Assignment 2

Weijia Sun | ws368

Table:Number of Comparisons – Dataset (Already Sorted)

|  |  |  |
| --- | --- | --- |
| dataset | shell sort | insert sort |
| data0.1024 | 3061 | 1023 |
| data0.2048 | 6133 | 2047 |
| data0.4096 | 12277 | 4095 |
| data0.8192 | 24565 | 8191 |
| data0.16384 | 49141 | 16383 |
| data0.32768 | 98293 | 32767 |

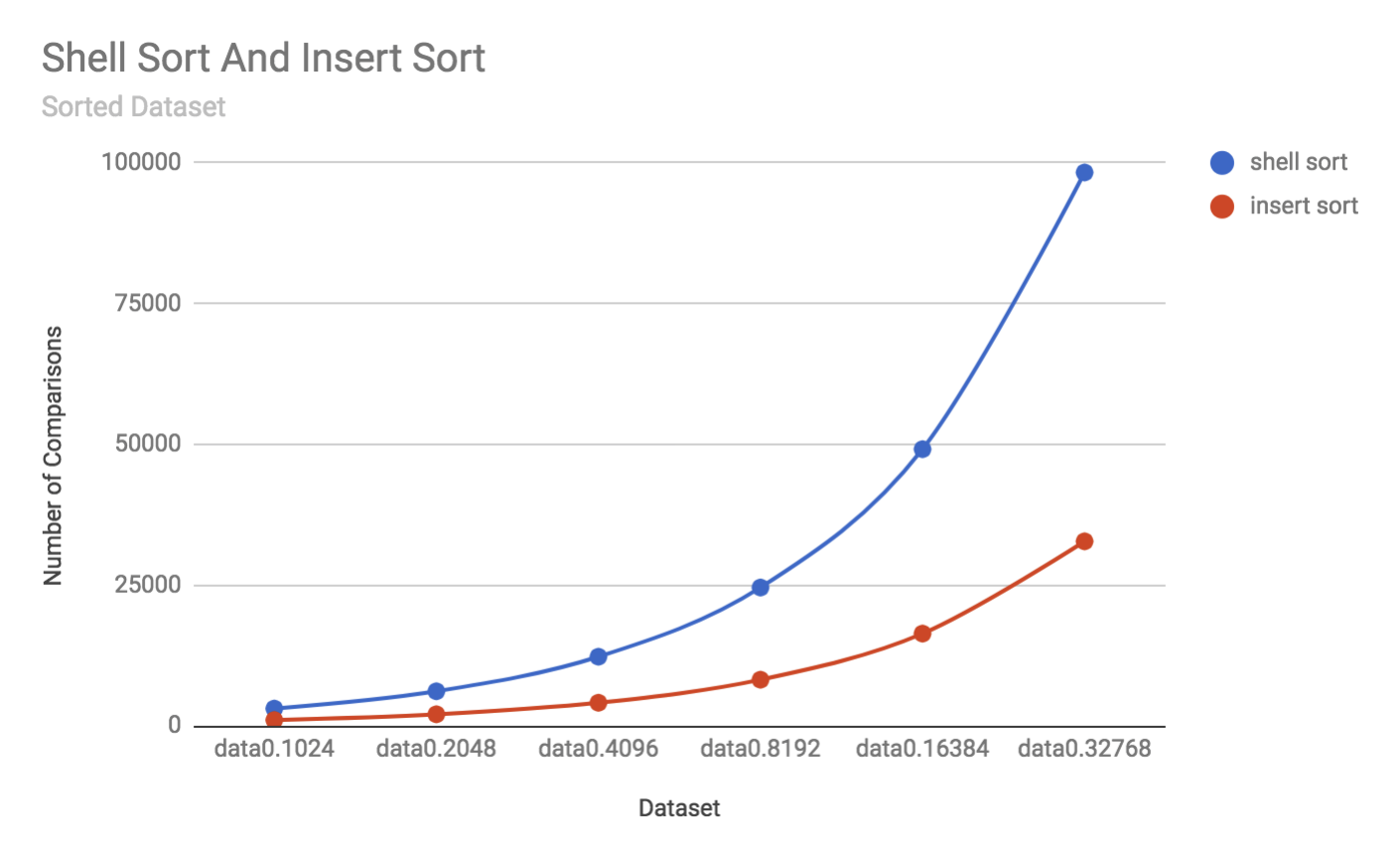
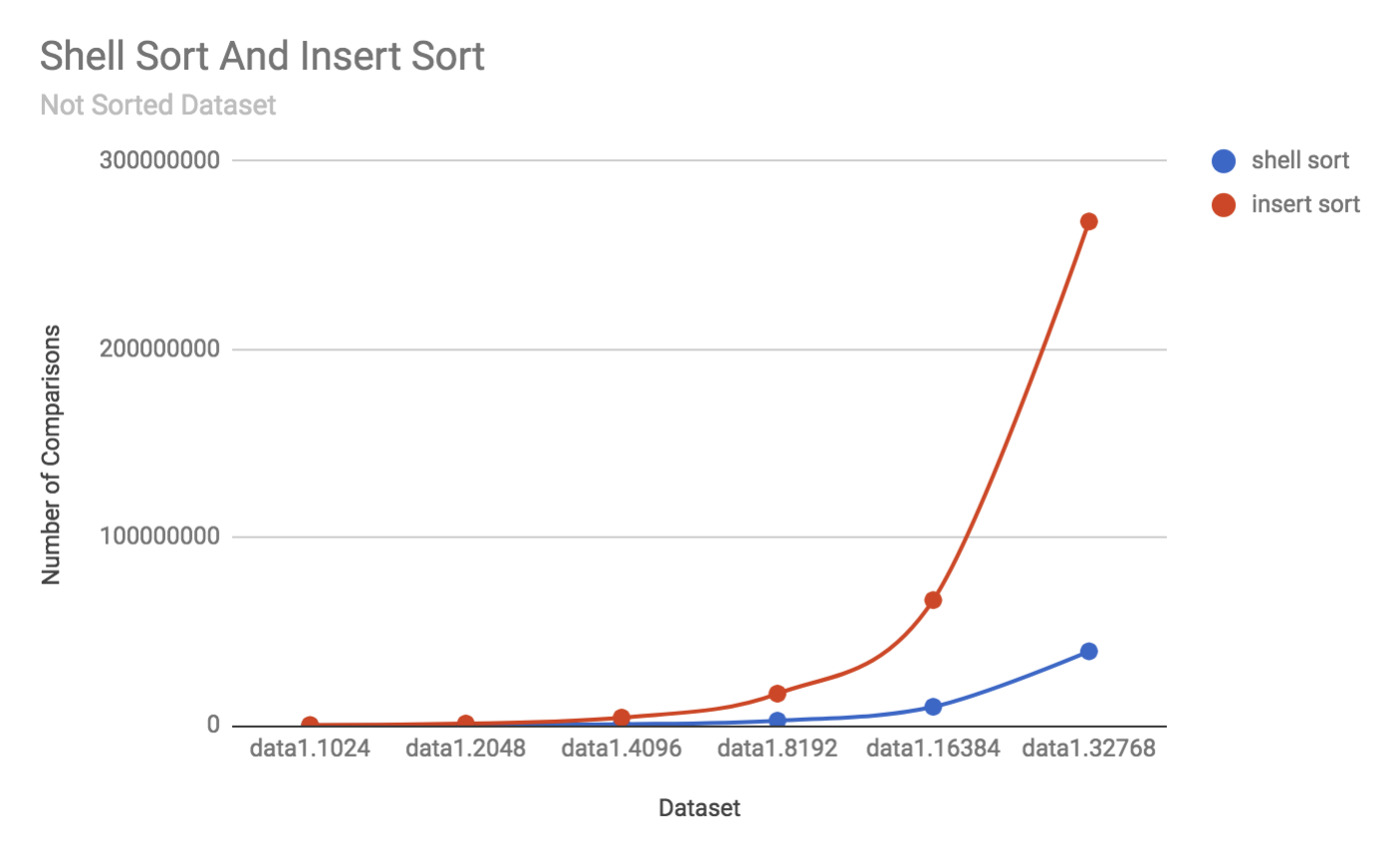


Table:Number of Comparisons – Dataset (Already Sorted)

|  |  |  |
| --- | --- | --- |
| dataset | shell sort | insert sort |
| data1.1024 | 46728 | 265553 |
| data1.2048 | 169042 | 1029278 |
| data1.4096 | 660619 | 4187890 |
| data1.8192 | 2576270 | 16936946 |
| data1.16384 | 9950922 | 66657561 |
| data1.32768 | 39442456 | 267966668 |



**Analyze：**

For the first data set which is already sorted. It is the best case for insertion sort. The number of comparisons is always N-1. But for the shell sort, it has to scan the array for several times when h equals to different value. When h=1, the number of comparisons has already been N-1. So for sorted array, insert sort does better than the shell sort.

For the second dataset which is randomly given. Shell sort does way better than insert sort. Because it efficiently avoid the long-distance 1-sort. It allows elements to move long distance by beginning with large h-value, which can reduce large amounts of disorder quickly. Then the data becomes partially sorted. In this situation, smaller h-sort can solve the problem efficiently. So that’s why shell sort works better than insert sort in random given array.

Table Name: Running Time-Data Size

|  |  |  |
| --- | --- | --- |
| dataset | Ordered Set | Shuffle Set |
| 1024 | 1109 | 645 |
| 2048 | 570 | 1417 |
| 4096 | 1225 | 2936 |
| 8192 | 2709 | 7073 |
| 16384 | 6551 | 15926 |
| 32768 | 17221 | 24988 |

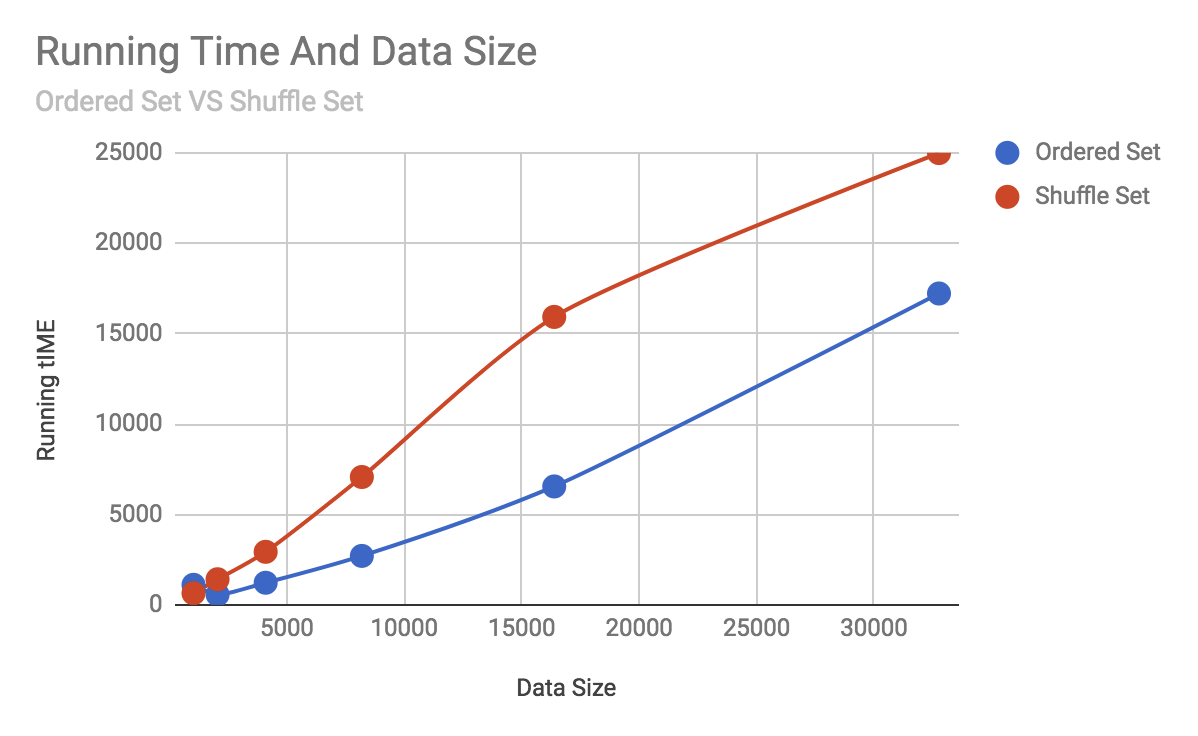


Table Name: Number of inversions – Dataset

|  |  |
| --- | --- |
| Dataset | Number of Inversions |
| data0.1024 | 0 |
| data0.2048 | 0 |
| data0.4096 | 0 |
| data0.8192 | 0 |
| data0.16384 | 0 |
| data0.32768 | 0 |
| data1.1024 | 264541 |
| data1.2048 | 1027236 |
| data1.4096 | 4183804 |
| data1.8192 | 16928767 |
| data1.16384 | 66641183 |
| data1.32768 | 267933908 |

Discuss: I use the thought of merge sort to count the number of inversion in complexity of O(nlogn). The key part o the algorithms is as follows:

When merge two subarray, assume i is the index for left subarray, j is the index of the right array. When array[i]>array[j]. For all the element in the left array at the right sort of array[i] is bigger than array[j] . So under this circumstance, there are mid-i+1 pair of inversions. And same for rest of them.

Basically, the time complexity for this algorithm is the same with merge sort. So it is O(nlogn).

The result of the running time is the average of 5 times calculationg.

For this problem, I use the counting sort method. Its’ time complexity is only o(n+r). The n represents the number of elements in the array. The r represents the biggest number in the array. The space complexity is O(r). Counting sort is not a comparison based sort, so for the time complexity, it is smaller than any comparison based sort method. So for this problem, I think it is the most efficient sorting method.

Table: Number of Comparisons(UB and BU) -- Dataset

|  |  |  |
| --- | --- | --- |
| dataset | number of comparisons(UB) | number of comparisons(BU) |
| 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 |

Explain:

As we can see, the number of comparisons needed for UB is same with BU’s. Because when number of element is the power of 2. Both the number of comparisons and the number of array access needed for BU and UB is same, just different at the order.