Sorting Algorithms

Marcello Capasso

April 2019

Abstract

This report aims to analyze the running time in the average case for the following sorting algorithms: Insertion sort, Quick sort, Heap sort, Counting sort, Radix sort, Bucket sort and Select algorithm. Moreover, the running time for the best and worst cases are analyzed for the Insertion and Quick sort algorithms.

1 Introduction

In order to evaluate the running time for the sorting algorithm, an array of random number is set as problem instance. For Insertion, quick, Heap, Counting, Radix and Select sorting the array elements are of integer type in the range (1, d) where $d = 1 * 10^5$, $1 * 10^6$. Bucket sort requires float numbers between 0 and 1. The size of the array (n) varies between 10 and $5 * 10^4$. In order to smooth out fluctuations, the average of 20 runs is taken.

2 Average Case

Figure 1 shows the running time as a function of the array size for the different sorting algorithms. A small comment about their performance is here given:

- Counting sort: this algorithm is by far the slowest. This is because this implementation works well with array elements of small magnitude. In fact the algorithm first finds the maximum value (m) contained in the array and then allocates in memory an array of m elements. This yields to a high running time as most of it is wasted allocating space in memory. A comparison with figure 2 immediately shows how reducing d by an order of magnitude already highly improves the running time.
- Bucket sort: this algorithm is the second worst one. Again, this is due to the high memory allocation needed for the Buckets. In fact, for each Bucket an array of n elements is allocated and initialized. The algorithm can be ran faster if assumptions about the maximum number of elements in each bucket are known.
- Insertion sort: this algorithm performs well with arrays of small size, however, for greater n, the quadratic nature of the its operation and space complexity takes over.
- Select algorithm: this algorithm is the third best for small arrays. It should be considered on itself as it returns the (n/2+1)th smallest element of the array and does not properly sort it.
- Heap sort: this algorithm is in the top three of the implemented ones for small and large arrays. It has a very similar behaviour as Quick sort.
- Quick sort: this algorithm is considered to be one of the fastest known. For small array this is clearly the case, whereas for large arrays Radix sort competes with it.
- Radix sort: as already mentioned it competes with Quick sort for large arrays. However, this could be due to fluctuations. The current implementation is based on Counting sort, using Quick sort would probably improve the running time.

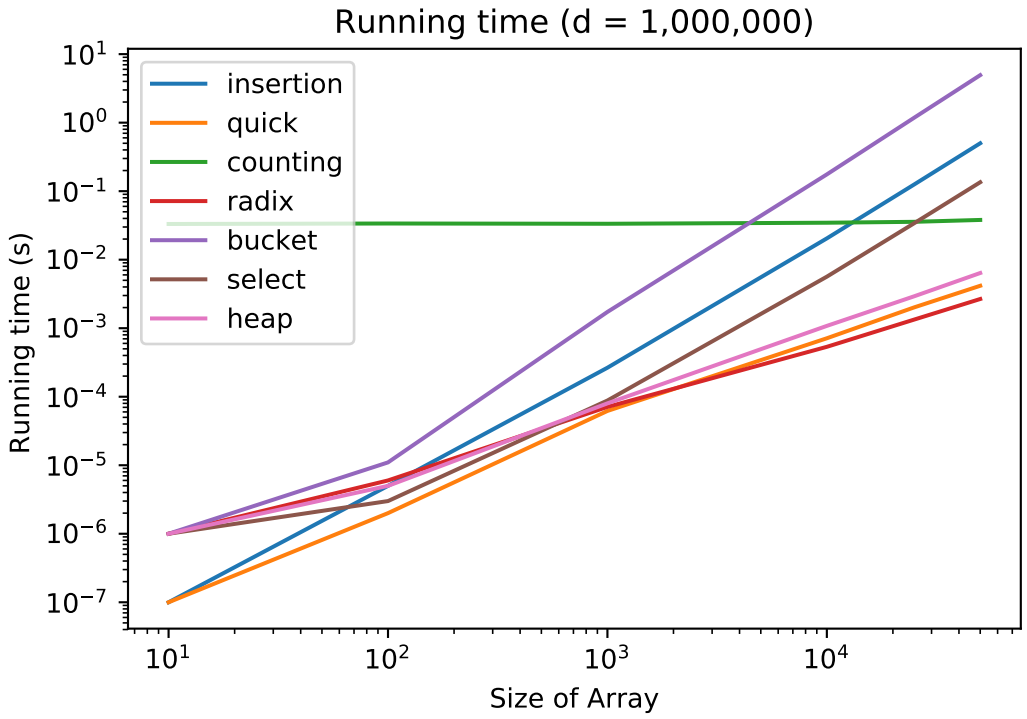

Figure 1: Running time in seconds ($d = 1 * 10^6$)

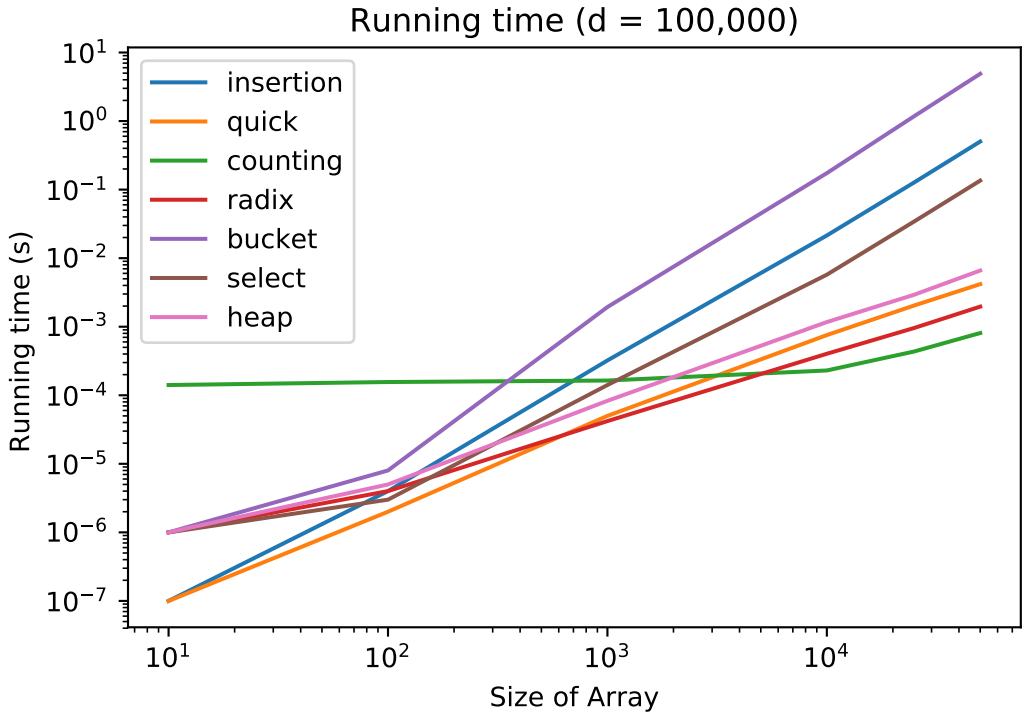

Figure 2: Running time in seconds ($d = 1 * 10^5$)

3 Best and Worst case scenarios

This paragraph aims to prove that Insertion sort and Quick sort take quadratic time in the worst case scenario (i.e. when the array is sorted in reverse order) and linear and $n * log(n)$ respectively for the best case scenario. The behaviours of the Insertion and Quick sort algorithms are analyzed in sub paragraphs 3.1 and 3.2 respectively.

3.1 Insertion Sort

For the worst case scenario, the running time of Insertion sort scales as

$$t(n) = an^2 + bn$$

where

$$a \approx 3.9 * 10^{-10},$$
$$b \approx 3.5 * 10^{-08}.$$

On the other side, in the best case scenario (i.e. when the array is already correctly sorted) the running time scales as

$$t(n) = an$$

where

$$a \approx 1.1 * 10^{-9}$$

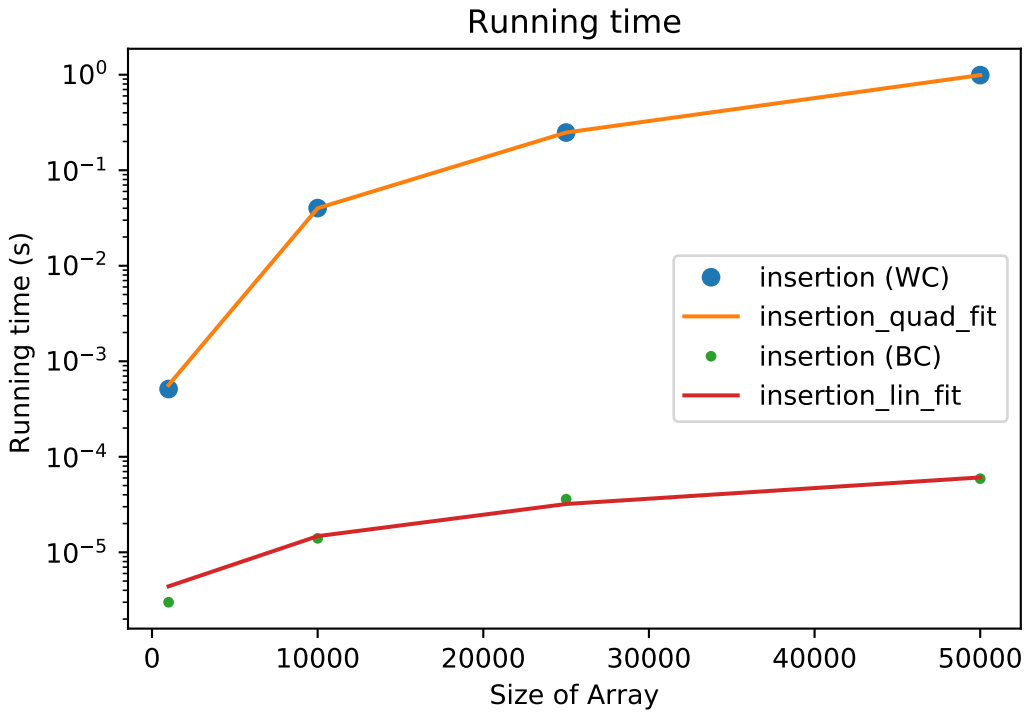

Figure 3: Running time in seconds. Worst case and Best case scenarios + quadratic and linear fits.

3.2 Quick Sort

For the worst case scenario (i.e. the array is already sorted in reverse order), the running time of Quick sort scales as

$$t(n) = ax^2 + bx$$

where

$$a \approx 3.7 * 10^{-10},$$
$$b \approx 1.1 * 10^{-6}$$

On the other side, in the best case scenario (i.e. when the pivot is always chosen to be the median of the array to be partitioned), the running time scales as

$$t(n) = a * (n) * log(n)$$

where

$$a \approx 2.5 * 10^{-9}$$

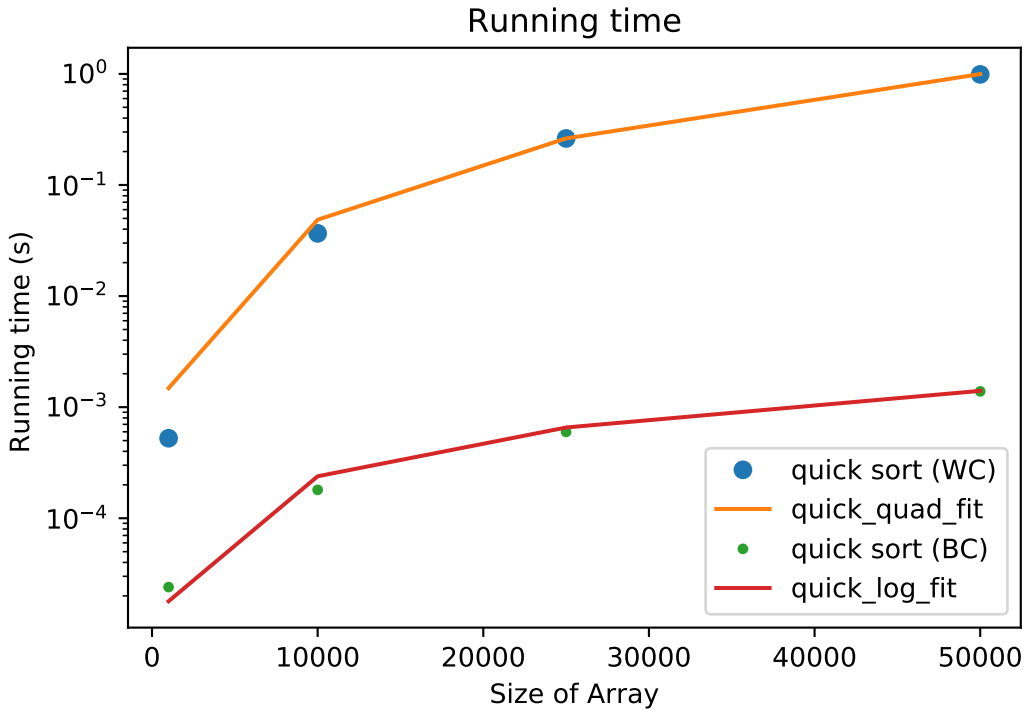

Figure 4: Running time in seconds. Worst case and Best case scenarios + quadratic and n*log(n) fits.