Yilin Yang

Data Structures HW 1 Report

Q1.

File Name	Shell Sort – Increment 7	Shell Sort – Increment 3	Shell Sort – Increment 1	Shell Sort - Total	Insertion Sort - Total
Data1.1024	37631	42769	81338	161738	264541
Data1.2048	146206	161122	309154	616482	1027236
Data1.4096	596750	644604	1245146	2486500	4183804
Data1.8192	2416547	2544475	4968304	9929326	16928767
Data1.16384	9517307	9887421	19419150	38823878	66641183
Data1.32768	38270741	39315471	77614953	155201165	267933908

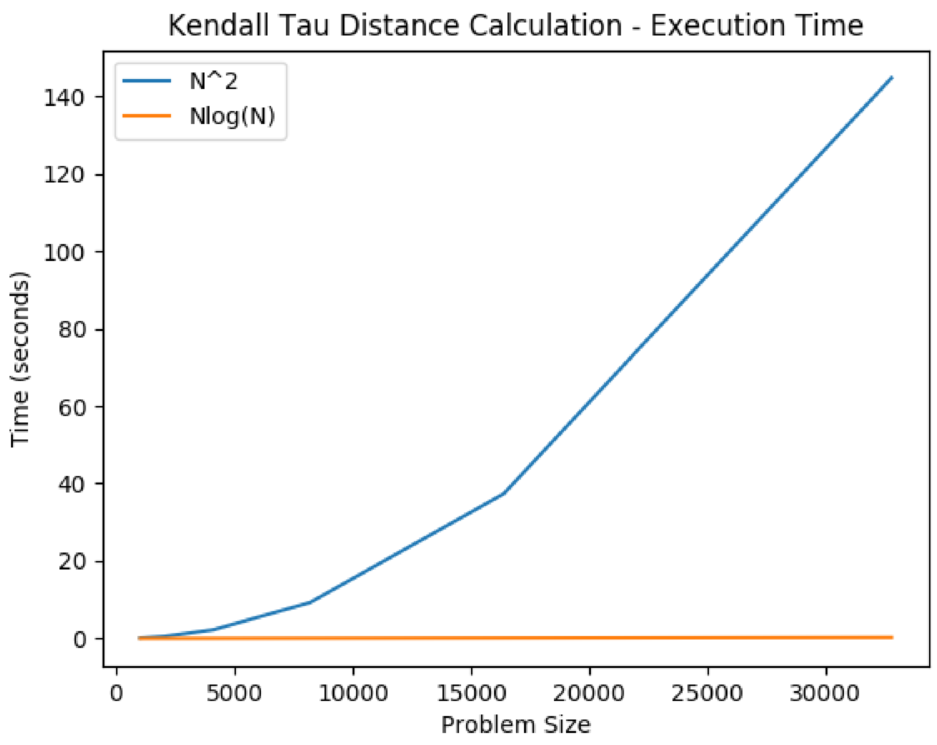
The table above shows the comparisons observed when running shell sort and insertion sort on the data provided. The contents of each data file consisted of randomly listed integers up to a specific power of 2 such as 1024, 2048, 4096, and so on. The output is a list of the integers in ascending order and each algorithm counts the number of comparisons made to determine whether an element in the list needed to be moved to be properly sorted. For the implementation of shell sort, a parameter known as the increment is provided to determine which elements in the list to sort: every 7th, every 3rd, or every single one. Code executed by the two sort algorithms is actually identical with the stipulation that insertion sort always does an increment of one; i.e. every element is inspected in one pass.

Shell sort is distinguished by sorting the data multiple times, each time refining its sort until it is complete. This distinction is responsible for the improvement over insertion sort. Shell sort was observed to make only ~60% of the comparisons needed by insertion sort to fully sort the list. The primary weakness of insertion sort relative to shell sort is that it must shift every element preceding the selected number when moving an element to its properly sorted place. For K elements, relocating the Nth smallest requires relocating K-N elements, which can be an enormous expense given a large problem size. In contrast, shell sort is only forced to disturb 1 other element when sorting, a much smaller cost. Obviously, the final result of shell sort may not be a fully sorted list if not enough iterations are performed, but by presorting the list with every pass, the final iteration, which is functionally equivalent to insertion sort, needs to make relatively few adjustments since multiple elements are already in their correct location.

Q2.

The graph below depicts runtimes for two implementations of Kendall Tau Distance calculation. A brute force method was implemented, calculating distance by iterating through a list of numbers twice; for every element in the list, the method checks every other element in search of inversions that would indicate discordant pairs. As a result, the complexity of brute force is N^2. To improve upon this, a second method was implemented to find Kendall Tau Distance using merge sort.

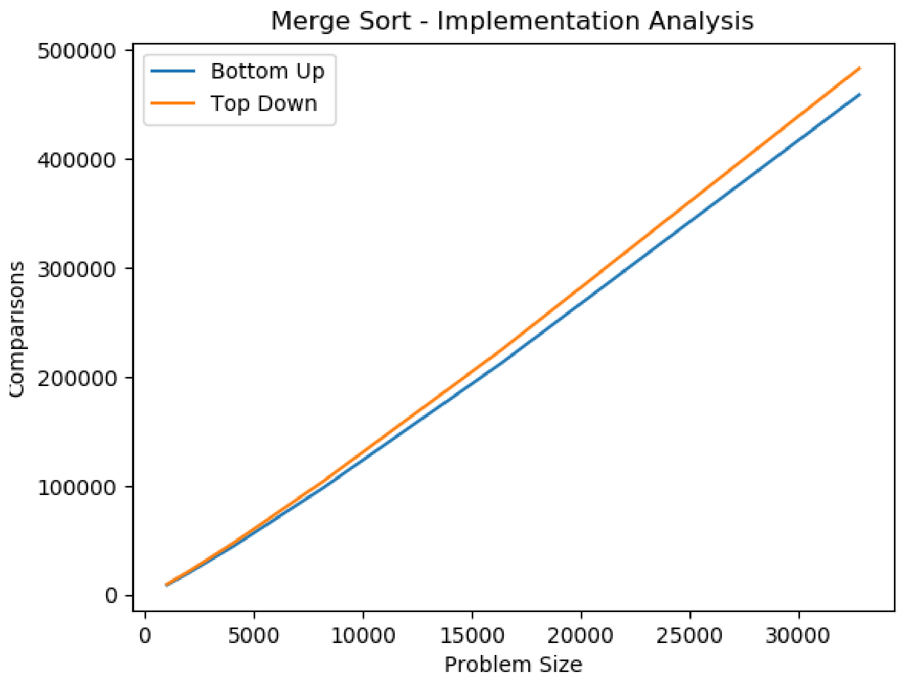
Merge sort recursively sorts the list, breaking the problem size into halves each time. This divide-and-conquer strategy offers a complexity of Nlog(N) which is considerably faster. Inversions can be located when merging the sublists back together, since its known that one list must be smaller (overall) than the other, if the larger list contains an element that contradicts this expectation, an inversion is detected.



Q3.

The data set for this problem has two noticeable characteristics. The first is that it contains multiple duplicates of numbers. The second is that it is already sorted. These traits would encumber several sorting methods, or at the very least make several of their comparisons wasted efforts since less work needs to be done than the algorithm expects. Because of this constraint and with the knowledge of the two characteristics, bucket sort was chosen to best fit this data set. Bucket sort iterates through the list only once and records instances of elements it encounters, filling a "bucket" as it goes. This bucket is implemented as a list. For example, if it encounters the element "11", it accesses index 11 in the bucket list and increments it by one to represent encountering an 11. If it were to encounter another 11, the same bucket would be filled. Other buckets exist for other elements. By making only one pass through the data set, the complexity of bubble sort is N+k where k is the number of buckets. Since we know there are only four possible values to find, the bucket count can be as small as four. With problem sizes of up to 32 thousand, this is trivially small, making the complexity very nearly N. Obviously, this type of performance can only be realized since so much information is already known about the data being sorted, but it is one of the most effective solutions for the given problem.

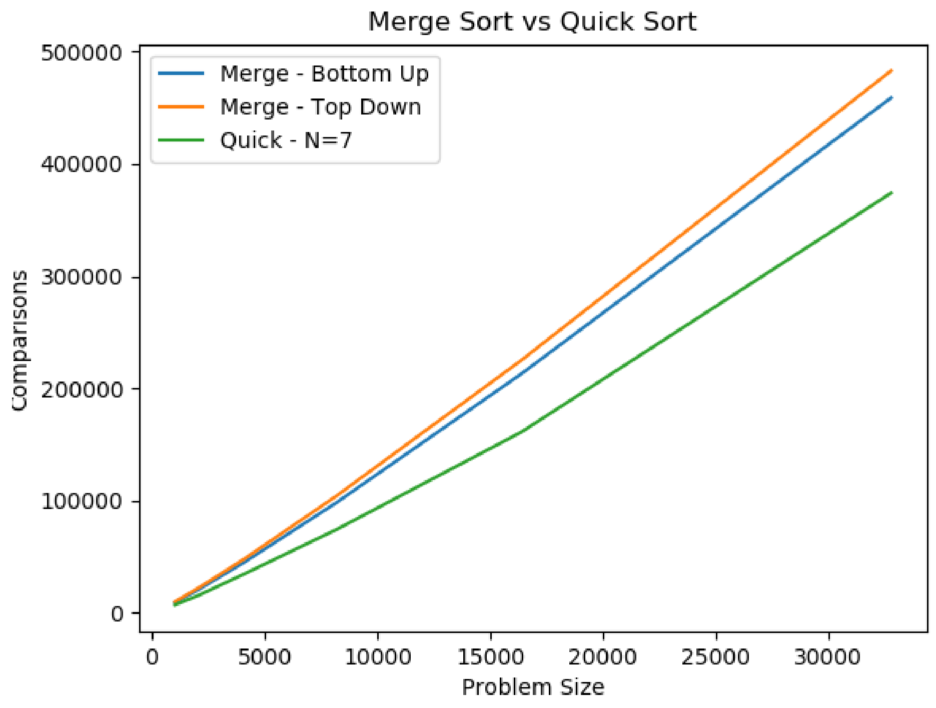
Q4.



Two implementations of merge sort, top-down and bottom-up, were evaluated based on the number of comparisons made when accessing a list of numbers to sort them. As can be seen in the graph, the two approaches are fairly similar, having very little difference for problems sizes less than 4096 and only diverging by a few thousand for problem sizes in the dozens of thousands. Consistently, the bottom-up approach appears to be more efficient in terms of comparisons made. This may be a result of how merge sort recursively breaks down the problem set in the top-down approach. A base case is needed to break recursion, in this case a list size of 1. As a result, top-down is inspecting sublists of size 1, something that bottom-up never does. This amounts to a modest but noticeable difference for larger and larger data sets, such as those exceeding 16 thousand. This difference has very light impact on runtime however, as the two are still highly competitive in such regard. Additionally, top-down approach is generally considered a more intuitive way of communicating and implementing the algorithm. For most purposes top-down is sufficient and easy to use, though bottom-up does offer some incentives for implementation.

Q5.

In addition to merge sort, quick sort was implemented as well with the same data set provided and measured by the same metric, number of comparisons. Like merge sort, quick sort is also recursive, continually breaking down the problem into smaller problems to be sorted. Unlike, merge sort, quick sort is somewhat input sensitive and will acknowledge if the data it receives is already sorted by performing very little work. The key to quick sort is that do work only if it detects a data point is misaligned relative to the pivot point. Because of this, data that is close to being fully sorted, likely after multiple iterations of quick sort, will need very few adjustments. As a result, quick sort does fewer and fewer operations the longer it has been working on a problem. This shows itself in the results graphed below; the improvement of quick sort over merge sort is more dramatic for very large problem sizes.

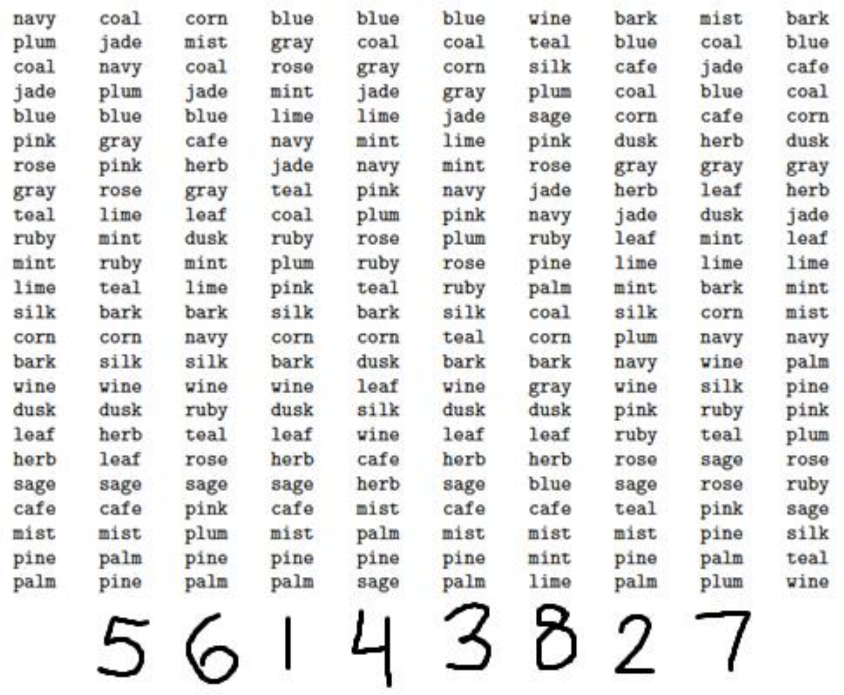


Quick sort is also optimized with the insertion sort cutoff. For smaller problems sizes, it is inefficient to perform quick sort given the scale. When the problem size is found to be smaller than a cutoff value, N, quick sort will instead hand the problem set to insertion sort, which is faster in the given context. N was set to 7 for the results shown above, however, different values were toyed with to observe the impact on number of comparisons made.

Cutoff Value (N)	Comparisons Made on Problem Size 1024
1	7231
3	7134
4	7106
5	7104
7	7184
9	7294
10	7345

The results of the experiments with the cutoff value N are shown in the table above. Note that to collect these results, the random shuffle of the data list, as prefered by quick sort, was temporarily disabled to obtain consistent results. A problem size of 1024 was used to observe how many comparisons quick sort needed with different cutoffs to insertion sort. The results suggest that a cutoff of around 4 or 5 appears to be a sweet spot; there is very little difference between picking either one. A value of 5 is ever so slightly more efficient, though randomization would make this improvement negligible. Performance starts to invert considerably with a cutoff value of 3 however. This is likely the point where quick sort is doing too much work that would be better solved by insertion sort instead. This point is mirrored with a cutoff value of 9, which produced similar results. Most likely, the opposite case is true here; too much work is being done inefficiently by insertion sort when it would be faster to work on it with quick sort.

Q6.



navy	coal	corn	blue	blue	blue	wine	bark	mist	bark
plum	jade	mist	gray	coal	coal	teal	blue	coal	blue
coal	navy	coal	rose	gray	corn	silk	cafe	jade	cafe
jade	plum	jade	mint	jade	gray	plum	coal	blue	coal
blue	blue	blue	lime	lime	jade	sage	corn	cafe	corn
pink	gray	cafe	navy	mint	lime	pink	dusk	herb	dusk
rose	pink	herb	jade	navy	mint	rose	gray	gray	gray
gray	rose	gray	teal	pink	navy	jade	herb	leaf	herb
teal	lime	leaf	coal	plum	pink	navy	jade	dusk	jade
ruby	mint	dusk	ruby	rose	plum	ruby	leaf	mint	leaf
mint	ruby	mint	plum	ruby	rose	pine	lime	lime	lime
lime	teal	lime	pink	teal	ruby	palm	mint	bark	mint
silk	bark	bark	silk	bark	silk	coal	silk	corn	mist
corn	corn	navy	corn	corn	teal	corn	plum	navy	navy
bark	silk	silk	bark	dusk	bark	bark	navy	wine	palm
wine	wine	wine	wine	leaf	wine	gray	wine	silk	pine
dusk	dusk	ruby	dusk	silk	dusk	dusk	pink	ruby	pink
leaf	herb	teal	leaf	wine	leaf	leaf	ruby	teal	plum
herb	leaf	rose	herb	cafe	herb	herb	rose	sage	rose
sage	sage	sage	sage	herb	sage	blue	sage	rose	ruby
cafe	cafe	pink	cafe	mist	cafe	cafe	teal	pink	sage
mist	mist	plum	mist	palm	mist	mist	mist	pine	silk
pine	palm	pine	pine	pine	pine	mint	pine	palm	teal
palm	pine	palm	palm	sage	palm	lime	palm	plum	wine
	5	6	1	4	3	8	2	7	

1. Knuth Shuffle – part of list is randomized, other part is unchanged

2. Selection Sort – the only list where currently sorted values are already in their final place

3. Insertion Sort – part of list is sorted in ascending order, other part is unchanged

4. Merge Sort (top-down) – list is sorted in two halves

5. Merge Sort (bottom-up) – entire list is sorted in groups of two

6. Quick Sort (standard) – everything sorted around a single pivot

7. Quick Sort (3-way) – everything sorted around two pivots

8. Heap Sort – last value is temporarily first