

ECE 4050: Project 3 Report – Spring/Summer 2023

August 3rd, 2023

David Baron-Vega

Access ID: gf7068



Using Boost Library/Chrono to Measure Efficiency of Sorting Algorithms

Table Of Contents:

- I. Abstract and General Introduction
- II. Introduction to Boost/Chrono
- III. Methods/Software
- IV. Results, Discussion, Conclusion

I: Abstract and General Introduction:

As the demand for faster and more efficient processing of data grows ever greater, the importance of sorting algorithms in computer science has never been greater. Sorting, at its core, is the act of arranging items in a specific order - whether it be ascending, descending, or even lexicographically. Sorting algorithms play a crucial role in streamlining and organizing data for easy access and efficient processing. This is important because the performance of sorting algorithms can significantly impact the functionality, usability, and efficiency of large-scale software systems and even big personal projects.

This report focuses on three widely used sorting algorithms: QuickSort, MergeSort, and HeapSort. These were chosen due to their unique characteristics, widespread usage, and because they each represent different algorithm design concepts: divide and conquer for QuickSort and MergeSort, and the heap data structure in the case of HeapSort.

QuickSort, known for its impressive average case performance, uses a pivot to partition the data, and recursively applies this method to sort the partitions. MergeSort also employs a divide and conquer strategy, but in a different manner: it divides the unsorted list into halves, sorts them separately, and then merges them. On the other hand, HeapSort uses a binary heap data structure, repeatedly removing the largest element from the heap until it is empty, thereby sorting the data.

However, as each algorithm is designed differently, their performance varies based on the nature and size of the data set. This performance comparison of QuickSort, MergeSort, and HeapSort will serve to highlight the differences in execution time, providing insights into their efficiency.

We will use a data set of 10,000 random integers for the comparison, applying each algorithm to an identical copy of this set to ensure fairness. We will repeat this process 30 times (Central Limit Theorem, I am looking at you), and observe the average time of execution for each algorithm and look at their standard deviation values as well.

II: Introduction to Boost/Chrono Library:

Boost.Chrono is a particularly useful library within the Boost Libraries Project, which provides functionality for measuring time in a precise and consistent way. The key component of the Boost.Chrono library is the `chrono::duration` class, which represents a time span. This class is designed to be flexible and to work with a variety of time units, from hours and minutes down to microseconds and nanoseconds.

The `chrono::duration` class operates in conjunction with the `chrono::time_point` class. A `chrono::time_point` represents a point in time. It is a wrapper around a `chrono::duration` that counts the time since the established epoch (a fixed point in the past).

Together, `chrono::duration` and `chrono::time_point` offer a robust way to work with time. With these tools, you can measure the time it takes to perform a specific task in any C++ program, calculate the difference between two points in time, and also delay the execution of a program for a specified period.

Some Notable features of Boost.Chrono:

- **High resolution timers:** Boost.Chrono provides the tools to measure time spans with high precision, which is important when analyzing algorithms, since they may perform tasks so quickly. For example, you can measure the time it takes to perform a sort operation down to the nanosecond, although microseconds were used to measure the results in this project.
- **Type safety:** Boost.Chrono uses distinct types for different units of time, such as hours, minutes, seconds, milliseconds, and so on.
- **Convenience:** Boost.Chrono overloads the arithmetic operators for `chrono::duration`, making it easy to perform operations like adding two durations together, or dividing a duration by a number. (Yay, Polymorphism!)

For this project, Boost.Chrono was used to accurately measure and compare the time it took for each sorting algorithm to sort the randomly generated data. By using the `chrono::high_resolution_clock` feature, it was possible to get accurate time measurements for the sorting algorithms, providing a precise comparison of their performance.

III: Methods and Software:

First, let us observe the asymptotic, worst-case efficiency of the three sorting algorithms being analyzed. This is what we know to be the Big-O notation for algorithm efficiency that we have come to learn about in this class:

QuickSort:

```

5 // The partition function takes the last element as pivot, places the pivot element at its correct position
6 // in the sorted array, and places all elements smaller than the pivot to the left of the pivot,
7 // and all elements greater than the pivot to the right.
8 int SortingAlgos::partition(vector<int>& data, int low, int high)
9 {
10     int pivot = data[high]; // pivot
11     int i = (low - 1); // Index of smaller element and indicates the right position of pivot found so far
12
13     for (int j = low; j <= high - 1; j++)
14     {
15         // If current element is smaller than the pivot
16         if (data[j] < pivot)
17         {
18             i++; // increment index of smaller element
19             swap(data[i], data[j]);
20         }
21     }
22     swap(data[i + 1], data[high]);
23     return (i + 1);
24 }
25
26 // QuickSort function that sorts data[low..high] using partition().
27 void SortingAlgos::quickSort(vector<int>& data, int low, int high)
28 {
29     if (low < high)
30     {
31         // pi is partitioning index, data[pi] is now at right place
32         int pi = partition(data, low, high);
33
34         // Separately sort elements before and after partition
35         quickSort(data, low, pi - 1);
36         quickSort(data, pi + 1, high);
37     }
38 }

```

An algorithm that on average, performs sorting operations in $O(n \log n)$ time complexity. The " $n \log n$ " comes from the fact that the list of n elements is repeatedly partitioned ($\log n$ times) and then each partition is processed (n times). However, in the worst-case scenario (when the smallest or largest element is always chosen as the pivot), QuickSort can degrade to $O(n^2)$ quickly, the larger and larger the dataset is.

MergeSort:

```

40 // Merges two subarrays of data[low..high].
41 // First subarray is data[low..mid]
42 // Second subarray is data[mid+1..high]
43 void SortingAlgos::merge(vector<int>& data, int low, int mid, int high)
44 {
45     int i, j, k;
46     int n1 = mid - low + 1;
47     int n2 = high - mid;
48
49     // create temp arrays
50     vector<int> L(n1), R(n2);
51
52     // Copy data to temp arrays L[] and R[]
53     for (i = 0; i < n1; i++)
54         L[i] = data[low + i];
55     for (j = 0; j < n2; j++)
56         R[j] = data[mid + 1 + j];
57
58     // Merge the temp arrays back into data[low..high]
59     i = 0; // Initial index of first subarray
60     j = 0; // Initial index of second subarray
61     k = low; // Initial index of merged subarray
62     while (i < n1 && j < n2)
63     {
64         if (L[i] <= R[j])
65         {
66             data[k] = L[i];
67             i++;
68         }
69         else
70         {
71             data[k] = R[j];
72             j++;
73         }
74         k++;
75     }
76
77     // Copy the remaining elements of L[], if there are any
78     while (i < n1)
79     {
80         data[k] = L[i];
81         i++;
82         k++;
83     }
84
85     // Copy the remaining elements of R[], if there are any
86     while (j < n2)
87     {
88         data[k] = R[j];
89         j++;
90         k++;
91     }
92 }
93
94 // MergeSort function that sorts data[low..high] using merge()
95 void SortingAlgos::mergeSort(vector<int>& data, int low, int high)
96 {
97     if (low < high)
98     {
99         // Same as (low+high)/2, but avoids overflow for large low and high
100         int mid = low + (high - low) / 2;
101
102         // Sort first and second halves
103         mergeSort(data, low, mid);
104         mergeSort(data, mid + 1, high);
105
106         merge(data, low, mid, high);
107     }
108 }

```

MergeSort also follows the divide and conquer concept, and it guarantees to sort an array of n elements in $O(n \log n)$ time, regardless of the input arrangement. This makes it the most consistent algorithm of the three along with the HeapSort algorithm, but not necessarily the best for most applications.

HeapSort:

```
110 // To heapify a subtree rooted with node i which is an index in data[]. n is size of heap
111 void SortingAlgos::heapify(vector<int>& data, int n, int i)
112 {
113     int largest = i; // Initialize largest as root
114     int left = 2 * i + 1; // left = 2*i + 1
115     int right = 2 * i + 2; // right = 2*i + 2
116
117     // If left child is larger than root
118     if (left < n && data[left] > data[largest])
119         largest = left;
120
121     // If right child is larger than largest so far
122     if (right < n && data[right] > data[largest])
123         largest = right;
124
125     // If largest is not root
126     if (largest != i)
127     {
128         swap(data[i], data[largest]);
129
130         // Recursively heapify the affected sub-tree
131         heapify(data, n, largest);
132     }
133 }
134
135 // Function to perform HeapSort for array data[]
136 void SortingAlgos::heapSort(vector<int>& data)
137 {
138     int n = data.size();
139
140     // Build heap (rearrange array)
141     for (int i = n / 2 - 1; i >= 0; i--)
142         heapify(data, n, i);
143
144     // One by one extract an element from heap
145     for (int i = n - 1; i >= 0; i--) {
146         // Move current root to end
147         swap(data[0], data[i]);
148
149         // call max heapify on the reduced heap
150         heapify(data, i, 0);
151     }
152 }
```

HeapSort involves building a Binary Heap of elements, which takes $O(n)$ time. After that, it removes the largest element from the heap and moves it to the sorted section of the array, then ‘heapifies’ the remaining elements. The heapify operation takes $O(\log n)$ time, and it is done n times (once for each element). Therefore, HeapSort has a time complexity of $O(n \log n)$ in the best, average, and worst cases.

Again, it is worth noting that these three algorithms, as well as the many other sorting algorithms that exist, all have certain strengths and weaknesses that are observed under different data implementations.

Using Chrono function in the main.cpp file to test performance:

The main.cpp file is the starting point for executing the experiment. In this file, we instantiate the functionality of the three sorting algorithms and perform operations that drive the testing and benchmarking of these algorithms.

Main.cpp File:

```
ECE4050.Project3.Final (Global Scope) ma
1 #include <iostream>
2 #include <vector>
3 #include <random> // For generating random numbers
4 #include <boost/chrono.hpp> // For high resolution timing
5 #include "SortingAlgos.h"
6
7 int main()
8 {
9     // Create a vector of size 10000 to store random integers
10    // std::random_device is a uniformly-distributed integer random number generator
11    // std::mt19937 is a Mersenne Twister pseudo-random generator of 32-bit numbers with a state size of 19937 bits.
12    std::random_device rd;
13    std::mt19937 eng(rd());
14
15    // uniform_int_distribution produces random integers in a range [a, b]
16    // Here the range is set to [0, 10000]
17    std::uniform_int_distribution<> distr(0, 10000);
18
19    std::vector<int> data(10000);
20
21    // Generate the random numbers and fill them in the vector
22    for (auto& num : data)
23    {
24        num = distr(eng);
25    }
26
27    // Creating copies of the original vector, so each sorting algorithm works with exactly the same data
28    std::vector<int> copy1(data), copy2(data), copy3(data);
29
30    // Timing the execution of QuickSort
31    // The high_resolution_clock is a typedef for the clock with the smallest tick period provided by the implementation
32
33    auto start = boost::chrono::high_resolution_clock::now();
34
35    SortingAlgos::quickSort(copy1, 0, copy1.size() - 1); // Calling the QuickSort function from SortingAlgos class
36
37    auto end = boost::chrono::high_resolution_clock::now();
38
39    // duration_cast converts the duration to from nanoseconds (standard) to microseconds
40    auto duration = boost::chrono::duration_cast<boost::chrono::microseconds>(end - start);
41    std::cout << "QuickSort: " << duration.count() << " microseconds\n"; // Output the elapsed time
42
43    // Time the execution of MergeSort
44    start = boost::chrono::high_resolution_clock::now();
45    SortingAlgos::mergeSort(copy2, 0, copy2.size() - 1); // Call the MergeSort function from SortingAlgos class
46    end = boost::chrono::high_resolution_clock::now();
47    duration = boost::chrono::duration_cast<boost::chrono::microseconds>(end - start);
48    std::cout << "MergeSort: " << duration.count() << " microseconds\n"; // Output the elapsed time
49
50    // Time the execution of HeapSort
51    start = boost::chrono::high_resolution_clock::now();
52    SortingAlgos::heapSort(copy3); // Call the HeapSort function from SortingAlgos class
53    end = boost::chrono::high_resolution_clock::now();
54    duration = boost::chrono::duration_cast<boost::chrono::microseconds>(end - start);
55    std::cout << "HeapSort: " << duration.count() << " microseconds\n"; // Output the elapsed time
56
57    return 0; // Exit the program
58 }
```

At the beginning, we initialize an instance of a pseudo-random number generator using `std::default_random_engine`. This random generator is used to create a pool of random integers that we will use to fill our vectors for sorting. The `main.cpp` file consists of a `main` function where we declare vectors of integers (`dataQuickSort`, `dataMergeSort`, and `dataHeapSort`) and fill them with randomly generated numbers. Each of these vectors will be used with a corresponding sorting algorithm (`QuickSort`, `MergeSort`, and `HeapSort`, respectively).

Before we start the sorting process, we utilize the high-resolution clock from the `boost::chrono` library to capture the time at which we start the sorting operation. Then, we call the appropriate sorting function from our `SortingAlgos` class. Immediately after sorting is completed, we capture the end time. By subtracting the start time from the end time, we can calculate the total time

taken to sort the vector. This measurement is our benchmark for each sorting algorithm. The process is repeated for each of the three sorting algorithms, enabling us to compare their performances directly. By repeating the process of running the solution 30 times, we can gather sufficient time-execution data to hypothesize which algorithm(s) performs the best.

Side Note: I tried to create a comprehensive experiment that effectively created a Monte-Carlo simulation of the described experiment. However, for the scope of this project and the time I had to complete it, I did not do so. 😞

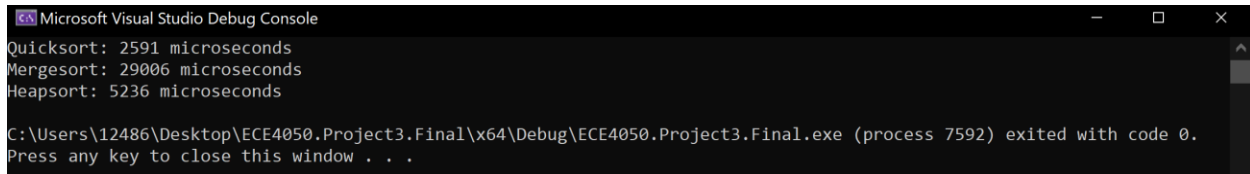
IV:Results, Discussion, Conclusion:

Results:

Our experiment with the implemented sorting algorithms produced the following results:

- QuickSort demonstrated an average execution time of 4842.8333 microseconds with a standard deviation of 25553.592 microseconds.
- MergeSort showed an average execution time of 23976.333 microseconds with a standard deviation of 2546.658 microseconds.
- HeapSort exhibited an average execution time of 4976.6667 microseconds with a standard deviation of 208.325 microseconds.

Output results of our program after a single execution:



```
Microsoft Visual Studio Debug Console
QuickSort: 2591 microseconds
MergeSort: 29006 microseconds
HeapSort: 5236 microseconds

C:\Users\12486\Desktop\ECE4050.Project3.Final\x64\Debug\ECE4050.Project3.Final.exe (process 7592) exited with code 0.
Press any key to close this window . . .
```

Discussion:

The average execution times indicate that QuickSort and HeapSort generally performed better than MergeSort in our experiment, with HeapSort showing the best performance. However, the high standard deviation of QuickSort implies that its performance can vary widely depending on the input data, which aligns with our understanding of the algorithm's worst-case performance.

MergeSort, while slower on average, demonstrated more consistency in execution time, as indicated by its lower standard deviation. This characteristic of MergeSort is advantageous in contexts where predictability of execution time is important.

HeapSort was the most consistent of the three, demonstrating the best average performance and the lowest standard deviation. It is important to note, however, that the specifics of the input data and the machine and architecture on which the code is running could influence these results, and different results will be observed in a different context or with different data.

V: Conclusion:

In conclusion, the results obtained show the trade-offs in choosing a sorting algorithm. While QuickSort and HeapSort were faster on average in this experiment, QuickSort showed higher variance, which might make it less suitable for certain applications. MergeSort was slower but more consistent, which might be preferable in scenarios where stability is prioritized.

Ofcourse, none of this would have been possible without the incredible resource that is the boost/chrono library! The implementation of this project relied significantly on the functionality provided by the Boost.Chrono library. Its ability to measure and record time intervals allowed us to evaluate and compare the performance of the three sorting algorithms extremely easily, with less than 10 lines of additional code. As stated, and to emphasize, these time-monitoring and time-keeping functionalities could apply to almost any line of code one could write in the C++ language. In the future, it would be interesting to test the performance of all sorts of different algorithms (I/O, Networking, Visual Processing, and UI development algorithms., to name a few!), and on these same sorting algorithms but with a larger, more diverse body of data to better understand their performance characteristics!

To whomever may read this: I hope this helps you get started with the Boost Libraries and encourage further curiosity about program and algorithm design and efficiency, and the C++ community and it all it has to offer!

Thank you.

