Name: Naga Venkata V Vinnakota

Netid:nvv13

Course: ECE 16:332:573 DSA

**Assignment-2**

Q1. **Implement Shellsort which reverts to insertion sort. (Use the increment  
   sequence 7, 3, 1). Create 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 Shellshort is more effective than Insertion sort in  
   this case. Also, discuss results for the relative (physical wall clock)  
   time taken when using (i) Shellsort that reverts to insertions sort, (ii)  
   Shellsort all the way.**

**Answer:**

**Problem Scenario:**

Develop an implementation for shell sort that reverts to insertion sort. i.e. when the gap is one call Insertion sort.

For the above implementation, the below observations are obtained for Data0 and Data1.

**Data0:**The number of comparisons made for the shell sorting(total) and Insertion phase for data0 are as below :

|  |  |  |
| --- | --- | --- |
| Size | Shell Sort(Total) | Insertion Sort phase |
| 1024 | 3061 | 1023 |
| 2048 | 6133 | 2047 |
| 4096 | 12277 | 4095 |
| 8192 | 24565 | 8191 |
| 16384 | 49141 | 16383 |
| 32768 | 98293 | 32767 |

The above graphs indicate that for a sorted data the number of comparisons made for Shell sort and Insertion sort will be Linear.

**Data1:**

Now let us have the data that is not sorted and see how many comparisons are required for sorting the data for both the Shel sort and Insertion phase of the Shell sort.

The below table shows the number of comparisons made in the sorting (both total and Insertion phase)

|  |  |  |
| --- | --- | --- |
| Size | Shell Sort(Total) | Insertion Sort phase |
| 1024 | 46768 | 1961 |
| 2048 | 169081 | 3873 |
| 4096 | 660673 | 7887 |
| 8192 | 2576322 | 15473 |
| 16384 | 9950984 | 30805 |
| 32768 | 39442505 | 61508 |

From the table and the graphs we can see that the comparisons follow quadratic trend for the entire sorting but it is almost linear for Insertion sort phase i.e. when gap is ‘1’.

The average complexity of Insertion sort is O(N^2), which is very slow for arrays with large size as the number of Inversion might be a high value. This was changed to linear with the help of shell sorting which has a complexity of less than O(N^2) in case of array size 32768.

**Why Shell Sort is effective than Insertion Sort:**

The Shell sort tries to sort the array elements which are known distance apart. This distance known as gap is reduced to a value using a sequence, once the array elements with this gap are sorted. The decrease in this gap is done until the gap is ‘1’ i.e. it reaches Insertion Sorting. The main purpose of this h-sorting or the gap is to reduce the number of inversions. Insertion Sort is the best sorting Algorithm when the number of Inversions is small.

**Comparison of Physical Wall clock for Shell sort that reverts to Insertion and Shell sort all the way:**

I have run the two implementations shell sort that reverts to insertion and shell sort all the way on data0 and data1.

Data0:

|  |  |  |
| --- | --- | --- |
| Size | Shellsort all the way | Reverts to insertion |
| 1024 | 0 | 996900 |
| 2048 | 0 | 4592400 |
| 4096 | 5169500 | 4151400 |
| 8192 | 8144800 | 6762700 |
| 16384 | 17396900 | 30366000 |
| 32768 | 38927800 | 33214500 |

Data1:

|  |  |  |
| --- | --- | --- |
| Size | Shellsort all the way | Reverts to insertion |
| 1024 | 7460500 | 41686400 |
| 2048 | 82984900 | 67412800 |
| 4096 | 333650200 | 213652000 |
| 8192 | 1144792600 | 890690700 |
| 16384 | 4243010500 | 4731353600 |
| 32768 | 13079526700 | 16048733700 |

From the above two tables and plots, we can observe that Shell sort that reverts to Insertion sort takes more time compared to sell sort all the way.

**Conclusion:**

Shell sorting is effective than insertion sorting as it reduces the number of inversions by sub-sorting the array, this leads to an almost Linear Insertion Sort for h=1 i.e gap=1 as seen in the above graphs.

Q2. **The Kendall Tau distance is a variant of the "number of inversions". 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.**

**Answer:**

The number of inversions i.e the Kendall Tau Distance can be determined by the number of swaps a sorting algorithm uses. The number of swaps or exchange of the array elements directly gives the number of inversions present in an array.

Any sorting Algorithm that has a worst complexity of NlgN gives the number of swaps made in less than quadratic time(N^2). One such Algorithm is Merge Sorting.

By using the sorted data, we can verify if the merge sort provides values that are valid for our Analysis.

**Data0:**

Number of pairs of Inversions for data0:

|  |  |
| --- | --- |
| Size | Kendall Distance |
| 1024 | 0 |
| 2048 | 0 |
| 4096 | 0 |
| 8192 | 0 |
| 16384 | 0 |
| 32768 | 0 |

Run time of the Algorithm for data0

|  |  |
| --- | --- |
| Size | Run time |
| 1024 | 7648400 |
| 2048 | 17631200 |
| 4096 | 35451400 |
| 8192 | 84180800 |
| 16384 | 204765800 |
| 32768 | 283124600 |

Here we can see that the runtime of the algorithm to check if there are any inversions is less than a quadratic curve.

With same algorithm, lets find out the number of inversions in the arrays of data set data1 i.e data which is not sorted

Number of Pairs of Inversions for data1

|  |  |
| --- | --- |
| size | Kendall Distance |
| 1024 | 4491 |
| 2048 | 9959 |
| 4096 | 21925 |
| 8192 | 48024 |
| 16384 | 104287 |
| 32768 | 225166 |

The runtime of the Algorithm for data1

|  |  |
| --- | --- |
| size | Run time |
| 1024 | 9018300 |
| 2048 | 19358700 |
| 4096 | 64731900 |
| 8192 | 88348700 |
| 16384 | 274464400 |
| 32768 | 375824100 |

For a data that is not sorted, the above tables and plots indicate that the average complexity of Merge sort is less than quadratic time, O(N^2), and hence It is one of the best sorting algorithm to find the Kendall Tau Distance in less than quadratic time on average.

Q3 **Implement the two versions of MergeSort 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. Discuss (i) relative number of  
   operations, (ii) relative (physical wall clock) time taken.**

**Answer:**

The two versions of Merge sort are

1. Top-Down approach which uses the recursive calls of the sort function
2. Bottom-Up approach which is an iterative implementation of merge sort.

We first use both the implementations on data0 which is already sorted.

The number of comparisons made for both the implementations on same data is as in the table below:

The below table shows the comparisons for data0:

|  |  |  |
| --- | --- | --- |
| Size | Recursive | Iterative |
| 1024 | 5120 | 5120 |
| 2048 | 11264 | 11264 |
| 4096 | 24576 | 24576 |
| 8192 | 53248 | 53248 |
| 16384 | 114688 | 114688 |
| 32768 | 245760 | 245760 |

It can be observed that both the implementations make the same number of comparisons.

The run time of each implementation for data0 is as below:

|  |  |  |
| --- | --- | --- |
| size | Recursive | Iterative |
| 1024 | 6399200 | 10793700 |
| 2048 | 4988600 | 16721400 |
| 4096 | 38327700 | 38955100 |
| 8192 | 128713100 | 60122200 |
| 16384 | 170249000 | 175382600 |
| 32768 | 299261100 | 259784800 |

It can be observed that even though both the implementations take almost same amount of execution time, Iterative implementation has a relatively low run time.

Now we implement both the implementations on data1.

Number of Comparisons made

|  |  |  |
| --- | --- | --- |
| size | Recursive | Iterative |
| 1024 | 8954 | 8954 |
| 2048 | 19934 | 19934 |
| 4096 | 43944 | 43944 |
| 8192 | 96074 | 96074 |
| 16384 | 208695 | 208695 |
| 32768 | 450132 | 450132 |

The run time of each implementation is as below:

|  |  |  |
| --- | --- | --- |
| size | Recursive | Iterative |
| 1024 | 14424900 | 15993900 |
| 2048 | 15511600 | 16039900 |
| 4096 | 68658900 | 65167700 |
| 8192 | 96882700 | 78172800 |
| 16384 | 142641900 | 126951500 |
| 32768 | 380046100 | 370339400 |

The above tables and plots indicate that both the implementations have the same number of comparisons made for a given size.

The runtime of both the algorithms follow the same trend as the size increases but the iterative implementation has relatively less run time.

Hence, it can be concluded that Iterative implementation of merge sort is faster than the recursive implementation as there is no additional calls to the function in iterative implementation.

Q4 **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.**

**Answer:**

The data created is already sorted and no further sorting is required.

However, It will be a healthy practice to check if the data is sorted and try to sort if not. The best approach for this will be the bubble sort algorithm with a check for already sorted array. The algorithm only makes N comparison for an array of size N for the best case i.e. sorted data and N^2 comparisons for worst case.

The main reason of selecting bubble sort with a check for this problem is it only takes N comparisons for a sorted data i.e. Linear comparisons.

Q5. **Implement Quicksort using median-of-three to determine the partition  
   element. Compare the performance of Quicksort with the Mergesort  
   implementation and dataset from Q3. Is there any noticable 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.**

**Answers:**

The run times of the Quick sort for both the data sets are as below.

Data0:

|  |  |
| --- | --- |
| Size | Runtime |
| 1024 | 2990400 |
| 2048 | 14780900 |
| 4096 | 19995400 |
| 8192 | 48081600 |
| 16384 | 73948200 |
| 32768 | 165932400 |

Data1:

|  |  |
| --- | --- |
| size | Runtime |
| 1024 | 5682500 |
| 2048 | 14561900 |
| 4096 | 33201100 |
| 8192 | 45684600 |
| 16384 | 127311500 |
| 32768 | 182118500 |

**Merge vs Quick Sort:**

Let us compare the runtimes in nano seconds for both the sorts on both the data sets. I have taken the runtime of iterative merge sort from the question 3.

**Data0:**

For Data 0, we can observe that Quick sort has a relatively less runtime compared to Merge sort.

|  |  |  |
| --- | --- | --- |
| Size | Quick Sort | Merge |
| 1024 | 2990400 | 10793700 |
| 2048 | 14780900 | 16721400 |
| 4096 | 19995400 | 38955100 |
| 8192 | 48081600 | 60122200 |
| 16384 | 73948200 | 175382600 |
| 32768 | 165932400 | 259784800 |

**Data1:**

For Data 1, we can observe that Quick sort has a relatively less runtime compared to Merge sort.

|  |  |  |
| --- | --- | --- |
| Size | Quick Sort | Merge |
| 1024 | 5682500 | 15993900 |
| 2048 | 14561900 | 16039900 |
| 4096 | 33201100 | 65167700 |
| 8192 | 45684600 | 78172800 |
| 16384 | 127311500 | 126951500 |
| 32768 | 182118500 | 370339400 |

From the above observations above, We can observe that Quick sort takes comparatively less time than the Merge sort (Here I have taken the case of bottom up approach, as it has less run time than the recursive merge).

**When N=7 is used as a cutoff to Insertion sort:**

Data0:

|  |  |  |
| --- | --- | --- |
| Size | Quick sort | Quick Insertion with cutoff=7 |
| 1024 | 2990400 | 2140800 |
| 2048 | 14780900 | 6212300 |
| 4096 | 19995400 | 21637700 |
| 8192 | 48081600 | 36225700 |
| 16384 | 73948200 | 120353200 |
| 32768 | 165932400 | 206092900 |

Data1:

|  |  |  |
| --- | --- | --- |
| size | Quick sort | Quick Insertion with cutoff=7 |
| 1024 | 5682500 | 5642400 |
| 2048 | 14561900 | 12698700 |
| 4096 | 33201100 | 27875500 |
| 8192 | 45684600 | 62812200 |
| 16384 | 127311500 | 148559900 |
| 32768 | 182118500 | 310754100 |

From the above two graphs we can observe that there is a noticeable difference when we use N=7 as cutoff to Insertion sort.

We can observe that the runtime is almost linear from 16384 for both the data.

**Experiment if a cutoff value causes inversion of performance:**

In order to find out a value of cut-off which inverts the performance, I have ran the function with cut-off ranging from 10 to 1010 by incrementing 100 each time. The data set 32768 from data0 and data1 was used for the purpose.

|  |  |  |
| --- | --- | --- |
| Cutoff | Data0 | Data1 |
| 10 | 264645500 | 203358300 |
| 110 | 214008900 | 273494900 |
| 210 | 212692300 | 249629900 |
| 310 | 206382900 | 307378900 |
| 410 | 177504400 | 336424900 |
| 510 | 216884600 | 216756000 |
| 610 | 203680300 | 398202000 |
| 710 | 263414100 | 319883500 |
| 810 | 295053200 | 238059400 |
| 910 | 327787800 | 207799100 |
| 1010 | 216300400 | 372774100 |

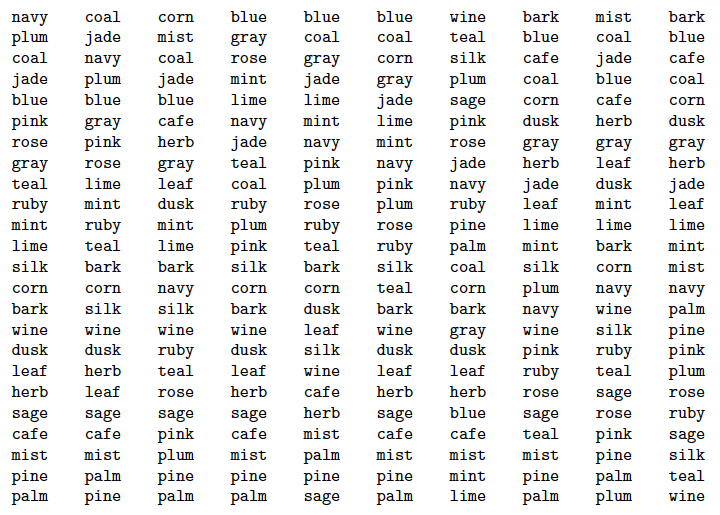
From the above graphs, we can observe that there is no value of cut-off to insertion which inverts the performance.

Q6.

**Extra Points: 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) Mergesort(top-down)(5) Mergesort (bottom-up) (6) Quicksort  
   (standard, no shuffle) (7) Quicksort (3-way, no shuffle) (8) Heapsort.**

**Answer:**

**Original 5 6 1 4 3 8 2 7 Sorted**



|| || || || || || || || ||

Original merge-TD quick K. shuffle Merge Insertion heap selection quick 3-way