ASSIGNMENT2

Data Structure & Algorithm

Weijia Sun

ws368 | 02/19/2018

Assignment 2

Weijia Sun | ws368

Q1.

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

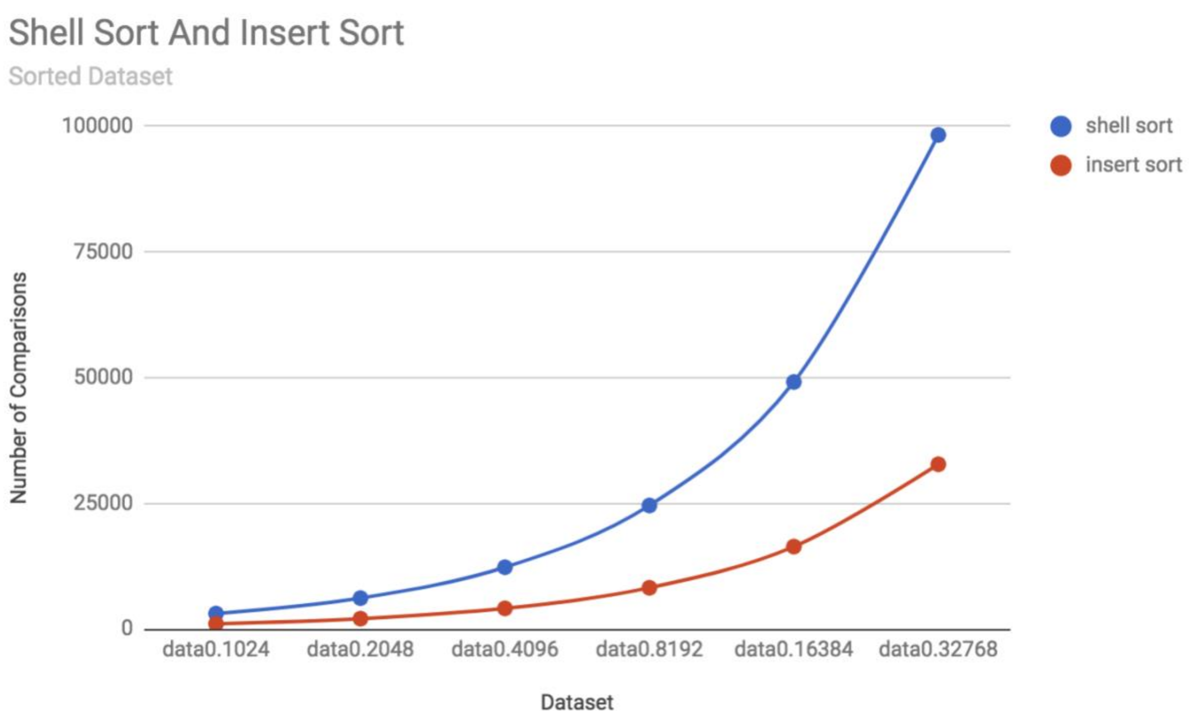
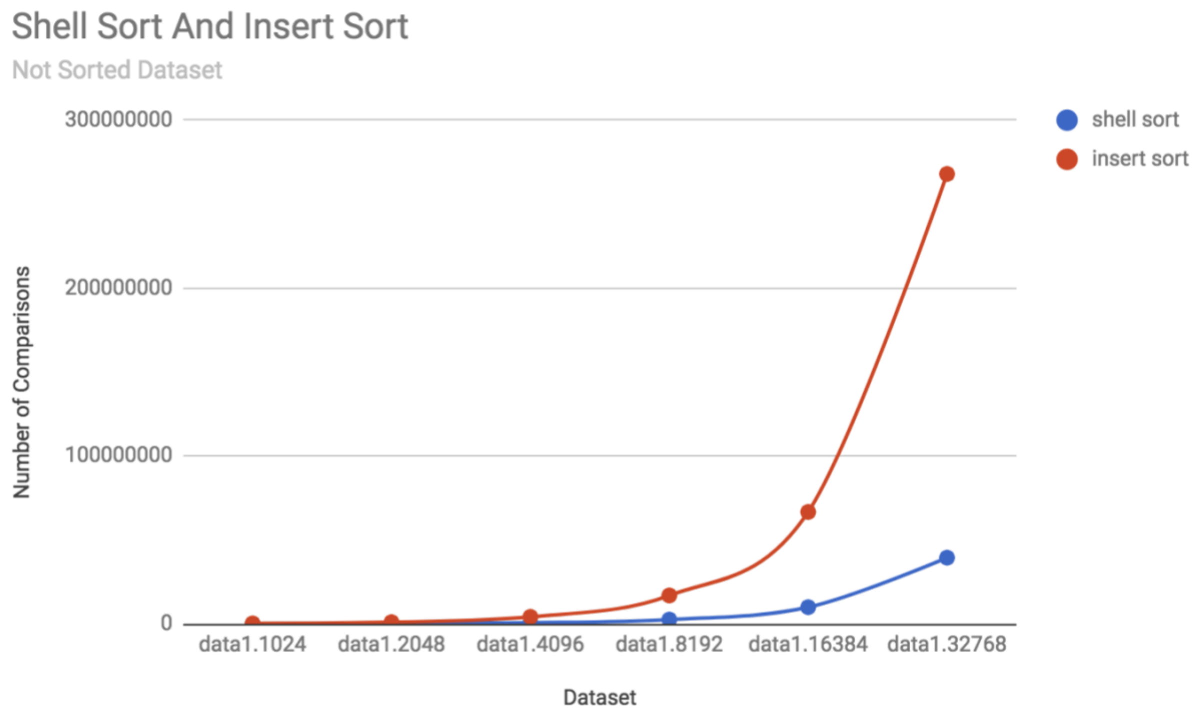


Table:Number of Comparisons – Dataset (Already Sorted)

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.

Q2.

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

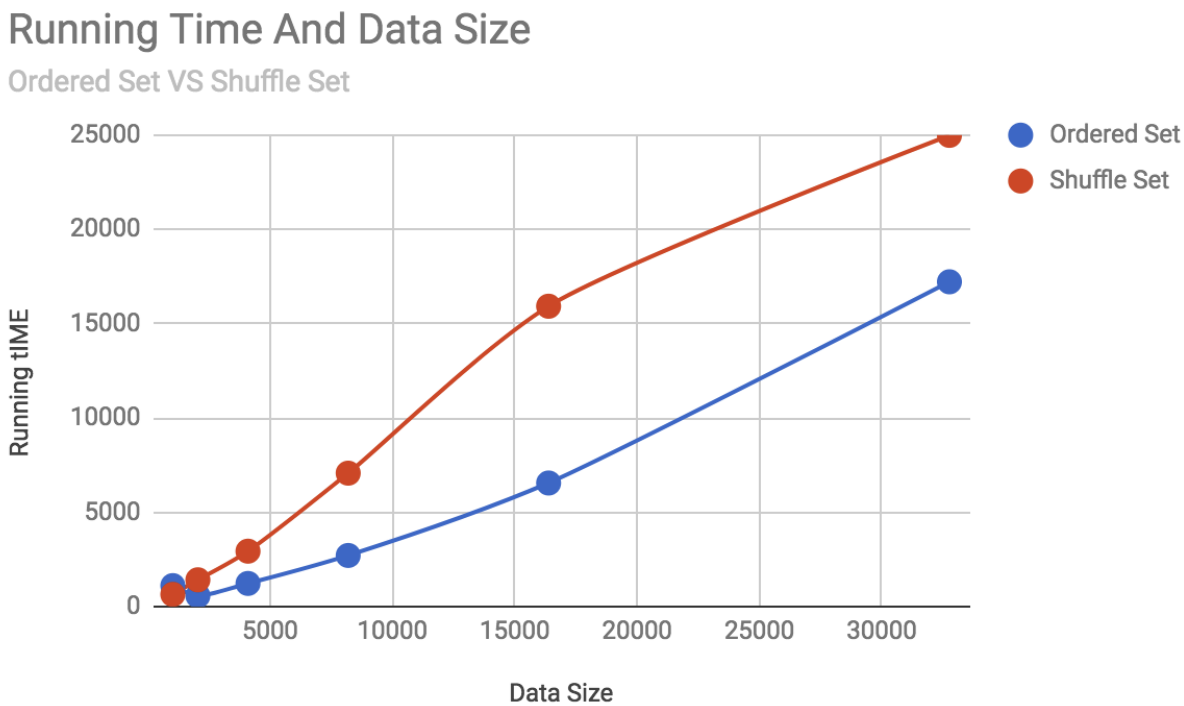


Table Name: Number of inversions – Dataset

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.

Q3.

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).

But for this question, we can assume we already know the values in the array in advance. So for the counting array ,we can directly create a size 4 array, and use the array record the number of times each value. And use the count array to output the result.

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.

Basically, the complexity of the modified counting sort here is O(n). Also, since the array is already sorted, both the bubble sort and the insertion sort perform as O(n) algorithm here, too. So for this question , I think modified counting sort, bubble sort and the insertion sort are the most efficient algorithm.

Q4.

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

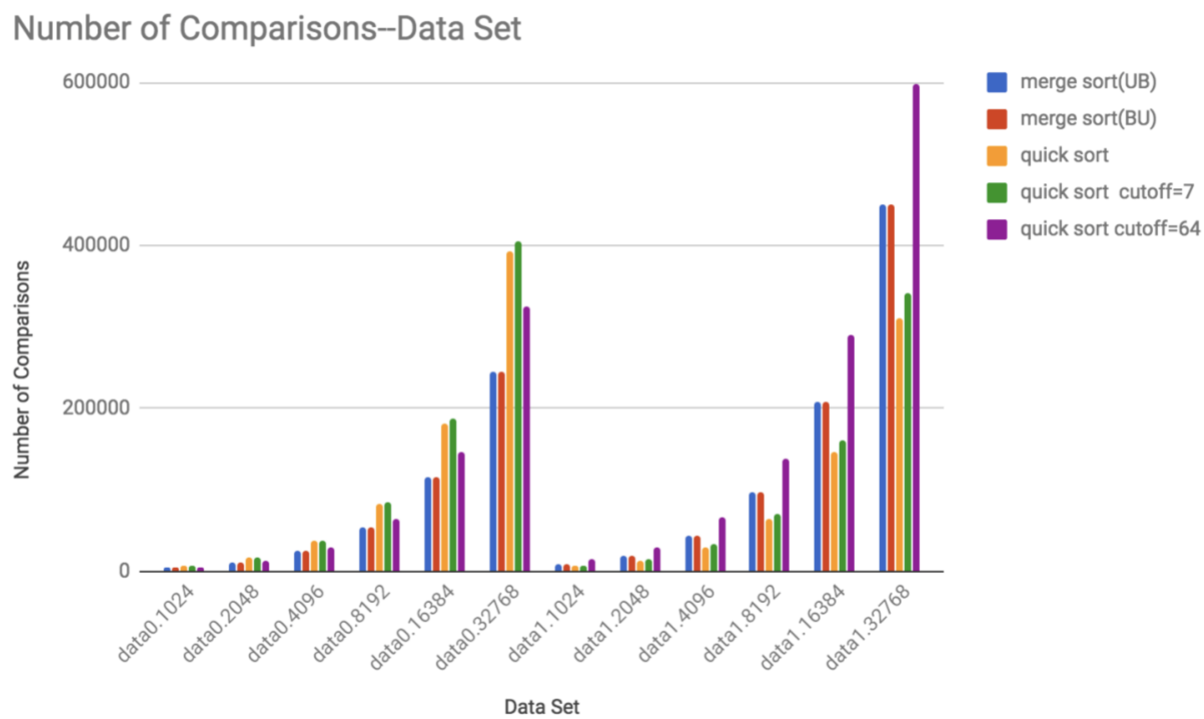
Explain:

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.

Q5.

dataset	merge sort(UB)	merge sort(BU)	quick sort	quick sort cutoff=7	quick sort cutoff=64
data0.1024	5120	5120	7181	7567	5048
data0.2048	11264	11264	16398	17168	12137
data0.4096	24576	24576	36879	38417	28362
data0.8192	53248	53248	81936	85010	64907
data0.16384	114688	114688	180241	186387	146188
data0.32768	245760	245760	393234	405524	325133
data1.1024	8954	8954	6046	7005	15231
data1.2048	19934	19934	12704	14637	29482
data1.4096	43944	43944	28724	32562	65785
data1.8192	96074	96074	63576	71200	139050
data1.16384	208695	208695	145509	160686	290796
data1.32768	450132	450132	310593	341283	598193

From this table, we can plot a graph:



For this question, we already know the data0 series data set is already sorted, the data1 series data set is unsorted and is not the worst case. So I don't shuffle the array in the quicksort.
As we can see, for quicksort, it performs worse than merge sort at the already sorted array and performs better than merge sort at the random array.

Because in the small array, the insertion sort is quicker than quick sort, since even in a very small subarray, quicksort always keep calling itself. So when we set a cutoff value to turn to insertion sort, the algorithm will do better.
When I set the cut-off value as 7. Also, merge sort is better at the best case, but the quick sort with cut-off value=7 works better when the array is random. When I keep raise the cut-off value, the quicksort's performance at sorted array becomes better(because for insertion sort, it performs excellent at sorted array) and worse at random array. When it comes to cut off value =64, for each dataset, it is worse than merge sort.

Q6.
Column2: Merge sort(Bottom-up), because every 4 elements has been sorted.

Column3: Quick sort(Standard, no shuffle), because we can observe that, before word "navy", all the element is "smaller" than navy, after "navy", all the element is bigger than "navy" (alphabetically). So the "navy" is the pivot here.

Column4:Kruth Shuffle: It seems keep implement Kruth Shuffle until word "silk".

Column5:Merge sort(top down), because the left half of the array has been sorted, the first half of the element in the last 12 element in the array has also been sorted, rest of them is still shuffled. So it is Merge sort(top down)

Column6: Insertion sort, because the invariants for insertion sort is: 1. Entries the left of index fixed and in ascending order. 2. Entries to the right of the index haven't been seen. Compare to column 8, we can assure column 6 is insertion sort, which up to "teal"

Column7: The current column is an max heap. So it is in an intermediate step of Heap Sort

Column8: Selection sort, because the invariants for selection sort is : 1. Entries the left of index fixed and in ascending order. 2. No entry to the right of index is smaller than any entry to the left of index. So it's selection sort up to "navy"

Column9:Quicksort(3-way,no shuffle),from the picture ,we can assure that the "navy" is the pivot here. We can tell the difference form the standard Quicksort by the place of the word"plum". For 3 way quick sort. "plum" is the first word to compare with the pivot "navy", since it is larger than "navy". So it exchange with the last word in the column, so "plum" is the last element in the column now.