Algorithms - HW 2

I. Names of team members:

1. Shashank Shekhar (Unity Id: sshekha4)
2. Rahul Ravindra(Unity Id: rravind)
3. Akshay Podila (Unity Id: apodila)

II. Implementation:

Code is implemented in C. The source code is obtained from GeeksforGeeks and modified as per requirement.

Bubble Sort: https://www.geeksforgeeks.org/bubble-sort/

Merge Sort: https://www.geeksforgeeks.org/merge-sort/

Quick Sort: https://www.geeksforgeeks.org/quick-sort/

III. Code and Data Repo: https://github.ncsu.edu/sshekha4/HW2

IV. Data Collection and Presentation

1. Data Table of User Process Times

For the below table, Mean Sort times of 5 different readings have been taken for files upto 50K in size. For larger files, only 1 sort reading is taken. For the data load times, average of 5 different readings for a file size across different sorts is taken.

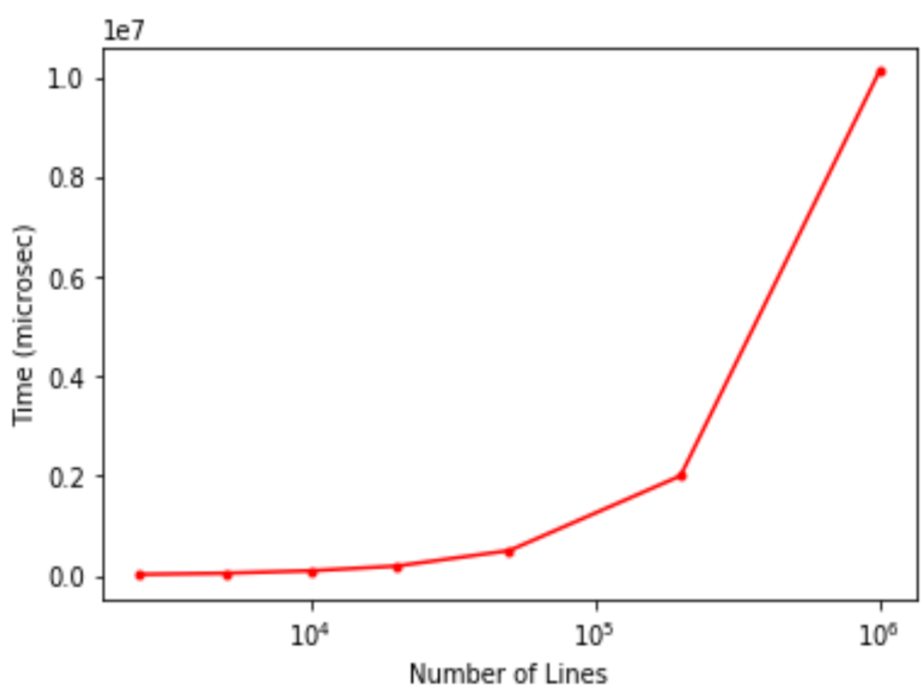
File Name	No. of Input Lines	Data Load Time (µs)	Sort Time (µs)	Total Time (µs)
syslog2500.log_bubble	2500	$0.01 * 10^6$	$0.16 * 10^6$	$0.17 * 10^6$
syslog2500.log_merge	2500	0	$0.01 * 10^6$	$0.01 * 10^6$
syslog2500.log_quick	2500	$0.01 * 10^6$	0	$0.01 * 10^6$
syslog5000.log_bubble	5000	$0.02 * 10^6$	$0.69 * 10^6$	$0.71 * 10^6$
syslog5000.log_merge	5000	$0.03 * 10^6$	0	$0.03 * 10^6$
syslog5000.log_quick	5000	$0.02 * 10^6$	$0.01 * 10^6$	$0.03 * 10^6$
syslog10k.log_bubble	10000	$0.06 * 10^6$	$2.97 * 10^6$	$3.03 * 10^6$
syslog10k.log_merge	10000	$0.06 * 10^6$	$0.02 * 10^6$	$0.08 * 10^6$
syslog10k.log_quick	10000	$0.06 * 10^6$	$0.01 * 10^6$	$0.07 * 10^6$
syslog20k.log_bubble	20000	$0.18 * 10^6$	$11.95 * 10^6$	$12.13 * 10^6$
syslog20k.log_merge	20000	$0.12 * 10^6$	$0.03 * 10^6$	$0.15 * 10^6$
syslog20k.log_quick	20000	$0.13 * 10^6$	$0.02 * 10^6$	$0.15 * 10^6$
syslog50k.log_bubble	50000	$0.34 * 10^6$	$75.90 * 10^6$	$76.24 * 10^6$
syslog50k.log_merge	50000	$0.31 * 10^6$	$0.11 * 10^6$	$0.42 * 10^6$
syslog50k.log_quick	50000	$0.31 * 10^6$	$0.06 * 10^6$	$0.37 * 10^6$

syslog200k.log_bubble	200000	$1.40 * 10^6$	$2271.257 * 10^6$	$2272.657 * 10^6$
syslog200k.log_merge	200000	$1.38 * 10^6$	$0.52 * 10^6$	$1.90 * 10^6$
syslog200k.log_quick	200000	$1.33 * 10^6$	$0.28 * 10^6$	$1.61 * 10^6$
syslog1Ma.log_bubble	1000000	$6.78 * 10^6$	$67804.474 * 10^6$	Unknown
syslog1Ma.log_merge	1000000	$6.74 * 10^6$	$2.94 * 10^6$	$9.68 * 10^6$
syslog1Ma.log_quick	1000000	$6.73 * 10^6$	$17000.236 * 10^6$	$17006.966 * 10^6$
syslog1Mb.log_bubble	1000000	$6.89 * 10^6$	$57438.867 * 10^6$	$57445.757 * 10^6$
syslog1Mb.log_merge	1000000	$6.84 * 10^6$	$3.03 * 10^6$	$9.87 * 10^6$
syslog1Mb.log_quick	1000000	$6.94 * 10^6$	$1.78 * 10^6$	$8.72 * 10^6$
syslog1Mc.log_bubble	1000000	$6.87 * 10^6$	$12378.033 * 10^6$	$12384.903 * 10^6$
syslog1Mc.log_merge	1000000	$6.83 * 10^6$	$2.92 * 10^6$	$9.75 * 10^6$
syslog1Mc.log_quick	1000000	$6.85 * 10^6$	$1594.206 * 10^6$	$1601.056 * 10^6$

2. Plot of Data Load Time vs Data Size (number of lines)

   Since the x-axis had values that were very close to each other, x-axis is log scaled.

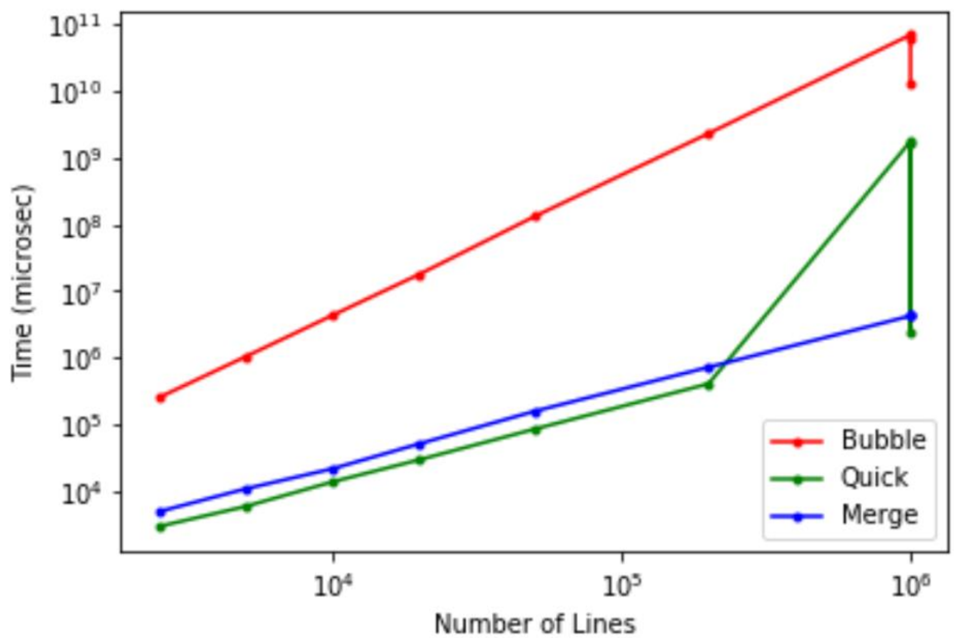
   The average data load times for a filesize across different sorts (bubblesort, quicksort, mergesort) for 5 different runs is taken.



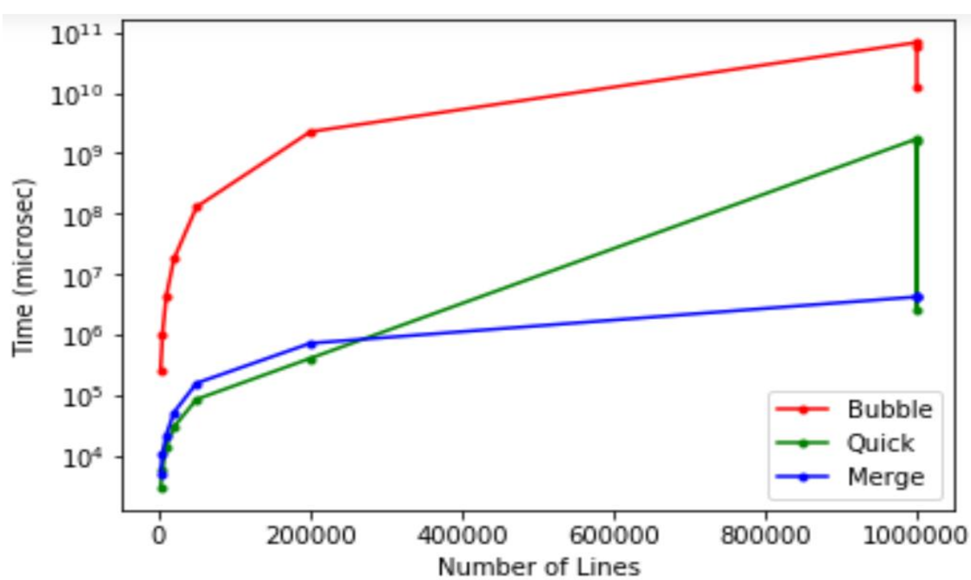
3. Plot of Sort Time vs Data Size (number of lines)

   The average sort times for 5 runs for file sizes upto 50K is taken. For 200K and 1M file sizes, only a single sort run is performed.

   For the below graph, both the x-axis and y-axis are log scaled.



   For the below graph, y-axis is log scaled.



4. Answers:

No. of times sort was executed before timing data was recorded	5
No. of executions performed	5
Whether the highest and lowest times were dropped	Lowest time dropped. Highest time almost constant.
Whether mean or median time is the one reported in the data table	Mean time is reported in the data table
Operating System	Red Hat Enterprise Linux Version: cpe:/o:redhat:enterprise_linux:7.7:GA:server
CPU Type and Speed	Intel(R) Xeon(R) CPU E7-8867 v3 @ 2.50GHz Cache Size: 46080 KB
Type and Size of Disk holding the data	512 GB of storage and type is Hard Disk Drive storage

V. Analysis Questions:

1. Data Structure used to hold the data in memory: Array
   It is a collection of items stored at contiguous memory locations where elements can be accessed randomly using indices of an array. Data type of all the elements must be the same.
2. Comparison function used:

```
// Comparator for the sort function

int cmpfunc (const void * a, const void * b) {
  return (((record*)a)->t - ((record*)b)->t);
}

// Structure of each record
typedef struct trec{
        time_t t;
        char rec[512];
}record;
```

   In the comparator function, the time values are compared. If the timestamp of the record on the left side is larger than the timestamp of the record on the right side, the records are swapped.
3. For the Bubble sort, the order of growth was expected to be $O(n^2)$. The order of growth observed closely resembled the curve $y=x^2$. Hence, the expected and the observed order of growth are similar. For syslog500k.log, syslog1Ma.log,

syslog1Mb.log and syslog1Mc.log, the algorithm did not finish as expected since it would take several days for the algorithms to finish in these cases.

For the Merge sort, the order of growth was expected to be O(nlogn). The oder of growth observed closely resembled the curve y=xlogx. Hence, the expected and observed order of growth are similar. The algorithm completed for all the syslog files.

For the Quick sort, the order of growth was expected to be $O(n^2)$ [worst case performance] and nlogn average case performance. The running time for syslog1Ma.log and syslog1Mc.log files could not be measured indicating that the quicksort probably hit its worstcase scenario in these cases (n^2 complexity). The running time for all the other files measured shows a y=xlogx curve indicating that the average performance was nlogn.

4. Both the load and sort times are comparable for mergesort and quicksort for smaller files. Load time for smaller files in case of bubble sort is less compared to the sort time.

   For larger files, data load time is significantly larger than the sort time for mergesort and quicksort (considering average case performance). Data load time for larger files in case of bubble sort is significantly less compared to the sort time which reaches large values for big files considering its $O(n^2)$ performance.
5. Three data files contained the same number of input lines (1 million). Yes, there were differences observed wrt. Quicksort for the 3 files which ran poorly for syslog1Ma.log and syslog1Mc.log files ($O(n^2)$ behavior) and ran syslog1Mb.log with its average case performance (nlogn behavior). For MergeSort, the performance was similar across all the 3 files. For bubble sort, the time to run the algorithm could not be measured since it would take several days to complete the execution given that it has $O(n^2)$ performance.
6. We start from the leftmost element and keep track of index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap current element with arr[i]. Otherwise we ignore current element.
7. In-place Sorting Algorithms: Bubble Sort and QuickSort

   Additional Memory Needed: Merge Sort
8. Bubble Sort is implemented iteratively. MergeSort and QuickSort are implemented recursively.
9. Bubble Sort, Merge Sort are stable sorts. The quicksort implementation is not a stable sort.
10. Larger Data that cannot fit into memory
    i. I would use Merge Sort to implement a solution.
    ii. Since the whole data cannot be loaded into memory, divide and conquer strategy would be effectively used here. We can perform Multistep Merge, where the entire data is broken into several runs. Each run is loaded into

memory and sorted. Then the sorted runs are written back to disk. Then there is a final merge step performed on all of these runs.

iii. Assuming that 10^8 lines can fit in memory at once, we will divide the whole data into 10^6 groups of 10^8 size each. Each of these run lists will then be sorted and written back to the disk. Since sorting 10^6 lines takes approximately 3 seconds by merge sort and merge sort is an nlogn sort where n is the number of lines, we essentially perform $20*10^6$ operations in total. So, to sort 10^8 lines, we will have to perform $27*10^8$ operations which will take 405 seconds to complete. Like this, we need to sort 10^6 different lists. Therefore, the total time to sort these lists would be **405 * 10^6 seconds**. Assuming sequential disk access speed is 127 MB/sec and 132 bytes/line on average, each list has 10^8 lines and hence the total size of each list is 10^8 * 132 bytes = 13200 MB. Therefore, to read a list to memory will take approximately 104 sec. So, to read 10^6 such lists would take **104 * 10^6 seconds.** Assuming disk write speed to be same as disk access speed of 127 MB/sec, to write each of 10^6 lists would take around **104 * 10^6 seconds**. Assuming random disk seek has the same speed as the sequential disk seek of 127 MB/sec, a total of **103937007 sec** will be required to read 132 * 10^{14} bytes (ie. 10^{14} lines each having 132 bytes / characters) = 132 * 10^8 MB of data for the purpose of sorting. This calculation makes an assumption that there is a very large input and output buffer with negligible time to read and write data from these buffers into main memory. As calculated before, 3 seconds are required to perform $2*10^6$ operations. Hence, to perform approximately $47*10^{14}$ operations, approximately **$71*10^8$ seconds** are required. Adding all the above time calculations, we get (103937007 + 7100000000 + (2*104000000) + 405000000) seconds = **7816937007 seconds = approximately 248 years.**