Sorting Algorithms and their Behaviour in Real-life Data

INF221 Term Paper, NMBU, Autumn 2019

Ghazal Azadi
ghazal.azadi@nmbu.no

Nasibeh Mohammadi
nasibeh.mohammadi@nmbu.no

Abstract

In this paper, we aim to not only analyse three sorting algorithms of merge, heap and quick sort in terms of their running times for sorting three cases of already-sorted, reverse-sorted and random numbers but also to see if theoretical expectations hold for their behaviours or not.
In this regard, asymptotic notations help us to compare algorithms' behaviour with theoretical expectations. Besides, using timeit function in Python we calculate running times of our sorting algorithms to see which algorithm performs better.

1. Introduction

Sorting a sequence of components is an important subject in computer science and data structure. Many reasons can be stated to support this issue. As an instance, data in sorted order makes it easier to efficiently search for information or makes it much faster to select a specific item in a set. [1]
In this regard, many sorting algorithms have been introduced so far. However, their performances differ and since we know that, if the algorithm is not fast enough, it can not be called functional, comparing performances of different sorting algorithms is the key to select the best one.
In this paper, we firstly aim to compare running time of three sorting algorithms of merge sort, heap sort and quick sort and secondly compare real-life behaviour of them with theoretical expectations.

2. Theory

2.1 Divide and Conquer

"Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related subproblems. These algorithms typically follow a divide-and-conquer approach: they break the problem into several subproblems that are similar to the original problem but smaller in size, solve the subproblems recursively, and then combine these solutions to create a solution to the original problem. The divide-and-conquer paradigm involves three steps at each level of the recursion:

- Divide the problem into a number of subproblems that are smaller instances of the same problem.
- Conquer the subproblems by solving them recursively. If the subproblem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- Combine the solutions to the subproblems into the solution for the original problem." Cormen et al. [2009, Ch. 2.3]

[1] https://algs4.cs.princeton.edu/25applications/

2.2 Sorting Algorithms

Since running time of sorting algorithms not only depends on input but also depends on the input size, we will consider three cases of sorted, reversed-sorted and randomised data sets with different sizes.
Reviewing details of algorithms being used in this paper, the pseudocode and the theoretical running time of each of algorithm are as follows.

2.2.1 *Merge sort Algorithm*

Pseudocode for the merge sort algorithm is shown in Listing. 1.

Listing 1 Merge sort algorithm from Cormen et al. [2009, Ch. 2.3].

```
MERGE-SORT($A, p, q, r$)
if $p < r$
    $q = (p + r)/2$
    Merge-Sort(A, p, q)
    Merge-Sort(A, q + 1, r)
    Merge(A, p, q, r)
```

Listing 2 Merge Pseudocode from Cormen et al. [2009, Ch. 2.3].

```
MERGE($A, p, q, r$)
$n_1 = q - p + 1$
$n_2 = r - q$
Let $L[1..n_1 + 1]$ and $R[1..n_2 + 1]$ be new arrays
for $i = 1$ to $n_1$
    $L[i] = A[p + i - 1]$
for $j = 1$ to $n_2$
    $R[j] = A[q + j]$
$L[n_1 + 1] = \infty$
$R[n_2 + 1] = \infty$
$i = 1$
$j = 1$
for $k = p$ to $r$
    if $L[i] \leq R[j]$
        $A[k] = L[i]$
        $i = i + 1$
    else $A[k] = R[j]$
        $j = j + 1$
```

Running time of merge sort algorithm from Cormen et al. [2009, Part. II] for both worst case and average case is:

$$T(n) = \Theta(n \lg n) .$$

2.2.2 *Heap Sort Algorithm*

Pseudocode for the heap sort algorithm is shown in Listing. 3. Running time of heap sort algorithm in both average case[2] and worst case [3] is:

$$T(n) = O(n \lg n) \ .$$

Listing 3 Heap sort algorithm from Cormen et al. [2009, Ch. 6.4].

```
HEAP-SORT(A)
Build-Max-Heap(A)
for i = A.Length downto 2
    swap A[1] and A[i]
    A.heap-size = A.heap-size −1
    Max-Heapify(A,1)
```

Listing 4 Build-Max-Heap Pseudocode from Cormen et al. [2009, Ch. 6.4].

```
BUILD-MAX-HEAP(A)
A.heap-size = A.Length
for i = ⌊A.Length/2⌋ downto 1
    Max-Heapify(A,i):
```

Listing 5 Max-Heapify Pseudocode from Cormen et al. [2009, Ch. 6.4].

```
MAX-HEAPIFY(A)
l = Left(i)
r = Right(i)
if l ≤ A.heap-size and A[l] > A[i]
    largest = l
else
    largest = i
if r ≤ A.heap-size and A[r] > A[largest]
    largest = r
if largest ! = i
    swap A[i] and A[largest]
    Max-Heapify(A, largest)
```

2.2.3 *Quick Sort Algorithm*

Pseudocode for the third algorithm is shown in Listing. 6. Running time of quick sort algorithm from Cormen et al. [2009, Part. II] for average case is:

$$T(n) = \Theta(n \lg n) \ .$$

and for worst case, it is:

$$T(n) = \Theta(n^2) \ .$$

[2]https://www.academia.edu/33897939/Performance_Comparison_of_Different_Sorting_Algorithms/

[3]Cormen et al. [2009, Part. II]

Listing 6 Quick sort algorithm from Cormen et al. [2009, Ch. 7.1].

```
QUICK-SORT(A, p, r)
if p < r
    q = Partition(A, p, r)
    Quick-Sort(A, p, q-1)
    Quick-Sort(A, q + 1, r)
```

Listing 7 Partition Pseudocode from Cormen et al. [2009, Ch. 7.1].

```
PARTITION(A, p, r)
x = A[r]
i = p − 1
for j = p to r − 1
    if A[j] ≤ x
        i = i + 1
        swap A[i] and A[j]
swap A[i + 1] and A[r]
return i + 1
```

3. Methods

The test-data being used in this paper has been generated by random number generator in Python. In order to compare runtime of sorting algorithms, we consider six sets of random numbers with sizes ranging from 1024 to 1048576. It is good to mention that these sizes have been selected based on powers of two functions.

Table 1: Sample Sizes

n_1	$2^{10} = 1024$
n_2	$2^{12} = 4096$
n_3	$2^{14} = 16384$
n_4	$2^{16} = 65536$
n_5	$2^{18} = 262144$
n_6	$2^{20} = 1048576$

In addition to this, we have considered three cases of already sorted, reversed-sorted and randomised data sets for every sizes mentioned above.

Running time of algorithms have been measured using "timeit" function in Python. The timeit module provides a simple interface for determining the execution time of small bits of Python code. It uses a platform-specific time function to provide the most accurate time calculation possible. It reduces the impact of startup or shutdown costs on the time calculation by executing the code repeatedly.[4]

The computer being used for running all the codes is MacBook Pro with macOs 10.15.1 software and 2,3 GHz Dual-Core Intel Core i5 processor.

The programming language being used to run the codes of this paper is Python 3.7.4. The NumPy library has been also used in this paper with the version of 1.16.4.

[4]https://pymotw.com/2/timeit/

4. Results

4.1 Performance of Merge Sort Algorithm

As we have already seen from the pseudocode of merge sort algorithm, "this sorting method works on the principle of divide and conquer. Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list." [5]
Considering three cases of sorted, reverse-sorted and randomised data sets with sizes of $n_1, n_2, n_3, n_4, n_5, n_6$, running time of merge sort algorithm would be as follows.

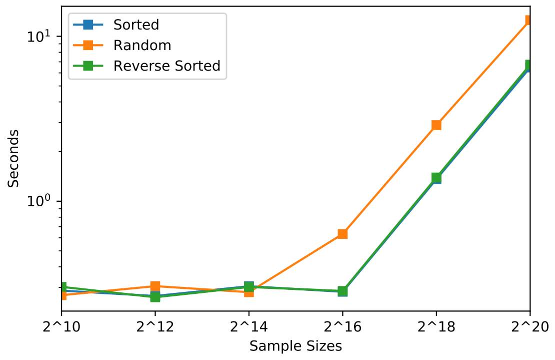

Figure 1: Running Times of Merge Sort Algorithm for Different Sample Sizes

Note that for better showing the difference between running times, we consider *log* of running times on y axis of the plot. The running time of merge sort algorithm ranges from $(0.2, 13)$. In addition, until sample size of $n_3 = 2^{14}$ three cases of data did not affect significantly performance of the merge sort. However increasing sample size to the amount of $n_4 = 2^{16}$ dramatically affects the performance of the algorithm for the randomised data set.
As we mentioned before, 'timeit' function helps us measuring the time takes to sort our data sets. This function considers seven times for executing and sorting the data sets. The table below shows the standard deviations of running times for each data set in different sizes.

Table 2: Standard Deviations of Merge Sort Running Times for Different Sample Sizes and Data Sets

Sample Size	Sorted	Reverse Sorted	Randomised
n_1	0.009	0.019	0.009
n_2	0.018	0.016	0.016
n_3	0.011	0.015	0.011
n_4	0.007	0.011	0.018
n_5	0.035	0.051	0.083
n_6	0.054	0.057	0.417

Based on what we have seen so far, we can generally say that among three cases of our data sets, randomised data set is considered as the worst case for merge sort algorithm. Already sorted and reverse-sorted data sets do not significantly change the performance of merge sort algorithm but for the large sample sizes, sorted data set can be grouped as the best case for merge sorting algorithm regarding the time takes for sorting.
For comparing the performance of merge sort algorithm with our theoretical expectations, we firstly remind that expected running time of merge sort algorithm is

$$T(n) = \Theta(n \lg n) \ .$$

which based on the definition in Cormen et al. [2009, Ch. 3.1] means:

$$T(n) \text{ is the set of all functions for which we can find constants of}$$
$$c_1 > 0, \ c_2, > 0, \ n_0 \in \mathbb{N} \text{ so that :}$$
$$0 \le c_1(n \lg n) \le T(n) \le c_2(n \lg n) for\ all\ n \ge n_0. \tag{1}$$

Therefore if for all sizes of $n_1, n_2, n_3, n_4, n_5, n_6$ we find c_1 and c_2 which fit to the above condition we would be able to say the theoretical expectations hold for real-life behaviour of merge sort algorithm.
Considering $c_1 = 0.003$ and $c_2 = 0.000001$ we will have an upper and a lower bound for our $T(n)$ which graphically would seem as figure 2.

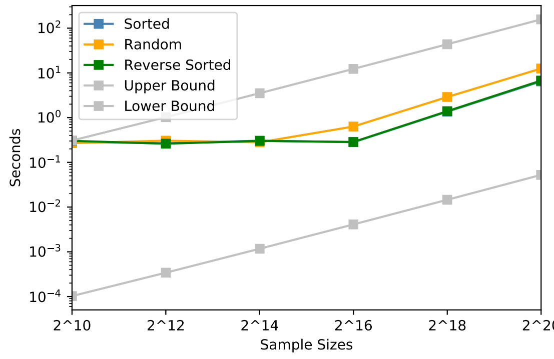

Figure 2: Upper and Lower Bounds for Merge Sort Running Time Function

So we can conclude that for merge sort running time function, $T(n) = \Theta(n \lg n)$ and theoretical expectations hold.

4.2 Performance of Heap Sort Algorithm

Heap sort involves building a Heap data structure from the given array and then utilising the Heap to sort the array. [6]
Considering three cases of sorted, reverse-sorted and randomised data sets with sizes of $n_1, n_2, n_3, n_4, n_5, n_6$, running time of heap sort algorithm would be as follows.
The running time of heap sort algorithm ranges from (0.2,30) seconds and generally randomised data set takes the longest time to be sorted by heap sort algorithm. What we again witness is that

[5]https://www.interviewbit.com/tutorial/merge-sort-algorithm/

[6]https://www.studytonight.com/data-structures/heap-sort

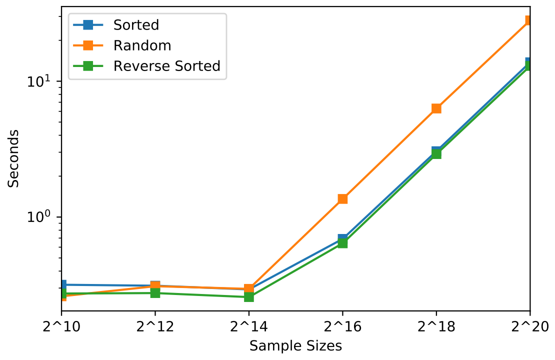

Figure 3: Running Times of Heap Sort Algorithm for Different Sample Sizes

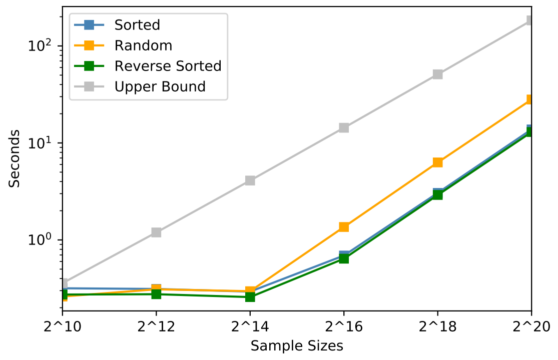

Figure 4: Upper Bound for Heap Sort Running Time Function

until the sample size of $n_3 = 2^{14}$ heap sort performance does not differ significantly however after increasing the sample size to the amount of $n_4 = 2^{16}$ situation considerably changes that we see for sample size of $n_6 = 2^{20}$ randomised data set takes 15 seconds more time to be sorted comparing to sorted and reverse sorted data sets. Therefore, among three cases of our data sets, randomised data set is the worst and reverse sorted data set is the best case for heap sort algorithm regarding the time takes for it to sort them.
The standard deviations of running times for each data set in different sizes are shown in the following table.

Table 3: Standard Deviations of Heap Sort Running Times for Different Sample Sizes and Data Sets

Sample Size	Sorted	Reverse Sorted	Randomised
n_1	0.010	0.012	0.010
n_2	0.017	0.016	0.015
n_3	0.019	0.010	0.013
n_4	0.030	0.039	0.043
n_5	0.062	0.051	0.186
n_6	0.079	0.073	0.068

Since expected running time of heap sort algorithm is $T(n) = O(n \lg n)$, we should show (2) for heap sort algorithm.
Based on the definition in Cormen et al. [2009, Ch. 3.1] we have:

$$T(n) \text{ is the set of all functions for which we can find constants of}$$
$$c > 0,\ n_0 \in \mathbb{N} \text{ so that :}$$
$$0 \le T(n) \le c(n \lg n) for\ all\ n \ge n_0. \tag{2}$$

In this regard, considering $c = 0.0035$, we will have an upper for $T(n) = O(n \lg n)$. The following plot shows the bounds graphically. So for heap sort algorithm we can say that the theoretical expectations hold, as well.

4.3 Performance of Quick Sort Algorithm

"Quick sort is one of the most efficient sorting algorithms and is based on the splitting of an array into smaller ones. Like merge sort, quick sort also falls into the category of divide and conquer approach of problem-solving methodology".[7]
An important point about using quick sort algorithm is that the higher sample size you have the more chance that recursion error occurs. In this regard, being able to measure the running time of quick sort algorithm for all of the sample sizes, we choose the position of pivot element, which is "a somewhat arbitrary element in the collection and using it helps to partition (or divide) the larger unsorted collection into two, smaller lists" [8] , as the median element of the data set. Consequently we will be able to use this algorithm for all the sample sizes we have without any recursion errors. Note that based on Cormen et al. [2009, Ch. 7.1] the pivot element is being choosed basically as the last element of sets.
Like the previous algorithms we have discussed so far, the results are shown in the following plot.

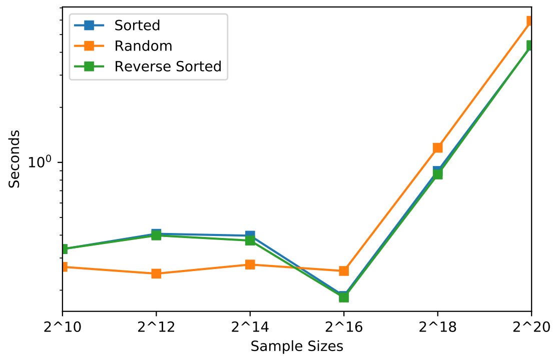

Figure 5: Running Times of Quick Sort Algorithm for Different Sample Sizes

The following table shows the standard deviations of running times for each data set in different sizes.

[7]https://www.interviewbit.com/tutorial/quicksort-algorithm/
[8]https://medium.com/basecs/pivoting-to-understand-quicksort-part-1-75178dfb9313

Table 4: Standard Deviations of Quick Sort Running Times for Different Sample Sizes and Data Sets

Sample Size	Sorted	Reverse Sorted	Randomised
n_1	0.011	0.012	0.008
n_2	0.014	0.022	0.014
n_3	0.014	0.017	0.013
n_4	0.021	0.012	0.009
n_5	0.037	0.037	0.0037
n_6	0.167	0.227	0.421

The running time of quick sort algorithm runs from 0.2 to 6 seconds. Moreover, when the sample size is less equal than $n_3 = 2^{14}$ the randomised data set can be considered as the best case for quick sort algorithm since the time takes to sort this data set is the lowest among three cases.
Since sorted data set is the worst case for quick sort algorithm, the expected running time for sorting this data set would be discussed separately. In this respect, considering $c_1 = 0.0.000001$ and $c_2 = 0.000000000001$ and $T(n) = \Theta(n^2)$ upper and lower bounds for sorted data set would be as figure (6).

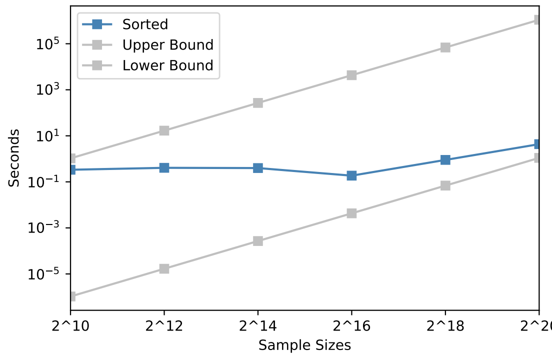

Figure 6: Upper and Lower Bounds for Quick Sort Running Time Function (Sorted Data Set)

For the two other data sets (reverse sorted and randomised data sets) the running function is the same as merge sort algorithm. In this regard, considering $c_1 = 0.0034$ and $c_2 = 0.00001$ we will have an upper and a lower bound for $T(n) = \Theta(n \lg n)$ as figure (7).
Thus, we can conclude that for quick sort algorithm the expected running time of sorted data set is $\Theta(n^2)$ and for two other sets (randomised and reverse sorted) is $\Theta(n \lg n)$ and this demonstrates our theoretical expectations.

5. Discussion

5.1 The Expected Running Time

Since we could set upper and lower bounds for merge and quick sort and set upper bound for heap sort running times' functions, we can conclude that theoretical expectations hold. Expected running time of merge (for all data sets) and quick sort (for two data sets of reverse sorted and randomised) is $T(n) = \Theta(n \lg n)$. Besides, running time

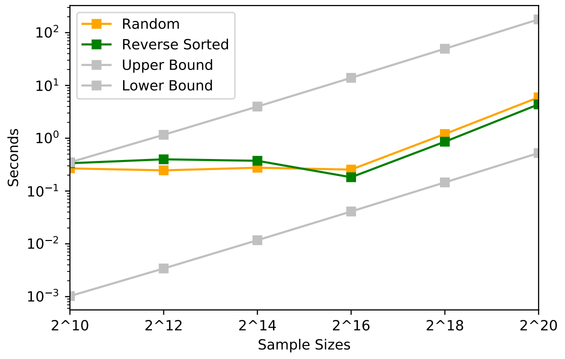

Figure 7: Upper and Lower Bounds for Quick Sort Running Time Function (Reverse Sorted and Randomised Data Sets)

of quick sort algorithm for sorted data set (which is considered as the worst case in the real world) is $\Theta(n^2)$. Lastly the expected running time for heap sort (for all data sets) is $T(n) = O(n \lg n)$.

5.2 Categorising Cases of Data Sets

We are also eager to know that for each sorting algorithm which data set could be the best one for it to be sorted during the shortest time and which data set can be considered as the worst one to be sorted in the longest period of time. In this regard, the following table wraps up the result of mentioned problem.

Table 5: Categorising Cases of Data Sets Based on Performance of Sorting Algorithms

Algorithms	$n \leq n_3 = 2^{14}$		$n \geq n_4 = 2^{16}$	
	Best Case	Worst Case	Best Case	Worst Case
Merge Sort	N.D	N.D	Sorted	Randomised
Heap Sort	N.D	N.D	R.sorted	Randomised
Quick Sort	Randomised	Sorted	R.sorted	Randomised

Note that in the table 5, N.D means no significant difference of algorithm's performance among data sets and R.sorted abbreviates reverse-sorted.

5.3 Comparing Performances of Sorting Algorithms

Summarising performances of all algorithms we can see that if the sample size is less than $n_3 = 2^{14}$ all of the sorting algorithms perform relatively equal. However for larger sample sizes, the heap sort algorithm has the poorest performance for especially the randomised data set. However, when we have the reverse sorted data set, quick sort algorithm can perform relatively fast.

Regardless of data set types, if we take average over running times of our sorting algorithms to generally see which algorithm performs the best, we will have the figure 8 which obviously shows that for relatively smaller sample sizes (n $\leq 2^{14}$) merge and heap sort are performing faster while for the relatively larger sample sizes (n $\geq 2^{16}$) quick sort outperforms and can be considered as a better algorithm to sort, regrading its running time.

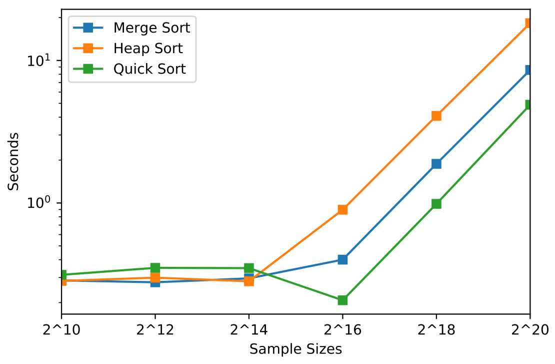

Figure 8: Running Times of Three Sorting Algorithms for Different Sample Sizes (on average)

Acknowledgments

We would like to show our gratitude to our professor Hans Ekkehard Plesser for sharing his knowledge with us during the course of this research. We also thank Krista Gilman, teacher assistant of the course of this research, for her patience for answering all of our questions.

References

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2009. *Introduction to Algorithms, Third Edition* (3rd ed.). The MIT Press, Cambridge, MA.