DSA Homework 2

Ruiyu Zhang || rz213

1. Implement Shell Sort which reverts to insertion sort. (Use the increment sequence 7, 3, 1). Create a table or a plot for the total number of comparisons made in the sorting the data for both cases (insertion sort phase and shell sort phase). Explain why Shell Sort is more effective than Insertion sort in this case.

A:

Code is written and tested using python3.5, ".py" file is attached

	data-size	512	1024	2048	4096	8192	16384
Time (ms)	insert	21.124	85.680	334.058	1354.999	5372.858	21759.449
	shell	4.556	17.510	67.523	260.893	1011.824	4059.274
Swaps	insert	64668	265060	1032286	4214951	16686014	67501453
	shell	11576	43692	163474	641751	2516854	10104167
Comps	insert	65114	253900	1061413	4221067	16581008	67032650
	shell	11756	43430	173545	643335	2471730	9942724

```
SHELL SORT
DATA-SIZE   COMPs     SWAPs     TIME
512         11756     13250     0005.42474 ms
1024        43430     46463     0020.17755 ms
2048        173545    179646    0081.11010 ms
4096        643335    655569    0301.39021 ms
8192        2471730   2496243   1144.46290 ms
16384       9941724   9990812   4591.31671 ms

INSERTION SORT
DATA-SIZE   COMPs     SWAPs     TIME
512         65114     65625     0024.55974 ms
1024        253900    254915    0097.63932 ms
2048        1061413   1063455   0410.98945 ms
4096        4221067   4225154   1630.45044 ms
8192        16581008  16589188  6423.59203 ms
16384       67032650  67049020  25874.03023 ms
```

From the result, we can easily tell that shell sort is much faster than insertion sort.
The reason is that shell sort has a worst-case complexity of

$$\mathbf{O(n^{3/2})}$$

for $\mathbf{2^k-1}$ sequence (1,3,7,etc)
While insertion sort has a worst-case complexity of

$$\mathbf{O(n^2)}$$

So as data size grow bigger, insertion sort grows faster in running time. And this is contributed a lot by number of swaps and comparisons performed, which is also included in the statistics above in the form.

2. The Kendall Tau distance is a variant of the "number of inversions" we discussed in class. It is defined as the number of pairs that are in different order in two permutations. Write an efficient program that computes the Kendall Tau distance in less than quadratic time on average. Plot your results and discuss. Use the dataset provided here. Note: data0.* for convenience is an ordered set of numbers (in powers of two). data1.* are shuffled data sets of sizes (as given by "*").

Data Set for Questions above: https://drive.google.com/file/d/0B4xMi5S-VFVRVWh0YzV6bmFLMjQ/view?usp=sharing

A:

Code is written and tested using python3.5, ".py" file is attached.

With inspiration from Prof. William R. Knight's paper "***A Computer Method for Calculating Kendall's Tau with Ungrouped Data***", published 1966.

Data Size	1024	2048	4096	8192	16384	32768
INVs Found	264541	1027236	4183804	16928767	66641183	267933908

```
DATA-SIZE   INVs-FOUND
1024        264541
2048        1027236
4096        4183804
8192        16928767
16384       66641183
32768       267933908
```

This algorithm has a complexity of

O(n*log n)

because it is based on Merge Sort algorithm.

Note: please modify the file path in the code before you do test on your device.

3. Create a data set of 8192 entries which has in the following order: 1024 repeats of 1, 2048 repeats of 11, 4096 repeats of 111 and 1024 repeats of 1111. Write a sort algorithm that you think will sort this set "most" effectively. Explain why you think so.

A:

Code is written and tested using python3.5, ".py" file is attached.
My algorithm has a **time-complexity** of

$$O(n)$$

and it will only scan every element of the data set for **exactly once**. However, it will **skip manipulating** (popping, in fact) element '111', which means it will manipulate the list for only 4096(=8192-4096) times. The **time complexity for item manipulation** is:

$$O(n/2)$$

as a result.

Method:
scan and pop on encountering '1', '11', '1111' (skips '111').
put popped data in temporary list.
put list back together by 'extending' list and put everything back at 'data'.

4. Implement the two versions of Merge Sort that we discussed in class. Create a table or a plot for the total number of comparisons to sort the data (using data set here) for both cases. Explain.

A:

Code is written and tested using python3.5, ".py" file is attached.

DATA-SIZE	ITERATE-COMPs	RECURSE-COMPs
1024	8954	8954
2048	19934	19934
4096	43944	43944
8192	96074	96074
16384	208695	208695
32768	450132	450132

The result shows that whichever way we use to perform merge-sort, the total comparison trials will remain the same.
The reason is simple: the total comparison times depends on the length of the array, regardless of either the way the program runs or whether the array is sorted or not. This example shows us the mutual parts of iteration and recursion.

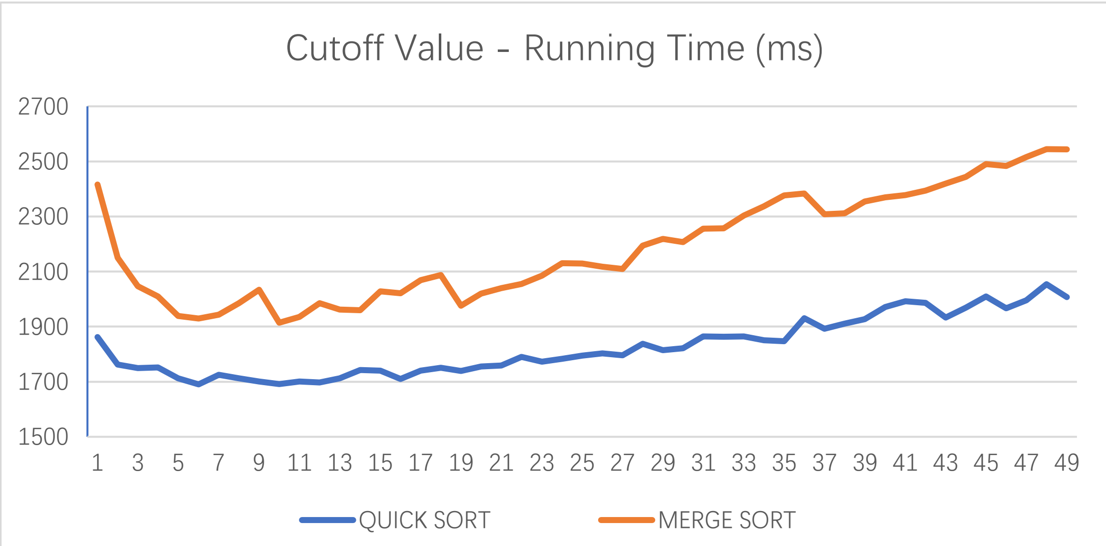
5. Implement Quicksort using median-of-three to determine the partition element. Compare the performance of Quicksort with the Merge Sort implementation and dataset from Q4. Is there any noticeable difference when you use N=7 as the cut-off to insertion sort. Experiment if there is any value of "cut-off to insertion" at which the performance inverts.

A:

Code is written and tested using python3.5, ".py" file is attached.

When N=7, Time for Quicksort is 1724.60 ms and time for Mergesort is 1942.92 ms.

My program runs 49 times in loop, with cutoff values ranging from 1 to 49. The result is plotted below. We can tell that both algorithms show similar trends, and the performance invert appear between N=5 and N=11.



The test is based on sorting a 300,000-membered random list. **Dataset from Q4 is not used here**, because the list is too short to show features of both kinds of sorting algorithms (program completed in microseconds, highly disturbed by compiler).

6. View the following Data Set here. The column on the left is the original input of strings to be sorted or shuffled; the column on the extreme right are the string in sorted order; the other columns are the contents at some intermediate step during one of the 8 algorithms listed below. Match up each algorithm under the corresponding column. Use each algorithm exactly once: (1) Knuth shuffle (2) Selection sort(3) Insertion sort (4) Merge Sort(top-down)(5) Merge Sort (bottom-up) (6) Quicksort (standard, no shuffle) (7) Quicksort (3-way, no shuffle) (8) Heapsort.

[location of data: https://sakai.rutgers.edu/access/content/group/f73f2fd4-280d-4e7c-8cf2-9cc34bcffcff/HW-DataSet/algorithm-stage.png]

A:

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

Column 1: Raw data.

Column 2: Merge sort, bottom to top.

Column 3: Quick sort, 3-way.

Column 4: Knuth shuffle.

Column 5: Merge sort, top to bottom.

Column 6: Insertion sort.

Column 7: Heap sort.

Column 8: Selection sort.

Column 9: Quick sort, standard.