Homework 4

Patrick Indri

June 7, 2019

1 Introduction

The aim of this homework is to implement some sorting algorithm (*Insertion Sort*, *Quick Sort*, *Heap Sort*, *Counting Sort*, *Radix Sort* and *Bucket Sort*) as well as the *Select Algorithm*. The implementations should be tested for correctness and performance: in particular, best and worst cases for *Quick* and *Select Sort* should be analysed. The `code/` folder contains the source codes implemented.

Compile and run

The `makefile` provided can be used to compile the code. Using `make` or `make all` the code gets compiled in a verbose version `main.x`, useful to test the implemented functions and see their output. Running `make benchmark` results in the compilation of `benchmark.x` which benchmarks the code and provides a cleaner output, ready to be analysed and plotted. Moreover, `run.sh` can be used to compile and run `benchmark.x`.

2 Implementation

Here, some insights of the various implementations are given.

`InsertionSort` trivially implements the sorting algorithm, make use of the utility function `Swap` to swap elements. The `QuickSort` function implements the *Quick Sort* algorithm using the first element of the to-be-sorted array as the pivot. In principle, for a

random array, the choice of the pivot is not relevant. The `HeapSort` implementation uses the binary heap implemented in *Homework 3* (treated as a max-heap). `CountingSort` requires the input array elements to be uniformly distributed in a bounded domain. This requirement is achieved by means of the `MAX_ELEM_VALUE` macro. `RadixSort` requires the input array to be constituted of elements of an equal number of digits. `RandomNDigitsFillArray` exploits modulo divisions by integer powers of `10` to achieve this result. The *Radix Sort* algorithm leans on a stable algorithm for sorting along the i-th digit: `RadixSort_aux` implements a *Counting Sort* algorithm, assuming `10` as the maximum number of digits. Lastly, `BucketSort` requires the input array to be uniformly distributed in $[0, 1)$. It has been implemented using an *Insertion Sort* implementation for doubles to order buckets. `BucketSort` handles buckets by means of the `AsVector` structure, which implements a dynamic vector. The vector is initialised as an empty container with a maximum capacity; `PushBackAsVector` is used to add elements to the vector. Once the vector has reached its maximum capacity, a `realloc` procedure doubles the capacity.

The *(Quick)Select Algorithm* has been implemented using a dichotomic approach. In particular, the pivot selection has been selected using the `rand()` function to avoid determinism: the median-of-three pivots approach can yield linear performance even on partially sorted arrays.

3 Benchmark

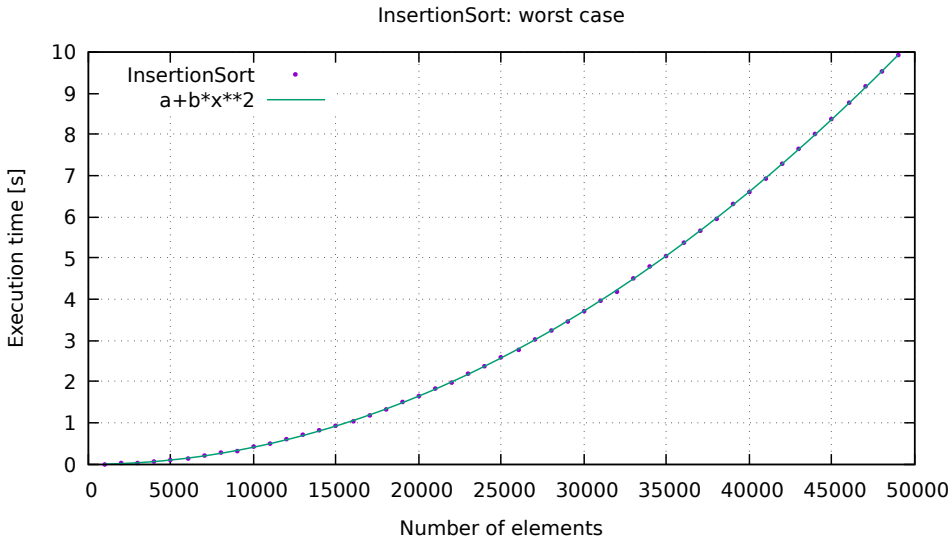
The assignment requires to verify the best and worst cases for both the *Insertion* and the *Quick Sort* algorithms. All the following results have been obtained by fitting the experimental data with the expected time complexities.

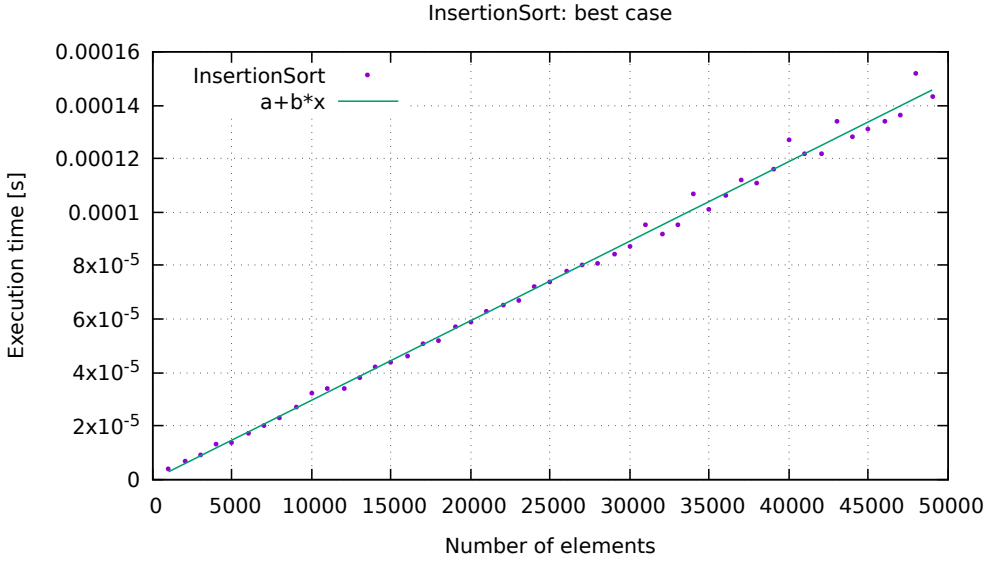
3.1 Insertion Sort

In the worst case, *Insertion Sort* takes time $O(n^2)$. This happens when the sorting algorithm is fed with a reversely sorted array.

On the other hand, the algorithm takes linear time when fed with an already sorted array.

The results satisfy the expectations, accurately reproducing the predicted behaviour. It should be noted how the worst case scenario not only scales worse but the execution is $\approx 10^5$ times slower if compared with the best case.





3.2 Quick Sort

In the worst case, *Quick Sort* takes time $O(n^2)$. This happens when the chosen pivot is always the smallest or the largest element or when all elements are equal. Only the latter case has been considered.

The best case performance is $O(n \log n)$. This scenario takes place when the selected pivot always splits in half the array. The `BestQuick` function uses an iterative procedure to produce the best array to feed to the sorting algorithm. In particular, it exploits the fact that the proposed *Quick Sort* implementation always chooses the first element as the pivot.

The predictions are accurately reproduced by the experimental results.