Divide-and-Conquer Sorting Algorithms

Jose Torres Postigo

2023-10-20

1 Objectives

The goal of this lab exercise is to practice with the Divide and Conquer method for implementing algorithms, considering the problem of sorting a list of elements. For this lab session, it is requested to implement three sorting methods: QuickSort, MergeSort and SelectionSort, this last one for comparison purposes.

There will be also an analysis for every algorithm implementing regarding the time required for sorting lists, depending on the number of elements contained in the list.

2 Experimental Setup

For this experiment, every algorithm is run with through a class given by the lecturer, which is going to feed the algorithm with lists and time how long the algorithm takes to accomplish the sort of the list.

For the QuickSort and Mergesort, the initial size of the list given is 100,000, increasing over every iteration up to 2,262,718. For the SelectionSort, this initial value is 2,000 up to 45,233.

Table 1: Computational environment considered.

CPU	Intel® Core™ i5 11600KF, 16GB
OS	Windows 11 Home 22H2
Java	openjdk 17.0.7 2023-04-18

3 Empirical Results

A summary of the experimental results is provided in Tables 3–4 in the Appendix, along with the statistical fitting of the data to different growth models.

In Figure 1 can be seen that the rate of growth for SelectionSort algorithm is much higher than the following algorithms. It takes the same time to sort a list of 40,000 elements with SelectionSort as a list of 750,000 elements aprox. with MergeSort, or a list of 1,125,000 elements with QuickSort. Thus, is not interesting to take a look at this algorithm for this lab session, only for comparative purposes.

In Figure 2 we can see a exponential growth with a lower rate than the previous algorithm. With a complexity of $\Theta(n \log n)$, the algorithm can sort lists with 2,000,000 elements in less than 4 seconds. Comparing the line graph with the empirical measurement obtained, can be said that is very ccurate for almost every result obtained.

Taking a look at Figure 3, it can be said that this is the quickest sorting algorithm by far. This algorithm takes about 2 seconds for sorting a 2,000,000 elements list. This is half of the time it takes the MergeSort algorithm, even though they have the same time complexity $\Theta(n \log n)$

Taking a look at the summary tables of the experimental results for the algorithms, we can certainly say that is very precise the model fittings as the standard errors are almost insignificant.

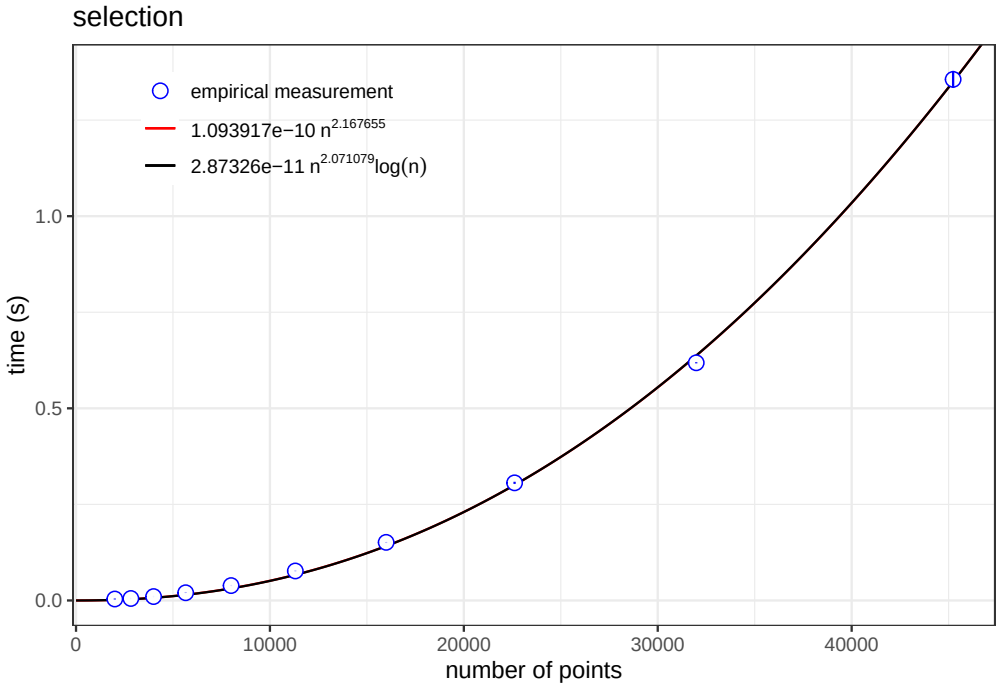

Figure 1: Time required for sorting lists of increasing size using selection.

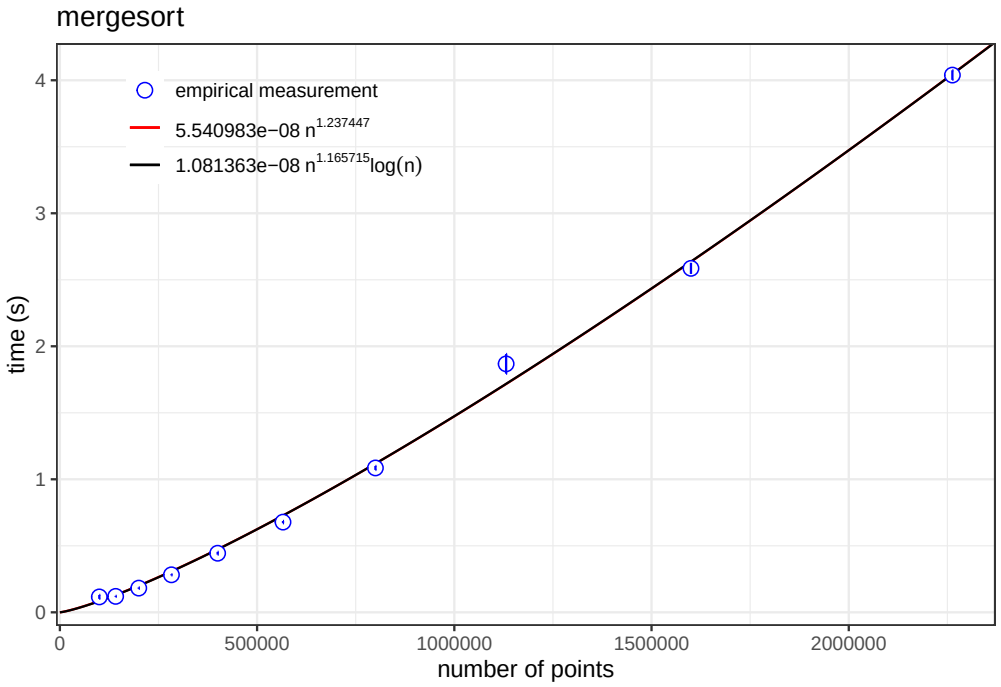

Figure 2: Time required for sorting lists of increasing size using mergesort.

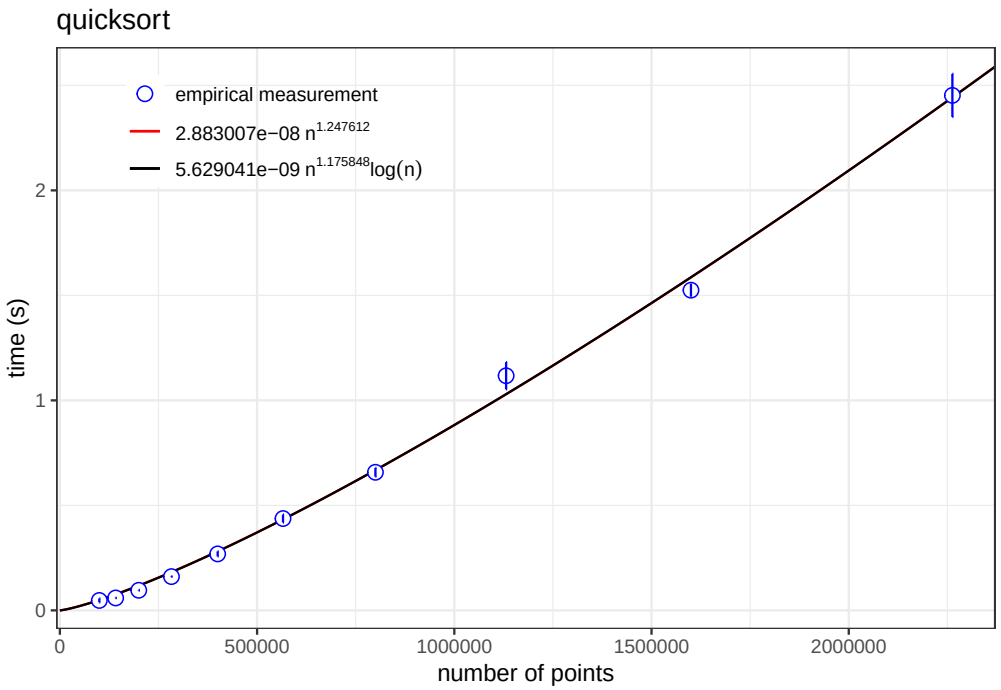

Figure 3: Time required for sorting lists of increasing size using quicksort.

4 Discussion

Provide your interpretation of the results: discuss whether the results match the theoretical predictions, whether some algorithm is better in practice than others, etc.

The empirical results I obtained highly match the theoretical predictions in general. There are some measurements which are not as fitted to the model as it should be, but this does not necessarily mean that there is a problem with the model nor with the implementation, but it could be the hardware used to take the readings.

Comparing the algorithms discussed in this lab session, there is a clear winner by far: The QuickSort. And not only this is a very fast algorithm, but also surprises me that the implementation is not so hard to work out.

Maybe there is algorithms faster than QuickSort, but only in a setup with the conditions necessary and few elements. On random sets of data, It seems to me that the QuickSort algorithm is the fastest.

A Appendix

A.1 Numerical Data

Mean and standard error of sorting lists of different sizes. All times measured in seconds.

```
## Warning: package 'knitr' was built under R version 4.3.1
```

Table 2: Summary of the experimental results for selection.

list size	mean time	std. error
2000	3.81e-03	1.20e-03
2828	5.15e-03	3.60e-05
3999	1.01e-02	2.90e-05
5655	2.01e-02	3.51e-04
7997	3.87e-02	1.29e-04
11309	7.68e-02	3.91e-04
15993	1.51e-01	3.04e-04
22617	3.06e-01	2.81e-03
31985	6.18e-01	1.49e-03
45233	1.36e+00	2.21e-02

Table 3: Summary of the experimental results for mergesort.

list size	mean time	std. error
100000	1.16e-01	1.47e-02
141421	1.20e-01	3.85e-03

list size	mean time	std. error
199999	1.82e-01	5.20e-03
282841	2.82e-01	4.25e-03
399997	4.44e-01	1.04e-02
565681	6.78e-01	8.57e-03
799993	1.09e+00	1.49e-02
1131360	1.87e+00	7.38e-02
1599984	2.59e+00	4.20e-02
2262718	4.04e+00	3.62e-02

Table 4: Summary of the experimental results for quicksort.

list size	mean time	std. error
100000	4.74e-02	9.41e-03
141421	5.92e-02	9.61e-04
199999	9.57e-02	1.53e-03
282841	1.61e-01	1.44e-03
399997	2.69e-01	1.21e-02
565681	4.37e-01	1.89e-02
799993	6.58e-01	2.09e-02
1131360	1.12e+00	6.47e-02
1599984	1.52e+00	3.13e-02
2262718	2.45e+00	1.03e-01

A.2 Model Fitting

```
## selection
## ---------
##
## Formula: val ~ a * size^b
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
## a 1.094e-10  3.351e-11   3.264   0.0115 *
## b 2.168e+00  2.884e-02  75.161 1.09e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.009288 on 8 degrees of freedom
##
## Number of iterations to convergence: 19
## Achieved convergence tolerance: 1.714e-06
##
##
## Formula: val ~ a * size^b * log(size)
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
```

```
## a 2.873e-11  9.017e-12   3.186   0.0129 *
## b 2.071e+00  2.954e-02  70.100 1.91e-12 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.009506 on 8 degrees of freedom
##
## Number of iterations to convergence: 19
## Achieved convergence tolerance: 1.756e-06
##
## mergesort
## ---------
##
## Formula: val ~ a * size^b
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
## a 5.541e-08  2.430e-08    2.28   0.0521 .
## b 1.237e+00  3.050e-02   40.58  1.5e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.06326 on 8 degrees of freedom
##
## Number of iterations to convergence: 10
## Achieved convergence tolerance: 1.873e-06
##
##
## Formula: val ~ a * size^b * log(size)
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
## a 1.081e-08  4.717e-09   2.292   0.0511 .
## b 1.166e+00  3.033e-02  38.436 2.31e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.06283 on 8 degrees of freedom
##
## Number of iterations to convergence: 10
## Achieved convergence tolerance: 1.771e-06
##
## quicksort
## ---------
##
## Formula: val ~ a * size^b
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
## a 2.883e-08  1.358e-08   2.122   0.0666 .
## b 1.248e+00  3.275e-02  38.090 2.48e-10 ***
```

```
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.04058 on 8 degrees of freedom
##
## Number of iterations to convergence: 23
## Achieved convergence tolerance: 2.738e-06
##
##
## Formula: val ~ a * size^b * log(size)
##
## Parameters:
##    Estimate Std. Error t value Pr(>|t|)
## a 5.629e-09  2.641e-09   2.131   0.0657 .
## b 1.176e+00  3.262e-02  36.052 3.84e-10 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 0.04035 on 8 degrees of freedom
##
## Number of iterations to convergence: 24
## Achieved convergence tolerance: 2.324e-06
```