

## SE 2XB3 Group 4 Report 3

Huang, Kehao	Jiao, Anhao
400235182	400251837
huangk53@mcmaster.ca	jiaoa3@mcmaster.ca
L01	L01

Ye, Xunzhou  
400268576  
yex33@mcmaster.ca  
L01

5 February 2021

# 1 Quicksort

## 1.1 In-Place Version

Our implementation has its natural in-place advantage over the given implementation which uses auxiliary memory to store different partitions for each recursion call. Specifically, our in-place implementation swaps elements in the array and uses two parameter `low` and `high` to mark the array partition on which a recursion call should work. The amount of memory used for the whole sorting process is independent of the input size. In this case, it is the length of the input array. On the other hand, the given non-in-place implementation copies elements from the input array to fresh allocated auxiliary arrays. Each auxiliary array is used as the new partition for the subsequent recursion call. And the returned sorted array is a concatenation of two sorted partitions and the pivot. Both the element copying action and list concatenation are costly, in terms of both time and space complexity.

A test on the average runtime of both versions of quicksort is then carried out. For  $n$  on the scale from  $10^4$  to  $10^7$ , an array of  $n$  random numbers is passed to both implementations. The runtime is plotted on a semi-log graph as shown in Figure 1. The experimental result did not precisely match our prediction. From  $n = 10^4$  to approximately  $n = 10^{5.5}$ , the in-place version has a slight longer runtime than the non-in-place version. Figure 2 is a zoomed-in view of the plot, which demonstrates the shorter runtime of the non-in-place version. As  $n$  increases from  $10^{5.5}$ , the in-place quicksort gradually starts to outperform the other. The disadvantage of using auxiliary arrays and deep copying data around memory becomes obvious as  $n$  passes  $10^6$ .

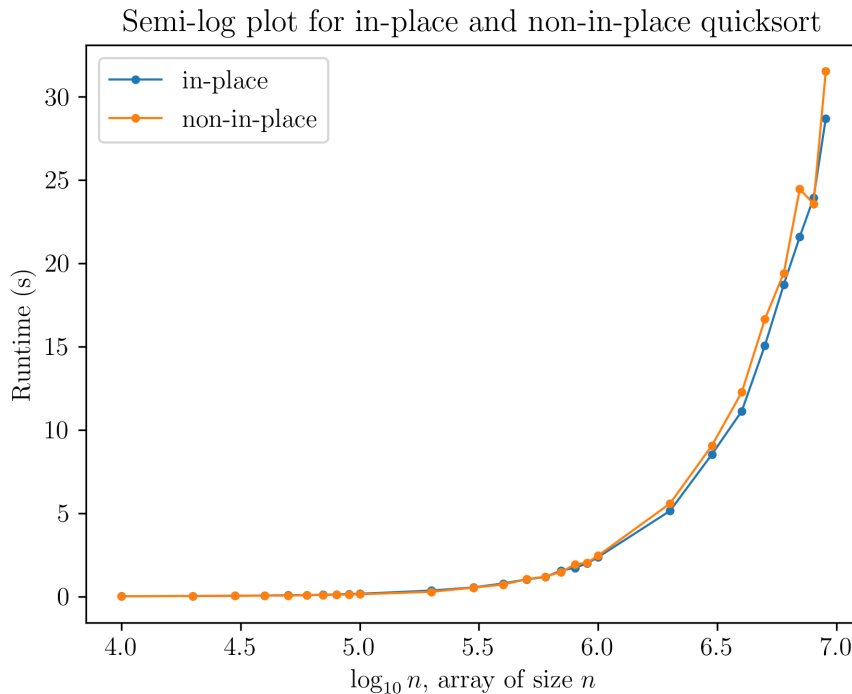


Figure 1: In-place and non-in-place quicksort comparison

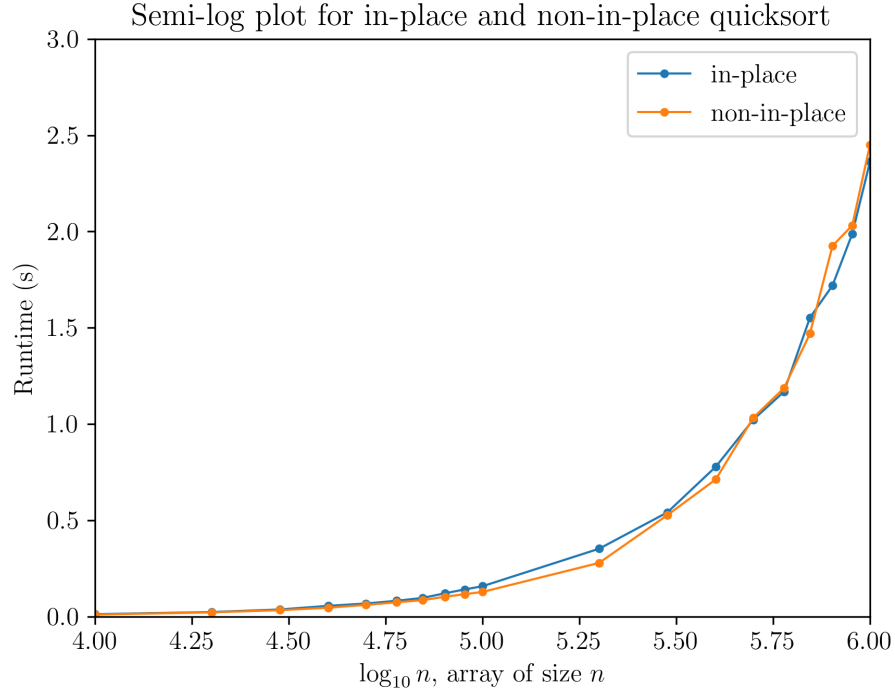


Figure 2: Quicksort versions comparison on lower scale  $n$

To quantify the performance difference between the two versions, we used a rather simple model. Either version of quicksort is known to have a complexity of  $\mathcal{O}(n \lg n)$ . We use  $c*n \lg n$  as an approximate model of the imperial runtime. Thus we take the average of the differences of the  $c$  constants of the two versions as the quantified performance difference. From the same data set plotted above, for  $10^4 \leq n < 10^7$ , the non-in-place version is on average 4% faster than the in-place version.

In conclusion, the non-in-place quicksort is slightly faster than the in-place one in practice. However, the in-place version has an observable speed improvement for an input size  $n > 10^6$ .