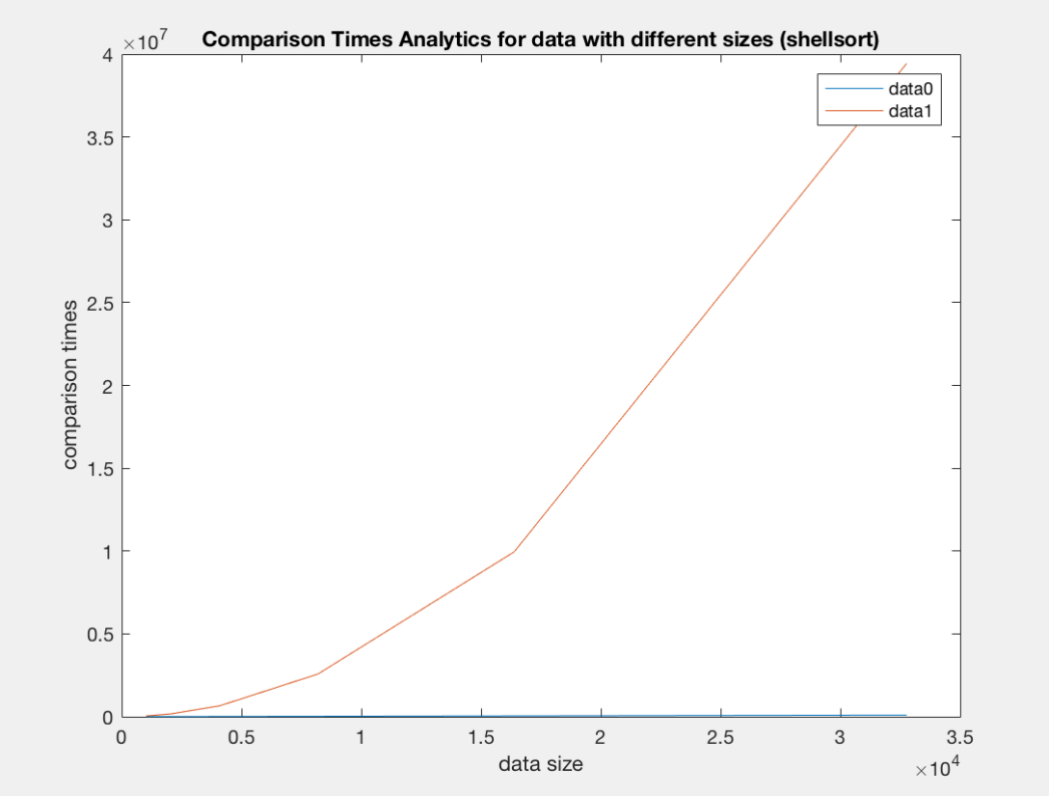
HW2

(Data Structure and Algorithms)

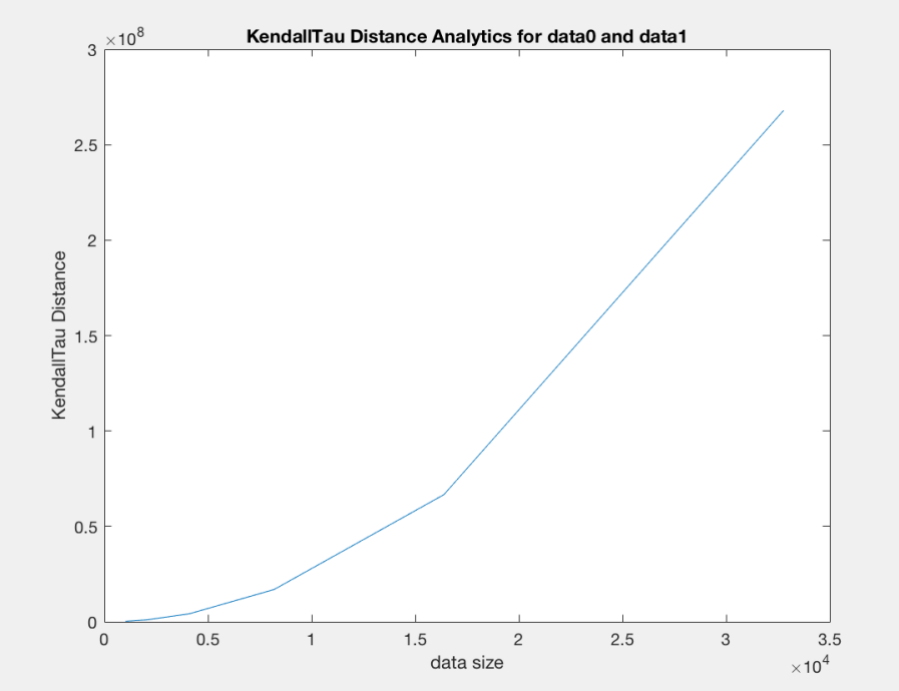
Name: Xiaoyi Tang
RUID: 174008332

Q1:



- Code: shellsort.java
- Why shellsort is more effective than insertion sort:
  - At the beginning, especially when the permutation is out of order, shellsort could divide it into shorter arrays and there is no doubt that it is faster for each shorter array to be sorted separately than sorting the whole sequence.
  - Then, the initial sequence has been partially sorted. Even though shellsort needs almost the same comparison times as insertion sort, shellsort needs fewer swap operations than insertion sort.
  - Therefore, shellsort is more effective than insertion sort in this case.

Q2:

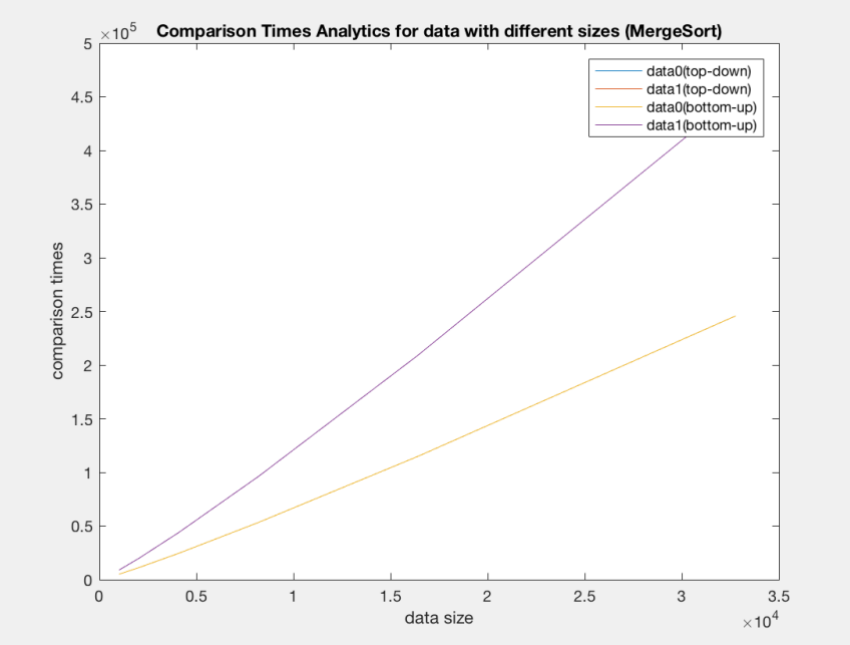


- Code: KendallTau.java
- Train of thought:
  - The Kendall Tau rank distance is a metric that counts the number of pairwise disagreements between two ranking lists.
  - In this case, as the datasets are special for data0 is sorted and data1 is unsorted, the Kendall Tau distance has something to do with the sorting of data1.
  - To satisfy the requirement of computing distance in less than quadratic time in average, I choose to sort the sequence in data1 with merge sort. When merging two subarrays, the Kendall Tau distance should be added by the current length of the first half if the entry in the second half is smaller than the entry in the first half.
- Graph Analytics:
  - Assume that $T(n) = an^b$
  - From the result of my program, I can know that
    when $n = 1024$, $T = 264541$
    when $n = 2048$, $T = 1027236$
    Then we can get that: $b = 1.96 < 2$
  - Therefore, when I compute the distance by merge sort, the time complexity is smaller than quadratic order.

Q3:

- Code: BestSort.java
- Why I choose this sort algorithm:
    - In this case, I choose insertion sort algorithm.
    - As the problem described, all entries in the sequence are in order so that we don't need to do swap operations at all. In insertion sort, we always have some part of the sequence sorted. Just from the second entry, we find the suitable position for it and then move it to the right position. In this best case, all entries just need to be compared with the previous one. We would find that all entries are not smaller than the previous one and don't need swap at all.
    - Therefore, the total time complexity for this algorithm is $O(n)$, which could be the best sorting algorithm among all. And that is why I choose insertion sort to solve this question.

Q4:

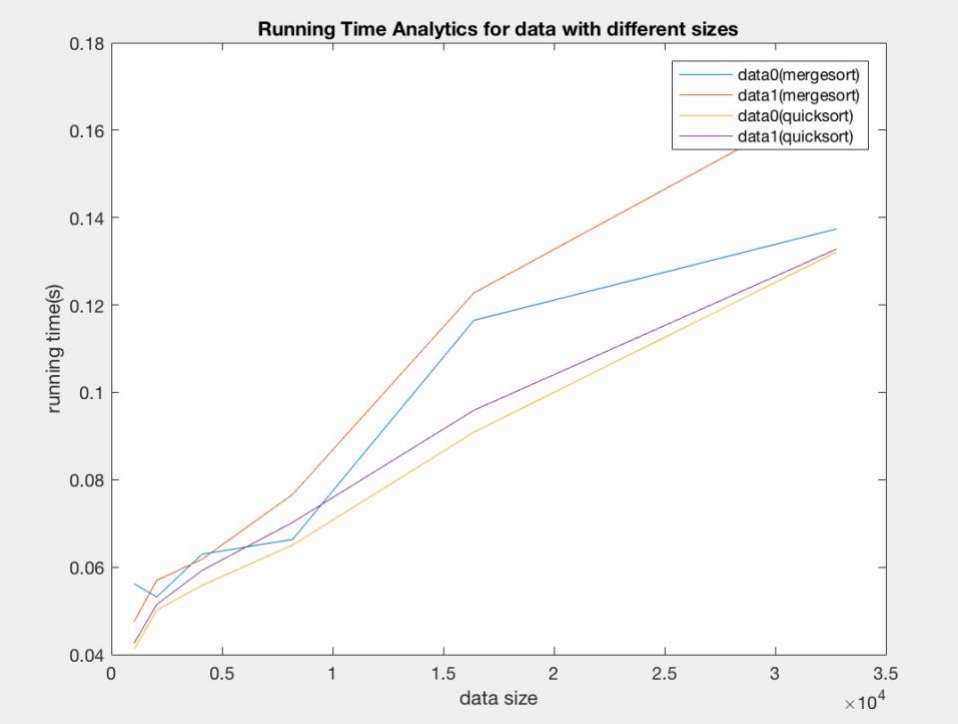


- Code: MergeSort.java (top-down) and BottomUpMS.java (bottom-up)
- Analytics:
    - What can be seen obviously from the graph is that the comparison times for two versions of MergeSort is the same.
    - The reason is that: top-down MergeSort is to divide the sequence and merge it at the same time. Before it starts to merge, it should divide the sequence first. Also, it increases the size of merge by 2 times every time. Bottom-up divides the sequence completely at the beginning and then merges it. It starts merging from the smallest size and increase it by 2 times every time. As a result, although the

two versions of mergesort realize the function in two different ways, the times and order for comparison are the same.

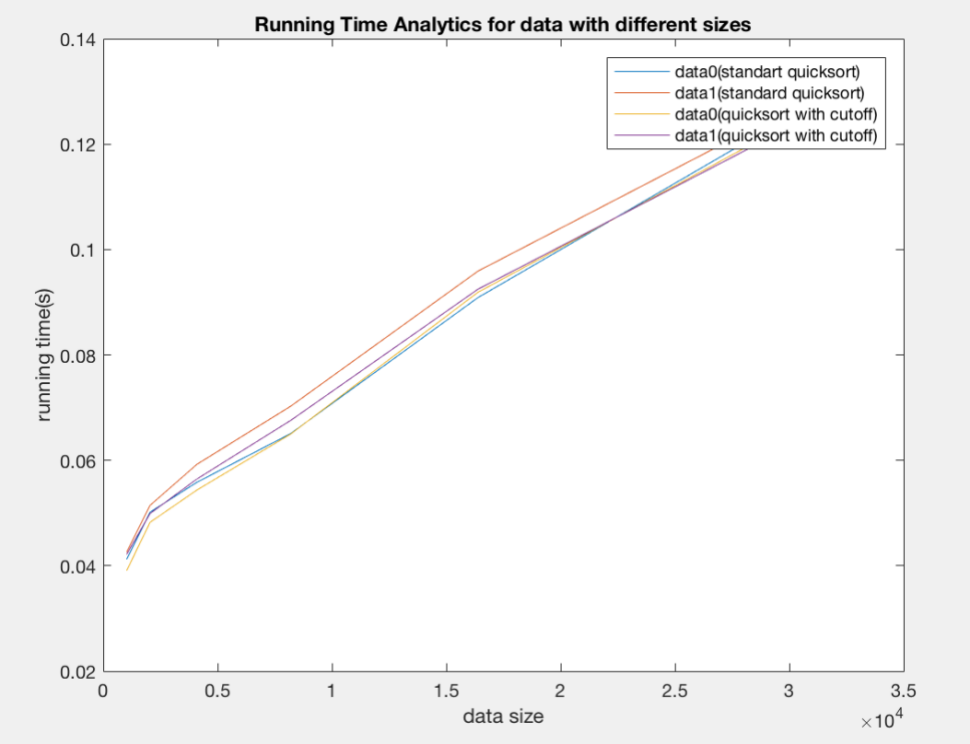
Q5:

- Code: QuickSort.java
- Comparison: shown as the graph



  - Data0 is the best case, data1 is average case
  - The time complexity for MergeSort and QuickSort in best case and average case are all $O(nlogn)$, but in my experiment, the performance of quicksort is a little better than mergesort

- About cufoff:



- We can find that the when we use $N = 7$ as the cutoff, the performance of quicksort is improved, but the change is not big and can be negligible.
- After massive experiments, I find that if the cutoff is no smaller than 8, the performance is worse than standard quicksort. But if the cutoff is no bigger than 7, the performance is better than standard quicksort. Therefore, the critical value of cutoff is 8.

Q6:

No.	Algorithm	Reanson
1	MergeSort (bottom-up)	The second half of the sequence has not been sorted but the first half has been partially sorted and the order satisfies the merge process of mergesort.
2	QuickSort (standard)	All strings smaller than “navy” are placed before “navy”, all strings bigger than “navy” are placed after “navy”
3	Knuth Shuffle	The second half has no change but the first half is unsorted randomly.
4	MergeSort (top-down)	The sequence has been merged as size = 4
5	Insertion Sort	The strings before “bark” are all sorted and satisfy the requirements of insertion sort.
6	HeapSort	It is obvious that the biggest value “wine” has been at the top of the heap.

7	Selection Sort	Every time the smallest value in the unsorted sequence will be choosed and be placed at the end of sorted sequence.
8	QuickSort (3-way)	All strings smaller than “navy” are placed before “navy”, all strings bigger than “navy” are placed after “navy”, and all strings equal to it are around it.