NovelSort and Choosing the Best Algorithm(s)
Lab Assignment #6

by
Justin Anthony Timberlake

CS 303 Algorithms and Data Structures
October 4, 2017

1. Problem Specification

The purpose of this lab was to first implement novelSort, which was only described in the in-class portion of the lab. This was to be tested on some input we generated. For the homework portion, we were to look at a given problem and use our knowledge and implementation of algorithms so far, taking into account elements such as time and space complexity to make the best choice of how to approach this problem. In my case, I sorted the records, first by time in 3 steps of CountingSort, which was all part of a modified RadixSort. After that, I used the hybrid insertMergeSort from a previous lab to sort the elements once more by their locations (as Strings).

2. Program Design

The steps taken to develop the program were:

a) Develop the novelSort algorithm in class and test that it worked.

b) Copy over some of the text read-in related code in the main method. Copy over basic reusable methods such as printArray and readInNums, which were altered to suit the data structures used in this specific assignment, as well as some things, which were deleted from the final code.

c) Copy over already implemented insertMergeSort, merge, insertionSort, and an implementation of countingSort, which I had implemented at another time from the algorithm in the book directly. These methods were then all changed to work with a new object type, Record, which is also defined as another class within the Java file.

d) Implement all constructors and accessor methods for the Record class and develop the code for the test input from the sample file, reading in the records and creating a Record object for every line with an uncertain final number of records being read into an ArrayList. For a number of reasons, however, this ArrayList would then be copied to an array for the actual calculations and sorting of the individual elements.

The following methods were used in the primary class:

a) readInNums (Scanner derp) : receives a Scanner object, locally referred to as “derp” and scans each line of the file, creating one Record object for each as it goes.

b) printArray(Record[] records) : prints the attributes of each Record in a given array according to the format presented in the example. In some cases, it will append additional leading zeroes t o match the format more precisely.

c) RadixSort(Record[] a) : This instance of radixSort contains 3 different passes through the otherwise iterative countingSort algorithm. However, due to the nature of the data being read in and each column having different attributes, the individual calls to countingSort were instead spelled out within the contents of this overarching RadixSort, which would usually just call countingSort sepearately. This is because of my use of the Record object, which has attributes that store seconds, minutes, and hours of the time listed all separately to be sorted accordingly. In the case of seconds and minutes, we know that the max value can only be at most 59 with a minimum value of 0, but the hours can only represent from 0 to 23. These differences and the fact that each attribute had to change (for instance, the calls to "getHours()" as opposed to "getMinutes()" for the comparison calculations), further disallowing RadixSort to hold its traditional implementation of calling an outside stable sorting method, such as, in this case, countingSort.

d) insertMergeSort(Record[] A, Record[] temp, int p, int r, int cutoff) : This works as it did in an earlier lab, yet it compares based on the attributes within the Record objects within the arrays it handles.

e) merge (Record[] A, Record[] temp, int p, int q, int r) : This also works as it did before, but takes Record arrays.

f) insertionSort(Record[] list, int p, int r) : This also works as it did before, but takes Record arrays.

The following methods were used in secondary class, Record.

a) public Record() - base constructor for objects of type Record.

b) public Record(String place, int hrs, int min, int sec) – Constructs an object of the type Record, given initializing values.

c) getCity() : returns the city attribute of Record.

d) getHours() : returns the hours attribute of Record.

e) getMinutes() : returns the minutes attribute of Record.

f) getSeconds() : returns the seconds attribute of Record.

3. Testing Plan

The testing plan for this was less about physically testing run times and more about testing to see which algorithm would best suit the given problem based on space and time complexity constraints and benefits. For example, I knew immediately that in order to sort the records by their time in the given format, I would first need to parse and sort using a form of RadixSort, which would be best implemented with countingSort. RadixSort uses a stable sort as its base so that in the future when comparing two of the same element, they remain in the same relative order to each other, maintaining the sorted nature of the columns which have been sorted already. With the dividers of seconds, minutes, and hours and knowing the constraints for each subset and that I was sorting integers, the conditions were best for the use of countingSort as opposed to the other stable sorts. Specifically, from the algorithms we have covered so far, InsertionSort was a bad choice from the beginning simply because of its average case run time of n^2, which we have already witnessed expanding at an exponentially faster rate than any other sort implemented. MergeSort was also not the best option due to its additional space complexity requirement and running in all cases O(nlog(n)) time, which admittedly is as fast as comparison sorts could achieve, but we could do better. This left bucketSort and countingSort with which we could implement RadixSort, both of which would run in linear time, and naturally the better choice was countingSort, as I was dealing with integers in this case, rather than fractional values between 0 and 1, such as floats.

The fourth call to sort the array of records needed a different algorithm as I needed to sort a different kind of data type within it, Strings. In the case of sorting strings with RadixSort, it could very possibly be done, however, in this case, it would not help because I could not assume some things which were given in the case of sorting the elements within the time displays. For one, there is no constraint given on the length of each String representing a city, so I would have to add an entire extra n time operation to scan the list and find the longest String in each set of records being sorted, which would defeat the purpose of trying to utilize a linear sorting algorithm. And unless there were such a constraint on the Strings allowed, which limited the Strings to length of 3 or less, this would also call for even more calls required to iteratively sort by each letter in the word and ultimately take much longer than the time did. BucketSort still does not apply either, because we are not dealing with a preferably applicable data type.. This meant that the next best thing was sorting in nlog(n) time. However, there was a catch. In order to maintain the sorting of the time elements I had already calculated, I needed a stable sort specifically. This meant that heapSort and quickSort were not good candidates. So, finally, I was faced with insertionSort vs. mergeSort. While I already knew that mergeSort could achieve the tine complexity I was aiming for, it is further optimized by instead calling insertionSort for only small sizes of sub-lists, so based on the experiments done in previous labs (referenced below), calling insertMergeSort with a cutoff value of k = 4 was the best bet to further optimize run time.

One last issue to consider was space complexity and the chosen base data structures used to implement this whole program. First, I scanned in the values with an ArrayList simply due to the unknown number of Records I would be reading in. However, then the ArrayList is immediately copied into an array. While at first this was just my lazy solution to using algorithms which had only been implemented using arrays, I actually did try to implement them with ArrayLists of the type Record. Again, this is possible, but it would necessarily affect the run time. When calling a specific instance of countingSort (one of the thirds of my implementation of RadixSort), a secondary array (or arraylist) has to be generated to project the new sorted values onto after they have been appropriately "counted" with the C array. With an array implementation of the algorithm, this can be done with all default values of an array of a certain size set to "null". However, the only thing that can be done in constant time with an ArrayList is declare its "capacity" and individual values still have to be added in n time. Thus, every call of RadixSort would add an additional 3n space and time complexity for one call when this problem was avoided by instead copying to an array, adding n time and space complexity only once as opposed to having to add blank entries to the secondary array each time it is declared, necessarily changing the algorithm and making it less efficient.

4. Test Cases

Insertion/Merge/Heap/QuickSort Run Times and Variations											
Input Size (number of values)	Time (milliseconds)										
	Insertion Sort	insertMergeSort, cutoff at values K				Merge Sort	Heap Sort (max heap)	QuickSort			
		K = 4	K = 8	K = 16	K = 32			Standard	Median of 3	Best or Worst Cases (1,024 values)	
100	0	0	0	0	0	1	0	0	0	**Already Sorted**	8
1,000	4	0	0	0	0	2	0	0	0		
5,000	33	0	0	0	0	2	2	0	0	**Reverse Sorted**	2
10,000	24	0	0	0	0	1	3	1	1		
50,000	371	0	0	0	0	10	12	6	5		
100,000	1,412	0	15	16	15	15	15	10	10	**Random**	0
500,000	36,819	47	47	62	63	71	88	52	53		

5. Analysis and Conclusions

In this case, based on sorting algorithms we have covered so far, the best method I could find to save on time and space complexities was to different algorithms when best applicable in each case. The final algorithm from start to finish was as follows:

1. Read in the lines provided using an ArrayList of Record objects. **[O(n)]**

2. Copy the contents of this ArrayList into an array. **[O(n)]**

3. Call RadixSort with CountingSort on the time in 3 parts of 60, 60, and 24 digits in that order.

[O(n*60+n*60+n*24)]

4. Sort the list again with respect to the Strings representing the locations associated with each record and base comparisons lexicographically instead of strictly numerically, using mergeSort for all cases greater than small sub-lists of size 4 or less, in which case, sort those using insertionSort.

[O(nlog(n))]

5. Print the values in their newly sorted order with respect to different locations and their associated times in order as well.

[O(n)]

6. References

All code in this assignment was either presented in prior labs and assignments or provided through means of pseudocode within the in-class portion of the lab. RadixSort was based on a modified version of the code for countingSort, called 3 times, as shown with the algorithm in the book