**ECE578 DSA HW2**

Zhongze Tang (zt67)

**Q1**

I use the dataset provided for Q2 to test the two algorithms, and I only count the number of key comparison.

As we can see clearly in the table and charts, when data is in-order (data0.\*), the Shell Sort is less effective than Insertion Sort. This is because under the best case, Insertion Sort needs to compare N-1 times when Shell Sort needs to compare nearly N/7 + N/3 + N ≈ 1.476N times.

And when using data1.\*, which means arbitrary data sets, things change. Shell Sort performances more effectively than Insertion Sort. It speeds up by making a tradeoff between size and partial order in the subsequences. We know that the Insertion Sort performances well in short sequences and partially sorted sequences. Shell Sort first divides the sequence into short subsequences and when sort later, the subsequences have been partially sorted. Both part of Shell Sort are using Insertion Sort, so that’s why Shell Sort is more effective in this case.

Chart 1 – Numbers of key comparison in two sorts using data0.\*

Chart 1 – Numbers of key comparison in two sorts using data1.\*

|  |  |  |
| --- | --- | --- |
| DataSet | InsertionSort | ShellSort |
| data0.1024 | 1023 | 1510 |
| data0.2048 | 2047 | 3021 |
| data0.4096 | 4095 | 6045 |
| data0.8192 | 8191 | 12091 |
| data0.16384 | 16383 | 24184 |
| data0.32768 | 32767 | 48370 |
| data1.1024 | 265553 | 209064 |
| data1.2048 | 1029278 | 803644 |
| data1.4096 | 4187890 | 3225240 |
| data1.8192 | 16936946 | 13095081 |
| data1.16384 | 66657561 | 51601294 |
| data1.32768 | 267966668 | 207354020 |

Table 1 – Numbers of key comparisons in different cases

**Q2**

Q3

At the earliest time, I write an optimized version of Shell Sort. It will check if the subsequences are already in order. In this case, the order of the algorithm will be O(N) for any sorted sequences. Actually, the algorithm just checks if all the subsequences are sorted recursively, which means all the *merge()* methods are skipped.

Furthermore, some other algorithms, like Insertion Sort and Bubble Sort, also performances well. The order of them is O(N) as well.

Later, I think about another sort, Counting Sort. It’s not a comparison sort, and it may work better than comparison sort in theory. So I implemented a simple version of Counting Sort, which is, count the number of 1, 11, 111 and 1111, and then write the four numbers into an ArrayList respectively, each number N times (N stands for the count result of it). This simplified method is more like a kind of “cheat”, because the algorithm can only sort the data set provided by the question.

The order of this algorithm is still O(N), but needs O(4) extra space. After the test of running time, it runs a little bit slower than the optimized Shell Sort, not to mention the full version of Counting Sort, which requires more operations and more space.

In summary, I think the optimized version of Merge Sort is the “most” effective algorithm to sort the data set.

See <https://en.wikipedia.org/wiki/Counting_sort> for more information about Counting Sort.

**Q4**

I only compare the key comparison, and we can safely conclude that the comparison times are totally the same for Topdown and Bottomup version of Merge Sort when the size of dataset is a power of 2.

|  |  |  |
| --- | --- | --- |
| DataSet | Result\_topdown | Result\_bottomup |
| data0.1024 | 5120 | 5120 |
| data0.2048 | 11264 | 11264 |
| data0.4096 | 24576 | 24576 |
| data0.8192 | 53248 | 53248 |
| data0.16384 | 114688 | 114688 |
| data0.32768 | 245760 | 245760 |
| data1.1024 | 8954 | 8954 |
| data1.2048 | 19934 | 19934 |
| data1.4096 | 43944 | 43944 |
| data1.8192 | 96074 | 96074 |
| data1.16384 | 208695 | 208695 |
| data1.32768 | 450132 | 450132 |

Table X – The comparison times of two versions of Merge Sort

The reason is obvious. Both versions of Merge Sort share the *merge()* method, and comparison only happens in it. At the same time, the frequencies of calling this method are the same for two versions because their only difference is the order of the calls.

However, when the data set size is not a power of 2, the comparison times will be different.