Comparison of Heapsort and Quicksort

The current document will compare the performance of quicksort and heapsort for various input sizes using the following two criteria: run time and the number of comparisons and swaps.

In order to generate the graphs below the following procedure was used:

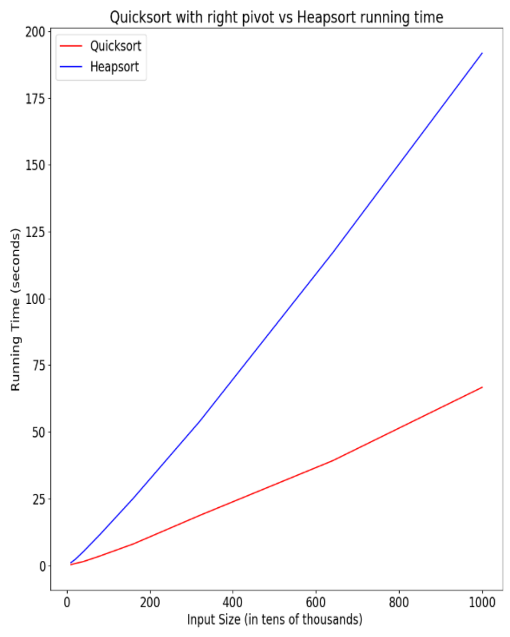
1) For each run of the experiment, 8 different input sizes were chosen in the range [100 000, 10 000 000]. The sizes remained the same for all of the runs.
2) For each input size, an array of random integers in the range (1, 10^12) was generated. It is important to specify that the given range is reasonably large to compensate the fact that random integers are used and not random real numbers**. (As discussed in the section that inspects running time, the size of the interval of random integers is relevant for the comparison)**
3) Both quicksort and heapsort were applied to the same array generated at step 2) and execution time and the number of comparisons and swaps were recorded and plotted for each run.
4) The experiment was repeated 10 times. The average time and number of comparisons and swaps was calculated for each of the input sizes and plotted together with the standard deviation.

The algorithms were implemented using Python3. It is important to note that the right most element of the array was chosen as the pivot element for the quicksort.

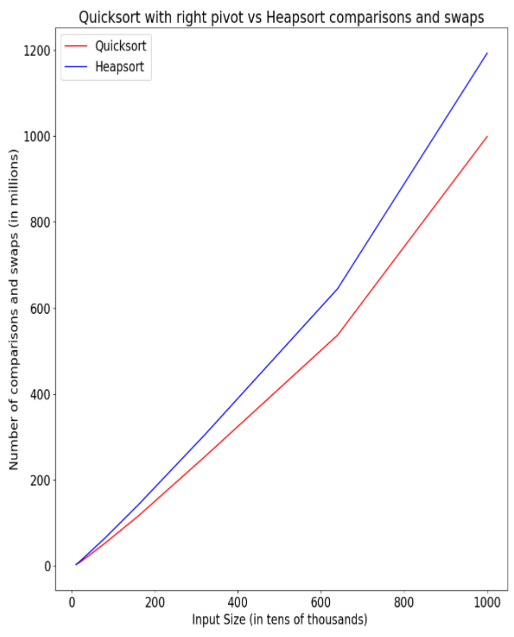
The next pages of this document contain the results of the ten runs of the experiment, followed by the graphs of the averages and by an analysis of the results.

Running Time

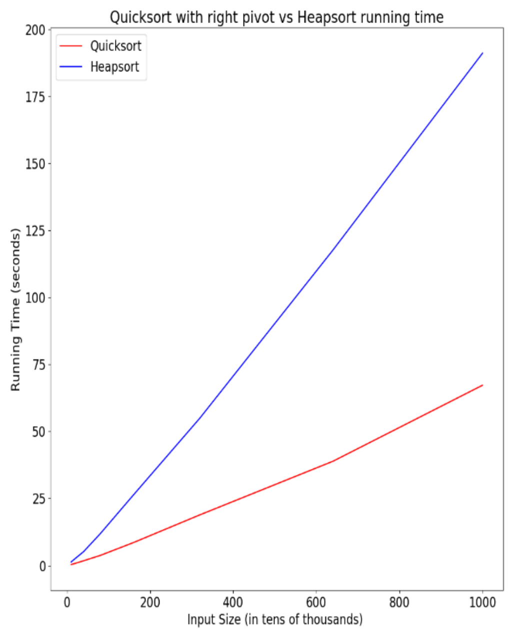
Number of Comparisons and Swaps



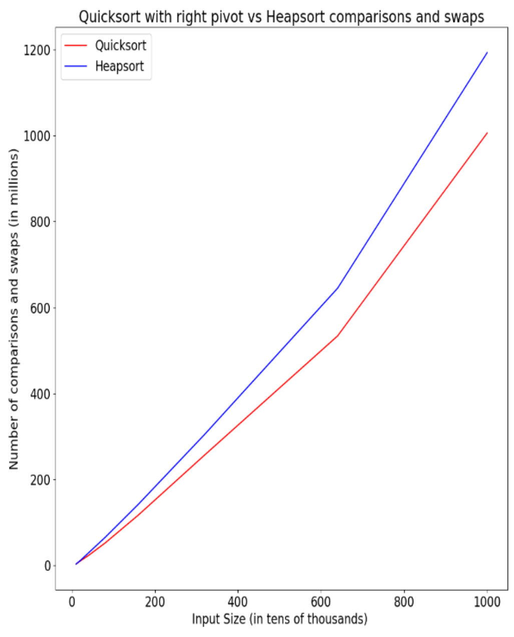
Run 1



Run 1



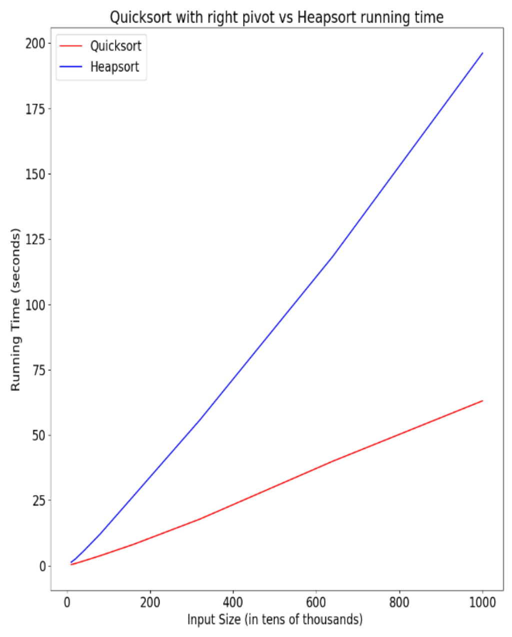
Run 2



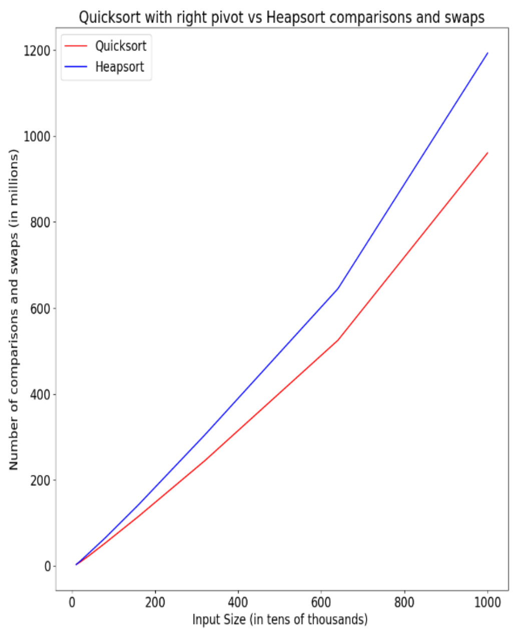
Run 2

Running Time

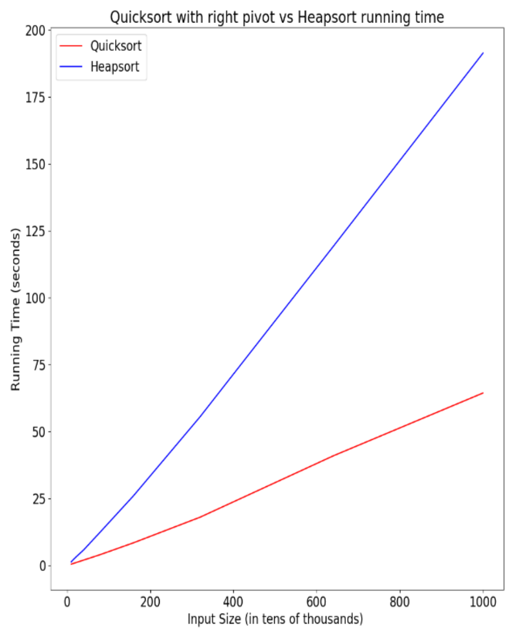
Number of Comparisons and Swaps



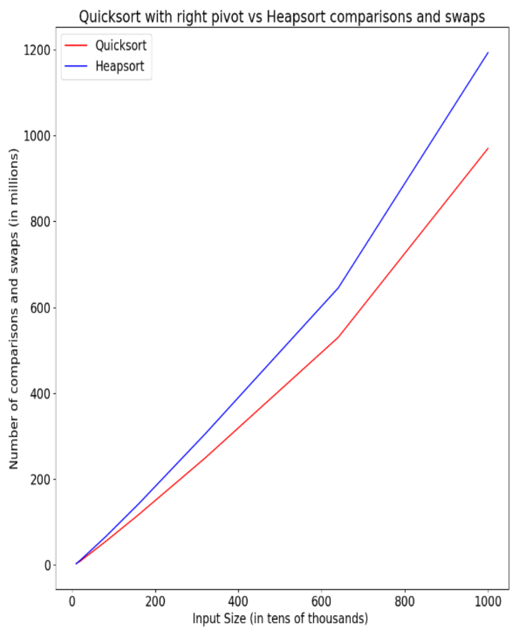
Run 3



Run 3



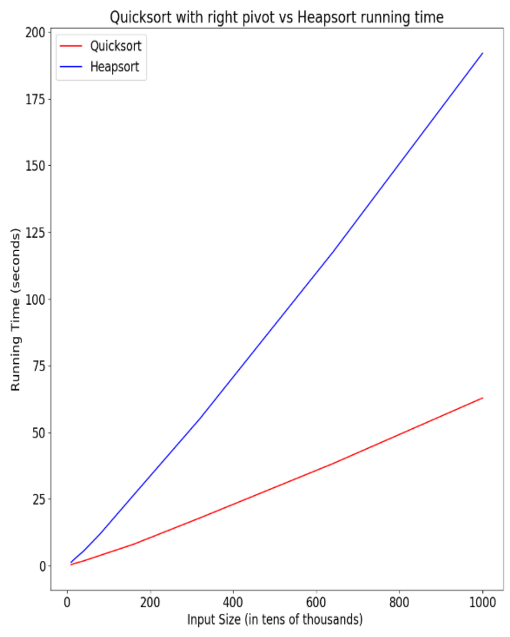
Run 4

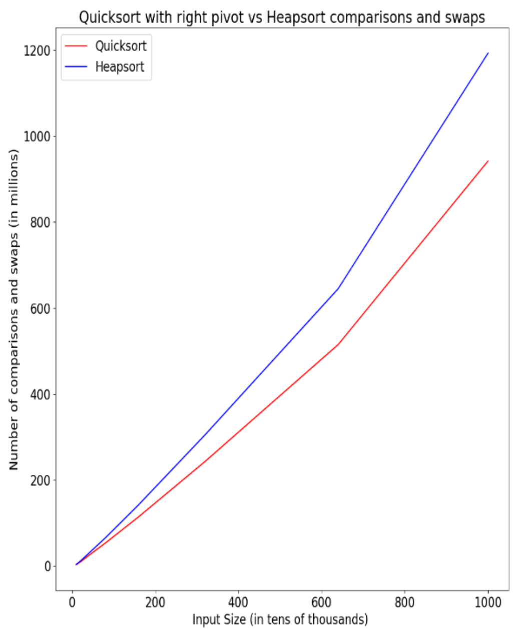


Run 4

Running Time

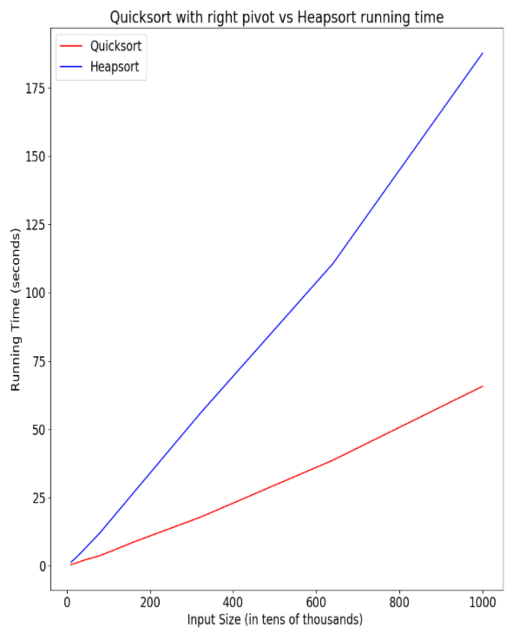
Number of Comparisons and Swaps

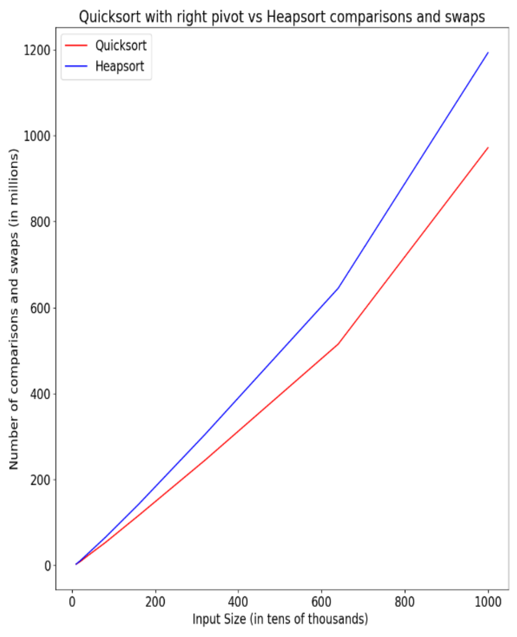




Run 7

Run 7



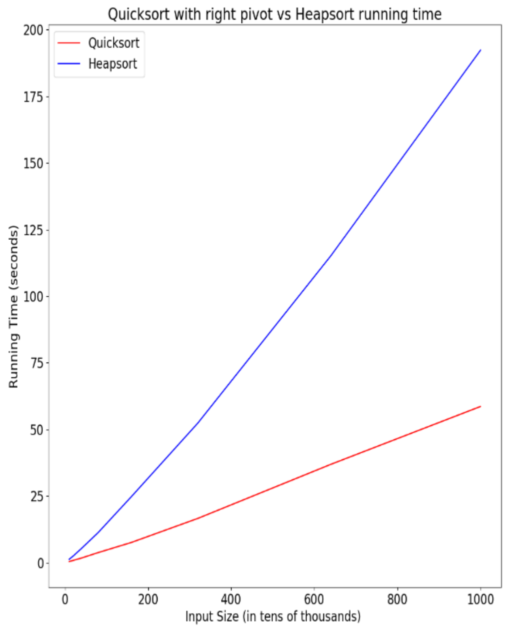


Run 8

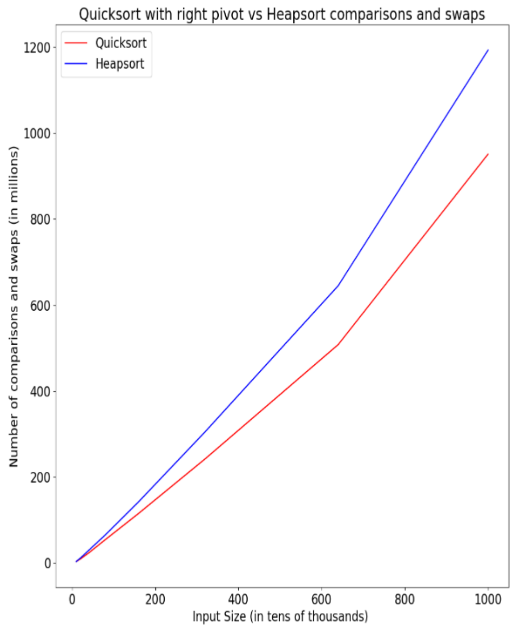
Run 8

Running Time

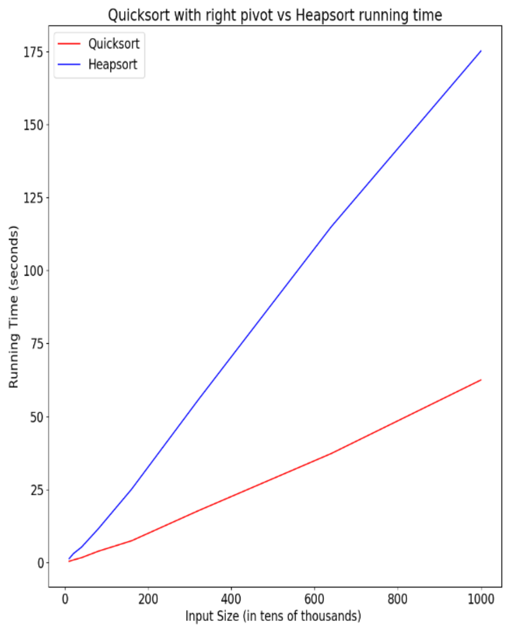
Number of Comparisons and Swaps



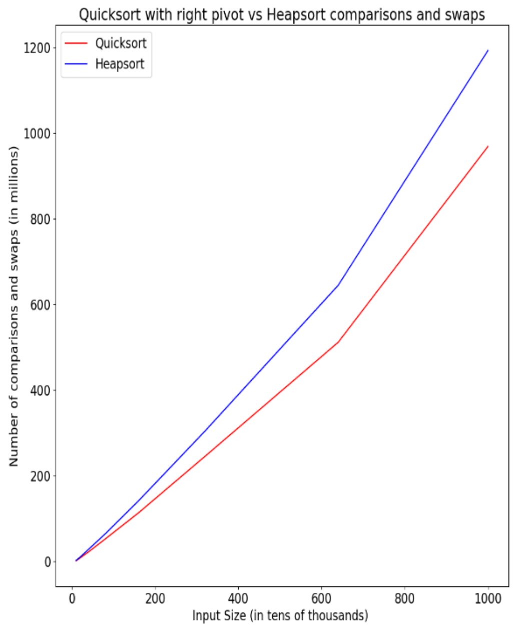
Run 9



Run 9



Run 10



Run 10

Average Running Time

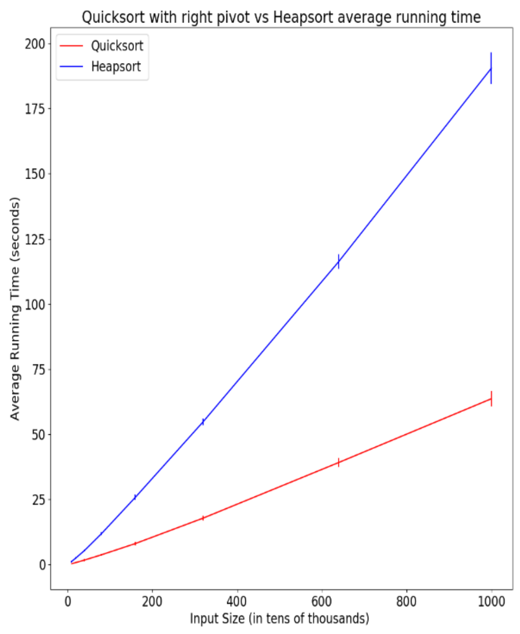


Figure 1.1

Average Number of Comparisons and Swaps

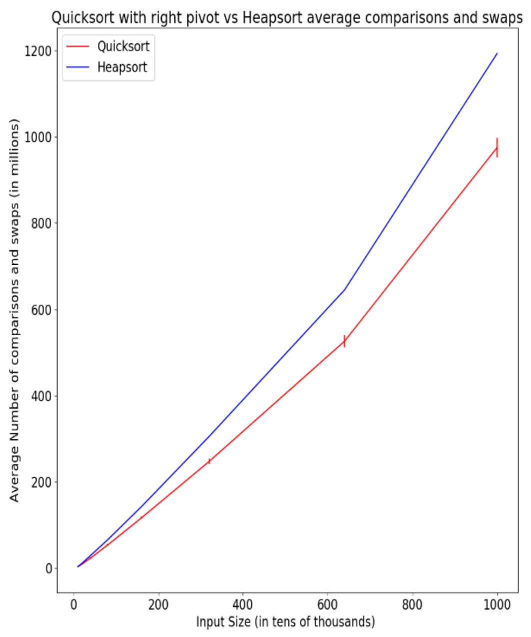


Figure 1.2

The above graphs can be summarized in two tables, showing the actual value of the averages and deviations.

Input Size	Heapsort average time (in seconds)	Quicksort average time (in seconds)	Heapsort time deviation	Quicksort time deviation
100 000	1.13	0.32	0.32	0.07
200 000	2.42	0.69	0.21	0.03
400 000	5.13	1.74	0.88	0.27
800 000	11.34	3.37	1.29	0.51
1 600 000	24.95	7.54	3.58	1.40
3 200 000	53.71	15.97	7.35	1.21
6 400 000	107.15	34.82	7.77	2.68
10 000 000	183.60	53.93	19.06	1.31

Table 1

Input Size	Heapsort average comparisons and swaps (in millions)	Quicksort average comparisons and swaps (in millions)	Heapsort comparisons and swaps deviation	Quicksort comparisons and swaps deviation
100 000	3.82	3.11	0.00072	0.14616
200 000	11.96	9.84	0.00106	0.28380
400 000	29.24	24.11	0.00113	0.50118
800 000	65.81	54.45	0.00217	1.10734
1 600 000	142.95	117.91	0.00406	1.01792
3 200 000	305.22	252.47	0.00334	1.79410
6 400 000	645.76	532.20	0.00629	7.74822
10 000 000	1193.74	982.37	0.00720	4.99232

Table 2

As stated before, the comparison of the two sorting algorithms is based on two criteria: running time and the number of comparisons and swaps. Let's first consider the number of comparisons and swaps.

By simply looking at the graphs for each run, it can be noticed that quicksort performs less comparisons and swaps than quicksort. One would also observe that as the input size gets bigger, the difference between the number of comparisons and swaps also increases. It is known that both algorithms have an average complexity of $O(\mathrm{n}\log n)$, therefore, the fact that the input size and the difference between the number of comparisons and swaps are directly proportional suggests that quicksort has smaller constant factors than heapsort. These two very simple remarks are supported by the graph of the average and the tables. It can be clearly seen that on random inputs, quicksort tends to perform better than heapsort.

By taking a closer look at the graphs and the tables above, there is an interesting phenomenon that emerges. The standard deviation of the number of comparisons and swaps for heapsort is extremely small, as seen in Table 2, and is not even visible on the graph. That shows that in average, heapsort has a nearly constant number of comparisons and swaps. This suggests that its performance is not affected as much by the actual values of the input array, but rather by the size of the array. On the other hand, in the case of quicksort, the comparisons and swaps varies, proving that the values of the input array has a much more noticeable effect on its performance. This observation will be detailed later in this section. In order to support this statement, the following graphs show the average number of comparisons and the average number of swaps independently, based on the same inputs.

Average Number of Comparisons

Average Number of Swaps

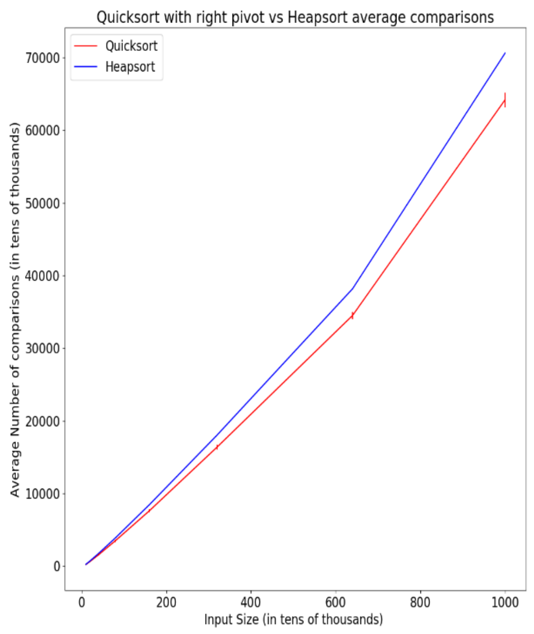


Figure 1.3

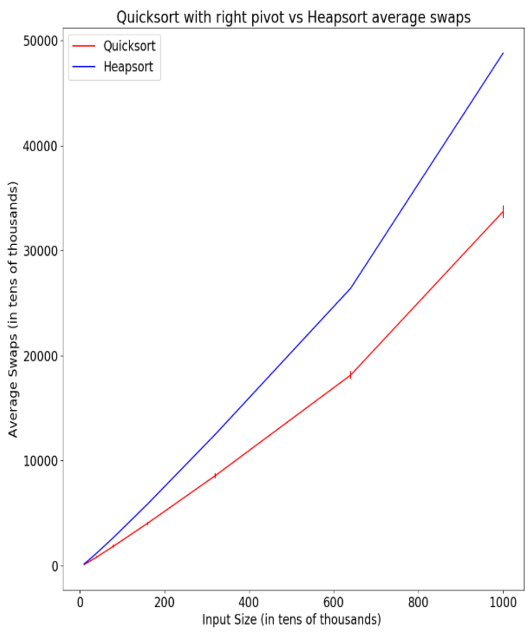


Figure 1.4

As seen in the above graphs, the same behavior applies when comparing the number of swaps and comparisons separately. A factor that contributes to the notable difference between the number of swaps is that in the case of heapsort, all the elements in the array are going to be swapped in the process of ordering, whilst quicksort does not swap elements that are already sorted.

There is a direct link between the number of comparisons and swaps and the running time of the two algorithms. Even though comparisons may not have an important effect on running time, swaps are considerably more time consuming and tend to impact the execution time. By looking at Figure 1.4, one can notice that heapsort performs in average 40% more swaps for an array of 10M elements, which significantly increases the running time. The correlation between the number of swaps and running time can be noticed by looking at Figure 1.1, which shows that for an array of 10M elements, quicksort runs 4 to 5 times faster than heapsort, this trend also holding for input sizes greater than 5M.

Another factor that causes quicksort to perform better than heapsort is the way in which the algorithms interact with the cache memory. Quicksort has a sequential access pattern, accessing contiguous memory locations, allowing a very effective use of caching and therefore speeding up the execution of the algorithm. On the other hand, heapsort does not have such an access pattern. In fact, heapsort jumps from one memory address to another, causing a large number of cache misses once the size of the input exceeds the size of the CPU cache.

So far, the running time comparison only took into consideration the case when the input array had random values (i.e. average case for both quicksort and heapsort). It is important to also consider the worst-case of the algorithms. For heapsort, the worst-case is still $O(\mathrm{n}\log n)$. For quicksort, the worst-case complexity is $O(n^2)$. The worst-case occurs when partitioning is always completely unbalanced. This causes the partition function to create two sub-arrays with n-1 and 0 elements. An example that causes quicksort to run in $O(n^2)$ is when the array is already ordered. However, the worst-case can be avoided by changing the pivot from the right most element to the middle element.

Another case that determines quicksort to run slower than the average is when considering the sorting of integer numbers. If the range of integers is too small, the input array will contain many repetitions and the partition function will create unbalanced sub-arrays, causing quicksort to run considerably slower. An example to illustrate this behavior can be seen in the graphs below. The graphs represent the average of ten runs of quicksort in which the random range was set to [1, 10^12] for the blue graph, while the red graph had a range of only 1,10^4].

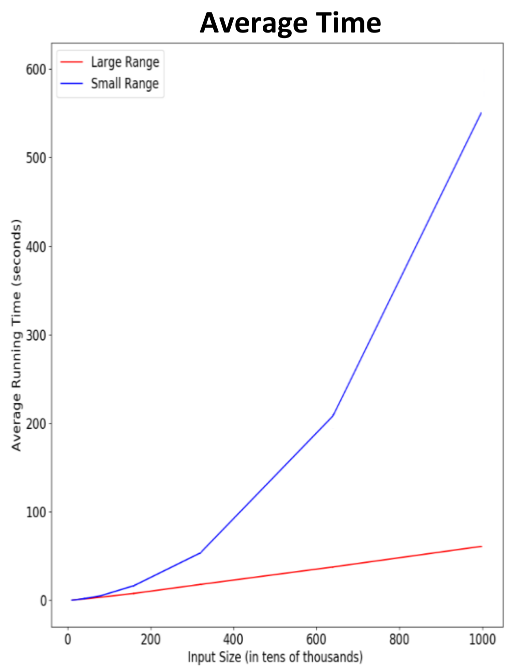


Figure 1.5

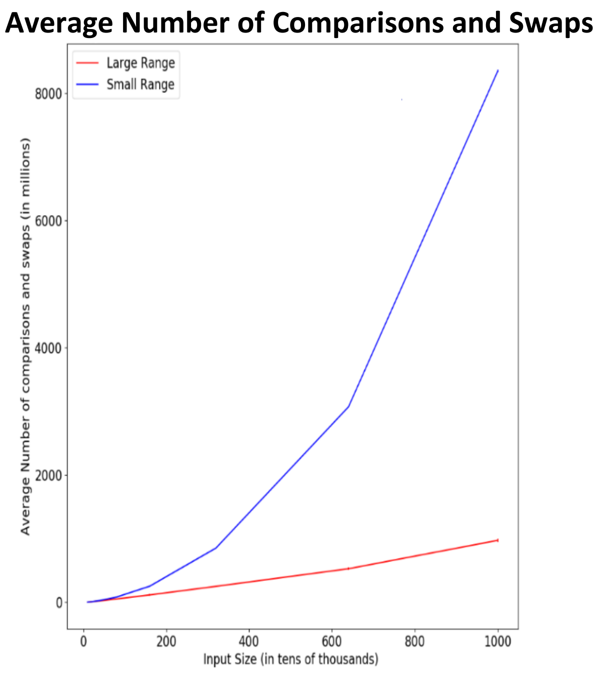


Figure 1.6

Having a considerable amount of repetitions in the input array causes heapsort to perform better than quicksort, as seen in Figures 1.7 and 1.8, that represent the average of 10 runs of quicksort and heapsort with the input array containing integers in the range [1, 10^4].

Average Time

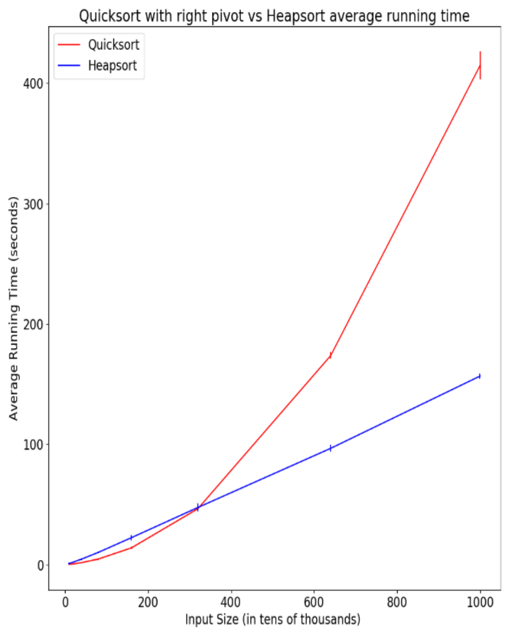


Figure 1.7

Average Number of Comparisons and Swaps

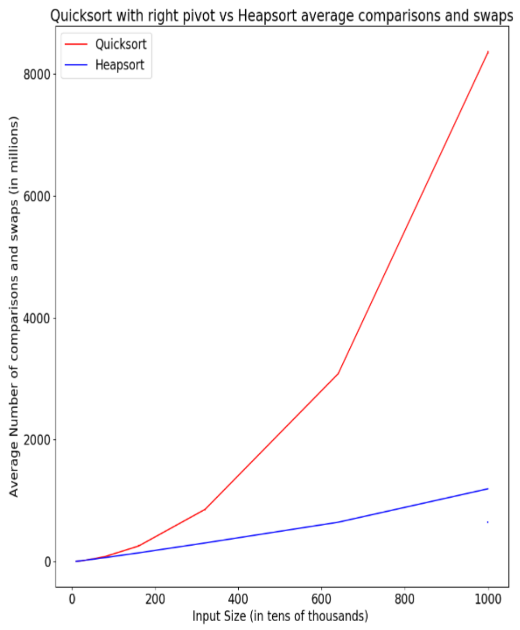


Figure 1.8

To summarize, both quicksort and heapsort have an average time complexity of $O(n\log n)$, as can be seen from the graphs who follow a nlog(n) curve, quicksort performing better than heapsort on random inputs that contain a very small number of repetitions due to better use of caching and less swapping operations. However, if the input contains repetitions or if the input array is already sorted, quicksort's performance drops significantly, causing it to be slower than heapsort. A change in the pivot element eliminates the possibility of the worst-case of $O(n^2)$ for quicksort. Therefore, one would choose to use quicksort if the input data is guaranteed to have few repetitions and to be in a random order, not already sorted, whilst heapsort's upper bound of $O(n\log n)$ makes it a choice when there is no guarantee of the input data to be randomized or to have few repetitions.