605.202: Data Structures

Lab 4 Analysis

Renee Ti Chou

Due Date: May 2, 2017

Lab 4 Analysis

Issues of Efficiency – (theoretical)

Time complexity

Quicksort

Average case: The cost of partition is linear to the size of the piece. In the average case, the array would be divided into two smaller pieces, each of them would then be divided into other two pieces, and so on. Thus, the overall cost of quicksort is sum of (# piece) * (size of piece), which is 1*n + 2*n/2 + 4*n/4 + … + n*1. Since there are lg n elements in this equation, the time complexity is then **O(n*lg n)**.

Worst case: In the worst case, the partition process generates relatively uneven pieces, with one of the pieces containing only the pivot. The time cost then becomes n + (n-1) + (n-2) + … + 1, and thus the time complexity is **O(n^2)**.

Heapsort

Insertion: In the average case, where data are uniformly distributed, the probability of the child larger than its parent is 1/2, and that of the child larger than its grandparent is 1/4, and so on. Thus, the cost of insertion is 1/2 + 1/4 + … = 1.

Deletion: The average cost of deletion is similar to the cost of the worst case, which is lg n, the height of the binary heap.

Overall, the dominant term of the time complexity of the heapsort is lg n * (2^(lg n)) (which is height * number of leaves) = lg n * n. Thus, the time complexity of the heapsort is **O(n*lg n)**.

Space complexity

Quicksort

The quicksort is an in-place sort, but there is an auxiliary stack to assist the sorting process in the code. Since the statement in the code is "stack = **new int**[n]", the additional space required in this case is linear to input data size. For the insertion sort combined with the quicksort strategy, because it is implemented using an array structure, and no further dynamic allocation is required, the space complexity is linear to the size of the partitions it takes over. Therefore, the overall space complexity of the quicksort is n + n = **O(2n)**.

Heapsort

The heapsort is also an in-place sort. Since no auxiliary space is needed, the space complexity is **O(n)**.

Analysis of Sorts with Different Types of Input Data

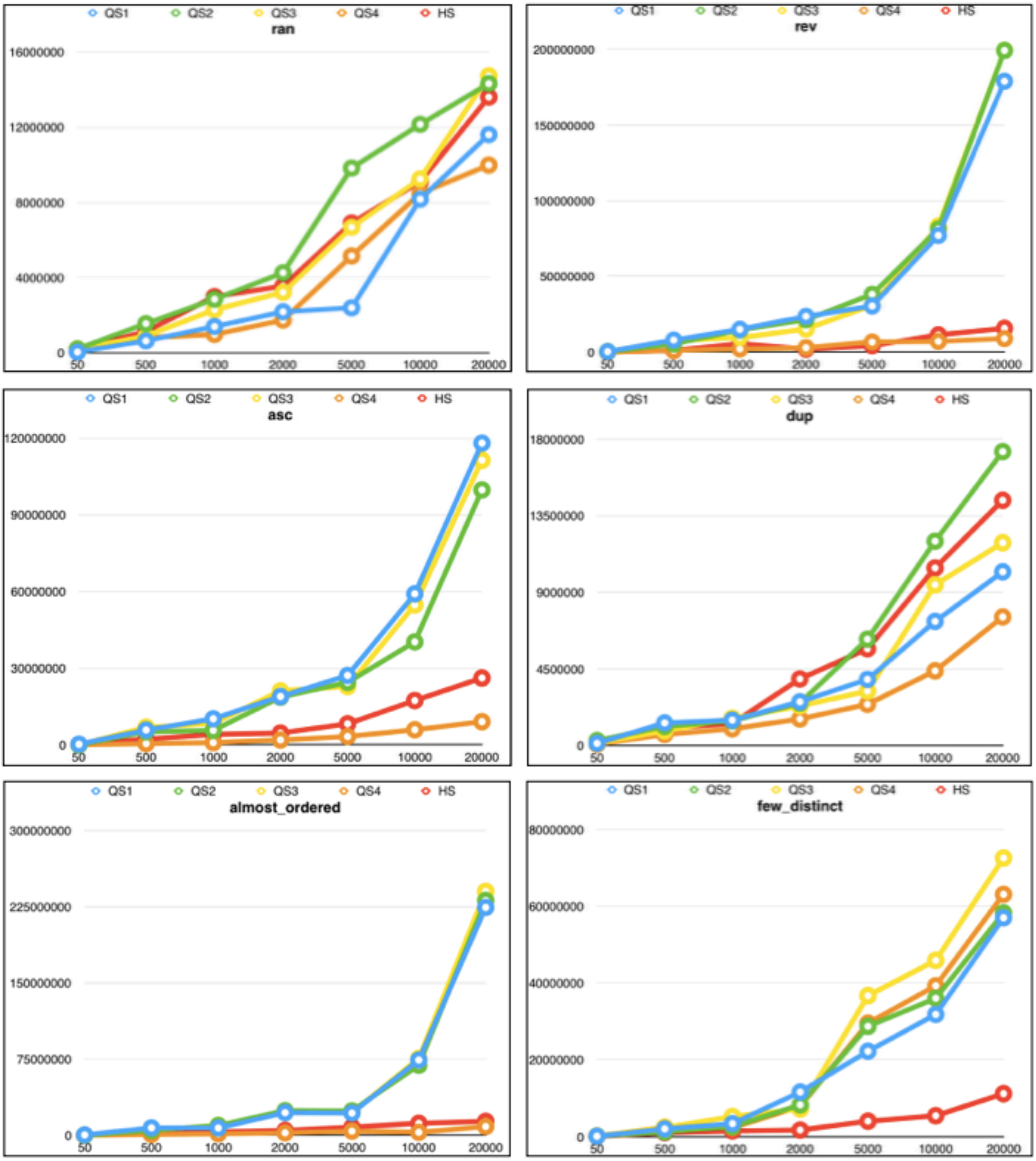

Figure 1. The performance (time in ns) of different sorts with different input types and sizes. *QS1* – first item of partition as pivot; *QS2* – insertion sort when partition size $\leq$ 100; *QS3* – insertion sort when partition size $\leq$ 50; *QS4* – median-of-three as pivot; *HS* – heapsort. *almost_sorted* – only 1 entry in the file is not in the sorted place; *few_distinct* – only contains 4 distinct values.

Table 1 **The performance (time in ns) of different sorts with different input types and sizes.**

ran	QS1	QS2	QS3	QS4	HS
50	43128	210700	213242	70182	155936
500	636708	1554957	904065	809625	1136763
1000	1401165	2848255	2290175	975898	2989725
2000	2184655	4258467	3222127	1737943	3548068
5000	2393239	9819274	6682345	5148055	6901215
10000	8164031	12145782	9239770	8474080	9098975
20000	11597011	14311050	14719641	9978500	13594845
rev	QS1	QS2	QS3	QS4	HS
50	310542	206777	386580	75082	92990
500	7956109	5369825	7408278	1154370	1057953
1000	15055018	14626053	9578065	2135600	5455645
2000	23618789	21345989	15247300	2855435	1938265
5000	30447636	38196750	30975308	6643612	4196210
10000	77075461	81305799	83250219	7104721	11414817
20000	179016964	199390080	199477476	8907306	15698203
asc	QS1	QS2	QS3	QS4	HS
50	230248	5161	10490	73852	126943
500	5792755	5087793	6849435	474024	2208770
1000	10184990	5642369	8403178	869236	4059613
2000	18895450	18650383	21093055	1894845	4607703
5000	27148071	24431501	22823381	3244713	8217941
10000	59035645	40253723	54734876	5894207	17325647
20000	117984813	99687687	111298569	9056220	26135945
dup	QS1	QS2	QS3	QS4	HS
50	132626	293192	136772	80002	132612
500	1335110	1124589	853640	652887	1014895
1000	1495268	1452419	1603430	983633	1297721
2000	2571655	2500694	2329178	1561548	3913991
5000	3887010	6250075	3195909	2423746	5686160
10000	7292215	12000699	9448510	4386546	10429750
20000	10204430	17264543	11910005	7555244	14404863
almost_ordered	QS1	QS2	QS3	QS4	HS
50	386445	24527	11319	75367	245163
500	7342377	4938041	5073584	919517	1790530
1000	7192746	9657326	9754780	1558375	3215670
2000	22258906	24351646	23148376	2434753	4818590
5000	21843391	24114596	22309796	4151308	8006375
10000	74133035	68931301	75679707	2945111	11827735
20000	224242538	231105581	240034015	8632005	13766831
few_distinct	QS1	QS2	QS3	QS4	HS
50	127295	166260	168240	183710	92530
500	2036218	1255490	2398436	2438854	1084133
1000	3410521	2661064	5257310	2331898	1510831
2000	11645148	8313385	7247714	7906809	1740195
5000	22271365	28761868	36735293	29579850	4053730
10000	31852288	36049137	45919804	39304645	5474300
20000	57016821	58304080	72566832	63108571	11221283

Heapsort Performance – (insensitive to order of data)

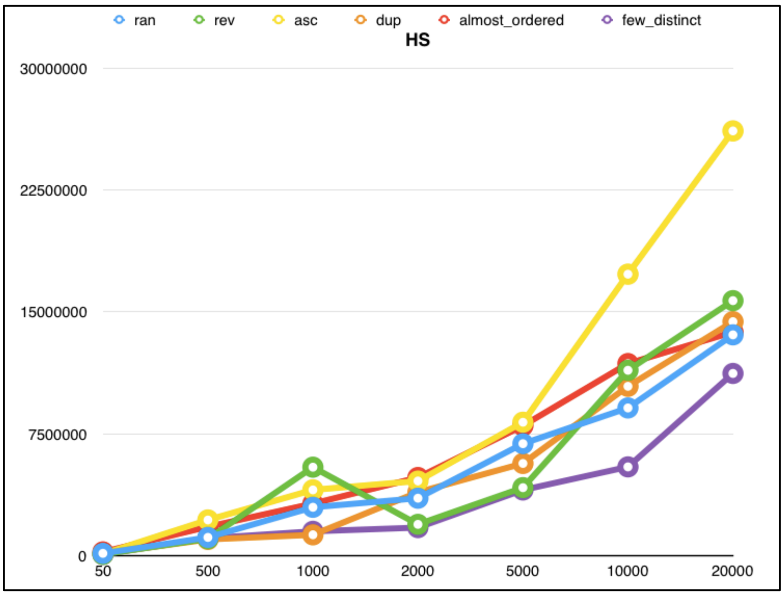

Figure 2 The performance of heapsort on different types of files.

From **Figure 2**, we can see that except for the ascending data with size 20K, there is minor difference among the performances of sorting different types of input files. This indicates that the heapsort is insensitive to the order of data, and thus can be viewed as the baseline in **Figure 1** for further analysis.

Pivot Selection of Quicksort – (QS1 vs. QS4)

The different strategies on selection a pivot can have an effect on the sorting performance. When the data is randomly ordered, both QS1 and QS4 can make even partitions, and there is no significant difference between their performances.

However, when the input data is in an ascending or reverse order, QS1 starts to partition the array in an extremely uneven way, and its performance begins to deteriorate (runtimes about 10 times larger than those of random data), making an obvious difference with QS4 as the size of data grows larger.

Insertion sort combination – (QS1 vs. QS2 vs. QS3)

In **Figure 1**, the combined insertion sort seems to have little effect on the performance of quicksort in, which infers that it may only change the lower ordered terms or constants of the time complexity.

To examine their potential effects, **Figure 3** shows the performance in smaller random data sets. For random data, insertion sort has an average cost of O(n^2). Since the cost of quicksort is O(n*lg n) for random data, the insertion sort could make the performance become more inefficient, when there is larger size of partition being sorted by the insertion sort.

Comparing QS2 and QS3, QS2 has partition size of 100 being sorted by the insertion sort, while QS3 has partition size of 50. Thus, QS2 may cost more time, and have worse performance than QS3. The statistical analysis result shown in **Figure 3** also demonstrates the effect of the insertion sort.

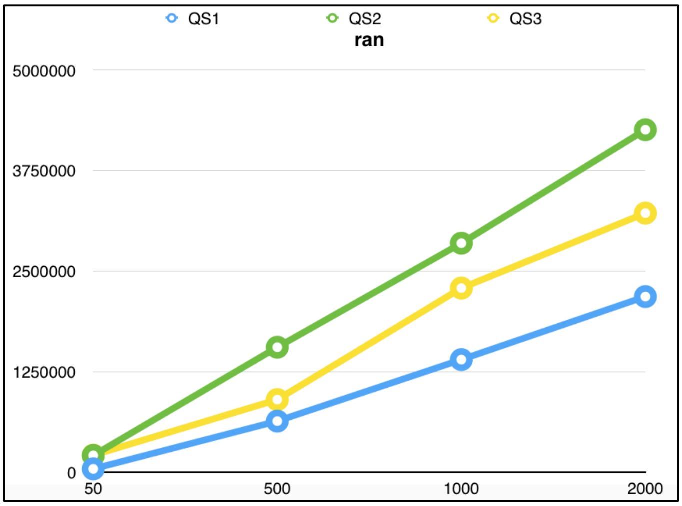

Figure 3 The performance of quicksort with different insertion sort combination in smaller data sets.

Number of Duplications – (dup vs. few_distinct)

The few_distinct data set only contains 4 distinct values, and thus it has a very high percentage of duplicates. From **Figure 1**, we can see that the performance of small portion of duplicates in random data (dup) is similar than that of random data. However, the performance of the various quicksorts become worse when the input files are few_distinct data sets (the heapsort performance as baseline).

The reason is that this type of data sets could create uneven partitions no matter which implementation of the sort – or more precisely, no matter which element is selected as pivot. Because of the partition bias, the duplicates make the partition uneven. The net result of partition would be all of the duplicates of the pivot being at the right or the left partition, and thus the performance is deteriorated.

Brief Summary – (Which factor has the most effect on efficiency?)

In the above analysis, the important factors that could affect sorting performance are: data size, data type, and implementation of sorting. I think the factor that has a big influence on performance is data type. Take the quicksort as an example, even if we use the strategy that would use the median value as the pivot to make the sizes of partitions even, the few_distince data type still breaks the plan. Thus, for sorts that are sensitive to the order of data, the data type could be the most impotent factor on efficiency.

Justification for Design Decisions

Lab4 Class

The Lab4 class is the main entry of the program, which reads in an input file with any formats, as long as it contains only integer values; and then calls different types of sorts to sort those values in ascending order. To make the program more practical and to make the output file more succinct, I chose to prompt the user to select only one sorting type, and the user can simply enter the type number to get the result.

Different Sort Classes

Choice of iteration

The heapsort and the four quicksort variations in this assignment are all iterative. In this case, both the heapify method of heapsort, and the partition process of quicksort have recurrence relation, and can be implemented recursively. The reason I chose to write the iterative versions is that it is easier to understand, but it may require an auxiliary stack, and in contrast, the recursive could be more intuitive

What about recursion?

If I chose to implement those sorts recursively, the length of the code would be reduced because of recursive calls and it would look more concise. However, the drawback is that recursion uses the system stack, so it has the risk of running out of system resources when the file size is extremely large.

Heapsort implantation

Instead of using the same heapify method for building the heap and extracting element in sorted order, I divided it into insert and delete method in order to decrease the lower order terms and constants of the time complexity. The insert method uses the bottom-up heapification strategy, where the node only compares to its parent node rather than two of its child nodes, and the insertion cost in average is O(1).

What I learned

1. I learned how to put the sorting material in class into practice. There are plenty of sorts introduced in the lecture, and I know the underlying concepts of them. However, it is more difficult than I have imaged to implement these sorts. Through this assignment, I learned how to write the ideas into pseudocode and then make them into real Java code.

2. Through the analysis, I learned how to interpret these collect data of time complexity. I learned that, to compare two different performances, we could make a baseline to normalize the scales. In this case, the baseline is the performance of the heapsort, which is insensitive to data types, and has the time cost O(n*lg n), which is the same as that of quicksort in its best case.

<u>What I might do differently next time</u>

Next time I may want to write code for other simple sorts such as bubble sort and simple selection sort, and compare their performances on different sets of data. The practice may help me get more sense of the mechanisms of different sorts and know how to choose an appropriate one in a particular situation.

Another thing I would like to try next time is to add the new feature/option of sorting in a descending order. This could make the program more user friendly since some user may want to have data sorted in reverse order. For example, they many want to have the data with higher scores at the top of the list.

<u>References</u>

1. Borrowed code for quicksort: http://www.geeksforgeeks.org/iterative-quick-sort/, contributed by Rajat Mishra.
2. Borrowed code for heapsort: http://quiz.geeksforgeeks.org/heap-sort/.