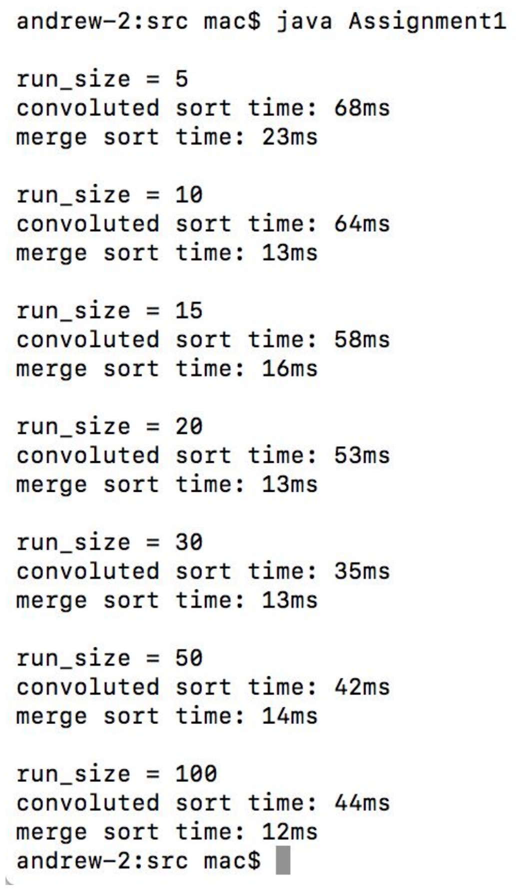
Andrew Pilon

CS 245
Assignment 1

I chose to implement a version of insertion sort on the unsorted segments in between the runs discovered within the randomly generated array. My main reasonings for choosing insertion sort was that it was one of the fastest of the in-place sorting algorithms we did in the practice assignments. I also chose this over other algorithms (quick sort, for example) because of its simplicity - insertion sort can be written with one while loop inside one for loop, and was very easy to ass into the code. This could also just be because I'm still not as familiar as I'd like to be with java structures, and I figured implementing a multi-function sorting algorithm would require me to call another class and perhaps that would slow it down more (although this is in part an assumption based on my non-knowledge). I also figured that because this sort would be restricted to data sizes of run_size or less, it would never have to work that hard, and possible time differences wouldn't be too significant.

To the right, I've added a picture of the run time comparisons between my convoluted sort and merge sort (I used the MergeSort.java file submitted in practice-assignment-05). I ran comparisons for various run sizes to see how that affected the difference and overall time of each sort as well. The array to sort contained 500,000 integers (it was easier to see time difference in ms than when I had it sorting 50,000). Merge sort was about the same each time, as it should be, since the run_size parameter only affects the convoluted sort (slight differences due to me generating a new random array for each run_size comparison). Convoluted sort got faster as the run_size increased *up to a certain point* (around run_size = 30-50), which is interesting as it means it was faster when it found less runs, and therefore while insertion sorting on bigger unsorted segments. I have no idea why it started getting slower again - something to do with the trade off between longer insertion sorts, but having to do less of them overall? I'm pretty sure that my merging of the array of runs could be done more efficiently; I wasn't sure how to get around the cases of an odd number of segments for merging, as well as an even number that becomes an odd



```
andrew-2:src mac$ java Assignment1

run_size = 5
convoluted sort time: 68ms
merge sort time: 23ms

run_size = 10
convoluted sort time: 64ms
merge sort time: 13ms

run_size = 15
convoluted sort time: 58ms
merge sort time: 16ms

run_size = 20
convoluted sort time: 53ms
merge sort time: 13ms

run_size = 30
convoluted sort time: 35ms
merge sort time: 13ms

run_size = 50
convoluted sort time: 42ms
merge sort time: 14ms

run_size = 100
convoluted sort time: 44ms
merge sort time: 12ms
andrew-2:src mac$
```

number later on (i.e. 6→3→? as opposed to 8→4→2→1), so I just explicitly checked for that with an if statement.

Regardless, merge sort was definitely faster than my convoluted method. I tried to minimize the amount of extra space by not separating and storing the runs/nonruns in another data structure, but instead storing the indices between each segment in an array to reference for merging later. The algorithm does one initial sweep of the random data array, in which it discovers runs and also sorts all random segments in between runs that have length<run_size, which ensures that by the end of the sweep, the data array is composed entirely of runs (of varying sizes). Then I wrote the merge method to use temp arrays for each merge, but just keep updating the original array as it goes. I could probably have generated a large temp array at the start and keep writing information into that as I merged, but again I wasn't entirely sure how to make that more generalized and be able to have it work while calling ConvolutedSort in the Assignment1 main function. I am now realizing that the insertion sort method for the nonrun segments also creates a temp array each time it's called, so actually I didn't do a super great job of avoiding unnecessary data structures (oops). This probably explains some of the time difference, as I wrote MergeSort to only generate one master temp array for all merges in order to save space. Other slowdowns could occur from insertion sort if the run_size is large enough for it to be impactful - this could probably be alleviated by implementing quick sort or some other method instead. Also, reducing the amount of arrays necessary by creating big ones at the start to be reused could help speed it up.