CSCU9A2 Practical 7B (Week 9) Spring 2017

Sorting experiments 16th March

If you get stuck or need help at any time during the practical, ask a demonstrator.

Remember: ***Week 7's checkpoints (sheets 5A and 5B) must be completed by the end of this week. You have until the end of week 11 to complete this week's checkpoints.***

THIS WORKSHEET:

In this practical you will do some timing experiments on the Selection sort and Quicksort algorithms.

TIMING SELECTION SORT

1. In a slight variation of what you have done in previous practicals, copy the ***folder*** **SelectionSort** from **Groups on Wide (V:)\CSCU9A2\Java** to your CSCU9A2 working folder.

   Launch BlueJ. Using the **Project** menu, **Open Non BlueJ...** create a BlueJ project in your **SelectionSort** folder – remember, when you navigate to it click *once* on the folder name then Open in BlueJ – do not double click to open the folder.

   Look at the classes: The structure is identical to the selection sort timer example seen in lectures: **ArrayUtil** is a general purpose *library* of useful methods, seen in lectures. **StopWatch** is a special purpose helper class to carry out timing (it is not necessary to read and understand this class). **SelectionSortTimer** is the main program – it contains a **main** method that can be launched, and the **selectionSort** method.

   SelectionSortTimer invites the user to enter the size of the array to be sorted, creates a new array of the appropriate size, fills it with a range of random integers, and then runs the **selectionSort** method to sort it, displaying the unsorted and sorted arrays in the terminal window. A new **StopWatch** is created and used to time the sorting process: it is *started* just *before* the call of **selectionSort**, is *stopped* immediately *afterwards*, and the elapsed time (in milliseconds) is reported in the terminal window. Placing **start** and **stop** just before and after the **selectionSort** call means that JVM start-up and program loading times are *not included* in the measurement of the elapsed time. The main program has no GUI, it just runs in BlueJ's terminal window – it must be launched repeatedly to carry out experiments.

   Compile and run the program — it should be fine – and carry out the experiments below.

2. To see the basic action of the program: Launch the **SelectionSortTimer main** method, enter 10 at the prompt in the terminal window, and look at the resulting output. Repeat this, entering 10 at the prompt again: notice that the array contains *different* numbers from the last time – the random number generator is working well! Repeat several times, entering, say, 20, 30, 100 for the array size. Everything should be fine. You will see that the elapsed time is (probably) reported as 0 milliseconds – that is not an error, the computer is simply so fast that it takes less than 1/1000 of a second to sort the array!

3. Run **SelectionSortTimer** again, this time entering 1000. The terminal window output starts to get a bit silly: many, many random numbers being displayed that are not actually very interesting, followed by the elapsed time which *is* interesting in this practical. Locate the two

lines in the main method that use **System.out.println** to display the array, and *comment them out*.

4. Now for some real timings: The best array sizes here depend on how fast your computer actually is, so experimentation may be needed. Try array sizes 1000, 10000, 100000 (and maybe some values in between) until you get a time that is perhaps around 100 or so milliseconds (the exact time does not matter) – on my PC it is an array size of 8000. In the instructions below, that size is referred to as the "basic size".

5 Repeat the timing for the basic size *five separate times*, and record the elapsed times. You should see that the timing varies quite a lot: This is because firstly, the JVM may be doing other management activities "behind the scenes" that add unpredictably to the time, and secondly Windows itself may be doing the same. This is a common problem with trying to *measure true execution times* by recording *overall elapsed times*. We will assume that an average of the measured times is a good representative of actual execution time. In some sorting algorithms, the actual time also varies with the precise collection of random numbers generated, but selection sort does *not* have that characteristic.

   So, calculate the average time to selection sort the basic size of array

6. Now try a range of array sizes building on that basic size, such as 2x the basic size, 3x, 5x, 10x, 20x, 30x, 50x, 100x, etc. For each size, repeat the measurement, say, five times, ***write down the time taken for each, and then calculate the average time for each array size***.

7. Sketch a graph showing your results with the array size along the horizontal x axis, and the time taken up the y axis. You should be able to see clearly that a line joining the points curves upwards towards the right.

8. Compare various pairs of average timings for array sizes *where one is double the other*, for example the basic size and 2x, or 10x and 20x, etc. In each case the ratio should be roughly 4 – meaning that if you double the amount of data that **selectionSort** has to sort, then it takes about 4 times as long.

9. Checkpoint [Sorting]: Now show a tutor your graph and your analysis of the ratios of pairs of readings. Answer any questions they ask you.

AN INTRODUCTION TO QUICKSORT

Quicksort is a *recursive* sorting method, rather like merge sort (seen in lectures). The merge sort algorithm splits an array into two equal partitions (copying into two new arrays), recursively merge sorts each partition, and then merges the sorted partitions back into the original array. The performance of both quicksort and merge sort is stated as O(n log n) – slower than a "linear" O(n) algorithm (there are no general purpose sorting algorithm that are that efficient), but faster than a "quadratic" O(n^2) algorithm such as selection sort.

Quicksort works roughly like this to sort an array:

1. Pick some value, the *first* value in the array is convenient (although not always the best) – this is called the "pivot".
2. In one (intricate) scan of the array, rearrange the values (within the array, no new arrays needed) so that all values less than the pivot are in lower index locations, all values greater than the pivot are in higher index locations with the pivot value in between. This gives two partitions that are already in the correct region of the array and never need to be compared with each other.
3. Recursively quicksort the two partitions.
4. Job done.

The quicksort algorithm has already been coded for you for this practical.

TIMING QUICKSORT

1. In the same way as above, copy the ***folder*** **Quicksort** from **Groups on Wide (V:)\CSCU9A2\Java** to your CSCU9A2 working folder.

   Launch BlueJ. Using the **Project** menu, **Open Non BlueJ...** create a BlueJ project in your **Quicksort** folder – remember, when you navigate to it click once on the folder name then Open in BlueJ – do not double click to open the folder.

   Look at the classes: The structure is identical to the selection sort timer example above. This time the main program is **QuickSortTimer**. **QuickSortTimer** is slightly different from **SelectionSortTimer**: *within one run,* it repeatedly prompts the user for the size of an array, constructs a new array of that size containing random integers, measures the time for quicksort to process the array and then reports the time to the user. It is easier to carry out repeated timings. The repetition is coded as a "while (true)" loop – automatically an infinite loop, but containing an if statement that "break;"s the loop if a negative array size is requested. There is a short commented-out section that builds the array to be sorted in a different way, for use later.

   Compile and run the program — it should be fine – and carry out the experiments below.

3. Now for some timings: Again the best array sizes here depend on how fast your computer actually is, so experimentation may be needed. Try array sizes 1000, 10000, 100000, 1000000 (and maybe some values in between) until you get a time that is perhaps around 100 or so milliseconds (the exact time does not matter) – on my PC it is about 1000000. You should already get the feeling that this is very different from selection sort! In the instructions below, this is referred to as the "basic size".

4. Now try a range of array sizes building on that basic size, such as 2x the basic size, 3x, 5x, 10x, 20x, 30x, 50x, 100x, etc. For each size, repeat the measurement, say, five times, ***write down the time taken for each, and then calculate the average time for each array size***.

5. Sketch a graph showing your results. Remember that quicksort is an O(n log n) algorithm – the graph should just curve gently upwards. Add to the graph a sketch of what an O(n^2) algorithm would look like: take the timing for the basic size, multiply it by 4 to get what the timing for 2x would be, by 16 to get what the timing for 4x would be and so on. Compare the two lines on the graph.

   Try to estimate how long selection sort would take to sort the *largest* array that you tried with quicksort. Compare the two timings.

6. Now open the **QuickSortTimer** class in BlueJ's editor, comment out the line that creates an array of random values, and uncomment the five lines below it that create an already sorted array of values.

   Try doing your sorting timings again. For large arrays it should fail disastrously with an *exception* – if you scroll to the top of the exception report, you will see that it is a "stack overflow": there have been too many recursive method calls inside each other for the JVM to keep track of and it has run out of memory space and has given up! There is no easy way to fix this in BlueJ, so open a *command window* at the folder with your project (review the earlier lab worksheet if you cannot remember how), and run your program like this:

   ```
   java   –Xss100M   QuickSortTimer
   ```

   The –Xss100M tells the JVM to allocate 100Mbytes of memory for its "thread stack", and you should now be able to repeat some of your timing measurements without getting exceptions.

   Record some timings and sketch the graph – it is bad: quicksort has now become an O(n^2) algorithm! It only behaves well if the starting array is randomly ordered, otherwise all values fall into just one partition, and it behaves similarly to selection sort! For best behaviour quicksort is *supposed* to use the *median value* for partitioning the array, and *not* the first value (then it always performs well) but finding the median value is hard and time consuming!

That's all! Reflect on what you have seen about the performance of these two sorting algorithms.

SBJ 15 March 2017