Katholieke Universiteit Leuven

Data Structures and Algorithms

Assignment 1

Author: Pieter Verlinden
Student ID: r0369547
Study: 1Ba Informatica

Lecturer: Dr. Ir. Philip Dutré

March 15, 2013

Contents

1 Introduction

The purpose of this assignment is to study and compare a number of sorting algorithms. A conclusion will be made based on two experiments.

In the first experiment, four sorting algorithms (selection sort, insertion sort, mergesort and quicksort) are tested on their number of compares made to sort an array. Both sorted and unsorted arrays will be tested.

The second experiment is a double ratio experiment. This implies measuring the time for a sorting algorithm to sort an array of size N. Dividing the time to sort an array of size $2N$ with its previous time, gives us the doubling ratio of that algorithm. In this experiment, insertion sort and quick sort will be tested and thoroughly discussed.

2 Experiments

2.1 Generic Sorting Algorithms

This first experiment implies counting the compares made by each sorting algorithm. Methods for generating immense amounts of data input and data output gives us significant data which is then plotted in a scatter graph (see Figures 1 and 2).

One hundred arrays respectively with size $(0..N-1)$ and random values from $(0..N-1)$ were sorted. The amount of compares that each test returned are plotted on the chart in Figure 1. As for the data in the chart in Figure 2, the same method is maintained but with arrays that are already sorted.

Experimental data

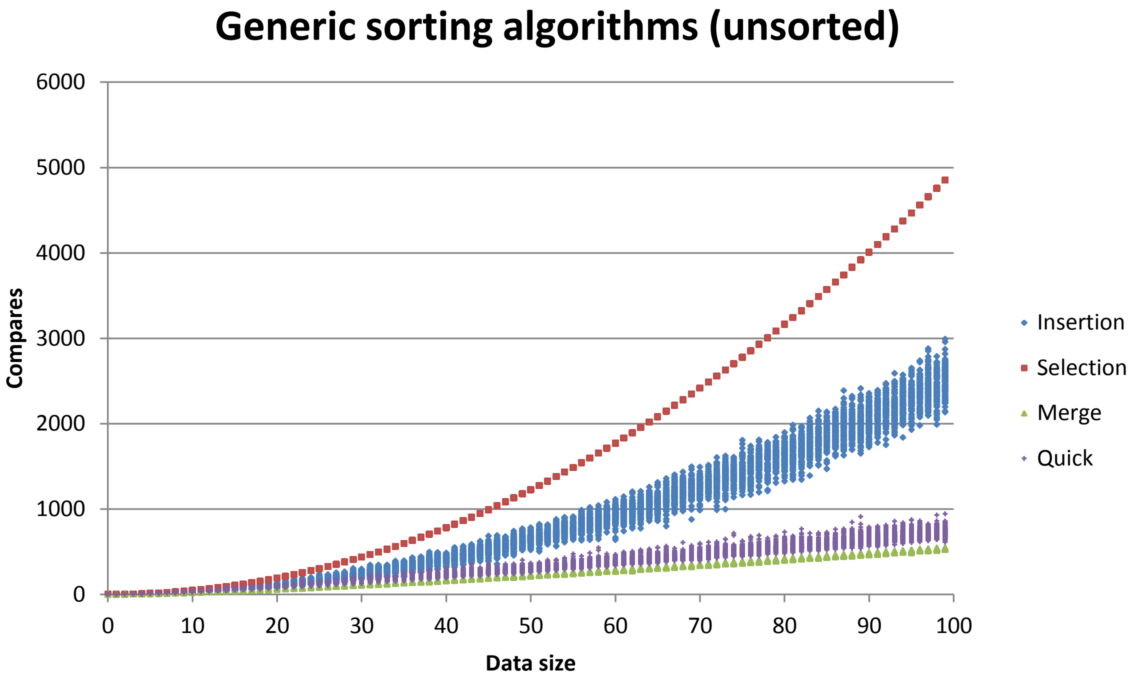


Figure 1: Generic sorting algorithms data with unsorted arrays as input

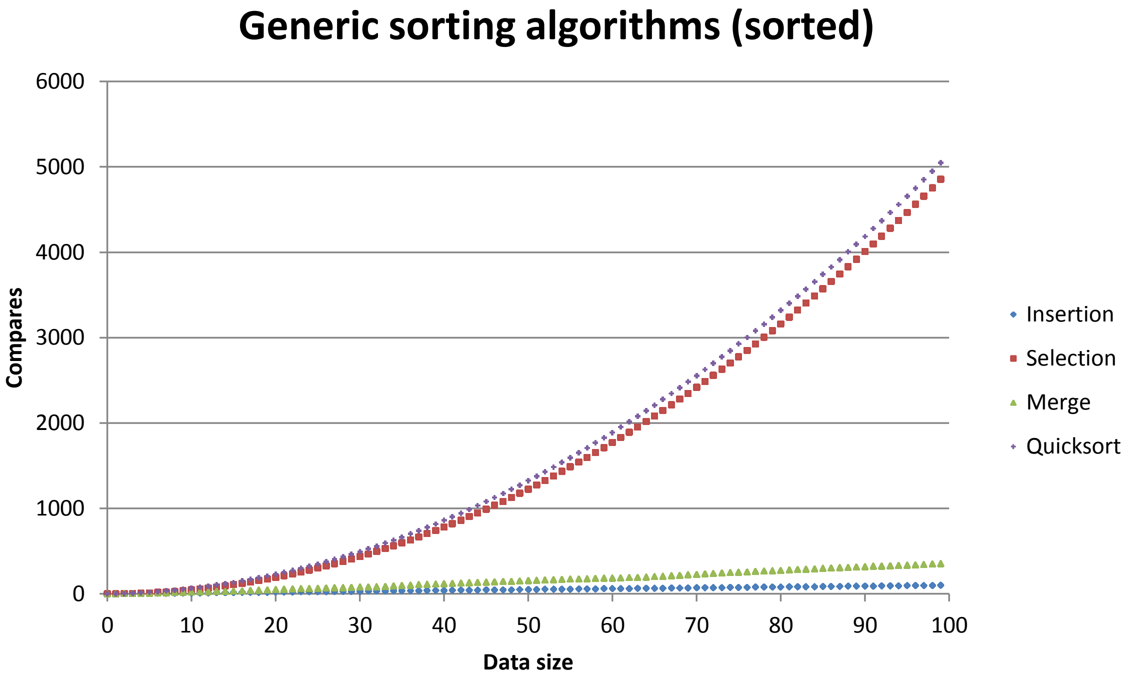


Figure 2: Generic sorting algorithms data with sorted arrays as input

2.1.1 Selection sort

Selection sort works by selecting the smallest element in the array and exchanging it with the first entry then selecting the second smallest element and exchanging it with the second entry until the array is sorted.

> ***Proposition A.*** *Selection sort uses* $\sim \frac{N^2}{2}$ *compares and* N *exchanges to sort an array of length* N*.*
> *(Source: [1])*

As stated by proposition A, the expected compares of the selection sort algorithm proceeds according

$$\sim \frac{N^2}{2}$$

Because selection sort checks every entry in an array, it is insensitive to input. This means that it takes about as long to run selection sort for an array that is sorted or for an array with all values equal as it does for an array with random elements.

As we can see in Figure 1, the compares of selection sort (red dotted line) grow quadratically. Because there are no values surrounding the line (like the thick dotted line of the compares of insertion) we can conclude that selection sort is insensitive to input. The data in Figure 2 confirms our statement of insensitivity by the fact that there is no difference between the compares of an array with random values or between the compares of an array with sorted values.

2.1.2 Insertion sort

Insertion sort works by inserting elements according to their size in the array. The items at the left side are in order during the sort, however they are not in their final position. They may have to be moved so other elements can be inserted.

> ***Proposition B.*** *Insertion sort uses* $\sim \frac{N^2}{4}$ *and* $\sim \frac{N^2}{4}$ *exchanges to sort a randomly ordered array of length N with distinct keys, on the average. The worst case is* $\sim \frac{N^2}{2}$ *compares and* $\sim \frac{N^2}{2}$ *exchanges and the best case is* $N-1$ *compares and* 0 *exchanges.*
> *(Source: [1])*

As stated by proposition B, the expected compares and exchanges proceeds according

$$\sim \frac{N^2}{4}$$

The insertion sort algorithm is very input sensitive. In Figure 1 we see that insertion sort has the most spread data which proves our theorem true. From the data in Figure 2 we can conclude that the insertion sort algorithm is the most efficient for sorted and nearly sorted arrays.

2.1.3 Mergesort

Mergesort is a recursive sorting algorithm based on the principle of splitting, sorting and merging. An array is recursively split until only auxiliary arrays of length one remain. Each auxiliary array then sorts and merges itself until the original array is sorted.

> ***Proposition F.*** *Top-down mergesort uses between* $\frac{1}{2}N \log N$ *and* $N \log N$ *compares to sort an array of length N.*
> *(Source: [1])*

As stated by proposition F, the expected compares proceeds according

$$\sim N \log N$$

One of the great advantages of mergesort is that it offers certainty about the time to sort. If we sort an array of size N, the time that is needed to sort this array will always be proportional to $N \log N$. One major let-down is the extra use of space proportional to N.

As we can see in Figure 1 and Figure 2, mergesort is a very time-stable algorithm and is not very sensitive to input. According to our results we could state that mergesort is the most efficient algorithm. However, the extra space that is needed makes it extensively slower due to memory access time.

2.1.4 Quicksort

Quicksort is also a recursive sorting algorithm based on partitioning an array into two subarrays. The two subarrays will then be sorted independently. It has many in common with mergesort where we break the array into two subarrays, sort and merge them. Quicksort splits the array in two subarrays of different length. the position where the array is split depends on the contents of the array.
The major advantage of quicksort is that, unlike mergesort, it uses far less space and makes it superior to the mergesort algorithm.

> ***Proposition K.*** *Quicksort uses $\sim 2N \log N$ compares (and one-sixth that many exchanges) on the average to sort an array of length N with distinct keys.*
> *(Source: [1])*

> ***Proposition L.*** *Quicksort uses $\sim \frac{N^2}{2}$ compares in the worst case, but random shuffling protects against this case.*
> *(Source: [1])*

As stated by proposition K the expected compares proceeds according

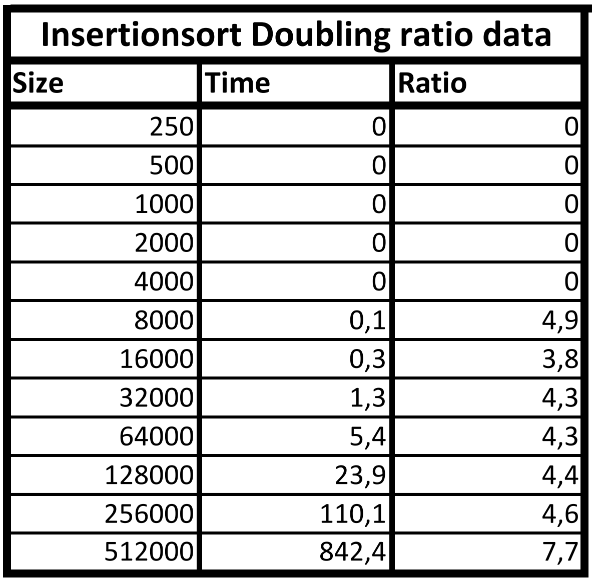
$$\sim 2N \log N$$

The data in Figure 2 confirms proposition L. An array that is already sorted is the worst case scenario for the used quicksort algorithm. This is due to the chose of pivot point. In the used algorithm, the first entry is chosen as pivot. As the array is already sorted, the first element will always be the smallest one which leads to a recursive call on whole the array minus one element. This will lead to a very inefficient algorithm. A random shuffle of the elements in the array protects against this occurrence. Since an array with random elements can be represented by a shuffled array, Figure 1 proves the shuffle theorem true.

2.2 Doubling ratio experiment

In this experiment, actual time will be measured to determine a certain ratio. This report only tests two of five algorithms, insertion sort and quicksort.

2.2.1 Insertion sort



Insertionsort Doubling ratio data

Size	Time	Ratio
250	0	0
500	0	0
1000	0	0
2000	0	0
4000	0	0
8000	0,1	4,9
16000	0,3	3,8
32000	1,3	4,3
64000	5,4	4,3
128000	23,9	4,4
256000	110,1	4,6
512000	842,4	7,7

Table 1: Doubling ratio data for insertion sort.

From the data listed in Table 1, we can conclude that the insertion sort algorithm has a doubling ratio value of ±4.
In the last row, the doubling ratio value is 7,7. This is due to the large amount of time the algorithm needs to sort an array of this size. If the process of sorting is interrupted by other processes running on the computer, the amount of time could increase extensively.

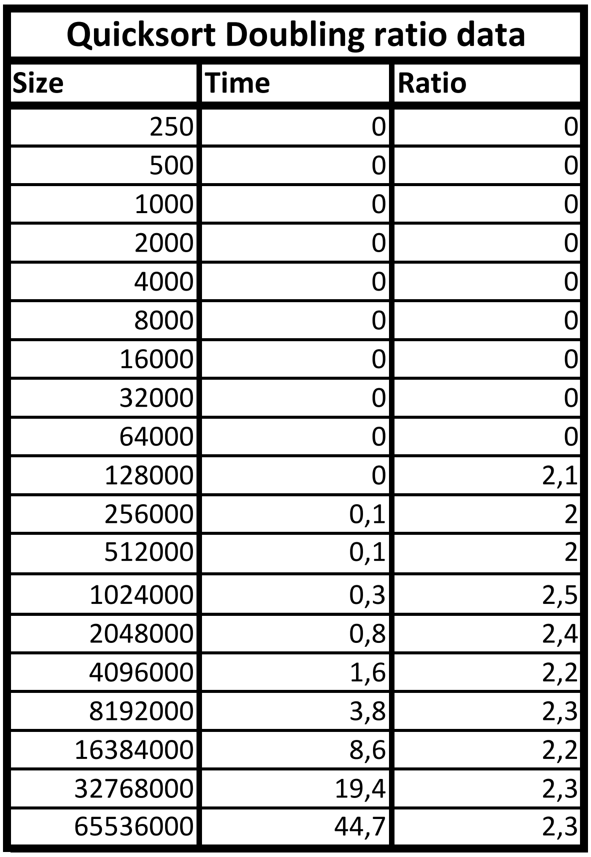
Section 2.1.2 states that the compares of insertion sort grow according $\sim \frac{N^2}{4}$. From this point we can derive the doubling ratio.

$$\frac{(2N)^2}{4} = \frac{4N^2}{4} = N^2$$

$$\text{ratio: } \frac{N^2}{\frac{N^2}{4}} = \frac{4N^2}{N^2} = 4$$

With this ratio we can predict the time needed to sort an array of size $2N$. e.g. the time needed to sort an array of size $512'000$ according to the time for sorting an array of size $256'000$, is $110s * 4 = 440s$ (see Table 1). This is significantly lower then the real sorting time but this is due to the fact stated above.

2.2.2 Quicksort



Quicksort Doubling ratio data

Size	Time	Ratio
250	0	0
500	0	0
1000	0	0
2000	0	0
4000	0	0
8000	0	0
16000	0	0
32000	0	0
64000	0	0
128000	0	2,1
256000	0,1	2
512000	0,1	2
1024000	0,3	2,5
2048000	0,8	2,4
4096000	1,6	2,2
8192000	3,8	2,3
16384000	8,6	2,2
32768000	19,4	2,3
65536000	44,7	2,3

Table 2: Doubling ratio data for quicksort.

From the data listed in Table 2, we can conclude that the quicksort algorithm has a doubling ratio value of ±2.

Section 2.1.4 states that the compares of quicksort grow according $\sim 2N \log N$. We apply the same method for deriving the doubling ratio as used in section 2.2.1.

$$2(2N)\log(2N) = 4N(\log N + \log 2)$$

$$\text{ratio: } \frac{4N(\log N + \log 2)}{2N \log N} = 2 * (\frac{\log 2}{\log N} + 1) = \frac{\log 4}{\log N} + 2$$

$$\Downarrow$$

$$\lim_{N \to +\infty} (\frac{\log 4}{\log N} + 2) = 2$$

Again we can predict the time needed to sort an array of size $2N$. e.g. the time needed to sort an array of size $131'072'000$, is $44{,}7s * 2 = 89{,}4s$ (see Table 2).

3 Conclusion

From the data of the executed experiments, we can conclude that our predictions, based on logical math, meet the experimental data.

In section 2.1, the experimental data stated that selection sort is an input insensitive sorting algorithm. In some cases this could be a desired algorithm, because it gives us certainty about the compares, which are always equal to $\sim \frac{N^2}{2}$ where N is the size of the array.

Unlike selection sort, insertion sort is very input sensitive. The data in Figure 1 confirms our expectations. With random data input, the amount of compares intersperse extensively while sorted data in Figure 2 is just a straight line.

We state that in the best case scenario, insertion sort has $N - 1$ compares and 0 exchanges. In the worst case scenario, compares and exchanges are equal to $\sim \frac{N^2}{2}$. We can conclude that insertion sort uses about $\sim \frac{N^2}{4}$ compares and exchanges.

Mergesort is a semi-input-sensitive sorting algorithm. Its compares will always be between $\frac{1}{2}N \log N$ and $N \log N$ which makes it both an efficient algorithm and a reliable algorithm.

The last algorithm we have tested, is the quicksort algorithm. Our experiment stated that this algorithm is rather input sensitive. Its worst case scenario has the same amount of compares that selection sort needs i.e. $\sim \frac{N^2}{2}$. However for random data, quicksort is a very reliable and fast algorithm. Therefore a random shuffle of the data protects against this case.

We can conclude that mergesort is a very efficient algorithm but due to the higher amount of data access, quicksort becomes superior to mergesort.

In section 2.2 we performed a doubling ratio experiment. This experiment stated that the doubling ratio of insertion sort was ±4 and the doubling ratio of quicksort was ±2. With these ratios it is possible to determine the time needed for these algorithms to sort an array of double the length of its previous array.

References

[1] Robert Sedgewick and Kevin Wayne. *Algorithms.* Addison-Wesley, 4th edition, 2011.