Sorting slowly: comparison between insertion, heap, merge and quick sort

Vladyslav Pallah

December 17, 2022

# Introduction

In this paper, we investigate a classical problem: sorting. The problem is to arrange a vector of *n* integers according to some total order which is computed in *O*(1). We use *<* as the total order.

In this paper we compare four comparison-based sorting algorithms: insertion sort, heap sort, merge sort and quicksort.

## insertion sort

Insertion sort repeatedly inserts elements into a sorted sequence.

Inserting an element into a sorted sequence is done by moving all elements that are larger than the value being inserted to the index one larger than currently occupied, starting from the largest. The value to be inserted is the moved to the array at index after the first value that is smaller or equal to the inserted value.

First, the element located at index 0 forms the sorted part of the array. The algorithm then performs *n*− 1 insertions, starting from the second element in the array to the last element.

The average complexity of insertion sort is *O*(*n*2). The insertion sort has a best case: the array sorted in an ascending order. In such a case, the complexity of insertion sort is *O*(*n*). The worst case of insertion sort is the array sorted in descending order. In such a case, the complexity of insertion sort is *O*(*n*2), same as average case.

The algorithm is stable and in-place.

## heap sort

The average complexity of selection sort is *O*(*n*2). There are no best and worst cases.

The algorithm is stable and in-place.

## merge sort

Bubble sort iterates over the input array, swapping two neighbouring values if they are not in order. One iteration over the input array moves the largest element to the last position in the array, and the last position can be excluded from any further comparisons. Since one iteration moves one element to its correct position, then we need no more than *n* iterations over the input array to sort the array. If no swaps happen during any pass of bubble sort, then the array is sorted and then algorithm is stopped.

The average complexity of bubble sort is *O*(*n*2). The bubble sort has a best case: the array sorted in an ascending order. In such a case, the complexity of bubble sort is *O*(*n*). The worst case of bubble sort is the array sorted in descending order. In such a case, the complexity of bubble sort is *O*(*n*2), same as average case.

The algorithm is stable and in-place.

## quick sort

Bubble sort iterates over the input array, swapping two neighbouring values if they are not in order. One iteration over the input array moves the largest element to the last position in the array, and the last position can be excluded from any further comparisons. Since one iteration moves one element to its correct position, then we need no more than *n* iterations over the input array to sort the array. If no swaps happen during any pass of bubble sort, then the array is sorted and then algorithm is stopped.

The average complexity of bubble sort is *O*(*n*2). The bubble sort has a best case: the array sorted in an ascending order. In such a case, the complexity of bubble sort is *O*(*n*). The worst case of bubble sort is the array sorted in descending order. In such a case, the complexity of bubble sort is *O*(*n*2), same as average case.

The algorithm is stable and in-place.

# Methodology

The algorithms were implemented in C++. The results were generated for two categories: sorted array (best case of insertion sort and bubble sort) and randomly shuffled arrays of values 0, 1, ..., *n*− 1.

The algorithms were tested on arrays of sizes from 100, 200, 300, ..., 10000. In both categories, each algorithm was given the same input data. Each vector size was tested 100 times.

The result for each size is the average time it took for each algorithm to sort the input array.

# Results

Graphs of results for the average case of all sorting algorithms are presented in Figures 1 and 2. Bubble sort takes several times longer than the other sorting algorithms. Insertion takes the least time, and selection sort takes about two times longer. For small array sizes shown in Figure 2, the results are the same.

Graphs of results for the best case of insertion sort and bubble sort are presented in Figure 5. Selection sort takes quadratic time regardless of how the array is sorted, and so it takes significantly longer than bubble sort and insertion sort, which take linear time. The same can be seen even for small arrays, in Figure 4.

Graphs of results for the best case of insertion sort and bubble sort are presented in Figure 5. Selection sort is excluded, since it makes it impossible to compare insertion and bubble sort. The graph is noisy, but general linear tendency of the results might be seen. On sorted arrays, insertion sort takes approximately two times as much time as bubble sort.

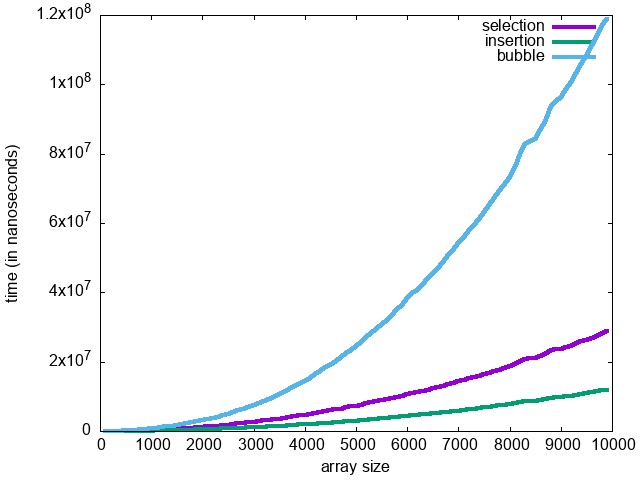


Figure 1: Average time each algorithm took to sort an array of randomly shuffled values from 0 to *n*− 1.

# Conclusions

Bubble sort performed best on sorted arrays, but in the average case, it performed several times worse than others. Selection sort performed worse than insertion sort only by a factor of about 2 in the average case, but it performed significantly worse than insertion sort.

We conclude, that insertion sort is best of the three sorts.

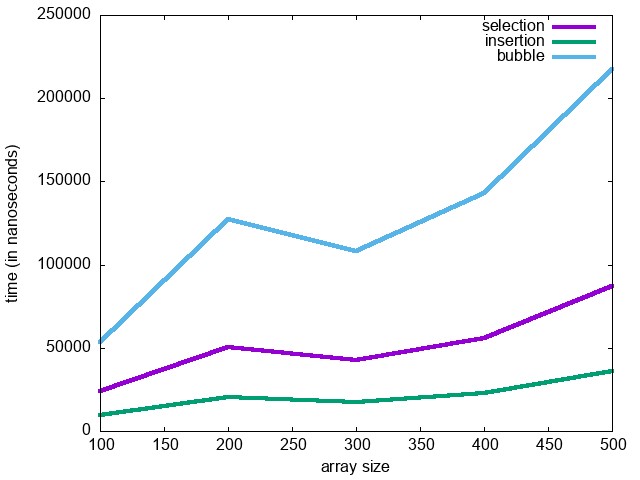


Figure 2: The graph shown in Figure 1 for small array sizes.

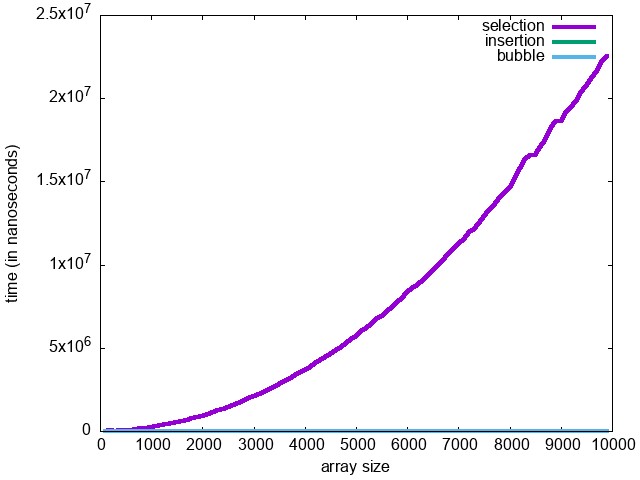


Figure 3: Average time each algorithm took to sort an array of sorted values

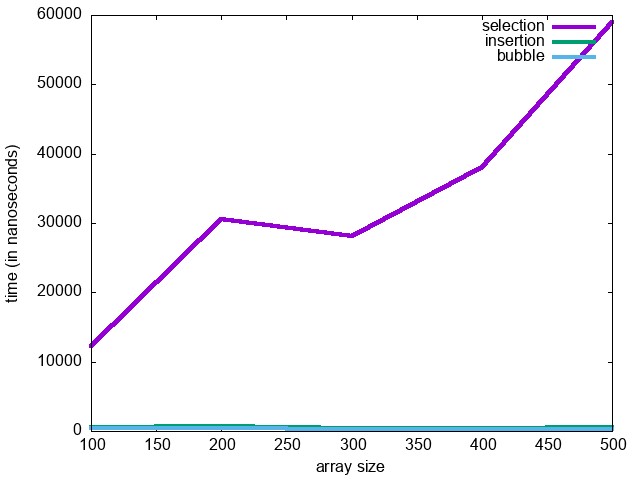


Figure 4: The graph shown in Figure 5 for small array sizes.

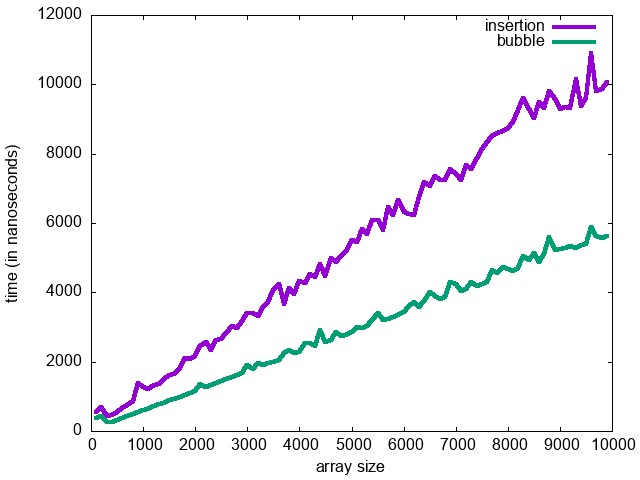


Figure 5: Average time each algorithm took to sort an array of sorted values, excluding selection sort.