CCDSALG Term 3, AY 2020 – 2021

Project 1 – Comparing Sorting Algorithms

Section	Names	Task 1	Task 2	Task 3	Task 4
S13	Tolentino, Enrico	X	X		X
	Cruz, Airon John	X	X	X	X
	Hernandez, Pierre Vincent	X	X		X

Fill this part with your section and names. For the tasks, put an X mark if you have performed the specified task. Please refer to the project specifications for the tasks.

LIST OF SORTING ALGORITHMS

Sorting Algorithm	Author (if available)	Downloaded From
1. Bubble sort	Obtained from GeeksforGeeks	https://www.geeksforgeeks.org/bubble-sort/
2. Insertion sort	Obtained from Faizan Parvez, Great Learning	https://www.mygreatlearning.com/blog/insertion-sort-algorithm/
3. Selection sort	Obtained from GeeksForGeeks	https://www.geeksforgeeks.org/selection-sort/
4. Merge sort	Obtained from Prof. Arren Antioquia's CCDSALG course slides (T3 2020-2021)	
5. Radix sort	Obtained from Austin G. Walters	https://austingwalters.com/radix-sort-in-c/

6. Gnome Sort	Obtained from GeeksforGeeks, improved by Sam007 and splevel62	https://www.geeksforgeeks.org/gnome-sort-a-stupid-one/

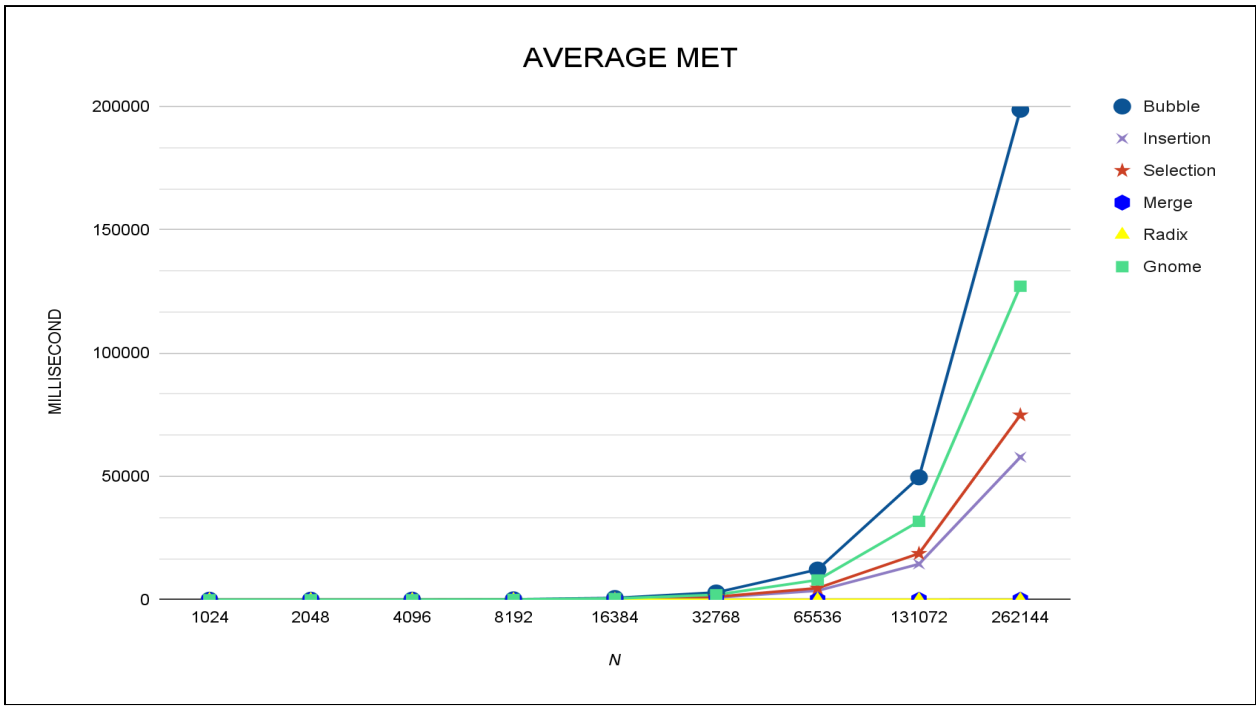
COMPARISON TABLE

M = (10)

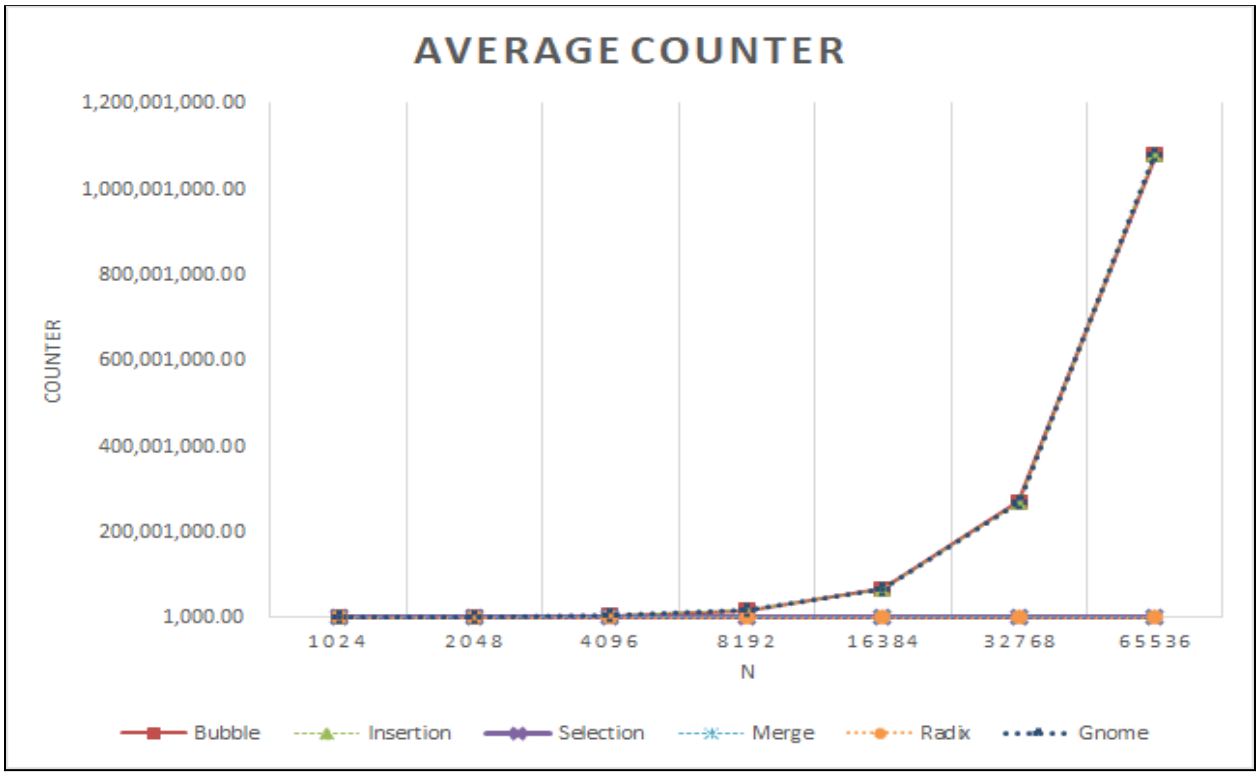
Size	Average Machine Execution Time (milliseconds)					
	Bubble	**Insertion**	**Selection**	**Merge**	**Radix**	**Gnome**
1024	**3.125000**	**1.562500**	**0.000000**	**0.000000**	**0.000000**	**3.125000**
2048	**7.812500**	**3.125000**	**4.687500**	**1.562500**	**0.000000**	**6.250000**
4096	**31.250000**	**15.625000**	**17.187500**	**1.562500**	**0.000000**	**31.250000**
8192	**160.937500**	**56.250000**	**79.687500**	**0.000000**	**1.562500**	**128.125000**
16384	**701.562500**	**226.562500**	**295.312500**	**3.125000**	**6.250000**	**493.750000**
32768	**2929.687500**	**912.500000**	**1178.125000**	**3.125000**	**1.562500**	**2003.125000**
65536	**12232.812500**	**3645.312500**	**4693.750000**	**7.812500**	**7.812500**	**7993.750000**
131072	**49615.625000**	**14465.625000**	**18737.500000**	**23.437500**	**9.375000**	**31789.062500**
262144	**198659.375000**	**57807.812500**	**74873.437500**	**46.875000**	**20.312500**	**127121.875000**

Size	Average Counter Value					
	Bubble	Insertion	Selection	Merge	Radix	Gnome
1024	257952	257952	5352	8941	4096	257952
2048	1043653	1043653	11971	19922	8192	1043653
4096	4151843	4151843	26874	43915	16384	4151843
8192	16624111	16624111	61917	96090	32768	16624111
16384	67020005	67020005	132858	208675	81920	67020005
32768	270269519	270269519	285128	449933	163840	270269519
65536	1079479994	1079479994	604813	965770	327680	1079479994
131072	-2147483648	-2147483648	1262227	2062600	655360	-2147483648
262144	-2147483648	-2147483648	2560369	4387183	1310720	-2147483648

GRAPHS



Graph 1. Average Machine Execution Time in milliseconds



Graph 2. Average Counter Value

DISCUSSION

I. The machine and its results

Airon's machine was used for the testing phase for it has a decent amount of memory that is eight Gigabytes (8 GB) and an AMD Ryzen 5 3550H that is a 4-core/8-threads APU (CPU with integrated graphics). This is to decrease the margin of error that can be obtained when using a slower and memory-restricted machine which could bottleneck the performance and runtime of the code.

The program (main.c) was compiled using the TDM-GCC compiler in Dev-C++ and was executed four times in Airon's machine due to the results that we're getting in the average machine execution time (MET). Some sorting algorithms seldom produce and display zero milliseconds (0.000000). An example of this is the Selection, Merge, and Radix sort when $N = 1024$. At first, it was suspected that there was a logical error in the source code. It was checked, traced, and tested a number of times but there were no logical errors found. It was all working as expected, from generating random N values stored in the original array, making a copy of the array, number of times it runs, the sorting algorithms that were called, resetting of values, copying the copy of unsorted array before calling another sorting algorithm and the memory allocation for the arrays.

The group formulated a hypothesis for these results. It is possible that the generated values might be among the following or all of it: the generated array for the current N value is almost sorted, there are values that are generated multiple times, or the highest place value of the highest integer value is until the hundreds or tens or ones place only (for the case of the Radix sort), so it is possible that the machine was able to execute it so fast. Since the program displays the average MET up to 6 decimal places, it is also possible that the average MET is very small, especially the sorting algorithms that displayed a zero millisecond result.

II. Machine Execution Time

The researchers have followed the advised n-value scale for this study, wherein 1,024 being the base value and 262,144 is the ceiling(see basis in Integer overflow). The results in this study are already the computed average Machine Execution Time (MET) from the 10 runs of each sorting algorithm using DevC as the researchers' chosen IDE. Based on the tests conducted by the researchers, results have shown that the Radix sorting algorithm has the lowest average MET for every increase in array size for only having 20.3125ms as its maximum average MET. Consequently, Bubble sort has been observed to list the highest MET for each level of size which peaks at 198,659.375ms at the maximum n-value. Under it is the Gnome Sorting algorithm which also resulted in 127,121.875ms as its average MET at the highest possible n-value.

Moreover, both Insertion and Selection sort algorithms which use non-recursive techniques are found to have close MET results from size values of 1,024 up until 8,192. Afterward, the Selection sort now seemed to depart from its head-to-head state against Insertion sort wherein there is already a visible increase in its MET as it steps on high array size(n) which are values greater than 32,768. On the other hand, Merge sort which is known to be an "out place" sort (requiring separate memory space for it to operate) was still able to list relatively low average MET wherein it only had a maximum MET of 46.875ms at 262,144 n-value during the multiple runs of varying array sizes**.**

Graph 1 shows the trend changes of all the seven sorting algorithms involved in this study concerning the increasing size of the array. In line with this, the researchers observed that Bubble sort accompanied by the second to the highest average MET, Gnome sort, had a drastic increase on their

tallied average MET during the 65,000 plus mark of variable n which curves their respective lines upward. However, other sorting algorithms such as the Selection and Insertion still showed an almost identical significant increase on their average MET at 260,000 plus mark of the variable n, respectively. Lastly, Merge sort alongside with Radix sort seemed to produce a horizontal line that shows both of their consistent low average MET from the previous runs.

III. Counter variable

The researchers thoroughly discussed the critical part of each sorting algorithms' code where the counter would increase. For Bubble Sort, two options were argued over at first, the number of times each value was checked and the number of times swaps between two values within the array. It was then decided that the swapping would be the critical part of the code as it is considered to be the part of the code with the highest degree. This logic is then applied when deciding the critical parts of each sorting algorithm. For Insertion Sort, the researchers have agreed to put the counter variable inside the while loop of the function—vital part. Specifically, that portion of the code is where the values are filtered and inserted into the array of sorted values.

On the other hand, the researchers decided to put the Selection sort counter variable inside the if-statement that checks for a value if it is less than the current minimum number within the unsorted array. The researchers have tested the algorithm to support the earlier decision; furthermore, they have found out that this particular part is where the counter variable fluctuates depending on the arrangement of unsorted numbers —sufficed by the random number generator. For Merge Sort, it was decided that the number of comparisons when merging each data would be the critical part of the code, but the researchers initially had two options, with the other one being the number of merges of each half. The former was decided as, in line with the logic being applied for deciding the critical parts, it had the highest degree.

For Radix Sort, it was relatively more difficult in deciding the critical part of the code, unlike the other sorting algorithms as it did not rely on comparisons. The other initial options for the critical part of the code are the number of digits with the highest value, the number of times the bucket is used, and the number of times each individual value is placed inside the bucket. The buckets are where the index of the sorted significant digits are stored and significant digits are powers of 10 denoting the degree of digits. Eventually, it was decided that the number of times a value is placed in the bucket would be the critical part of the code as it had the highest degree among all the other options. Lastly for Gnome Sort, given that it was very similar to bubble sort, it was decided the swaps too were the critical part of the code as the only significant difference of note is the movement of comparing pairs of numbers in the given array.

Based on the results, it can be observed that the Bubble, Insertion, and Gnome sort have the same counter value per set of N values. The swaps or inserts of these three sorting algorithms have the same behavior but different implementations. As for the Selection sort, it has the lowest counter value among other sorting algorithms due to the placement where the counter variable increments by one. It is possible that the counter of the said sort could have the same behavior as the first three sorts if the placement of the counter variable is on the swap method since it swaps every after checking the inner loop where the condition of the minimum value is being checked. For the Radix sort, the counter value can easily be calculated by the number of digits of the highest value in the array multiplied by the number of elements that are in the array. Lastly, the Merge sort could possibly be the lowest counter value in the test if only the placement of the counter variable in the selection sort is placed or incremented after every swap.

IV. Integer Overflow

The counters for each sorting algorithm worked as expected. However, starting in N = 131,072, the counters of the sorting algorithms with $O(n^2)$ growth rate such as Bubble Sort, Insertion Sort, and Gnome Sort grew too large that it had an integer overflow, causing the counters to wraparound, resulting in the -2,147,483,648 output. Since the integer data type is only 4 bytes, it can only store values from -2,147,483,648 to 2,147,483,647 range. As such, upon reaching 2,147,483,647, the next integer value would be -2,147,483,648 and then -2,147,483,647 when continuously incrementing by one. It is suspected that this occurrence is not limited to the sorting algorithms with the said rate of growth. At some point, the other sorting algorithms are bound to run into this problem too as the value of N grows larger.

V. Replit as Secondary Source

The program for this project was also compiled using the C-lang compiler in Repl.it and was executed to serve as a secondary source. The behavior of each sorting algorithm was based on its computed average MET in every *N* value. The compiler, processor, and memory size of the server were considered when this was done on the said website. It is possible that the server that the program is running to may possibly have a 32- or 64-core processor and a hundred Gigabyte memory. Additionally, it is possible that it's executing multiple programs at once from a wide variety of users around the globe.

The program exited after executing the program at *N* = 32768, so the binary flip that occurred in Airon's machine after this *N* value was not observed. The average MET of the Merge and Radix sorting algorithm still displays a decimal value while on Airon's machine it displayed zero milliseconds in *N* = 1024. Since it was the same code that was compiled and ran on both machines, it means that the program is working as expected it should be and it could support the hypotheses stated regarding the few of the sorting algorithms that displayed zero for their average MET.

VI. The Conclusion

Based on the gathered results that are also represented on the graphs, it is safe to conclude that when there is *N* number of data where *N* <= 16,384, there are no notable significant differences with the speed of each sorting algorithm. But as it increases from that range, the differences between the speed of each sorting algorithm also increases. Though the usage and the amount of memory being used by the sorting algorithm may vary. Some sorting algorithms are in-place such as the Bubble sort, Insert Sort, Selection Sort, and Gnome sort while others require an extra array to operate such as the Merge sort and Radix sort. These in-place sorting algorithms may not take up that much memory but since it's a comparative type of sorting algorithm, it may take some time to compare each data of an array. Though Merge sort is also a comparative type of sorting and it uses the divide-and-conquer method. It's faster and has fewer comparisons though it may take up some memory for the extra arrays that it's creating and using. Radix sort also uses an extra array but its difference is that it's a distributive type of sorting which saves up some computing processes for the machine. The downside to this sorting algorithm is that it only works on numerical values, so sorting other values like string values may need to use either one of the other sorting algorithms.