Lab 9 — Sorting Algorithms

Data Structures and Algorithms – Spring 2024

---

1. Download **lab09_files.zip** from the class web page and unzip it on your Desktop, then open **Sort.java** in DrJava. Go to *Settings* → *Interactions Pane* and set *Interactions Working Directory* to your lab09_files folder. Click OK, then click the *Reset* button at the top of the DrJava window.

2. The file **Sort.java** contains definitions for several different sorting algorithms. Compile the file, then type the following command at the DrJava prompt:

   ```
   > java Sort bubble
   ```

   This creates an array of 30 random integers and runs the **bubbleSort** method on it. To specify a different array size, simply include the desired size as an extra parameter. For example: `java Sort bubble 100`

   To test different sorting algorithms, you can use the following keywords:
   bubble, selection, insertion, shell, merge, quick, count1, count2

3. Test **selectionSort**, **insertionSort**, and **shellSort** on random arrays of various sizes using the Sort program.

4. Now let's examine the behavior of bubbleSort as the array sizes get larger. To do this, we can run the program SortTimer, specifying on the command line the sizes of the arrays to test. The command below will run bubbleSort on random arrays of integers of sizes from 10000 to 30000 elements, in increments of 1000, and will display the number of milliseconds taken in each case:

   ```
   > java SortTimer bubble 10000 30000 data.txt
   ```

   It will also record this timing data in a text file called "data.txt", which can be plotted using the Gnuplot program, with the commands shown below (note: the `gnuplot>` prompt is not part of the command):

   ```
   gnuplot> set style data lines
   gnuplot> plot "data.txt"
   ```

   Based on the shape of the curve, do you think the time complexity of bubbleSort is linear, quadratic, cubic, or something else? Extending the array size up to 50000 elements may give a clearer picture of the shape.

5. Next, generate some timing data for the other sorting algorithms and view it in Gnuplot. You can specify a different array increment size (the default is 1000) by including it as an extra command-line parameter after the starting and ending sizes. For example, to compare all four algorithms on array sizes from 10000 to 40000 in increments of 2500, you could do this:

   ```
   > java SortTimer bubble 10000 40000 2500 bubble.txt
   > java SortTimer selection 10000 40000 2500 selection.txt
   > java SortTimer insertion 10000 40000 2500 insertion.txt
   > java SortTimer shell 10000 40000 2500 shell.txt
   ```

   and then plot all of the results on the same graph with the Gnuplot command:

   ```
   gnuplot> plot [][-100:] "bubble.txt","selection.txt","insertion.txt","shell.txt"
   ```

   where `[][-100:]` tells Gnuplot to label the y-axis starting from –100 instead of 0.

6. Next, test **mergeSort** on some random arrays of various sizes to make sure it works correctly. Then run some timing tests to compare its rate of growth to the other sorting algorithms.

7. Finish the implementation of **quickSort** by completing the definitions of the **choosePivot** and **partition** methods. The choosePivot method can just return the element at position *first* in the array (but you could also experiment with other choices, for comparison). An outline of the partition algorithm is given below:

```
partition(data, first, last, pivot):
   initialize left index i to array position first
   initialize right index j to array position last
   loop:
      move left index i rightward until we encounter an element ≥ pivot
      move right index j leftward until we encounter an element ≤ pivot
      if indices have met or crossed, quit the loop
      otherwise swap elements at positions i and j
      move i to right and j to left one position, and continue loop
   return j as the midpoint position of the partition
```

This method rearranges the elements in a subregion of the *data* array from positions *first* to *last*, around a *pivot* value, such that only elements less than or equal to *pivot* end up on the left side of the subregion, and only elements greater than or equal to *pivot* end up on the right side of the subregion. When the loop finishes, all elements at positions *first* to *j* will be $\leq$ *pivot*, and all elements at positions *j*+1 to *last* will be $\geq$ *pivot*. (The elements, however, will not necessarily be in sorted order.) Test your quickSort implementation by running the commands:

```
> java Sort quick
> java Sort quick 200
```

8. If we know nothing at all about the elements we are sorting other than how to compare them, the fastest general sorting algorithms run in O(n log n) time. But if we know more about our elements, we may be able to take advantage of that information to improve the time complexity. For example, if we know that the elements to be sorted are guaranteed to be integers in the range 0 to 100, we can sort them in O(n) time by making use of an auxiliary array of 101 counters (one counter for each possible integer from 0 to 100).

   Based on this idea, implement an O(n) sorting algorithm for arrays of integers with a minimum value of 0 and a maximum value of 100, by completing the definition of the **countSort1** method in Sort.java. To test countSort1, uncomment the line in the method generateTestCase at the top of Sort.java that returns random arrays of values from 0 to 100. Then test countSort1 using the commands:

```
> java Sort count1
> java Sort count1 200
```

9. Next, plot timing data for countSort1 for array sizes from 1 million to 25 million elements in steps of 500,000. You may run other tests if you like. Do these results indicate a time complexity of O(n)? What is the big-O *space complexity* of countSort1? That is, how fast does the amount of storage required by countSort1 increase as a function of the number of elements to sort? Note: depending on how much memory your computer has, you may need to increase the heap size setting in DrJava for the larger tests (located under *Settings* → *Miscellaneous* → *JVMs*).

10. Rather than always assuming elements in the range 0 to 100, generalize your algorithm so that it determines the maximum element in the array and uses this information to sort the array, by completing the **countSort2** method in Sort.java. You should still assume that the smallest possible element is 0. To test countSort2, change generateTestCase to return arrays of integers from 0 to the size of the array, and run the commands:

```
> java Sort count2
> java Sort count2 200
```

Is the time complexity of countSort2 still O(n)? What is its big-O space complexity? Generate some timing data for countSort2 and plot the running times of countSort1 and countSort2 on the same graph. How do the curves compare?