Another Attempt to improve the Quicksort Algorithm with Hybrid Strategies

Calvin Smyk

Department of Computer Science
Florida State University
Tallahassee, FL 32304
cs22bh@fsu.edu

April 15, 2023

Abstract

The QuickSort algorithm is a popular sorting algorithm that employs the divide-and-conquer strategy. Its performance, however, is determined on the pivot selection approach employed and the available data. In this project, I hope to deal with the QuickSort algorithm's flaws and eliminate these by merging different sorting algorithms to create a hybrid sorting algorithm. The algorithms that will be studies include the original quicksort, quicksort with random pivot selection, quicksort with median of means pivot selection, the insertion sort algorithm and the merge sort algorithm.The algorithm's performance will be measured in terms of execution time and number of comparisons. The findings of this project will give insight on the flaws of the quicksort algorithm as well as on how to deal with them

1 Objective

The objective of this project is to identify the flaws of the Quicksort algorithm and to eliminate these using different approaches. These approaches include, the combination of different approaches, as well as stacking of different approaches. Hence, the goal is create an efficient sorting algorithm using any of the approaches that will lower or eliminate the likelihood of experiencing the worst-case scenario with the complexity of $O(n^2)$.

2 Literature Review

In total there are five sorting algorithms that will be implemented and evaluated.

2.1 Quicksort with pivot as the first or last element of the list

The QuickSort sorting algorithm is well-known for sorting an array or a list of elements. It is based on the divide-and-conquer principle and has an $O(nlogn)$ best-case time complexity and an $O(n^2)$ worst-case time complexity, where n is the number of members in the array or list. In the worst-case scenario, the pivot element is either the largest or smallest entry in the list, which occurs when the list is either sorted or reversed sorted, resulting in one partition being significantly smaller than the other. Since this is not ideal, there are methods that try to avoid this worst-case scenario.

2.2 Quicksort with a pivot that is calculated through the median of means

The median of medians pivot selection is one such approach, which assures that the pivot selected is always the list's median element. This increases the likelihood of a well-balanced partition, resulting in more efficient sorting.

2.3 Quicksort with a random pivot element

Another technique is random pivot selection, which involves selecting a pivot at random from a list. This technique has the advantage of avoiding the worst-case scenario in the majority of cases, but it could also add some additional complexity.

2.4 Insertion sort algorithm

Apart from the Quicksort sorting algorithm there is also the insertion sort algorithm that sorts an array by comparing each element with the ones that come before it and inserting it into the correct position in the sorted array. It is an in-place sorting algorithm, which means it sorts the array without requiring any additional memory space. The insertion sort algorithm works by maintaining a sorted sublist within the array. Initially, the first element of the array is considered as the sorted sublist. The algorithm then compares the second element with the first element and inserts it in the correct position in the sorted sublist. The third element is then compared with the elements in the sorted sublist and inserted in the correct position. This process is repeated for all the elements in the array, resulting in a fully sorted array. It has an average time complexity of $O(n^2)$, which makes it less efficient than other sorting algorithms like merge sort and quicksort. However, it is often used for small datasets or as a subroutine in other sorting algorithms.

2.5 Merge sort algorithm

At last a basic overview of the Merge sort algorithm is need which is a divide-and-conquer algorithm that sorts an array by recursively dividing it into halves, sorting the halves separately, and then merging the sorted halves back together. The merge sort algorithm works by first dividing the input array into two halves until the size of each half is 1 or 0. The algorithm then merges the sorted halves back together by comparing and merging the elements from each half. This process continues recursively until the entire array is sorted. Merge sort has an average and worst-case time complexity of $O(nlogn)$, which makes it more efficient than other sorting algorithms like insertion sort and quicksort.

3 Implementation

To begin with, a specific set of rules needs to be identified to ensure a unbiased and fair experiment.

1. No sorting algorithm is allowed to use any assisting libraries, which would give this algorithm an advantage. The only functions that are allowed for each algorithm, is calling the $len()$ function as well as using the *random* library.

2. To ensure that every algorithm is running in the same environment and has the same memory capacity and cores available the university's linprog server is utilized for the testing phase.

3. Additionally each algorithm needs to be implemented in the same language, since each language is differently capable of handling recursive calls.

4. In order to be able to compare the algorithms, they need to be tested on the same data.

3.1 Data generation

For the generation of the data, a total of 8 different dataset sizes will be used : $[10, 500, 2500, 10000, 50000, 100000, 250000, 500000, 1000000]$. To ensure that each algorithm will use the same data, 5 different datasets will be created for each dataset size. Three of the five will be randomly shuffled, the last two, however will be in sorted and reversed sorted order. This means that at the end a total of 40 datasets will be created that will all be used to evaluate the performance of the different algorithms. Thereby, each dataset ranges from 0 to n, where n is the size of the dataset.

3.2 Set up of the evaluation pipeline

For the evaluation three parameters are of relevance.

1. *Size of the dataset*: This parameter will inform about the dataset size used.

2. *Comparisons*: This parameter will give information about how many comparisons were needed to sort the array of the given length. Therefore a counter is implemented that will track the number of comparisons at each step for each algorithm. The number of comparisons in theory is calculated with :

$$\sum_{a=1}^{n-1} \sum_{b=a+1}^{n} \times E[X_{a,b}]$$

   Hence, the counter will be incremented each time that two arrays of arbitrary size are compared.

3. *Time taken*: This parameter will track how long the algorithm needs to sort the whole array. To keep this parameter unbiased, it will only include the time taken for the sorting and neglect the time taken for loading the data and the evaluation process.

As previously stated, for each dataset size, 5 different datasets exists. At the end of each evaluation, the number of comparisons and the time taken will be averaged over all 5 runs in order to give runs with a very bad time complexity less weight. The final result will be a dataframe containing the different dataset lengths as well as the averages results over all 5 runs. Additionally for each testing algorithm two plots will be created, one showing the number of comparisons over the length of the dataset, and the other one showing the time taken over the length of the dataset.

4 Testing phase

After setting up the environment that will be used for the testing of the different algorithms, the next step is to define the approaches that will be tested in this project. Recalling, that the objective is to find and deal with the flaws of the quicksort algorithm the first step is to identify the flaws. Therefore all three of the introduced quicksort sorting algorithms will be tested and compared. This will identify the flaws with each of the three approaches and as the next step ideas will be presented that will eliminate the identified problems.

4.1 Initial testing of algorithms

Before starting the testing, it will be verified that the three quicksort sorting algorithms work. Therefore the data example from the lecture :

7	6	3	5	1	2	4

will be taken into account to check if the number of comparisons as well as the time taken make sense. In this example the number of items that need to be sorted are $n = 7$. Therefore the upper limit of the number of comparisons is

$$2 \times n \times \ln(n) = 2 \times 7 \times \ln(7) \approx 27.25$$

and the time taken to run the sorting needs to be

$$O(n \log(n) \leq O(n^2)$$

From the following graph we can learn about the performance of each algorithm with the given data example. This experiment will be run a total of 5 times.

Algorithm	Number of comparisons	Time taken for sorting
Quicksort	11	0.0000025
Quicksort rndm Pivot	Range : 11 - 21	0.0000045
Quicksort Median of Means	11	0.0000035

Figure 1: Initial testing of variations of the Quicksort

According to the picture, the standard quicksort performs the best regarding the time taken and equally good with the Median of Means variation regarding the number of comparisons. The number of comparisons stays for each run the same, since the strategy of picking the pivot nor the sorting of the arrays is changed. The variation with the random pivot however takes each run a different number of comparisons, since the pivot strategy changes, and since the running time is dominated by the time to do the comparisons, this variation takes the longest time in this case.

Considering the structure and implementation of each algorithm all these results make sense. However, since the data is of very short length, the results cannot be generalized which is why the next step is to run all of the three algorithms through the evaluation pipeline and see how the performance changed depending on the length of data that needs to be sorted.

4.2 Evaluation of three Quicksort variations

After validating that all three variations work, they will be applied to the evaluation data. Here we will look at the results differently. The following graph will show the performance of each algorithm for each of the five different sorting of the datasets for the first 4 dataset sizes. First, for the randomly sorted datasets, none of the three variations seem to have any difficulties with sorting the arrays. The specific evaluation will be conducted later, while the focus now

	Random sorting				In order				In reverse order			
Algorithm	**10**	**500**	**2500**	**10000**	**10**	**500**	**2500**	**10000**	**10**	**500**	**2500**	**10000**
Quicksort	✓	✓	✓	✓	✓	✓	✗	✗	✓	✓	✗	✗
Quicksort rndm Pivot	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Quicksort Median of Means	✓	✓	✓	✓	✓	✓	✓	✗	✓	✓	✓	✗

Figure 2: Finding flaws of standard QS and QS with median of means variation

will be on why the standard quicksort and the median of means variation struggle with sorting arrays that are sorted, while the random Pivot variation does not struggle with it.

The reason for this problem lies on the divide-and-conquer strategy of the quicksort algorithm. In the last two sorting cases, the partition will tend to split the array into two subarrays of uneven sizes, with one subarray being empty or nearly empty, which will lead to an many unnecessary recursive calls, as the algorithm will continue to partition and sort the already sorted subarrays. This leads to an recursive error thrown by python, since it's lacking the tail recursion elimination. The variation with the random pivot selection will randomly choose a pivot, which has a higher chance of success regarding the partition.

Standard Quicksort

	Datalength	Time	comparisions
0	10.0	0.000040	2.700000e+01
1	500.0	0.001868	4.869333e+03
2	2500.0	0.011501	3.300233e+04
3	10000.0	0.044327	1.518900e+05
4	50000.0	0.236941	9.247637e+05
5	100000.0	0.505250	2.005008e+06
6	250000.0	1.387818	5.558798e+06
7	500000.0	3.006561	1.192077e+07
8	1000000.0	6.410068	2.568063e+07

Quicksort with random Pivot

	Datalength	Time	comparisions
0	10.0	0.000051	2.466667e+01
1	500.0	0.001560	4.479000e+03
2	2500.0	0.009534	3.277067e+04
3	10000.0	0.042730	1.617200e+05
4	50000.0	0.239709	9.380223e+05
5	100000.0	0.520834	2.019697e+06
6	250000.0	1.367541	5.478041e+06
7	500000.0	2.864143	1.163374e+07
8	1000000.0	6.027657	2.439018e+07

Quicksort with median of means

	Datalength	Time	comparisions
0	10.0	0.000035	1.300000e+01
1	500.0	0.001300	1.384667e+03
2	2500.0	0.008109	8.189667e+03
3	10000.0	0.037407	3.771167e+04
4	50000.0	0.216647	2.150300e+05
5	100000.0	0.452612	4.568597e+05
6	250000.0	1.228133	1.217415e+06
7	500000.0	2.568235	2.555585e+06
8	1000000.0	5.474403	5.338557e+06

Figure 3: Performance comparison of the three quicksort variations

Looking at this comparison, it needs to be pointed out that the averaged results for each dataset size only contain the first three sorting datasets, since the median of means and standard quicksort fail through the recursive calls for the in order, and reverse order sorted strategy. If they would be considered, the average measured time and average number of comparisons would be much higher since it would result in their worst-case time complexity of $O(n^2)$.

Apart from this limitation, the picture shows that for arrays that are randomly shuffled and have no particular order, the variation with the median of means is by far the quickest, using the least number of comparisons and least time to sort the array. However, with increasing order in the data, it will get more and more inefficient all the way to the point where the data is fully sorted which will lead to a worst-time complexity of $O(n^2)$.

The same applies to the standard quicksort, only that it will reach the point of inefficiency faster. The random pivot variation however may take a little bit longer and needs more comparisons, than the median of means variation, since the pivot is randomly chosen which may led to an unequal partition, yet it is

the only variation that does not encounter a recursive error. Therefore, the best strategy chosen in order to eliminate the issue of the recursive call error, is the random pivot strategy. The next step is to improve this variation to the point that it is as fast as the median of means variation but avoiding the recursive call error.

4.3 Combinations of approaches to increase performance

After coming to the conclusion, that the quicksort variation with the random pivot selection is the only strategy that is able to eliminate the recursive call error, this sorting algorithm is chosen as the base.

4.3.1 Approach introduction

For the enhancement two different approaches are suggested:

1. **Quicksort with random pivot + insertion sort** : This approach is taking into account, that the random pivot variation is more likely to chose a bad pivot when the array is smaller, leading to a higher number of comparisons needed, hence longer sorting time. While the insertion sort is independent of the pivot and has a consistent performance for small arrays. Therefore, the insertion sort algorithm will be used sort arrays that have a size smaller than 2501 and the quicksort with random pivot selection will be used for any dataset size bigger than 2500.

2. **Quicksort with random pivot + insertion sort + merge sort** : This approach builds upon the first approach, but in this case merge sort will replace quicksort for array larger than 499999. Due to it's property that Merge sort has a guaranteed worst-case time complexity of $O(n \log(n))$, regardless of the input data. Additionally, merge sort has good cache locality because it accesses adjacent elements in memory, which can improve performance for large arrays that cannot fit entirely in the CPU cache.

4.3.2 Comparison of approaches with original variation

In this part, the performance of the quicksort variation with random pivot will be compared to the performance of the two newly introduced approaches. For this, all 5 differently sorted datasets will be used. The following picture shows the performance of all three algorithms:

Quicksort with random Pivot

	Datalength	Time	comparisions
0	10.0	0.000055	24.8
1	500.0	0.001886	4727.0
2	2500.0	0.011011	31687.2
3	10000.0	0.049566	154390.0
4	50000.0	0.277222	910731.0
5	100000.0	0.602704	2014635.2
6	250000.0	1.593993	5524271.0
7	500000.0	3.390451	11770883.4
8	1000000.0	7.122770	24574864.6

First approach

	Datalength	Time	comparisions
0	10.0	0.000039	18.6
1	500.0	0.020194	62064.6
2	2500.0	0.557878	1558500.8
3	10000.0	0.046849	149412.2
4	50000.0	0.269627	949477.6
5	100000.0	0.578028	2046069.4
6	250000.0	1.514512	5453778.8
7	500000.0	3.219694	11602276.2
8	1000000.0	6.904196	25011329.0

Second approach

	Datalength	Time	comparisions
0	10.0	0.000030	18.6
1	500.0	0.020833	62064.6
2	2500.0	0.585980	1558500.8
3	10000.0	0.050893	160271.0
4	50000.0	0.280194	929626.2
5	100000.0	0.598872	2030938.4
6	250000.0	1.606960	5538354.2
7	500000.0	3.062702	7697696.0
8	1000000.0	6.248560	16194963.6

Figure 4: Performance comparison of the quicksort with random pivot selection and the two approaches

Comparing the quicksort with random pivot selection with the first approach, the focus should be on the first three rows since this is where the insertion sort is applied. One can see that for the dataset size of 10, the insertion sort is quicker and needs less comparisons, even when keeping in mind that these results are averaged over all 5 runs. Yet, when the dataset size increases the performance of the insertion sort drops significantly. This is because it is not easily parallelizable because its algorithm depends on comparing adjacent elements in the array. This means that it is difficult to divide the problem into independent subproblems that can be solved simultaneously.

Looking at the second approach, the same interpretation can be drawn for the small arrays. However for the large arrays the inclusion of the merge sort into the hybrid algorithm does result in another positive impact. While for the dataset size of 500000 the difference is not as big, with increasing size the merge sort dominates the quicksort in both time taken and number of comparisons. Therefore the goal of increasing the algorithms performance worked out perfectly especially for the large arrays, which takes the longest time to sort.

4.3.3 Comparing the best approach with the previous ideal, quicksort with the median of means variation

The prior goal was to eliminate the recursive call error, avoid the worst-case time complexity and improve the hybrid algorithm in a way that it is fast than the ideal quicksort with the median of means variation with the ideal case of fully randomized arrays. Since by now it is proven that even the median of means variation's performance will decrease with increasing order in the dataset. Therefore, as a last step we will compare the ideal quicksort with median of means variation with the second approach. In the following picture the results of both methods are listed.

Quicksort with median of means

	Datalength	Time	comparisions
0	10.0	0.000035	1.300000e+01
1	500.0	0.001300	1.384667e+03
2	2500.0	0.008109	8.189667e+03
3	10000.0	0.037407	3.771167e+04
4	50000.0	0.216647	2.150300e+05
5	100000.0	0.452612	4.568597e+05
6	250000.0	1.228133	1.217415e+06
7	500000.0	2.568235	2.555585e+06
8	1000000.0	5.474403	5.338557e+06

Second approach

	Datalength	Time	comparisions
0	10.0	0.000030	18.6
1	500.0	0.020833	62064.6
2	2500.0	0.585980	1558500.8
3	10000.0	0.050893	160271.0
4	50000.0	0.280194	929626.2
5	100000.0	0.598872	2030938.4
6	250000.0	1.606960	5538354.2
7	500000.0	3.062702	7697696.0
8	1000000.0	6.248560	16194963.6

Figure 5: Performance comparison of the median of means variation with the second approach

In order to be able to understand this comparison, one needs to keep in mind that it depends on shuffled data for the median of means variation, which plays into its hand. However in a real-world scenario it is almost never the case that data is perfectly shuffled. And as already discussed previously, with higher order in the data this variation's performance will decrease rapidly. On the other hand the approach which combines the insertion sort, quicksort with random pivot selection and merge sort, can handle any data, shuffled or not. That being said, looking at the performance, the second approach does not fully reach the performance of the quicksort variation. Apart from the first dataset size, the median of means variation outperforms the second approach. For example with the biggest dataset size, the second approach takes almost 3 times as many comparisons and 0.8 seconds longer.

4.4 Problems during implementation

The first problem that was encountered during the experiment, is the selection of the coding language. While recursive algorithms can be implemented in Python, for instance C or C++ would have been a better choice. One reason for that is that C++ is a compiled language, while Python is an interpreted language, meaning that C++ code will be faster executed. Another main reason is that recursive algorithms often involve repeated function calls, which is handled better by C++ than Python. Additionally, C++ provides more control over memory allocation and management, which can be important when implementing recursive algorithms that require efficient memory usage.

Another problem was the implementation of the algorithms. Since each algorithm was implemented independently the way it was written can have a significant impact on the execution time of the algorithm. Even if the algorithm itself is efficient in terms of time complexity, a poorly optimized implementation can result in slow execution times.

Also, the hardware used for this time depended experiment was of relevance. Factors such as the processor speed, amount of available memory and how much resources the computer was able to allocate to the execution of the algorithm constantly changed even while using the linprog server. This made a very unbiased comparison quite difficult.

5 Conclusion

All in all, it can be said that this experiment was able to give some insight in how the quicksort algorithm works and where it encounters problems as well as how it can be improved using the combination of the insertion sort, quicksort with random pivot selection and merge sort.

Yet, it needs to be said that all these results are strongly depended on the programming language used. One of the main problems that have been identified is that the quicksort algorithm definitely does struggle with sorting arrays that are already sorted, reverse sorted or almost sorted, since it is getting very close to the worst-case time complexity of $O(n^2)$, leading to an recursive call error in python, which makes the execution of the algorithm impossible. Because the quicksort with the random pivot selection is the only one of the three variations that is avoiding the recursive call error, this variation was chosen for the further improvement of the performance.

Therefore a hybrid approach was build, combining the insertion sort algorithm with the quicksort with random pivot selection and the merge sort algorithm. This hybrid algorithm was able to outperform the performance of the original quicksort and the quicksort with random pivot selection. Yet, the performance did not fully reach the ideal performance of the median of means variation.

However, it can be said that this attempt was a full success, since the approach did outperform the only quicksort variation that was not receiving an recursive call error by almost 10%. Which means that the hybrid algorithm is able to avoid the worst-case time complexity more frequently even if the arrays are fully or almost completely sorted.

References

[Geea] GeekforGeeks. *Median of an unsorted array using Quick Select Algorithm*. URL: `https://www.geeksforgeeks.org/median-of-an-unsorted-array-in-liner-time-on/`.

[Geeb] GeekforGeeks. *Quicksort*. URL: `https://www.geeksforgeeks.org/quick-sort/`.

[Geec] GeekforGeeks. *Quicksort using random Pivoting*. URL: `https://www.geeksforgeeks.org/quicksort-using-random-pivoting/`.

[Hoa62] C. A. R. Hoare. "Quicksort". In: *The Computer Journal* 5.1 (1962), pp. 10–16.

[Kur16] Noriyuki Kurosawa. "Quicksort with median of medians is considered practical". In: *arXiv:1608.04852* (2016).

[Lis05] Beatrice List. "Randomized Quicksort and the Entropy of the Random Source". In: *Springer* (2005).

[Smy] Calvin Smyk. *The github:* URL: `https://github.com/CalvinSmyk`.