ECE 573 – DATA STRUCT & ALGS

ASSIGNMENT2

Sorting

Zhongze Tang (zt67)

2018-2-20

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 comparisons.

As we can see clearly in the table and charts, when data is in-order (data0.*), the Shell Sort is far less effective than Insertion Sort. This is because, in the best case, Insertion Sort needs to compare (N-1) times when Shell Sort needs to compare (N-7) + (N-3) + (N-1) = (3*N – 11) 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 parts of Shell Sort are using Insertion Sort, so that’s why Shell Sort is more effective in this case.

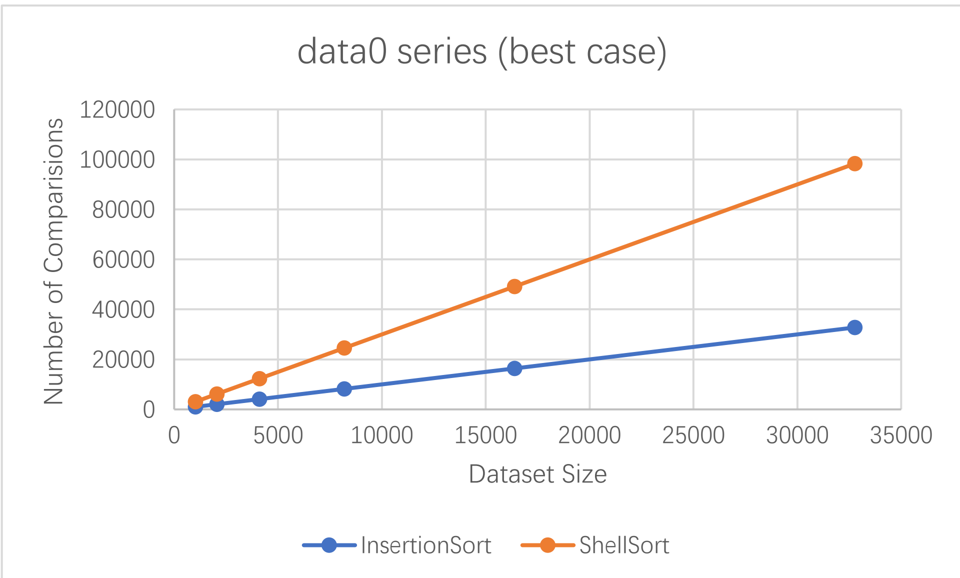

Chart 1 – Numbers of key comparison in two sorts using data0.*

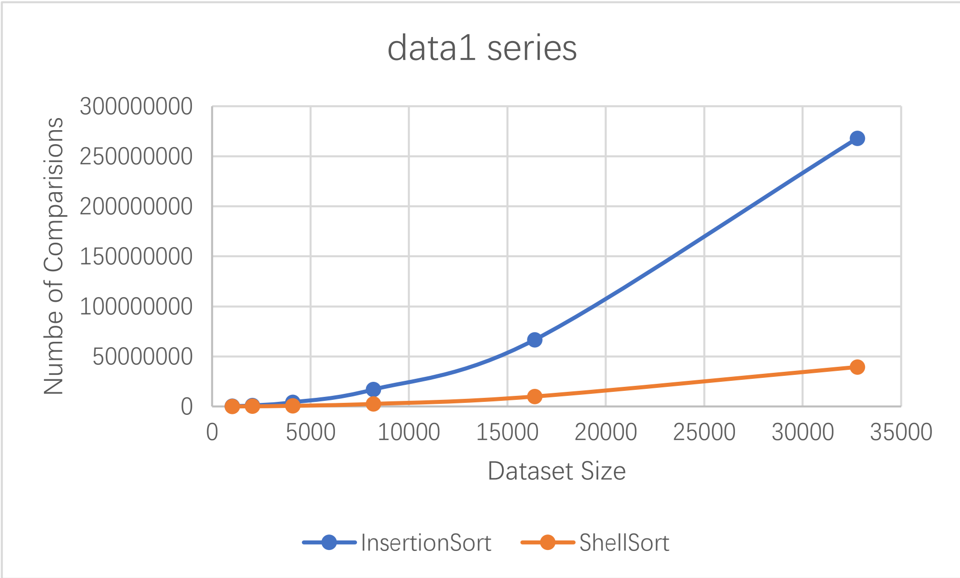

Chart 2 – Numbers of key comparison in two sorts using data1.*

DataSet	InsertionSort	ShellSort
data0.1024	1023	3061
data0.2048	2047	6133
data0.4096	4095	12277
data0.8192	8191	24565
data0.16384	16383	49141
data0.32768	32767	98293
data1.1024	265553	46728
data1.2048	1029278	169042
data1.4096	4187890	660619
data1.8192	16936946	2576270
data1.16384	66657561	9950922
data1.32768	267966668	39442456

Table 1 – Numbers of key comparisons in different cases

Q2

DataSet	RunTime(ns)	Distance
data1.1024	1167183	264541
data1.2048	1741291	1027236
data1.4096	3401951	4183804
data1.8192	5024185	16928767
data1.16384	7206814	66641183
data1.32768	11946244	267933908

Table 2 – RunTime and Kendall Tau distance in different cases

Suppose that all the data is from 0 to 2^N-1 and each number appears only once. All the numbers are reduced by 1 when read from the file to avoid out-of-range errors.

To calculate the Kendall Tau distance between array A and array B, we first get an array C that C[A[i]] = i, i from 0 to A.length -1. And find an array D that D[i] = C[B[i]], i from 0 to A.length -1. Then just figure out the number of inversions of D, and that's the result.

I use Merge Sort to find the number of inversions of D. So the order of this algorithm is supposed to look like O(nlgn).

Because we have to map the data to get D, there will be an item of x in the model. So, I suppose that f(x) = a*x*log2(x) + b*x and finally the curve fits well. Obviously it's less than quadratic time on average.

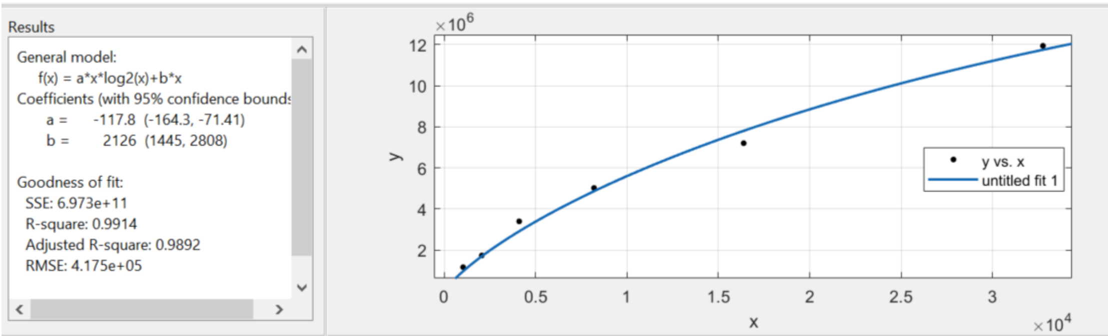

Chart 3 – Curve Fitting

In fact, since one of the input data set has been sorted, the question is the same as the one to find the number of inversions in an arbitrary data set.

Q3

At the earliest time, I write an optimized version of Merge 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 Merge 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 the 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 3 – 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.

Q5

First I compare the running time of Merge Sort (Naïve bottom-up) with Quick Sort(No cut off) and Quick Sort (cut off = 7). I do not shuffle the data before the Quick Sort because data0.* are the best cases and data1.* are the data sets that shuffled from data0.*.

DataSet	MergeSort	Quicksort	QuickSort(CUTOFF=7)
data0.1024	1053462	589544	571070
data0.2048	348882	638153	178012
data0.4096	743745	305282	209050
data0.8192	1024395	595538	255113
data0.16384	1645962	549557	314314
data0.32768	3730553	1491022	749493
data1.1024	984490	804753	627150
data1.2048	773797	671817	777575
data1.4096	1031456	848599	426804
data1.8192	1526329	1032770	1226219
data1.16384	3739257	2274920	2117598
data1.32768	8384261	7046044	5930014

Table 4 – Running time (ns) of different algorithms

As we can see in the two figures below, Quick Sort is faster than Merge Sort in general. And when cut off = 7, Quick Sort costs less time to finish the sort.

All the algorithms' running time looks like a function of NlgN, just as expected.

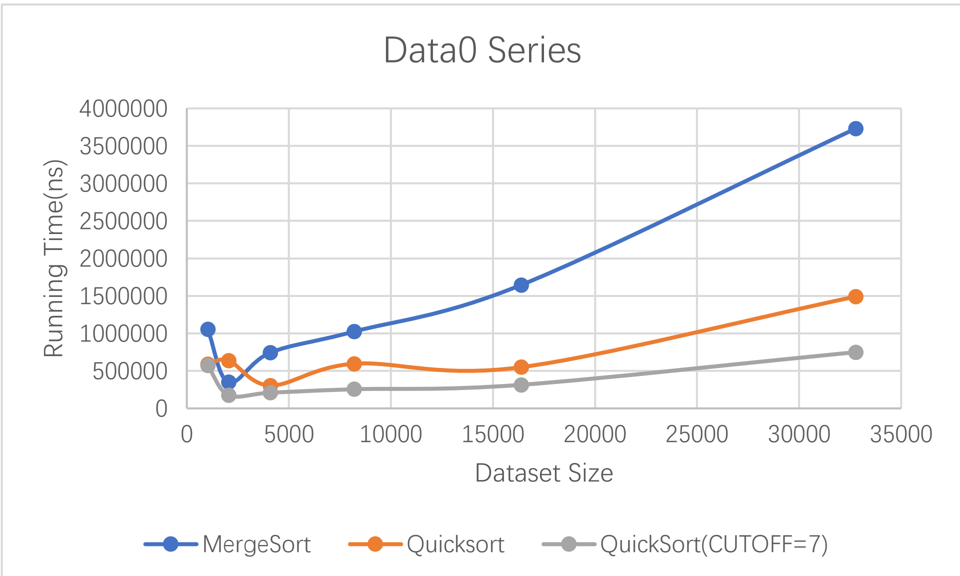

Chart 4 – Running time of different algorithms, using data0.*

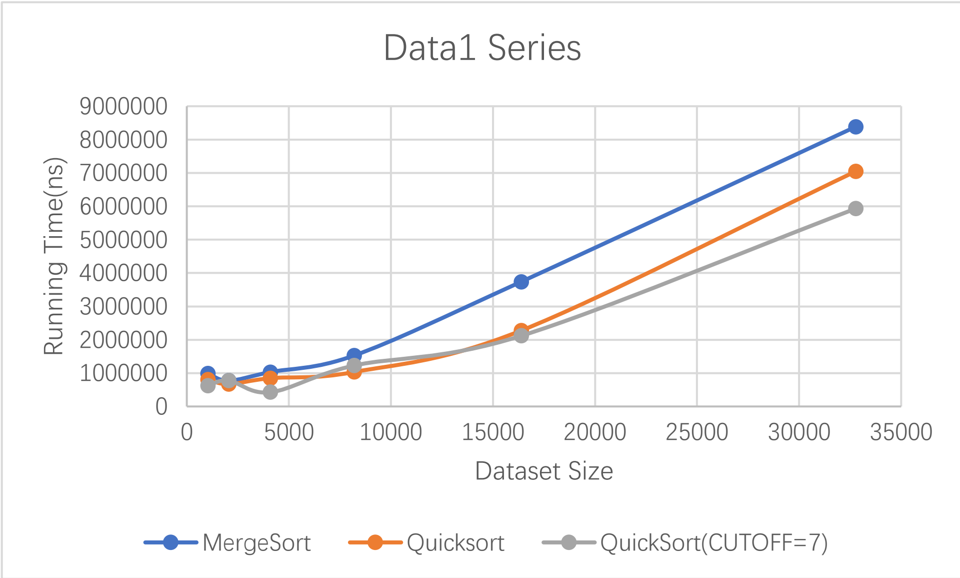

Chart 5 – Running time of different algorithms, using data1.*

Furthermore, I find that Quick Sort with cutoff performs better when the data set has been sorted compared to when the data set is arbitrary. That's because the Insertion Sort works better in a sorted sequence with an order of O(n), while it's O(n^2) on average.

Then I do an experiment on what's the best value of the cutoff. The result is not that ideal, but we can still find that a cutoff between 7 and 10 is a good choice.

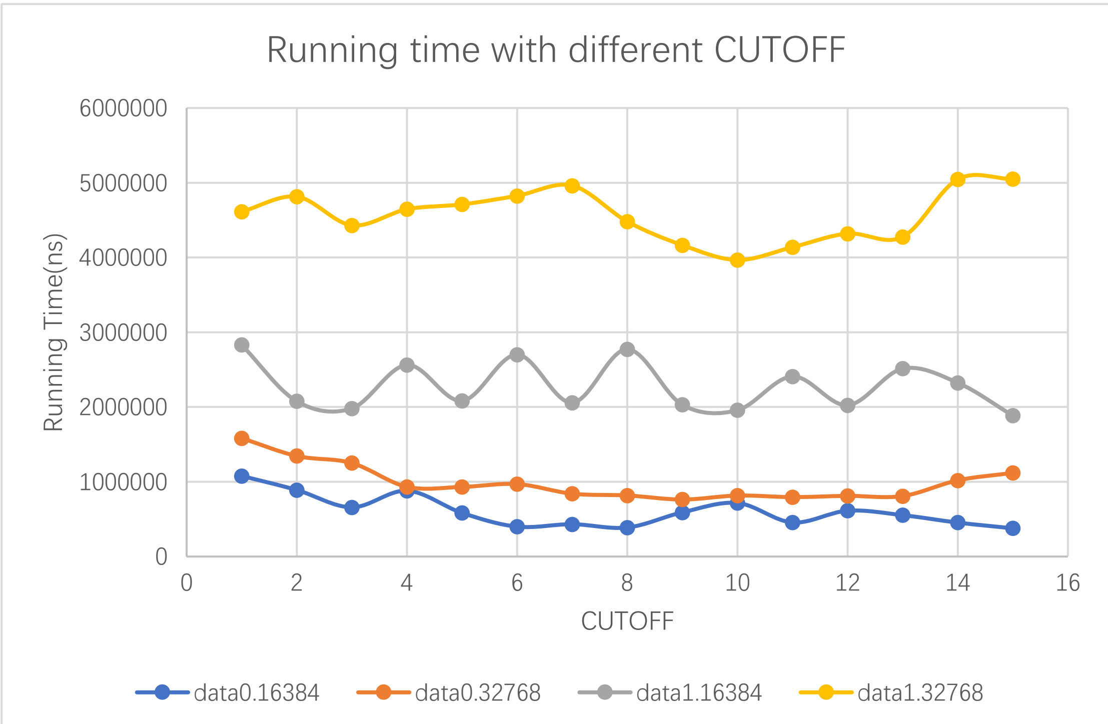

Chart 6 – Running time of different cutoffs

Q6

Col.2: Merge Sort (bottom up). Every four elements are sorted.

Col.3: Quick Sort (standard, no-shuffle). The elements before “navy” are smaller than it while the elements after “navy” are bigger than it. And the last two elements, “palm” and “pine” are still in place.

Col.4: Knuth shuffle. All the elements before “silk” are shuffled, and all the elements after “silk” are still in place.

Col.5: Merge Sort (top down). The first half of the array has been sorted, and the first half of the last 12 elements and the second half of the last 12 elements are sorted respectively, too. If it’s bottom-up, it should be that the first 16 elements are sorted.

Col.6: Insertion Sort. All the elements before “teal” (including itself) are sorted but different from the Col.10, and the elements after it are still in place.

Col.7: Heap Sort. It’s an array of a max heap, so it’s supposed to be a heap sort.

Col.8: Selection Sort. All the elements before “mint” (including itself) are sorted and are the same as the Col.10, and the elements after it are still in place.

Col.9: Quick Sort (3-way, no-shuffle). Just like the standard Quick Sort, the elements before “navy” are smaller than it while the elements after “navy” are bigger than it. However, the last one is “plum”, which is the first element in the array that is greater than “navy”. In the 3-way Quick Sort, the word “plum” exchanges with the array[hi], and then gt--, so that’s why it can keep in place.