Homework 8 Sorting Algorithm Analysis

Investigation I

When investigating the performance of quicksort algorithm with respect to the change of MIN_SIZE (the threshold of the size of an array to switch to insertion sort), my hypothesis was that the sorting time would be reduced at most with a certain value or a range of values for MIN_SIZE, so that the graph of sorting time / MIN_SIZE would be a parabola. Therefore I started looking for the best MIN_SIZE value by adding code in main() function. Since the fillAndShuffle() could be really random, I wrote loops such that the computer would generate 10 arrays and sort 10 times for each MIN_SIZE. I then wrote code to calculate the average of the sorting time for each MIN_SIZE and performed the hypothesis testing on WolframAlpha for data in the same MIN_SIZE value. It turned out that the data has very little fluctuation when the array size was small (1000000) and was very reliable. This means that the exact ordering of the array doesn't really matter given the size of the array was really big.

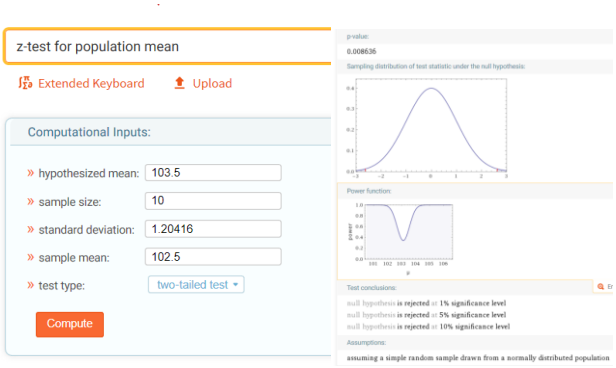
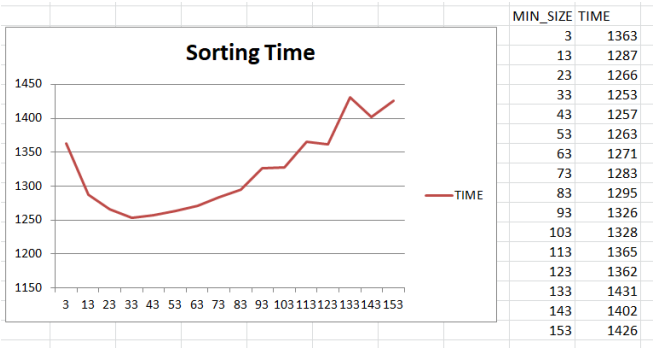


Figure 1. Hypothesis testing with sorting time{100, 104, 102, 102,104, 104, 103, 102, 102, 102} when MIN_SIZE = 45 and array length = 1000000

I set a range of MIN_SIZE in a for loop in main(), lessen the trials for each MIN_SIZE and hoped to spot the best time for quick sort to switch to insertion sort. I set MIN_SIZE as 3-19 with gap = 1 at first, but found out that the sorting time decreases when MIN_SIZE increases. I then set MIN_SIZE as 3-153 with gap = 10 and found that there seems to be a turning point between MIN_SIZE = 20-75. Therefore I tighten the boundary to that range and set the gap to 5. I run the program for 2 times, once increase the number of trials to 100 times and the other increase the array size to 10000000 (10 times). Each took some time to run.



MIN_SIZE	TIME
3	1363
13	1287
23	1266
33	1253
43	1257
53	1263
63	1271
73	1283
83	1295
93	1326
103	1328
113	1365
123	1362
133	1431
143	1402
153	1426

Figure 2. Roughly finding the optimal MIN_SIZE. MIN_SIZE on x-axis.

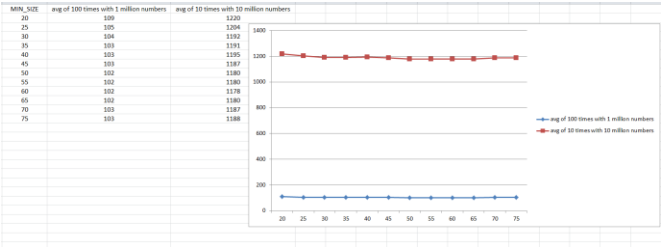


Figure 3. Finding the optimal MIN_SIZE in a tighter range and exploring the performance of algorithm with different array length.

The conclusion that surprised me was that the performances of this algorithm in this tighter range of MIN_SIZE were medial.

And then I checked the situation where there are lots of repeated numbers. I modified the code to a[k] = k/100; so that it would produce each for 100 times. Setting the array length as 1000000, I found out that the optimal MIN_SIZE is hardly affected by repeated numbers. Lastly, I made the array to ascending and descending order by commenting the “shuffle” and modifying the code to a[k] = a.length – k and the range of optimal MIN_SIZE did not change.

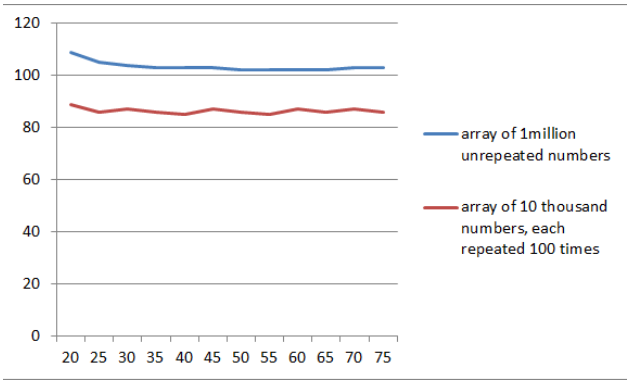


Figure 4. Investigation on repeated numbers. The algorithm spent less time in sorting the array of repeated numbers.

Next I explored the sorting time with respect to different lengths of the array in different MIN_SIZE slots and the results indicated that MIN_SIZE is (almost) not related to the original size of the array despite that some points are considered as outliers.

MIN_SIZE										
20	0	57	110	167	224	286	355	434	484	571
25	0	57	108	168	224	315	359	436	476	535
30	0	50	104	164	220	315	340	402	486	546
35	0	50	105	161	232	278	338	396	462	525
40	0	50	102	159	217	280	336	463	469	529
45	0	49	116	172	247	306	379	398	459	521
50	0	48	102	158	216	280	336	409	565	521
55	0	49	107	161	223	381	335	393	715	709
60	0	50	103	170	221	280	339	407	489	668
65	0	55	125	163	234	282	337	406	471	531
70	0	53	106	162	253	286	349	393	463	517
75	0	49	101	159	223	280	351	400	462	526

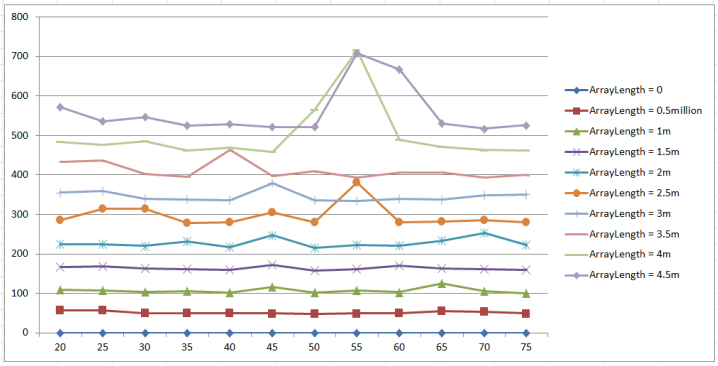


Figure 5. Sorting time with respect to different array length.

Investigation II

In this investigation I explored radix sort with its optimized version, the msdRadixSort.

The way that msdRadixSort works was to put the words into buckets by their initial letter and then uses standard radix sort to separately sort each of the individual. My hypothesis was that msdRadixSort would have a performance significantly better than radixSort because msdRadixSort used more space, divided the work of radix sorting (reduces n for time complexity) and saved the last step of sorting. Therefore I checked by testing lists of words in size of 100k, 200k and 300k and added code to print the runtime so that I could collect the data. I run each algorithm 5 times and averaged them for more reliable conclusion, and the results were the same as my hypothesis. I encountered an error message that there was insufficient memory for the Java Runtime Environment to continue, so I made my word list to size of 1000. In this size, radixSort performed much better than msdRadixSort. This reminds me of the fact that log(n) > n for some n < N0 when we were doing asymptotic analysis. Therefore I tried more sizes for word list and found out that the point where msdRadixSort started to perform better than radixSort was roughly (there were variations in running the code) with the size of the list at 10k to 20k.

word #	msdRadixSort	radixSort
1000	8	1
2000	9	2
3000	13	4
4000	12	6
5000	18	6
6000	24	8
10000	28	19
20000	43	52
30000	56	88
40000	63	123
50000	66	152
100000	117	243
200000	200	500
300000	313	679

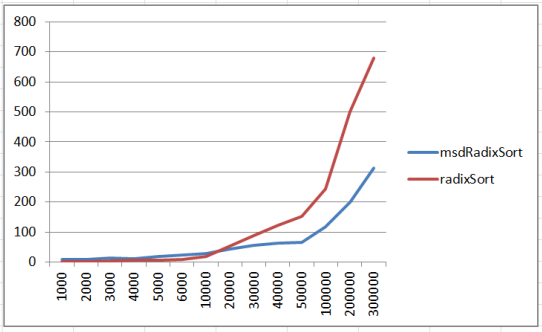


Figure 6. Performance of msdRadixSort and radixSort. The x-axis was the amount of words in the list and was not draw in-scale.

I also tried to investigate the change of sorting time with respect to the made up of the word list. I checked the dictionary and there were 17096 words with initial letter of “a”, so I set my load function to randomly choose the first 17096, 34192 and 68384 words to see the difference between msdRadixSort and radixSort.

word #	word range	msd	radixSort
100000	17096	194	70
200000	17096	373	226
100000	34192	179	143
200000	34192	154	266
100000	68384	147	148
200000	68384	188	363
100000	235886	145	230
200000	235886	208	549

Figure 7. Performance of msdRadixSort and radixSort to words with similar initial letters.

235886 is the whole range of words in the text file. MsdRadixSort was hardly affected when 1/3 of the list was covered (from “a” to “fascet”). Overall, the original radixSort algorithm performs better only when the words in the list has similar initial letters, that is to say, when the extra step in msdRadixSort failed to create enough buckets.

One possible improvement I can make about this investigation is that I could fix the “not enough space” error in my code and run each sorting method for multiple times. This would largely improve my efficiency in data collection (rather than executing the code for multiple times) and I believe it could to some extent reduce the uncertainty in computer performance.