithm to test the array for duplicates.
- Accumulate some data about the above things. (Global variables allowed here.)

Your program will be doing *many* trials and then printing some output.

What data are you recording?

- When you call the BF method, track the amount of CPU time it uses. E.g. check the system time before & after you call the method. Add the elapsed time to a global counter to track the *total CPU time for BF over all trials*.
- When you call the T&C method, track its CPU time (separately) the same way.
- Whenever the BF method returns "true", increment a global counter for total number of times that duplicates were found.
- Whenever the T&C method returns "true", same. *Obviously these two global counters should end up identical.* This can also serve as a sanity check for whether your two methods are correct. (It did for me—that's how I discovered that I had a typo in one of them!)

Program output

Your program should display the following output (one time, after all the trials are completed):

- Size of array being used (N)
- Range used for random numbers (max value RANGE aka "0 to RANGE-1")
- Number of trials performed (NUM_TRIALS)
- How many times the array contained duplicates (a value X somewhere in the range 0 to NUM_TRIALS inclusive)
- Probability of array containing duplicates (expressed as a percentage: X / NUM_TRIALS * 100)
- Total amount of CPU time used for BF testing (sum of NUM_TRIALS individual results), given in milliseconds
- Average amount of CPU time used for BF testing (total BF CPU / NUM_TRIALS), given in milliseconds
- Total amount of CPU time used for T&C testing (sum of NUM_TRIALS individual results), given in milliseconds
- Average amount of CPU time used for T&C testing (total T&C CPU / NUM_TRIALS), given in milliseconds

Note: Some of the above calculations have floating point results, but all the data are integers. Watch out for integer-division mistakes.

Expected results

The value for "Probability of array containing duplicates" is probably the number that will give you the best idea (while you are testing/debugging) of whether your code is on the right track. This value is theoretically predetermined by your settings for array size (N) and random number RANGE. However, it will vary slightly each execution because of the randomness.

Here are some settings you can experiment with. This is not "required input" for you to code into your program, and 50% and 37% are not "required output" either. This section is just some numbers for your convenience in testing.

These values for N and RANGE should give you results right around 50% for "Probability of array containing duplicates":

```
10, 70
23, 365
100, 7000
5000, 18000000
```

These values for N and RANGE should give you results right around 37% for "Probability of array containing duplicates":

```
10, 100
50, 2650
100, 10700
3737, 10000000
```

The more trials you can do, the "better" your numbers will usually be – more consistent/clustered and generally closer to the expected results (50 and 37). But see note below about Random Numbers.

Also note: With the exception of ONE pair of values above, I got all of those numbers above by imprecise trial and error. There is some actual math that you could use to find settings that would more accurately give you probabilities closer to 50% or 37% or whatever else you want. But I was not feeling like doing the math. In the case of the one that is precise, the actual probability is more like 50.7%.

Once your code is all working

There is a worksheet to finish. (See separate Word document "Lab5 Worksheet – HANDMEIN".)

The worksheet is about comparing the running time of our two algorithms on different input sizes.

Other interesting data things

It's interesting to tinker with different combinations of N and RANGE, and see how likely it is that you will end up with arrays containing duplicates.

- If you put RANGE as any number <N, the "probability" will *always* be 100%. There's no way you can pick N whole numbers that are all less than N, and have them all be different!

- The bigger you make RANGE (way bigger than N), the more likely it is that you will get an array that contains all-different numbers. Try this! Set N to some particular value, then run your program several times with larger and larger values of RANGE. You should see the "Probability of array containing duplicates" getting smaller and smaller. Eventually it may be small enough that you have to do many trials just to get any random array without duplicates.

Random number notes

Whenever you're dealing with random numbers like this, the more trials you can perform, the "better" your data will usually be, i.e. the closer you might get to some theoretically-expected values. So it's better to do more trials if you can. More as in *thousands*. Tens or hundreds of thousands. Even *millions* ... if you can.

But you are also dealing partly with a "slow" algorithm here (the BF one), which means that the bigger your array is, the longer every single trial will take (increasing proportionally to N^2), and thus you will be more and more limited in *how many* trials you can do without severely testing your patience. Trade-offs.

When you are doing the worksheet, NUM_TRIALS=1000 is about as high as you can reasonably go before things start running a little bit slow.

Point breakdown

- Main program (4 points)
- BF algo (2 points)
- T&C algo (2 points)
- Coding style (4 points)
- Worksheet (8 points)

How, what, and when to submit:

Please submit the following to the dropbox on Learning Hub:

- Java source code
- Completed worksheet (separate Word Doc)
- File names not important

You may (and should!) discuss the lab and coding techniques with your classmates, but all of the work you submit to Learning Hub must be your own.

This lab is worth 20 points.

It is due at midnight next Wednesday.

Virtual donut #1

Look at what happens to the probability that the array will contain duplicates if you put RANGE *exactly equal to* N (showing some selected values here):

```
N == RANGE ==  2 --> 50%
```

```
N == RANGE ==  3 --> 77.777778%
N == RANGE ==  4 --> 90.625%
N == RANGE ==  5 --> 96.16%
N == RANGE ==  6 --> 98.456790%
N == RANGE ==  7 --> 99.388010%
N == RANGE == 10 --> 99.963712%
N == RANGE == 13 --> 99.997944%
N == RANGE == 16 --> 99.999886%
N == RANGE == 19 --> 99.999993%
```

Can you find a formula (based on N) that predicts these answers? Hint: think about what it really means when RANGE == N. What is the VSP asking in that situation?

Virtual donut #2

Suppose you have a fixed array size of a particular N. As noted above, the bigger RANGE gets, the less likely there will be duplicates in the randomly-generated array. For example, below is some output for arrays of size N=100 (based on 2000 trials per test).

```
RANGE: 5000 ... %WithDupes: 64.5
RANGE: 6000 ... %WithDupes: 54.5
RANGE: 7000 ... %WithDupes: 50.8
RANGE: 8000 ... %WithDupes: 47.75
RANGE: 9000 ... %WithDupes: 43.85
RANGE: 10000 ... %WithDupes: 37.4
RANGE: 11000 ... %WithDupes: 35.6
RANGE: 12000 ... %WithDupes: 33.8
RANGE: 13000 ... %WithDupes: 32.9
RANGE: 14000 ... %WithDupes: 28.95
RANGE: 15000 ... %WithDupes: 26.8
```

These numbers are not precise because of randomness, but the trend is clear. The question is: can we find a formula for the precise probabilities? E.g. Find a formula for RANGE as a function of N that tells when the probability of duplicates gets lower than 37%?

This is not an "algorithms" problem. It's really a math/probability problem, and a kinda messy one at that. It's almost more of a "figure out the right thing to google" problem, but even once you find the right thing to google, there's some math for you to figure out the answer. If you get this one, you will have earned some serious donut respect. May be better to focus on midterms.