Jacob Imlay
Joel Ross
CS 261

Homework 9

Section I -- Data

Bubble Sort Comparisons/Swaps* vs. List Size

List Size	Least Comparisons/Swaps	Average Comparisons/Swaps	Most Comparisons/Swaps
2	1 / 0	1.5 / .5	2 / 1
3	2 / 0	4.33 / 1.5	6 / 3
4	3 / 0	8.625 / 3.0	12 / 6
5	4 / 0	14.433 / 5.0	20 / 10
6	5 / 0	21.813 / 7.5	30 / 15
7	6 / 0	30.801 / 10.5	42 / 21
8	7 / 0	41.429 / 14.0	56 / 28
9	8 / 0	53.719 / 18.0	72 / 36
10	9 / 0	67.691 / 22.5	90 / 45
11	10 / 0	83.361 / 27.5	110 / 55
12	11 / 0	100.745**/ 33	132 / 66

Selection Sort Comparisons/Swaps* vs. List Size

List Size	Least Comparisons/Swaps	Average Comparisons/Swaps	Most Comparisons/Swaps
2	1 / 0	1.0 / 0.5	1 / 1
3	3 / 0	3.0 / 1.166	3 / 2
4	6 / 0	6.0 / 1.917	6 / 3
5	10 / 0	10.0 / 2.717	10 / 4
6	15 / 0	15.0 / 3.55	15 / 5
7	21 / 0	21.0 / 4.407	21 / 6
8	28 / 0	28.0 / 5.282	28 / 7
9	36 / 0	36.0 / 6.171	36 / 8
10	45 / 0	45.0 / 7.071	45 / 9
11	55 / 0	55^^ / 7.980	55 / 10
12	66 / 0	66^^ / 8.968	66 / 11

Insertion Sort Comparisons/Swaps* vs. List Size

List Size	Least Comparisons/Swaps	Average Comparisons/Swaps	Most Comparisons/Swaps
2	1 / 0	1.0^	1 / 2
3	2 / 0	2.666^	3 / 5
4	3 / 0	4.917^	6 / 9
5	4 / 0	7.716^	10 / 14

6	5 / 0	11.05^	15 / 20
7	6 / 0	14.907^	21 / 27
8	7 / 0	19.282^	28 / 35
9	8 / 0	24.171^	36 / 44
10	9 / 0	29.571^	45 / 54
11	10 / 0	35.480^	55 / 65
12	11 / 0	41.897^	66 / 77

*Minimum and maximum number of Comparisons and Swaps do not necessarily correspond to each other. Average numbers of Comparisons and Swaps should correspond by definition as averages. This applies for all the tables that follow

**At List Size = 12, the size of values gathered during the sum calculation for bubble sorting resulted in inaccuracies when determining the average. This value was computed by hand using AnalyzeableHashMap values.

^For insertion sort, the averages of data gathered were always equal.

^^For permutations of length 11 and 12, the Java machine was unable to accurately determine the average number of comparisons due to size limits with the amount of data handled (not a heap space error – just very large numbers). However, from the definition of a selection sort, there are at least and at most n(n-1)/2 comparisons made during a selection sort, and so the average number of comparisons for this sort must also be n(n-1)/2.

Here is a Sample of data gathered using the AnalyzeableHashMap when performing bubble sorts on permutations of length 12. Key refers to the number of comparisons, while count refers to the number of times that many comparisons were made when sorting an array. This kind of table can be made using the printTable method of that class.

```
key: 11, count: 1
key: 22, count: 2047
key: 33, count: 116050
key: 44, count: 1454766
key: 55, count: 7802136
key: 66, count: 24217320
key: 77, count: 51114960
key: 88, count: 80443440
key: 99, count: 99388800
key: 110, count: 98340480
key: 121, count: 76204800
key: 132, count: 39916800
```

Merge Sort Comparisons/Swaps* vs. List Size for 10000 permutations

List Size	Least Comparisons/Swaps	Average Comparisons/Swaps	Most Comparisons/Swaps
10	16 / 8	22.662 / 25.015	25 / 34
100	516 / 530	541.797 / 573.008	561 / 613
1000	8637 / 8831	8707.3498 / 8976.600	8770 / 9116
10000	120236 / 123098	120450.4631 / 123617.7457	120664 / 124065
100000	1535691 / 1567512	1536366.241 / 1568933.287	1536992 / 1570260

Due to limitations on Java Heap Memory Space, with my AnalyzeableHashMap data structure, it was not possible to perform tests on lists of size 1,000,000 or greater. This problem could be handled by redesigning the AnalyzeableHashMap data structure to use a hashing function, rather than bin sorting to avoid wasting space. EDIT – I switched my code to use the AnalyzeableArrayList I designed. This data structure is able to handle the larger number of comparisons, although the structure has limits on the number of permutations it can handle. With this change, an array of length 1000000 (…Test(n, 7)) could be tested, although this proved to take too much time (over 30 minutes), and so I ended the test before completion.

Heap Sort* Comparisons/Swaps* vs. List Size for 10000 permutations**

List Size	Least Comparisons/Swaps	Average Comparisons/Swaps	Most Comparisons/Swaps
10	13 / 20	20.827 / 28.318	30 / 39
100	491 / 574	539.445 / 623.394	595 / 680
1000	8582 / 9427	8744.361 / 9592.678	8921 / 9794
10000	120462 / 123098	121058.230 / 129551.552	121683 / 130161
100000	1542041 / 1626954	1536366.241 / 1628833.204	1546455 / 1631419

***Although these numbers are fairly similar if not greater than those of the merge sort, the average time to complete these tests was much less than that of the merge sorts. I will not report that data here, but it can be found when running the fast sort test method.

Quick Sort Comparisons/Swaps* vs. List Size for 10000 permutations

List Size	Least Comparisons/Swaps	Average Comparisons/Swaps	Most Comparisons/Swaps
10	13 / 20	20.848 / 28.337	31 / 38
100	488 / 573	539.503 / 623.442	594 / 681
1000	8569 / 9414	8744.205 / 9592.508	8928 / 9794
10000	120342 / 128852	121060.990 / 129554.278	121583 / 130087
100000	1541899 / 1626889	1543889.917 / 1628826.563	1545730 / 1630815

Bubble Sort vs. Bin Sort Run Time on Permutations of Different List Sizes

List Size	Bubble Sort	Bin Sort
6	5ms	3ms
7	17ms	2ms
8	22ms	18ms
9	190ms	124ms
10	2395ms	1338ms
11	30663ms	16205ms
12	434016ms	210121ms

Due to the speed at which the machine can perform sorts on the permutations for lists of size 2-5, this data is not included. In particular, for these list sizes, the machine could perform the tests in either 0 or 1ms, which is not significant enough data to display the differences in runtime speed for the two sorts.

Merge Sort vs. Bin Sort Run Time© for Various List Sizes with 10000 permutations

List Size	Merge Sort Runtime	Bin Sort Runtime
10	152ms	11ms
100	160ms	35ms
1000	1779ms	365ms
10000	22468ms	4521ms
100000	279313ms	54811ms

© Ran out of symbols…anyway, this was the merge sort that kept track of the comparisons and swaps, and so the merge sort runtime data might be larger than the actual runtime.

Section II – Analysis of Quadratic Sorts

The different quadratic sorts produced a variety of different values for the largest, smallest, and average number of comparisons. The bubble and insertion sort algorithms were tied for the lowest number of comparisons in any permutation at n-1 comparisons. This is presumably when performed on a fully sorted array, as otherwise more comparisons would need to be made. The selection sort, on the other hand, at the greatest minimum number of comparisons at n (n-1) / 2 comparisons in each case. This is a result of the selection sort algorithm's definition: it must make this many comparisons. When looking at the most comparisons, we find that the selection and insertion sort algorithms both produced n (n-1) / 2 comparisons, which was to be expected from the definition of the algorithm. The bubble sort, on the other hand, needed more comparisons in the worst case. The bubble sort had an expected n(n-1) / 2 greatest number of swaps in the worst case, which follows expectations. The insertion sort had a large number of swaps than the bubble sort in the worst case, whereas the selection sort needed at most n – 1 swaps. This is because the selection sort only swaps if the swap will result in an item reaching its final location, enabling the sort to have the least number of swaps. When looking at the average case, the bubble sort had the largest number of comparisons on average, while insertion had the lowest on average. However, the insertion sort resulted in the largest number of average swaps, with selection having the least number of swaps.

From these results, we can infer several things. We can infer that the selection sort provides the best performance of the quadratic sorting algorithms in terms of number of swaps. At the same time, the selection sort has the most comparisons in the minimum case, and so it has the drawback of needing a relatively large number of comparisons regardless of the data in the array. We can also infer that the insertion sort has the best case average number of comparisons, and so it exceeds the other sorts in this area. Overall, from this data I would recommend the selection sort as the best sorting algorithm among the quadratic sorts, as it provides the best performance of swaps and is the second most efficient algorithm in the average case for comparisons. I would also put forth the bubble sort as the least efficient among these sorts, as it holds the worst average case performance for comparisons, and also has the largest worst case comparison number.

Section III – Analysis of Fast Sorts

I would begin this analysis by stating that each fast sort had very similar average case performances. Yes, these averages differed, but they were within very similar ranges of values. I will admit this may be a fault of my data. With the data structures I made for the purposes of analysis, I was limited in the number of permutations I could make. For these tests, as stated in my Data section, I used 10,000 permutations on lists with sizes at different powers of ten. With many more permutations available at larger list lengths, the accuracy of my average comparison and swap numbers by definition becomes less representative of the genuine average case for larger list sizes.

With all this said, I will discuss the data I do have, including the runtime data that is not formally reported in the Data section. In terms of the minimum number of comparisons, the merge sort had the lowest number of comparisons for all test sizes. The merge sort also had the lowest number of comparisons in the worst case among those tested as well. However, the merge sort had the largest of the average number of comparisons. As for the other sorts, the heap and quick sorts had relatively similar performance, differing by no more than 2000 in the number of comparisons in any minimum or maximum number of comparisons. Both of these sorts had better average case comparisons than the merge sort. Interestingly, these same statements hold true for the number of swaps as well, with merge providing the best performance in the best and worst swapping cases, but having the worst average case performance. An addendum to this is the fact the heap sort was much faster than the merge sort in terms of run time during tested (not reported).

In comparison to the quadratic sorts, the only relevant data would be the tests performed on lists of length 10 (the only data that is matched in both test sets). In these cases, the quadratic tests had lower maximum and average numbers of comparisons across all three sorts. However, the minimum number of comparisons was higher for the fast sorts. This may be the result of not testing all permutations for lists of length 10.

In conclusion, I would infer that the merge sort is able to perform best in extreme cases where a list is highly unsorted or is relatively sorted. However, the heap and quick sorts are able to perform better than the merge sort on average. I will state that further testing may reveal inaccuracies with these statements; however with the data gathered these are the conclusions I may reach.

Section IV – Analysis and Explanation of Other Sort

For my new sorting algorithm, I decided to use a bin sorting algorithm, which I learned in my algorithms course in Budapest, Hungary. A bin sort takes a list of unique natural numbers, and works as such:

```
1. Find the largest number R in the list.
2. Construct a new array SORTED of size R.
3. For each natural number N:
4.     Put N in SORTED at index N.
5. Return SORTED.
```

By taking advantage of array indexing, a bin sort is able to sort an array in O(R) time, where R is the largest number in the unsorted array. This can either be good or bad: if R is close to or less than the length of the array, then the bin sort will run in O(n) time. However, if R is n^2, then the sort will run in O(n^2) time: if R is the product of a function f(x) on n, then the runtime will be O(f(n)). Thus, the bin sort can be an exceptionally fast sort, but only on a very distinct input. Otherwise, the sort will either be ineffective or will have a runtime potentially worse than quadratic.

In the case of this test, we sorted an array that has the natural numbers 1 through n, where n was the length of the unsorted array. As we sorted unique numbers in this way, the bin sorting algorithm was a fit for being compared to the other sorts in terms of machine run time. As shown in our data, the bin sort proved to be exceptionally fast compared to both the bubble sort and the merge sort. In tests on lists of length 10000, the bin sort was able to complete sorting 10000 permutations in nearly ¼ the time of the merge sort. I mentioned this in the data section, but my data may be skewed by how the comparison and swap data were tallied during the bubble and merge sorts whereas this data was not handled. Future testing, wherein the comparison and swap data is not tallied, may reveal the merge and bubble sort are closer to the runtime of the bin sort for the given arrays.

Section V - Notes

In performing my analysis, I wanted to have the computer perform as much of the needed analysis (computing max, min, and average performance), as it could. Therefore, I created these data structures to facilitate my tests gathering and working with the data from the sorting algorithms.

I was surprised to discover the need for specific data structures for analyzing the data. When I initially attempted to run my tests, I only reached permutations of size 4 or 5 before encountering the Java Heap Space error when I was using my AnalyzeableArrayList. I then needed to create the AnalyzeableHashMap structure, which, although it is not an ideal HashMap, is still representative of a hashing function.

In designing my analyzing data structures, I learned a new use of HashMaps, and I also learned more about the limitations of arrayLists. The array list can handle many comparisons, but only a certain number of permutations. On the other hand, the hash map as I designed it cannot handle an excessive number of comparisons (on tests of arrays longer than length 100,000). These were lessons I did not expect, but it feels rewarding to have discovered these during my completion of the assignment.