Georgios Ioannou

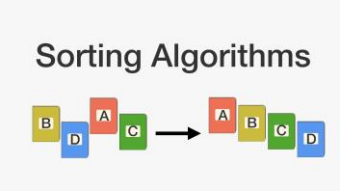
Professor Ahmet C. Yuksel

CSC 22000

27 July 2021

Revealing the mystery of the Sorting Algorithms:

Reporting and analyzing the compiler time of six Sorting Algorithms



Georgios Ioannou

Algorithms

July 27, 2021

Abstract:

In this programming assignment, I investigated the compiler time of six different Sorting Algorithms for different input sizes. The six Sorting Algorithms analyzed in this report are: Insertion Sort, Merge Sort, Heap Sort, Quick Sort, Randomized Quick Sort, and LSD Radix Sort. A code that was built from scratch was used to generate random arrays of different sizes from 10 to 1000000 and excessively test each Sorting Algorithm. The code is based on the pseudocodes found in the textbook: "Introduction to Algorithms, THIRD EDITION" by Cormen, Leiserson,

Rivest, and Stein and published by The MIT Press. The program was repeated twice and the average compiler time was reported.

Introduction:

This programming assignment focused on the Time Complexity of some Sorting Algorithms. Sorting is extremely critical in many applications and especially in computer science. Sorting is a prerequisite of searching and searching is everything. However, time is money and people want to perform tasks in the least amount of time possible. Therefore, there is a need for good and fast Sorting Algorithms so that tasks can be completed quickly. Sorting is an extremely well-solved problem and several computer scientists and mathematicians developed different Sorting Algorithms. These Sorting Algorithms are first compared according to their Time Complexities and then according to other properties such as Space Complexity and stability. Therefore, let us produce a table that lists the Time Complexities of the six Sorting Algorithms studied in this programming assignment to have an idea of what compiler times we expect.

Sorting Algorithm	**Time Complexity**		
	Best Case	**Average Case**	**Worst Case**
Insertion Sort	$\Theta(n)$	$O(n^2)$	$\Theta(n^2)$
Merge Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n\log n)$
Heap Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$O(n\log n)$
Quick Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n^2)$
Randomized Quick Sort	$\Theta(n\log n)$	$\Theta(n\log n)$	$\Theta(n^2)$
LSD Radix Sort	$\Theta(n+k)$	$\Theta(n+k)$	$\Theta(n+k)$

Table 1. Time Complexities of six Sorting Algorithms

My hypothesis was driven by the completion of Table 1 as I knew what to expect from the compiler times. My hypothesis stated that the fastest Algorithm will be LSD Radix Sort as it is a linear Sorting Algorithm and the slowest Algorithm will be Insertion Sort as it is a quadratic Sorting Algorithm. Regarding the other Algorithms, my hypothesis stated that Randomized Quick Sort beats Quick Sort which beats Heap Sort which beats Merge Sort.

Materials and Methods:

To complete this programming assignment I needed:

1. Laptop or Desktop Computer
2. Integrated Development Environment(To be able to write and edit the code)
3. Compiler(To read and compile the code written in C++)
4. Microsoft Excel(To import data and produce tables and graphs)

Difficulties, Implementation issues, and preferences/decisions:

For this programming assignment, I included some libraries so that I can use certain functions. The libraries that I used are:

1. c standard library(Provided me the srand() and rand() functions)
2. ctime library(Provided me the time() function)
3. cmath library(Provided me the floor(), pow(), and log10() functions)
4. limits library(Provided me the std::numeric_limits)
5. chrono library(Provided me the high_resolution clock and duration_cast function)
6. cassert library(Provided me the macro assert function)
7. iostream library(Provided me the cout and endl)

The range of elements that I chose no fill the arrays is [0-1000] and this was because I needed to have both small and large numbers so that I can compare better and simulate real-world scenarios. In addition to this, I chose a huge seed value(1293810238120) so that the pseudo-random number generator can generate completely random elements for the arrays.

For reporting the compiler time, I first chose nanoseconds as my timing unit. However, when it came to sorting the arrays of sizes N = 100000 and N = 1000000 using Insertion Sort, then the time in nanoseconds was huge and the program crashed. Moreover, choosing nanoseconds as the timing unit, makes it difficult for someone to study and compare what is going on because the numbers are extremely large. Therefore, for the above two reasons, I decided to choose microseconds as my timing unit.

However, there is only one exception that is important and needs to be mentioned. Insertion Sort is a quadratic Sorting Algorithm and therefore it does not perform well on large arrays. Therefore, when the array size was N = 1000000 Insertion Sort took 18 minutes to sort the array. 18 minutes is 1080000000 microseconds which made the program crashed. Therefore, I decided to change the time unit for this particular case only from microseconds to minutes.

During the implementation, I did not face many issues as the pseudocodes were straightforward to implement. However, I needed to modify all the pseudocodes to start from index 0 and not from index 1 because the programming language C++ starts from index 0.

Another issue that I faced in the implementation was that I could not declare static arrays because when the size was huge such as in N = 1000000 the static memory crashed. Therefore, I decided to use only dynamic arrays which solved the problem. However, I used the macro assert function to make sure that the allocation of the arrays happened successfully.

Finally, the biggest issue that I faced was with the LSD Radix Sort Algorithm. I needed to make many modifications to the pseudocodes of LSD Radix Sort and Counting Sort. One of these modifications and the hardest part was to compute the highest-order digit d. Another modification that I did and it was also difficult was on extracting the exact digit that I was performing the sorting on. I needed to refer back to my Mathematics textbooks to find out how to compute the number of digits in a number, how to shifts numbers to the right, and how to extract the right-most digit.

One of the preferences that I included in the implementation is in the randomizedPartition() function. In this function, I used the time() function to assign the seed value so that every time I generate a completely random pivot.

In addition to this, I decided to run the program twice and take the average of the two results so that I can get more precise and accurate results.

Results:

The following three tables illustrate the compiler time for each Sorting Algorithm and each input size N. Table 1 is the first trial. Table 2 is the second trial. Table 3 is the average of Table 1 and Table 2. Figure 1 is the graph representation of Table 3.

N	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Randomized Quick Sort	LSD Radix Sort
10	36	42	5	3	33	56
100	16	150	41	21	57	35
1000	1002	1066	610	296	637	273
10000	103442	10968	8457	4158	7429	3281
100000	11561580	125834	131178	59566	106683	36123
1000000	1080000000	1232985	1439518	1975219	2532630	340594

Table 1. Compiler time(microseconds) for each Sorting Algorithm(Trial 1)

Image 1. Output of first trial

N	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Randomized Quick Sort	LSD Radix Sort
10	8	45	4	3	45	53
100	21	153	41	22	79	34
1000	1477	1062	614	317	1086	274
10000	138336	11174	9972	4367	7818	3286
100000	11338951	122752	110026	59295	107777	34542
1000000	1080000000	1287599	1443588	1960633	2518113	341827

Table 2. Compiler time(microseconds) for each Sorting Algorithm(Trial 2)

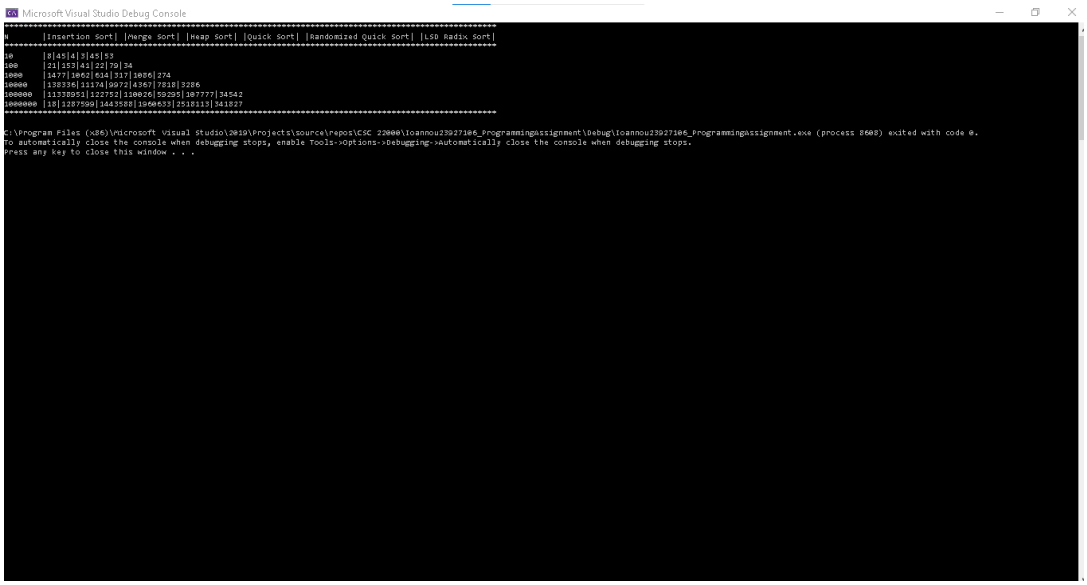


Image 2. Output of second trial

N	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Randomized Quick Sort	LSD Radix Sort
10	22	43.5	4.5	3	39	54.5
100	18.5	151.5	41	21.5	68	34.5
1000	1239.5	1064	612	306.5	861.5	273.5
10000	120889	11071	9214.5	4262.5	7623.5	3283.5
100000	11450265.5	124293	120602	59430.5	107230	35332.5
1000000	1080000000	1260292	1441553	1967926	2525371.5	341210.5

Table 3. Compiler time(microseconds) for each Sorting Algorithm(Average of Table 1 and Table 2)

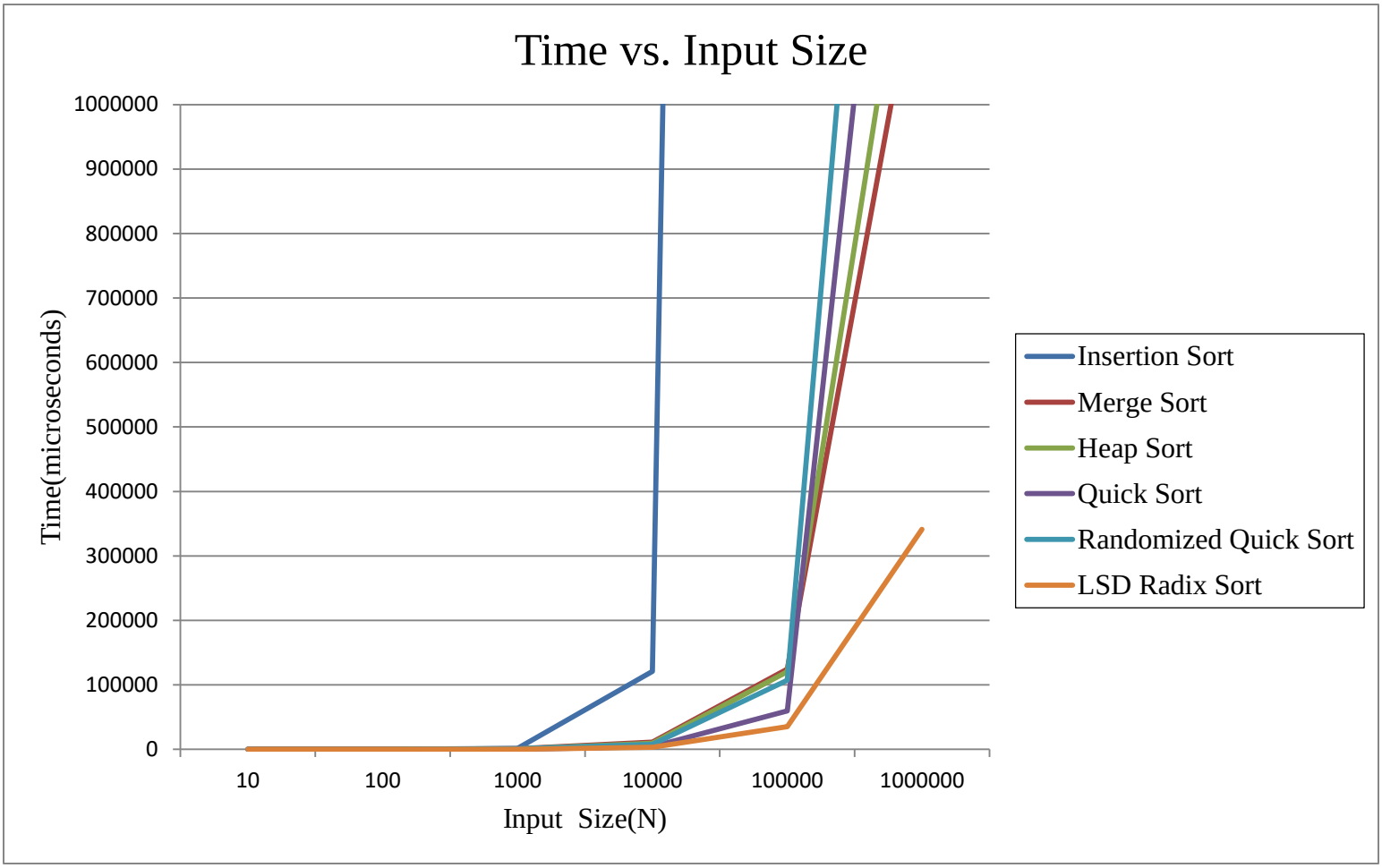


Figure 1. The line chart illustrates how each Sorting Algorithm behaves as input size N gets larger

Analysis:

As I stated in my hypothesis and by studying Table 3 and Figure 1, the slowest Sorting Algorithm is Insertion sort and the fastest Sorting Algorithm is LSD Radix Sort among these six Algorithms. I was also correct that Quick Sort beats Heap Sort which beats Merge Sort. However, Randomized Quick Sort(pick random pivot) is not faster than Quick Sort(pick the last element as the pivot). In fact, Randomized Quick Sort is sometimes slower than Heap Sort such as in the case when the input size is N = 1000000.

However, as the table shows, Insertion Sort even though it is a quadratic Sorting Algorithm it is faster than Merge Sort, Randomized Quick Sort, and LSD Radix Sort for small array sizes and nearly sorted arrays. Insertion Sort starts to be slow when the input size N gets large enough.

Conclusion:

In this programming assignment, the compiler times of six different Sorting Algorithms were investigated for different input sizes. The fastest Sorting Algorithm is the LSD Radix Sort and the slowest sorting Algorithm is the Insertion Sort. Then, as the tables shows, Quick Sort beats Heap Sort which beats Merge Sort. Randomized Quick Sort's behavior was strange as it was sometimes faster and sometimes slower than Heap Sort and Merge Sort. This programming assignment helped me to understand and analyze better these six Sorting Algorithms and their behaviors. I was also able to learn how to implement these Sorting Algorithms in a programming language which will be extremely useful in the future. Moreover, I was able to study and learn from all the difficulties and implementation issues that I had while doing this programming assignment. For instance, before this programming assignment, I did not know how to use the std::numeric_limits to assign huge values such as infinity to an element. I learned this while implementing the merge() function in Merge Sort.

This programming assignment can be extended by including more Sorting Algorithms and making more comparisons and as a result producing more complex tables and graphs.