Project 1

Analysing the Quicksort and Insertion Sort algorithms

CSE 5211: Analysis of Algorithms

Dr. William Shoaff

Zubin Kadva

1. Problem Description

The main purpose behind this project is to compare and contrast some algorithms used for sorting. Sorting algorithms arranges elements in a list in a certain order, commonly in ascending order. Most sorting algorithms require comparisons between the elements and swapping of those elements. [3]

2. Known algorithms

Sorting algorithms are classified based on: [1]

- Size of the list
- Computational complexity with respect to the no. of swaps
- Memory usage
- Recursion
- Comparisons

Some of them include:

- Quicksort
- Heapsort
- Merge sort
- Insertion sort
- Bubble sort
- Selection sort

In this project, we study **quicksort** and the **insertion sort** algorithms.

3. Algorithm description

3.1. Quicksort

Quicksort is a sorting algorithm, which partitions the list into smaller lists. It does so by selecting an element known as the pivot. Thus, the list consists of three sections: (i) All elements less than the pivot, (ii) the pivot and (ii) all elements greater than the pivot. Then, the quicksort algorithm is applied recursively to the first and third part of the list.

```
function quicksort(array, start, end)
      if(start < end)
            pivot := partition(array, start, end)
            quicksort(array, start, pivot - 1)
            quicksort(array, pivot + 1, end)
```

Pseudocode of the quicksort routine

```
function partition(array, start, end)
      pivot := start
      for i = start to end
            if(array[i] <= array[end])
                  pivot++
                  swap(i, pivot)
            swap(pivot, end)
      return pivot
```

Pseudocode of the partitioning routine

3.2. Insertion sort

Insertion sort loops over all positions in the sub list and then inserts the element at its proper position within the list. Thus, each element (known as key) is compared with the element on the left. This continues until the key is at its proper place. Thus, the list is sort.

```
function insertionSort(array)
      for i = 0 to array.length – 1
            temp = array[i + 1]
            j = i + 1
             while (j > 0 && array[j - 1] >= temp)
                  array[j] = array[j - 1]
                  j--
             array[j] = temp
```

Pseudocode of the insertion sort routine

4. Analysis

4.1. Quicksort

Best case analysis

The `partition` sub routine is expressed as:

$$T(n) = \sum_{i=start}^{end} (1) \quad = n$$

Thus, the complexity is $\boldsymbol{\Omega(n)}$.

The `quicksort` routine is expressed as:

$$T(n) = n + T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right)$$

$$\therefore\ T(n) = n + 2T\left(\frac{n}{2}\right)$$

$$\therefore T\left(\frac{n}{2}\right) = n + 2\left[\frac{n}{2} + 2T\left(\frac{n}{4}\right)\right]$$

$$\therefore T\left(\frac{n}{2}\right) = 2n + 4T\left(\frac{n}{4}\right)$$

$$\therefore T\left(\frac{n}{4}\right) = 2n + 4\left[n + 2T\left(\frac{n}{8}\right)\right]$$

$$\therefore T\left(\frac{n}{4}\right) = 3n + 8T\left(\frac{n}{8}\right)$$

The general equation is:

$$T(n) = kn + 2^k T\left(\frac{n}{2^k}\right)$$

$$Base\ case, T(1) = 0$$

$$\therefore n = 2^k\ \ \therefore k = \lg n$$

$$\therefore T(n) = n \lg n$$

Thus the complexity is $\boldsymbol{\Omega(n \lg n)}$.

<u>Worst case analysis</u>

The `partition` sub routine is expressed as:

$$T(n) = \sum_{i=start}^{n} (1) \quad = n$$

Thus, the complexity is $\boldsymbol{O(n)}$.

The `quickSort` routine is expressed as:

$$T(n) = n + T(n-1)$$

$$\therefore T(n-1) = n + [n - 1 + T(n-2)]$$

$$\therefore T(n-1) = 2n - 1 + T(n-2)$$

$$\therefore T(n-2) = 2n - 1 + [n - 2 + T(n-3)]$$

$$\therefore T(n-2) = 3n - 3 + T(n-3)$$

The general form of the equation is:

$$T(n) = kn - k + 1 + T(n - k)$$

$$Base\ case, T(1) = 0$$

$$\therefore n - k = 1 \therefore k = n$$

$$\therefore T(n) = n^2 - n + 1$$

$$\therefore T(n) = n^2$$

Thus, the complexity is $\boldsymbol{O(n^2)}$.

4.2. Insertion Sort

Best case analysis

The `insertionSort` routine is expressed as:

$$T(n) = \sum_{i=1}^{n-1} (1)$$

$$\therefore T(n) = n$$

Thus, the complexity is $\boldsymbol{\Omega(n)}$.

Worst case analysis

The `insertionSort` routine is expressed as:

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=1}^{i} (1)$$

$$\therefore T(n) = \sum_{i=1}^{n-1} i$$

$$\therefore T(n) = 1 + 2 + \cdots + n - 1$$

$$\therefore T(n) = \frac{n(n-1)}{2}$$

$$\therefore T(n) = O(n^2)$$

Thus, the complexity is $\boldsymbol{O(n^2)}$.

This can be summarized as:

Algorithm → / Complexity ↓	Quicksort	Insertion Sort
Best case	$\Omega(n \lg n)$	$\Omega(n)$
Average case	$\theta(n \lg n)$	$\theta(n^2)$
Worst case	$O(n^2)$	$O(n^2)$

Table: Algorithm complexities

5. Generating random data

```
/* Author: Zubin Kadva
 * Class: Analysis of Algorithms, Spring 2017
 * Project: Quicksort and Insertion Sort
 */
import java.util.Random;

public class Main {
    public static void main(String a[]) {
        int SIZE = 10;
        int[] ar = new int[SIZE];
        Random random = new Random();
        for (int i = 0; i < ar.length; i++) {
            ar[i] = random.nextInt(Integer.MAX_VALUE);
        }
        Quicksort quicksort = new Quicksort();
        quicksort.quicksort(ar, 0, ar.length - 1);
        InsertionSort insertionSort = new InsertionSort();
        insertionSort.insertionSort(ar);
    }
}
```

6. Implementation

6.1. Quicksort

```
/* Author: Zubin Kadva
 * Class: Analysis of Algorithms, Spring 2017
 * Project: Quicksort
 */
public class Quicksort {
    public void quicksort(int[] array, int start, int end) {
        // Check if list is not empty
        if (start < end) {
            // Calculate pivot
            int pivot = partition(array, start, end);
            // Recursive call on the two sublists
            quicksort(array, start, pivot - 1);
            quicksort(array, pivot + 1, end);
        }
    }

    private int partition(int[] array, int start, int end) {
        int pivot = start;
        for (int i = start; i < end; i++) {
            // Check if pivot less than end of list
            if (array[i] <= array[end]) {
                // Swap and increase pivot
                swap(array, i, pivot);
                pivot++;
            }
        }
        // Swap pivot and end of list to create two sublists
        swap(array, pivot, end);
        return pivot;
    }
```

```
    // Swap elements
    private void swap(int[] array, int a, int b) {
        int temp = array[a];
        array[a] = array[b];
        array[b] = temp;
    }
}
```

6.2. Insertion sort

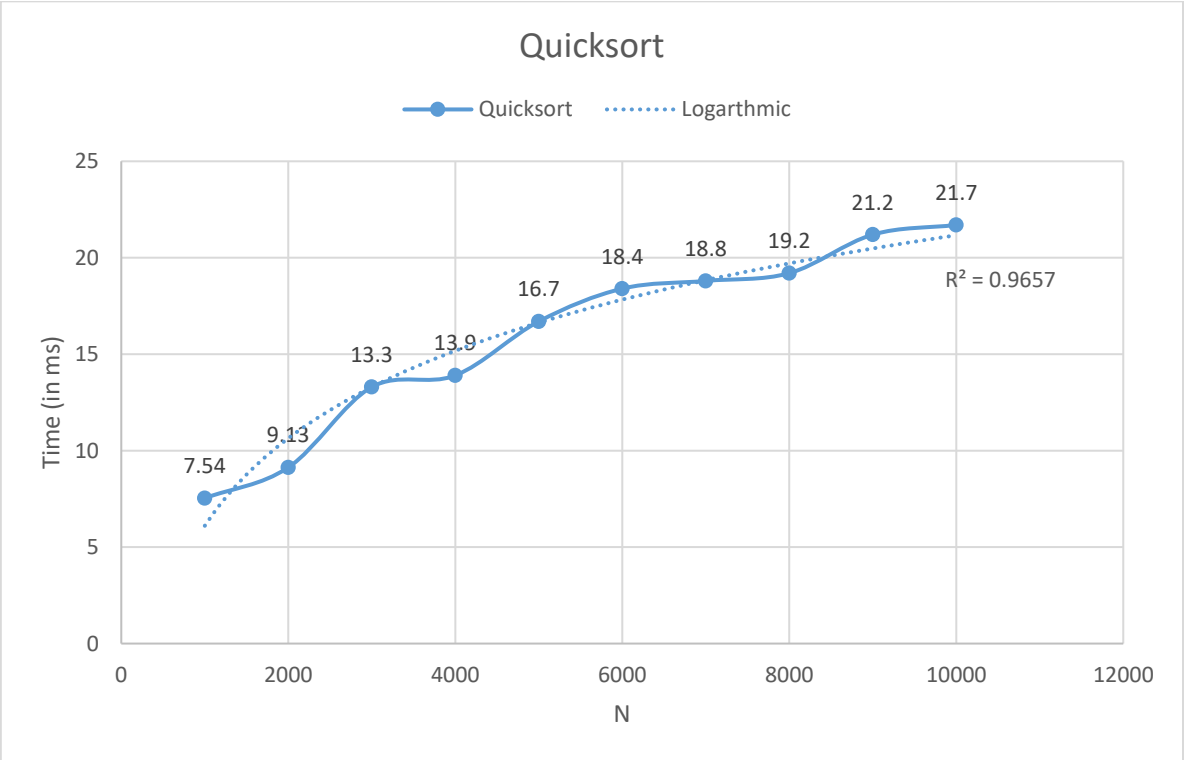
```
/* Author: Zubin Kadva
 * Class: Analysis of Algorithms, Spring 2017
 * Project: Insertion Sort
 */
public class InsertionSort {
    public void insertionSort(int[] array) {
        for (int i = 0; i < array.length - 1; i++) {
            // Create a copy of the next element and its index
            int temp = array[i + 1];
            int j = i + 1;
            // Insert the element to its proper place
            while (j > 0 && array[j - 1] >= temp) {
                // Insert the element and decrement index
                array[j] = array[j - 1];
                j--;
            }
            // Place the copied element
            array[j] = temp;
        }
    }
}
```

7. Execution time

7.1. Quicksort

N	Time (in ms)
1000	7.54
2000	9.13
3000	13.3
4000	13.9
5000	16.7
6000	18.4
7000	18.8
8000	19.2
9000	21.2
10000	21.7

Table: Quicksort execution time from n=1,000 to 10,000

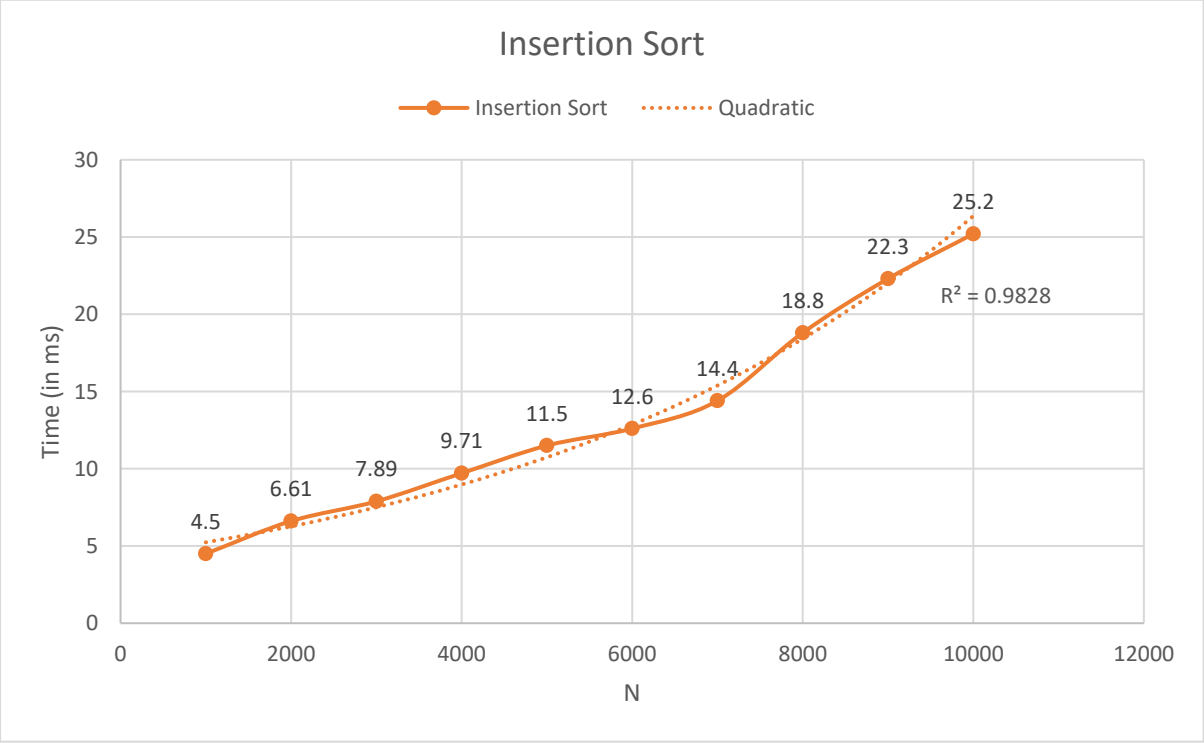


Graph: Scatter plot of the above data with a logarithmic trend line

7.2. Insertion Sort

N	Time (in ms)
1000	4.5
2000	6.61
3000	7.89
4000	9.71
5000	11.5
6000	12.6
7000	14.4
8000	18.8
9000	22.3
10000	25.2

Table: Insertion sort execution time from n=1,000 to 10,000



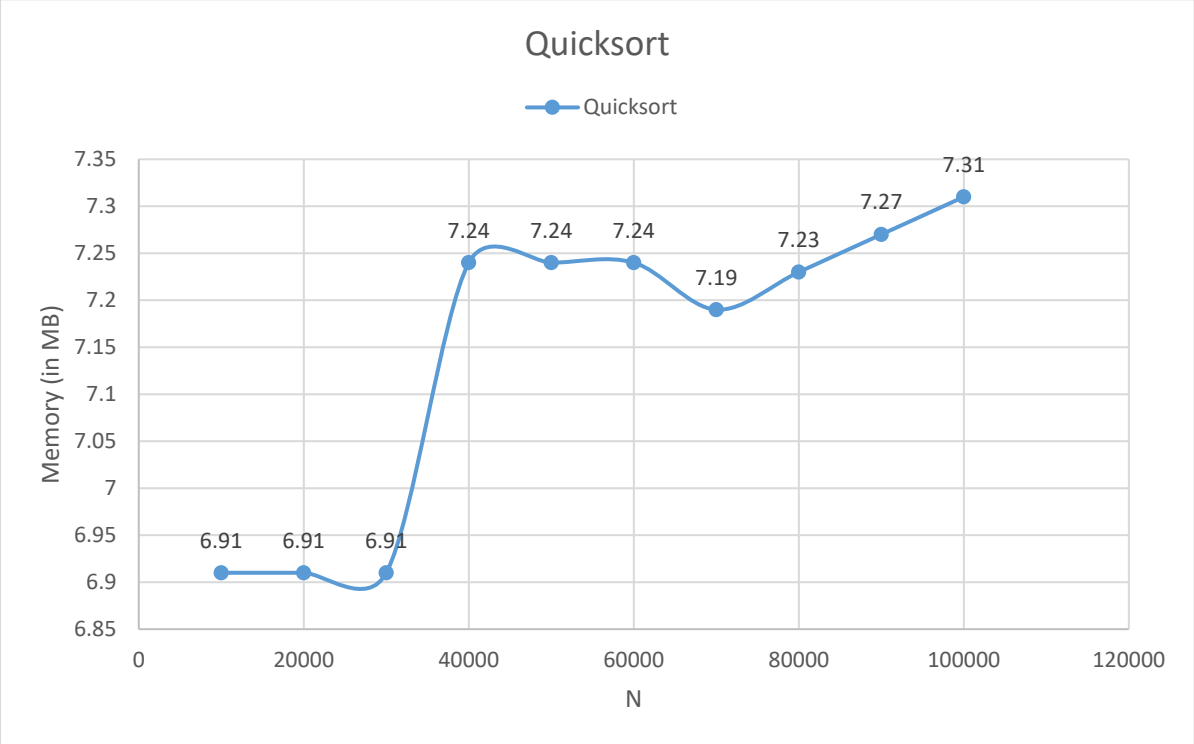
Graph: Scatter plot of the above data with a quadratic trend line

8. Memory consumption

8.1. Quicksort

N	Memory (in MB)
10000	6.91
20000	6.91
30000	6.91
40000	7.24
50000	7.24
60000	7.24
70000	7.19
80000	7.23
90000	7.27
100000	7.31

Table: Quicksort memory consumption from n=10,000 to 100,000

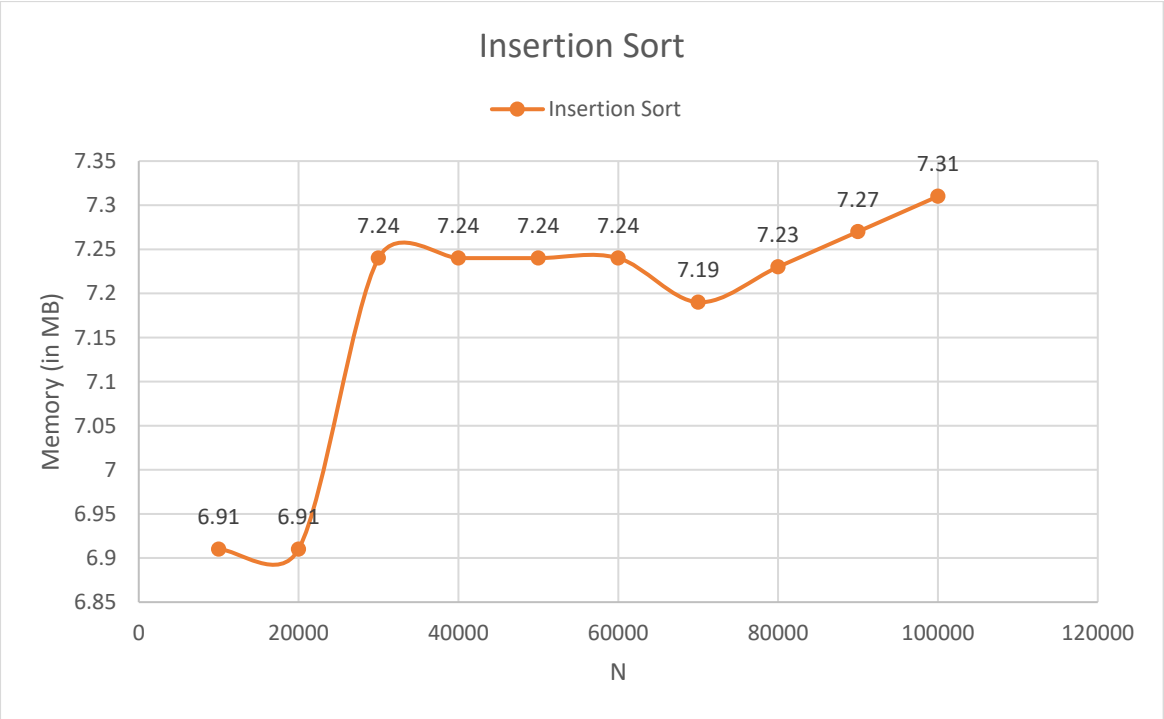


Graph: Scatter plot of the above data

8.2. Insertion Sort

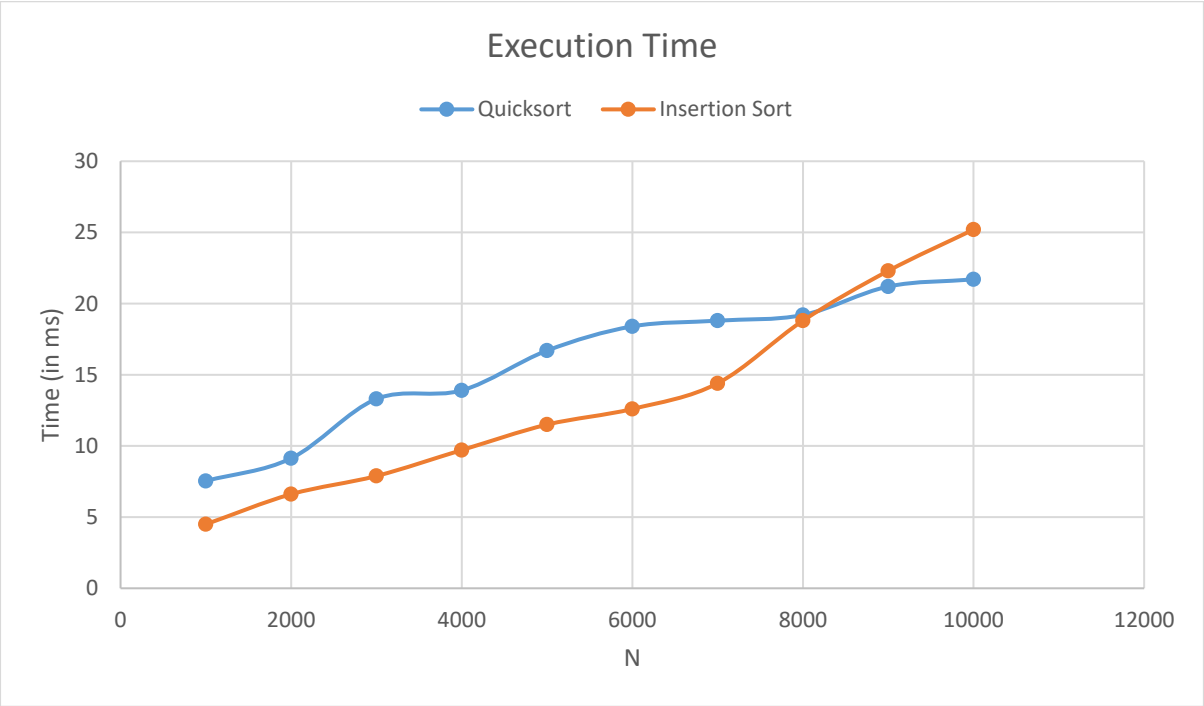
N	Insertion Sort
10000	6.91
20000	6.91
30000	7.24
40000	7.24
50000	7.24
60000	7.24
70000	7.19
80000	7.23
90000	7.27
100000	7.31

Table: Insertion sort memory consumption from n=10,000 to 100,000

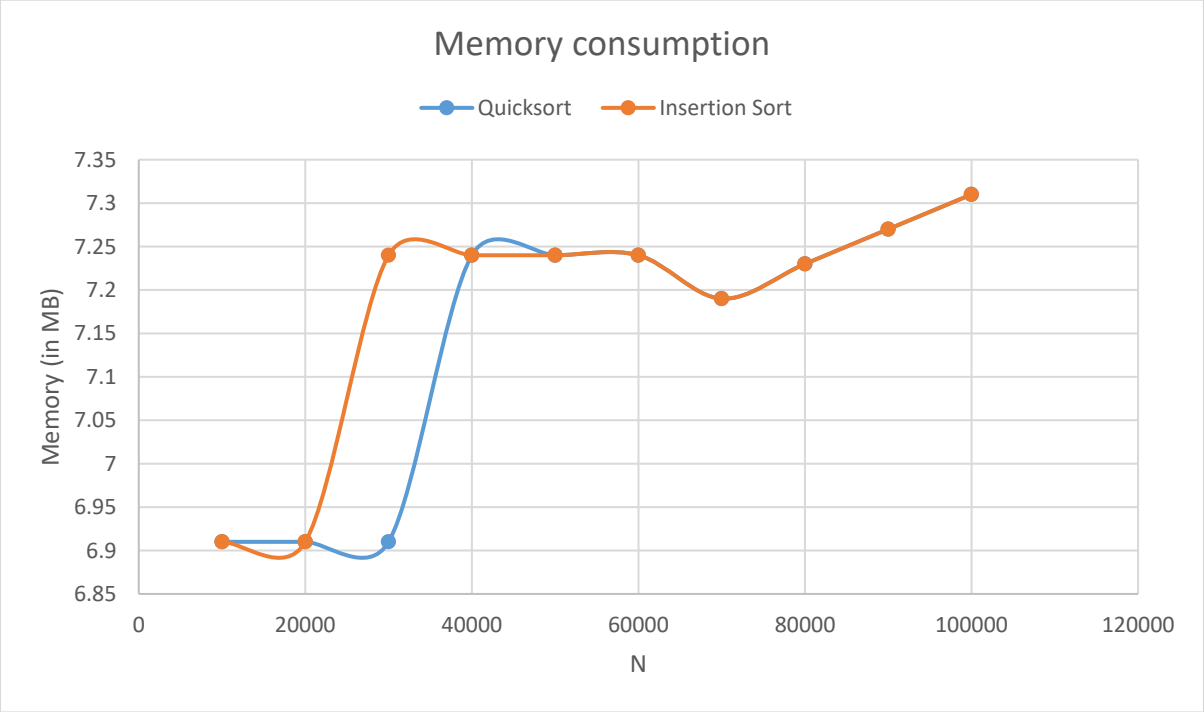


Graph: Scatter plot of the above data

9. Comparisons



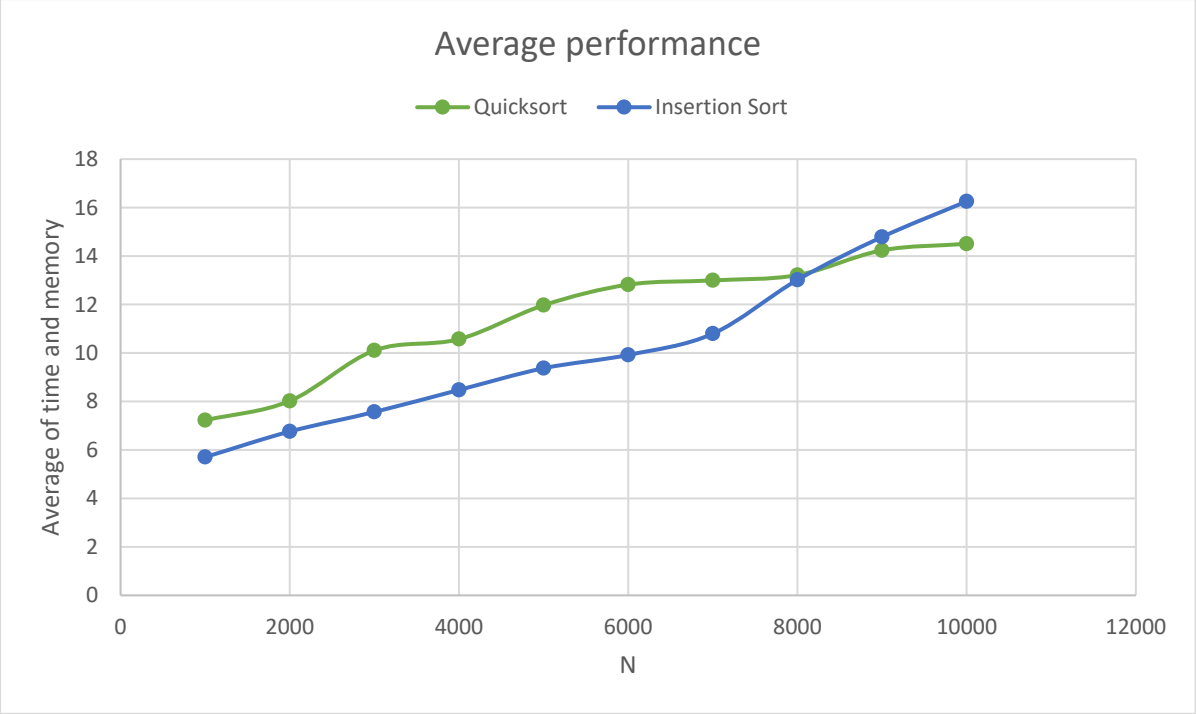
Graph: Execution time comparison



Graph: Memory consumption comparison

N	Quicksort	Insertion Sort
1000	7.225	5.705
2000	8.02	6.76
3000	10.105	7.565
4000	10.57	8.475
5000	11.97	9.37
6000	12.82	9.92
7000	12.995	10.795
8000	13.215	13.015
9000	14.235	14.785
10000	14.505	16.255

Table: Average performance for quicksort and insertion sort



Graph: Scatter plot of the above data

10. References and tools

[1] Sorting algorithms from https://en.wikipedia.org/wiki/Sorting_algorithm

[2] Knuth, D. E. (1998), "*The Art of Computer Programming*", Volume 3: (2nd edition) sorting and Searching, Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA

[3] Corman, T. H., Leiserson, C. E., Rivest, R. L., and Stein, C. (2009), "*Introduction to Algorithms",* MIT Press, 3rd edition

The NetBeans profiler for profiling the performance of the two algorithms.

Microsoft Excel for plotting graphs of the gathered data.