# Comparision Based Sorting Algorithms

Arpit Gupta

## Abstract

This document presents a brief discussion on sorting algorithms. Algorithms for `Quicksort` is provided in this document and its working is explained. Further, a proof of lower bounds on sorting is presented in this document. Most of the content presented here is created by referring and reproducing contents from one of the widely followed book on Algorithms by Cormen et al. [1]. **We do not claim originality of this work**. This document is prepared as part of an assignment for the Software Lab Course (CS251) to learn LaTeX.

Declaration: I have prepared this document using LaTeXwithout any unfair means. Further, while the document is prepared by me, I do not claim the ownership of the ideas presented in this document.

## 1 Introduction

Sorting is one of the most fundamental operations in computer science useful in numerous applications. Given a sequence of numbers as input, the output should provide a non decreasing sequence of numbers as output. More formally, we define a sorting problem as follows [1],
**Input:** A sequence of n numbers $\langle a_1, a_2, ..., a_n \rangle$.
**Output:** A reordered sequence (of size n) $\langle a'_1, a'_2, ..., a'_n \rangle$ of the input sequence such that $a'_1 \leq a'_2 \leq .... \leq a'_3$.
Consider the following example. Given an input sequence $\langle 8, 34, 7, 9, 15, 91, 15 \rangle$ , a sorting algorithm should return $\langle 7, 8, 9, 15, 15, 34, 91 \rangle$ as output.

A fundamental problem like this sorting has affected many researchers who contributed with innovative algorithms to solve the problem of soring *efficiently* [3]. Efficiency of an algorithm depends on primarily two aspects,

- **Time Complexity** is a formalism that captures running time of an algorithm in terms of the input size. Normally, *asymptotic* behaviour on the input size is used to analyze the time complexity of algorithms.

- **Space Complexity** is a formalism that captures amount of memory used by an algorithm in terms of input size. Like time complexity analysisasymptotic analysis is used for space complexity.

In the branch of algorithms and complexity in computer science, space complexity takes a back seat compared to time complexity. Recently, another parameter of computing i.e., energy consumption has become popular. Roy et al. [4] proposed an energy complexity model for algorithms. In this document, we will deal with time complexity of sorting algorithms.

One class of algorithms which are based on *elementary comparison* are commonly known as *comparision based sorting algorithms*. In this document we will provide a brief overview of `Quicksort`, a commonly used sorting algorithm [2]. Quicksort is a sorting algorithm based on *divide-and-conquer* paradigm of algorithm design. Further, we will derive the lower bound of any comparision based sorting algorithm to be $\Omega(n \log_2 n)$ for an input of size n.

## 2 Quicksort

Quicksort is designed as a three-step divide-and-conquer process for sorting an input sequence in an array [1]. For any given subarray, A[$i..j$], the process is as follows,
**Divide:** The array A[$i..j$] is partitioned into two subarrays A[$i..k$] and A[$k+1..j$] such that all elements in A[$i..k$] is less than or equal to all elements in A[$k+1..j$]. A partitioning procedure is called to determine $k$ such that at the end of partirioning, the element at the $k^{th}$ position (A[$k$]) does not change its position in the final output array.

**Algorithm 1** Partition procedure of `Quicksort` algorithm

1: **procedure** Partition($A$,$i$,$j$)
   ▷ A is an array of N integers, $A[1...N]$
   ▷ $i$ and $j$ are the start and end of subarray
2:    $x \leftarrow A[i]$
3:    $y \leftarrow i\text{-}1$
4:    $z \leftarrow j\text{+}1$
5:    **while** (*true*) **do**
6:       $z \leftarrow z$ - 1
7:       **while** A[z] > x **do**
8:          $z \leftarrow z\text{-}1$
9:       **end while**
10:       $y \leftarrow y + 1$
11:       **while** A[y] < x **do**
12:          $y \leftarrow y\text{+}1$
13:       **end while**
14:       **if** y < z **then**
15:          Swap A[y] $\leftrightarrow$ A[z]
16:       **else**
17:          return z
18:       **end if**
19:    **end while**
20: **end procedure**

**Conquer:** Recursively invoke `Quicksort` on the two subarrays. This procedure conquers the complexity by applying the same operations in two subarrays.

**Merge:** Quicksort does not require merge or combine operatoiin as the entire array A[i..j] is sorted in place.

In the heart of `Quicksort`, there is a partition procedure as shown in Algorithm 1. A pivot element x is selected. The first inner while loop (line #6) continues examining elements until it finds an element that is smaller than or equal to pivot element.Similarly, second inner while loop (line #9) continues examining elements until it finds an element greater than or equal to the pivot element. If indices y and z have not exchanged their sides around the pivot, the elements at A[$y$] and A[$z$] are exchanged. Otherwise, the procedure returns to the index $z$, such that all the elements to the left of z are smaller than or equal to A[$z$] and all the elements to the right of z are greater than or equal to A[$z$].

The main recursive procedure for `Quicksort` is

**Algorithm 2** `Quicksort` recursion

1: **procedure** Quicksort($A$,$i$,$j$)
   ▷ Quicksort procedure called with A,1,N
2:    **if** i < j **then**
3:       k $\leftarrow$ Partition(A,$i$,$j$)
4:       Quicksort(A,$i$,$k$)
5:       Quicksort(A,$k$+1,$j$)
6:    **end if**
7: **end procedure**

presented in Algorithm . Initial invocatioon is formed by call QUICKSORT(A,1,N) wher $N$ is the length of array N.

## 2.1 Time complexity analysis of `Quicksort`

The worst case of `Quicksort` occurs when array of length$N$, gets partitioined into two subarrays of size N-1 and 1in every recursion invocatoion of QUICKSORT procedure on ALgorithm 2. The procedurepartitioning procedure presented in ALgorithm 1, takes $\Theta$ (n) time, the recurrence relation for running time is,

$$T(n) = T(n-1) + \Theta(n)$$

As it is evident that $T(1) = \Theta(1)$, the recurrence is solved as follows,

$$T(n) = T(n-1) + \Theta(n)$$
$$= \sum_{k=1}^{n} \Theta(k)$$
$$= \Theta\left(\sum_{k=1}^{n}(k)\right)$$
$$= \Theta(n^2)$$

Therefore, if the partitioning is always maximally unbalanced, the running time is $\Theta(n^2)$. Intuitively, if an input sequence is almost sorted, `Quicksort` will perform poorly. In the best case, partitioning divides the array into two equal parts. Thus, the recurrence for the best case is given by,

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

which evaluates to $\Theta(n \log_2 n)$. Using a comparatively involved analysis, the average running time of `Quicksort` can be determined to be $O(n \lg n)$.

# 3 Lower bounds on comparison sorts

An interesting question arises about sorting algorithms based on comparisons is the following: What is the lower bound of this class of sorting algorithms? This question is important for algorithm researchers to further improve the sorting algorithm.

A decision tree based analysis leads to the following theorem [1].

**Theorem 1.** *Any decision tree that sorts $n$ elements has height $\Omega(n \log_2 n)$.*

*Proof.* Consider a decision tree of height $h$ that sorts n elements Since there are $n!$ permutations of n elements, each permutatioon representing a distinct sorted order, the tree must have at least $n!$ leaves. Since a binary tree of height $h$ has no more than $2^h$ leaves. So,

$$n! \leq 2^h$$

Applying logarithmic ($\log_2$) the inequality becomes,

$$h \geq lg(n!)$$

Applying Stirling's approximation,

$$n! > \left(\frac{n}{e}\right)^n$$

where $e$ is natural base of logarithms. Further,

$$
\begin{aligned}
h &\geq lg\left(\frac{n}{e}\right)^n \\
&= n\ lgn - n\ lge \\
&= \Omega(n\ lgn)
\end{aligned}
$$

$\square$

# 4 Conclusion

In this document, we have provided a discussion on sorting algorithms. We included algorithms for `Quicksort` and explained its working. Further, a proof of lower bounds on sorting is presented in this document. Most of the content presented here is created by referring and reproducing contents from one of the widely followed book on Algorithms by Cormen et al. [1]. We do not claim originality of this work. This document is prepared as part of an assignment for the Software Lab Course (CS251) to learn LaTeX.

# References

[1] CORMEN, T. H., LEISERSON, C.E., RIVEST, R.L., AND STEIN, C. *Introduction to Algorithms, Third Edition*, 3rd ed. The MIT Press, 2009.

[2] HOARE, C. A. R. Algorithm 64: Quicksort. *Communications of ACM 4*,7 (1961), 321-.

[3] MARTIN, W. A. Sorting. *ACM Computing Survey 3*, 4 (1971), 147-174.

[4] ROY, S., RUDRA, A., AND VERMA,A. An energy complexity model for algorithms. In *Proceedings of the 4th Conference on Innovations in Theoretical Computer Science* (2013), ITCS '13,pp. 283-304.