Analyzing the Runtime of the Quick Sort Algorithm
Lab #5

By
Michael Harden

CS 303 Algorithms and Data Structures
September 28, 2021

1. Problem Specification

The goal of this lab was to implement the Quick Sort Algorithm and choose the pivot in two different ways. 1) By always using the last element in the partition. 2) By using the median of 3 elements from the partition. Then to compare the runtime of these two different implementations of Quick Sort. And finally, compare Quick Sort to the run time of Heap Sort, Merge Sort, and Insertion Sort.

2. Program Design

This program requires two classes, Sorts and SortsDriver. The Sorts class contains the implementation for each of the sorting algorithms explored in previous labs and Quick Sort implementation. The SortsDriver class contains methods to parse the text files containing the integers to be sorted. The files can be parsed using any delimiter. The driver class also calculates the time taken for each sorting algorithm to sort all given files; compare the two implementations of Quick Sort; and finally, test the correctness of the Quick Sort implementations.

To implement the Heap Sort, Insertion Sort, and Merge Sort algorithm, the same process was used as in labs #2 #3, and #4

The Quick Sort algorithm was implemented with the following process.

a) Take the array which is to be sorted and choose a element within the current partition of the array to be the pivot , and put the pivot element at the end of the partition array. (on the first pass, the partition is the whole array)
b) Make two pointers, a left, and a right pointer. The left pointer starts at the first element of the current partition of the array, and the right pointer starts at the second to last element of the partition of the array. (The last element of the partition is the pivot)
c) If the left pointer, points to an element greater than the pivot and the right pointer, points to an element less than the pivot, then the two elements are swapped.
d) If the element at the left pointer is less than or equal to the pivot element, the index of the left pointer is increased by 1.
e) If the element at the right pointer is greater than or equal to the pivot element, the index of the right pointer is decreased by 1.
f) Steps C – E continue until the index of the left pointer is greater than the index of the right pointer. Once this happens, all the element from the left index to the end of the partition is greater than the pivot and everything to the right index to the start of the array is less than the pivot.
g) Finally swap the pivot with the element at the left index.
h) All element to the right and left of the pivot are now in their proper half with respect to the pivot, and the pivot is in its proper place.
i) All elements to the left and right of the pivot become the two new partitions respectively.

The pivot was chosen in two different ways.

1) The last element of the partition was always the pivot
2) The median between the elements at the first, last, and middle indices was taken, and that was the pivot

To calculate the run times for each required file, the following procedure was followed.

a) Parse the content of the file into a list of integers, starting with the smallest file and for each iteration move towards the largest file.
b) Create an instance of the Sorts class and pass the list of parsed integers into the Sorts constructor's parameter.
c) Store the current time just before sorting. Once the list has been completely sorted, the difference between the time directly after sorting and the stored time just before sorting is calculated.
d) The time is logged to the console
e) Continue to the next iteration and sort the next required text file.
f) The file sizes range from 100 to 500,000 elements.
g) This process takes place for Quick Sort, Merge Sort, Heap Sort, and Insertion Sort. This process also take place for the files ranging from 16 to 8192 and the Randomly sorted, Sorted, and Reverse Sorted files to compare the two implementations of Quick Sort.

The following constructor and methods are defined within the Sorts class.

a) __init__(data):

    Defines the instance variable, data, and sets its value to the value passed in the data parameter.

b) Each of the two Quick Sort implementations are split into 2 methods.
    1. quick_sort():

        calls the quick_sort_helper method and sets the first index to 0 and the last index to one less than the size of the array

    2. quick_sort_helper(first_idx, last_idx):

        sorts the array as described above
    3. quick_sort_med_piv():

        calls the quick_sort_med_piv_helper method and sets the first index to 0 and the last index to one less than the size of the array
    4. quick_sort_helper_med_piv(first_idx, last_idx):

        sorts the array with the median of 3 pivot value as described above.

c) Heap, Merge and Insertion sort are implemented in the same way as covered in the previous lab reports

The following constructor and methods are defined within the SortsDriver class.

a) test_quick_sort (lst):
This method tests the implementation of Quick Sort, with the last index as the pivot, by sorting the lists passed through the parameter and sorting it with Sorts.heap_sort(). The sorted list is then returned

b) test_quick_sort_med(lst):
This method tests the implementation of Quick Sort, with the median of 3 as the pivot, in the same way test_quick_sort works.

c) All other methods are consistent with the descriptions provided in lab #2, #3, and #4.

To parse the files, the open() and read() built in methods are used.

3. Testing

To test both sorting algorithms seven test cases were used to attempts and break the heap sort algorithm in different ways. The first contains both negative and positive numbers. The second contains multiple duplicates of the same number. The third contains only negative numbers. The forth contains two numbers reverse sorted. The fifth only one single element. The sixth is reverse sorted, and the last is an empty list.

Test Number	**Input**	**Expected Output**
#1	[10, 4, 6, 3, 2, 9, 16, 0, 3, -1]	[-1, 0, 2, 3, 3, 4, 6, 9, 10, 16]
#2	[4, 3, 3, 3, 2, -2]	[-2, 2, 3, 3, 3, 4]
#3	[-3, -103, - 5, -2, -10, -44, -31]	[-103, -44, -31, -10, -5, -3, -2]
#4	[4, 2]	[2, 4]
#5	[10]	[10]
#6	[10, 9, 8, 7, 6, 5, 4, 3, 2, 1]	[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
#7	[]	[]

4. Timing

The times to sort the provided files are as follows:

File Size	**Insertion Sort Time** (seconds)	**Merge Sort Time** (seconds)	**Heap Sort Time** (seconds)	**Quick Sort Time** (seconds)
100	0.000738144	0.000239849	0.000893831	0.000271082
1,000	0.076004982	0.004236221	0.013286114	0.003712177
5,000	1.765846729	0.07001996	0.094236851	0.024173021
10,000	6.662191868	0.139968872	0.151299715	0.056014061
50,000	187.4775789	0.60493803	0.885379076	0.321298122
100,000	742.8829157	1.234847069	1.978494167	0.741900921
500,000		5.956631899	10.73291397	4.087168932

The time differences between the two Quick Sort implementations are as follows:

File Size	**Last Index Pivot** (seconds)	**Median of 3 Pivot** (seconds)
16	2.79E-05	3.00E-05
32	6.70E-05	5.79E-05
64	0.000133038	0.000252962
128	0.000306368	0.000347853
256	0.000687122	0.000855207
512	0.001523018	0.001850128
1024	0.003994703	0.004128933
2048	0.009481907	0.00948596
4096	0.020676851	0.023358107
8192	0.041854858	0.049839735

The time differences between the two Quick Sort implementations for Randomly Sorted, Reverse Sorted, and Sorted are as follows:

File Status	**Last Index Pivot** (seconds)	**Median of 3 Pivot** (seconds)
Random	0.004504681	0.004585028
Reverse Sorted	0.033709764	0.15813303
Sorted	0.031879902	0.088108063

5. Analysis and Conclusions

In conclusion, the times shown demonstrates how Quick Sort is comparable to Merge and Heap sort, having a time complexity of O(n log n) when the pivot is chosen correctly. One advantage to using Quick Sort opposed to Merge Sort is how it is an in place sort, no auxiliary arrays are needed.

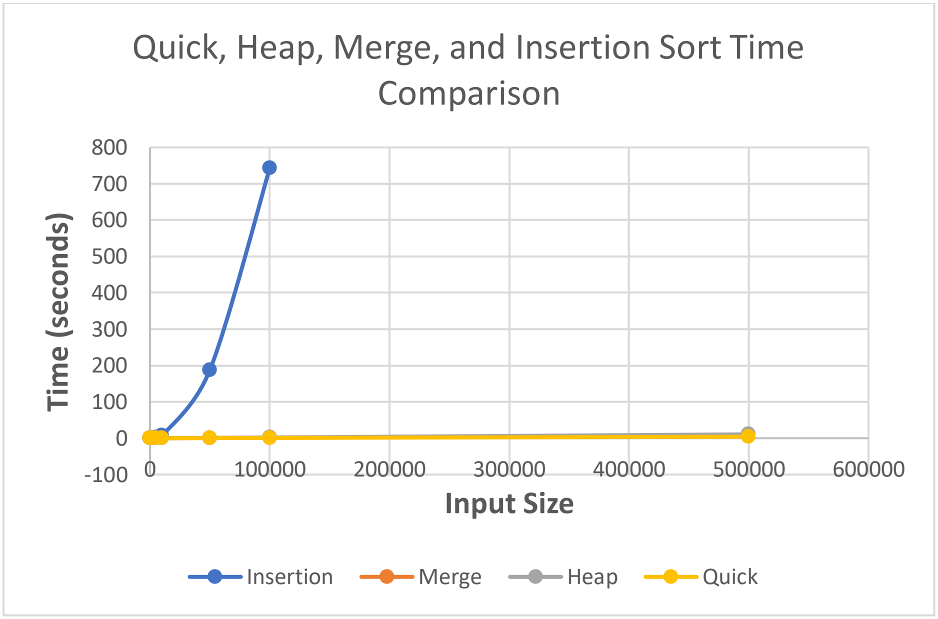
A weakness of quick sort is that it is an unstable sorting algorithm. If you want to resort an already sorted array on a new property, the previous sorting will be undone.

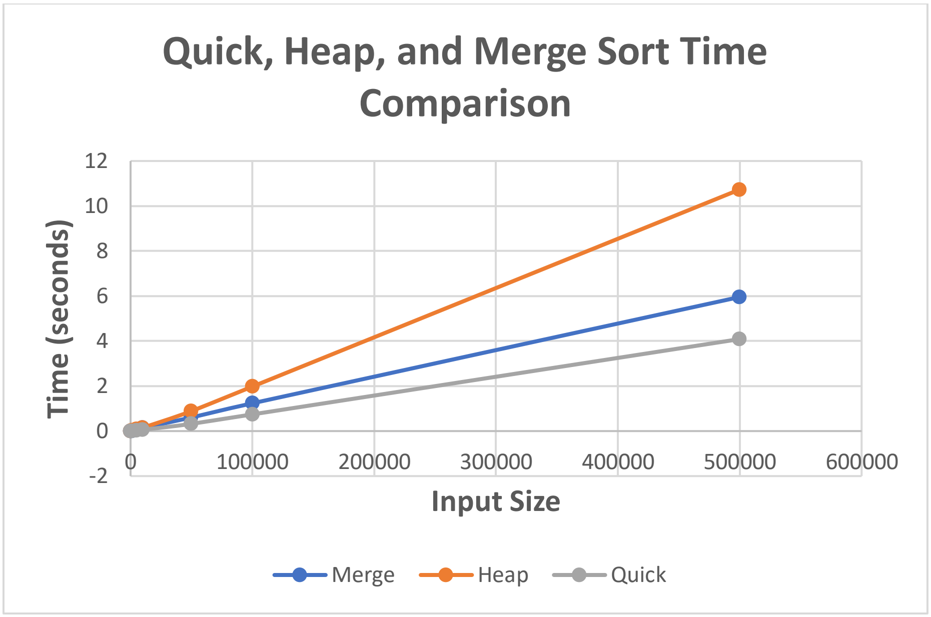
Quick Sorts best case time complexity is O(n log n) so when sorting data that is almost completely sorted, insertion sort would still be optimal, with a best case of O(n)

The following graphs displays how the sorting time grows in relation to a linearly growing input size. The first chart compares the time complexity of Quick, Heap, Merge and Insertion Sort. The second chart only compares Quick, Heap and Merge sort, as their differences are not great enough to show up when comparing to Insertion Sort.

The following graph uses the Quick Sort method that picks the median of three for the pivot.

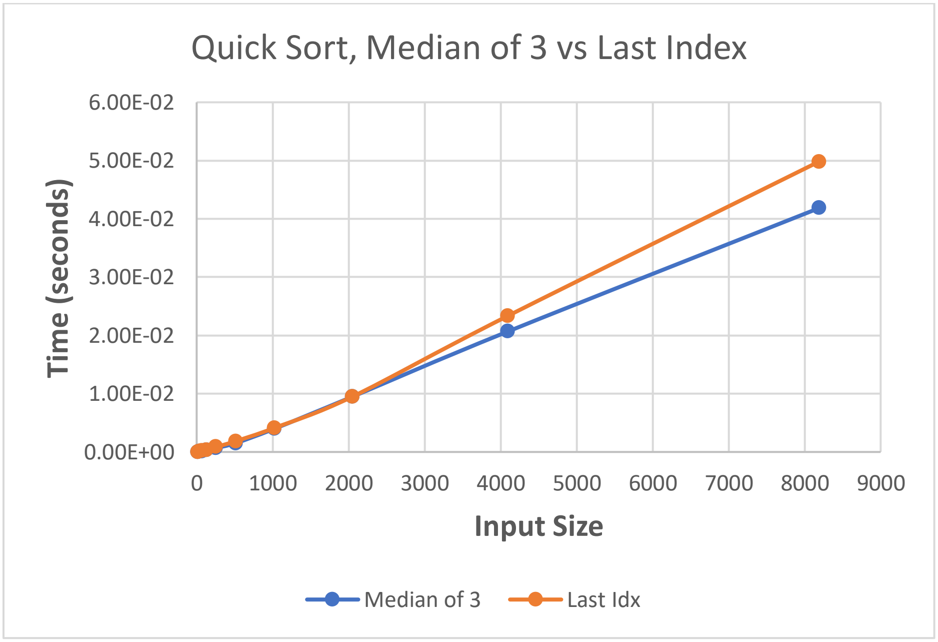
(data collected with provided text files):

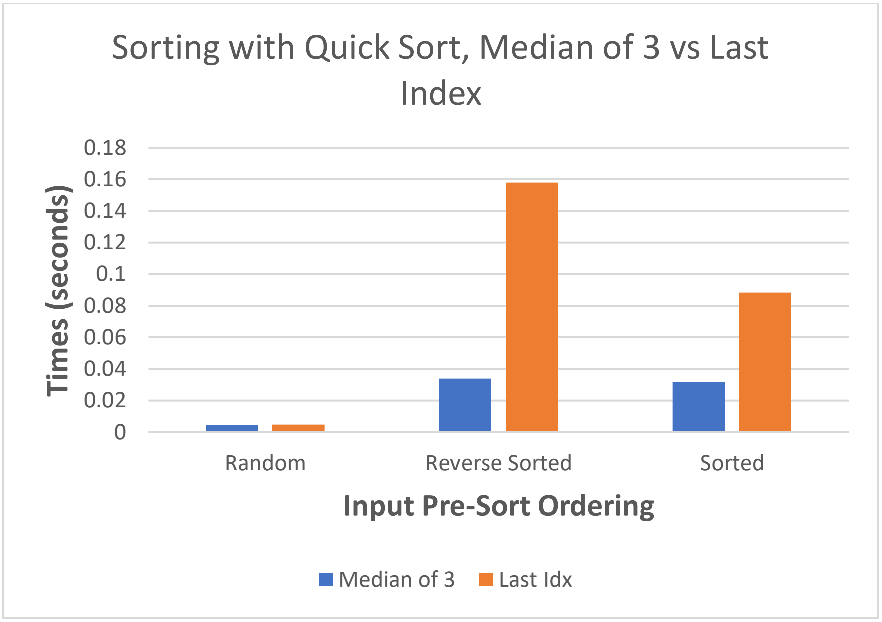




The data collected also shows the importance of picking a proper pivot, to minimize partition sizes. This is displayed in the tables below. As is shown when always choosing the last index as the pivot, the time on average is worse than when choosing the pivot as a median of three. Furthermore, on extreme cases where the data already sorted, or reverse sorted, the time comparisons was drastically different

The first chart shows the time comparisons for unsorted lists. The second chart shows the Time comparisons for random, sorted, and reverse sorted data.





6. Credits

- For the Quick Sort algorithm, the pseudo code in the lab4 document was used.
- As well as Algoexpert.com and opengenus.io
- Parts of my lab reports 2, 3, and 4 were also used in this report.
- For the Report the sample report provided on canvas was used as a guide.