



Quicksort

Abstract

In this project, we examine Quicksort as a divide-and-conquer sorting algorithm. After presenting the algorithm and its design, we analyze its time complexity, as well as its scaling properties, in order to explain why it has become such a widely popular choice in computer science. We also introduce the problem domain of computational geometry, where the solutions to many problems can be seen as generalizations of Quicksort. Finally, as a practical application, we present the convex hull problem and the Quickhull algorithm, featuring analogous runtime and scaling properties.

Student 1

Pratyai Mazumder

Student 2

Lodovico Mazzei

Student 3

Michele Chersich

1 Introduction

Sorting is a ubiquitous problem in computer science, with applications in practically every subfield – data structures, computation geometry, search engines, just to give a few examples.

The sorting problem can be defined as the reordering of a sequence of elements, based on some ordering criterion. More specifically, to be able to sort it, a sequence must possess following two properties¹:

- Its elements are *swappable*: So that the list can be permuted in-place².
- Any two of its elements are *comparable*: i.e. the elements have total ordering [1].

Additionally, for many sorting algorithms, such as Quicksort, the sequence must also be *randomly accessible*.

Many different algorithms were invented for the comparison sort problem. Quicksort, originally invented in 1962 by C. A. Hoare [4], is still one of the most commonly used sorting algorithms. The particularly interesting nature of Quicksort is that, unlike many other sorting algorithms, it does not even attempt to achieve the theoretical lower bound of worst-case complexity, i.e. $\Omega(n \log n)$ [3, p. 193]. Instead it has a fast average case and a much simpler implementation, making it a very practical choice.

There are many variants of the Quicksort algorithm, however, we will only focus on the worst-case and average-case analysis of one representative example.

2 The algorithm

2.1 The partitioning problem

Quicksort belongs to the family of *partition sort* algorithms, where a partitioning routine is called recursively until the whole sequence is sorted.

Given an array $A[1, \dots, N]$, the partitioning algorithm should split a permutation of it, A' , into two partitions $U = A'[1, \dots, p]$ and $V = A'[p + 1, \dots, N]$ such that $U[i] \leq V[j] \ \forall i \in [1, p], j \in [p + 1, N]$. By recursively partitioning U and V themselves, base cases will eventually be reached, consisting of single-element or empty arrays, which are already sorted. Figure 1 illustrates the recursive routine.

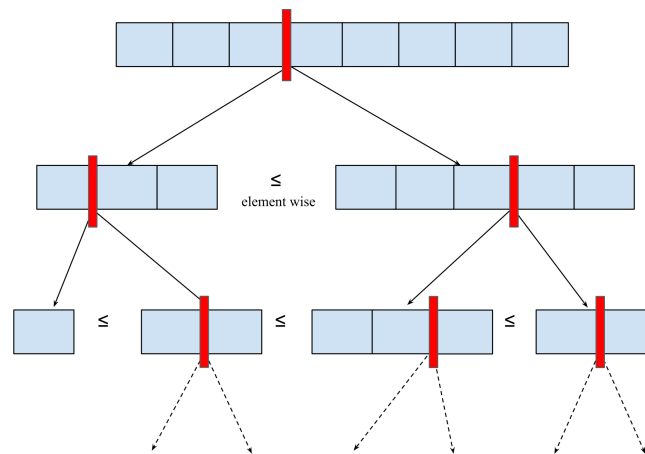


Figure 1. Tree representation of the partitioning process. At each level the array is permuted and split into two subarrays, where all elements in the left subarray are smaller than or equal to all elements in the right subarray. The base case of a branch is reached when an array is reduced to one or zero elements.

The following pseudocode recursively sorts a given subarray of A . It references a *partition* routine which will be discussed in the following sections. Also, the *partition* routine will move one element (called *pivot*) to its final location, so that element can be excluded from the recursive calls.

¹Either of these requirements can be relaxed for a relaxed variant of the sorting problem.

²I.e. modifying the original sequence object, instead of creating a new one.

Algorithm 1: Quicksort ($A, low, high$)

```
1 if  $low \geq 0$  &&  $high \geq 0$  &&  $low < high$  then
2   pivot := partition( $A, low, high$ )
3   quicksort( $A, low, pivot-1$ )
4   quicksort( $A, pivot+1, high$ )
5 end
```

2.2 Pivot selection

The partitioning in Quicksort relies on pivoting. This consists in picking a value γ to be our pivot. Given an array A , we permute it such that every element before γ is smaller than it, and every one after it is larger. The internal order in these subarrays is irrelevant, as it will be addressed in further iterations of the algorithm.

Clearly, the pivot choice will have an impact on the efficiency of the algorithm. We divide the selection methods into two categories: one-step and multi-step. These are discussed using an example array $A[1, \dots, N]$, as follows.

2.2.1 One-step selection

These methods only require choosing a single value at each iteration and using it as the pivot. Some examples are:

- *First element*: select $A[1]$ as the pivot.
- *Last element*: select $A[N]$ as the pivot.
- *Central element*: letting $k = \frac{N+1}{2}$, select $A[\text{floor}(k)]$ or $A[\text{ceil}(k)]$ as the pivot.
- *Random element*: select $A[r]$ as the pivot, where r is a random number such that $1 \leq r \leq N$.

2.2.2 Multi-step selection

Given that the best partitioning is achieved when picking the median of some array sample of size T , where T is an odd number, typically $3 \leq T \leq 9$.

- *Median-of- T with fixed selection*: T integers $1 \leq n_1, \dots, n_T \leq N$ are fixed beforehand. The median of $\{A[n_1], \dots, A[n_T]\}$ is used as the pivot.
- *Median-of- T with random selection*: T random integers $1 \leq r_1, \dots, r_T \leq N$ are generated. The median of $\{A[r_1], \dots, A[r_T]\}$ is used as the pivot.

2.3 Partitioning around the pivot

There are several schemes to partition a list around the pivot. Two common choices are the Hoare scheme and the Lomuto scheme. For the purpose of this project, we will focus on the former.

Figure 2 illustrates the process of partitioning around a pivot. We start with two iterators i and j , initialized at offsets 0 and $N + 1$ respectively, and move the iterators towards each other while maintaining the conditions $A[i] < \text{pivot}$ and $A[j] \geq \text{pivot}$. When both iterators fail their conditions, we swap $A[i]$ and $A[j]$. When i and j meet, the partitioning step is complete. This process is illustrated in Algorithm 2.

Algorithm 2: partition($A, low, high$)

```
1 pivot = selectPivot(A)
2 i, j := low - 1, high + 1
3 loop forever
4   do i := i + 1 while  $A[i] < pivot$ 
5   do j := j - 1 while  $A[j] \geq pivot$ 
6   if  $i \geq j$  then
7     return
8   end
9   swap  $A[i]$  with  $A[j]$ 
```

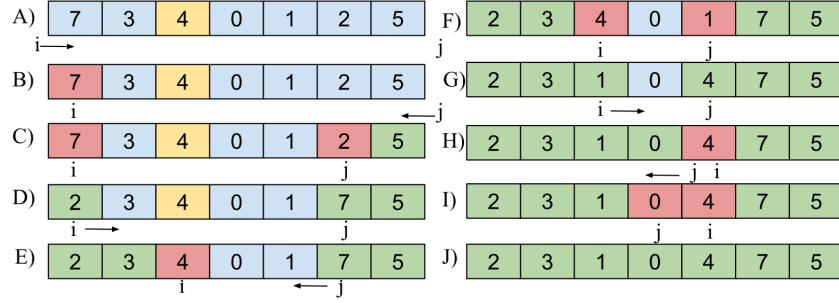


Figure 2. Partitioning of the array around the pivot $p = 4$ (yellow). Blue elements are yet to be checked; red elements need to be swapped; green elements are already in the correct partition.

A-B-C) Start with i at position 0 and j at position $N + 1$: i stops at 7, j stops at 2: $i < j$, so $A[i]$ and $A[j]$ are swapped. **D-E-F)** i next stops at 4, j stops at 1: $i < j$, so $A[i]$ and $A[j]$ are swapped. **G-H-I)** i and j move until they cross. **J)** End of iteration: all elements to the right of the pivot 4 are greater or equal than it, and all the ones to its left are smaller.

3 Complexity analysis

We will consider the notion of algorithm complexity to be the growth of its computational cost with respect to some input parameters. In general the cost can be any runtime resources the algorithm needs. Here, we will restrict the analysis to an estimation of how fast the computation time grows with the size of the input sequence.

We will also assume an extremely simplified model of execution that does not consider cache, locality of reference, hardware architecture, etc., all of which have significant impact on computation time. In this model, the only expensive operations are comparison and swap, each costing $\Theta(1)$. Everything else, including the recursive function calls and random number generations, is negligible.

3.1 Analysis of partition routine

Let's consider the Hoare scheme outlined in Algorithm 2. The first operation is the selection of the pivot, which is $\Theta(1)$ for all selection methods mentioned above.

Once the pivot is selected, we scan the subarray in $n = high - low + 1$ steps, each involving exactly one comparison between the current pointees, one occasional comparison between the pointers themselves, and one occasional swap between the current pointees. So, each step is $\Theta(1)$, and the total complexity of the partition routine is:

$$T_{partition}(n) = \Theta(1) + n\Theta(1) \approx \Theta(n)$$

This complexity estimation holds true for all the pivot selection and partitioning schemes we have discussed, and also for any input data.

3.2 Analysis of quicksort routine

Now, let's consider the recursive quicksort routine outlined in Algorithm 1. The first operation is a comparison to check the base case, which requires no additional computation. Otherwise, we incur into exactly one call to the

partition routine and two calls to the quicksort routine with the two partitions. If the two partitions have sizes u and $n - u - 1$, the complexity of the quicksort routine is:

$$T(n) = T_{\text{partition}}(n) + T(u) + T(n - u - 1) = T(u) + T(n - u - 1) + \Theta(n)$$

This is a recursive formula where the partition sizes, u and $n - u - 1$, depend on the input data and the pivot selection method. So, in order to get an explicit formula without recursion, we need to consider different input and pivot scenarios. We will do this for the following two important cases, which are also the most typical for complexity analysis.

3.2.1 Worst-case analysis

The worst case for the quicksort routine occurs when the partitions are the least balanced, i.e. the pivot element ends up being the largest or the smallest in the subarray. Therefore,

$$\begin{aligned} T_{\text{worst}}(0) &= \Theta(1) \\ T_{\text{worst}}(1) &= \Theta(1) \\ T_{\text{worst}}(n) &= T_{\text{worst}}(n - 1) + T_{\text{worst}}(0) + \Theta(n) \end{aligned}$$

Expanding this recurrence until the base case, we find

$$T_{\text{worst}}(n) = \sum_{k=1}^n \Theta(k) = \Theta(n^2)$$

3.2.2 Average-case analysis

For deterministic pivot selection methods, there is always some hostile input data for which the algorithm will meet the worst case. However, in practical scenarios, running into hostile input data is fairly unlikely. Instead, we are often interested in the average case analysis, that accounts for the prior distribution of the input data to compute the average cost.

Since the locations of the pivots determine the runtime of Quicksort for a given input data, and there is no other information about the pivot, we will consider the prior distribution of the pivot location at each call to the quicksort routine to be uniform, i.e. if the size of the given array is n , then its final location can be anywhere between and including 1 and n . So, the average sorting time will be

$$\begin{aligned} T(n) &= \frac{1}{n} \sum_{k=1}^n (T(k - 1) + T(n - k)) + \Theta(n) \\ &= \frac{1}{n} \left(\sum_{k=1}^n T(k - 1) + \sum_{k=1}^n T(n - k) \right) + \Theta(n) \\ &= \frac{2}{n} \sum_{k=0}^{n-1} T(k) + cn \quad [\text{for some constant } c] \\ nT(n) &= 2 \sum_{k=0}^{n-1} T(k) + cn^2 \\ nT(n) - (n - 1)T(n - 1) &= 2T(n - 1) + 2cn - c \\ \implies nT(n) &\approx (n + 1)T(n - 1) + 2cn \end{aligned}$$

From this recurrence we get the set of equations

$$\begin{aligned} \frac{T(n)}{n + 1} &= \frac{T(n - 1)}{n} + \frac{2c}{n + 1} \\ \frac{T(n - 1)}{n} &= \frac{T(n - 2)}{n - 1} + \frac{2c}{n} \\ &\vdots \\ \frac{T(2)}{3} &= \frac{T(1)}{2} + \frac{2c}{3} \end{aligned}$$

Adding them up yields

$$\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2c \sum_{k=3}^{n+1} \frac{1}{k} = \frac{T(1)}{2} + 2c \left(\log(n+1) + \gamma - \frac{3}{2} \right) \quad [\gamma \approx 0.577 \text{ is the Euler's constant}]$$

$$\Rightarrow T(n) = O(n \log n)$$

3.3 Analysis of randomized Quicksort

Instead of deterministic pivots, we can also select a random element from the subarray, with uniform probability. This way we obtain the exact same analysis of the average case and the complexity order $O(n \log n)$. However, there are a few notable distinctions:

- Our execution model ignores the cost of random number generation (RNG). With randomized pivots, we generate $O(n \log n)$ random numbers, which can have substantial impact, depending on the RNG implementation.
- With deterministic pivot selection, there is always some hostile permutation of the input array for which the algorithm will degenerate to the worst case. With randomized pivots, for any input data the expectation is the average case. Theoretically, the RNG can still select a pivot on the boundary everytime. However, it is extremely unlikely in practice for any decent RNG implementation and a reasonably large array.

4 Parallelizing Quicksort

Quicksort is fairly simple to parallelize with a few modifications. One distributed approach, called *sample sorting*, is briefly outlined here.

The algorithm starts with the input sequence of size n , evenly distributed over p processors, connected by some network. Every processor holds a partition of roughly $\frac{n}{p}$ size. The procedure of the distributed version of Quicksort is given in Algorithm 3.

Algorithm 3: Distributed Quicksort

- 1 Each processor selects s randomly sampled elements as candidate pivots from the data it holds, and sends them to the *manager* processor.
 - 2 The manager processor sorts the candidate pivots locally and select $p - 1$ evenly spaced pivots, and sends the selected pivots to every worker processor.
 - 3 Every processor locally p -partitions its own data using the $p - 1$ pivots received from the manager processor. It may happen that some partitions turn out empty this way.
 - 4 Every processor sends its local k -th partition to the k -th processor in the network.
 - 5 Every processor collects all the local partitions it received and sorts them locally, resulting in a distributed sorted permutation of the original input sequence.
-

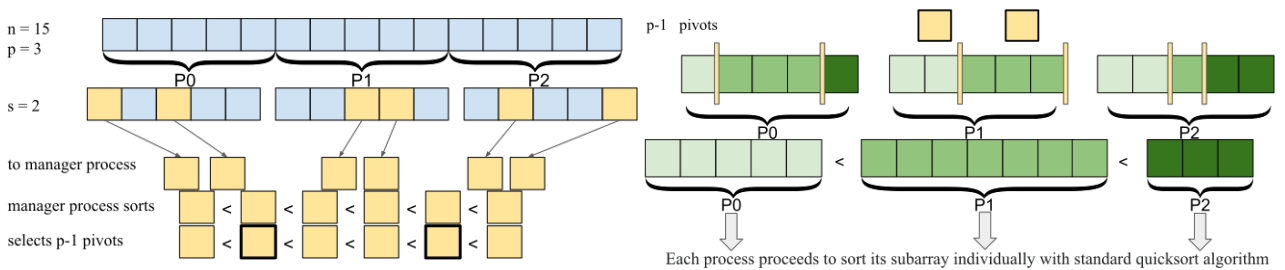


Figure 3. Distributed sample sorting: in the pivot selection process (left), each processor selects 2 candidate pivots (yellow) from the local data, then the manager process sorts all the candidate pivots to pick 2 pivots (yellow with thick border). All of the selected pivots are passed to each processor (right), which p -partitions the data with these pivots (the partitions are highlighted in shades of green) and send each partition to the appropriate processor. Finally each processor locally sorts the accumulated partitions.

The challenges of such distributed sorting algorithms are about optimizing inter-processor communication, which usually cost much more than local operations.

5 Computational geometry and the convex hull problem

Computational geometry is the study of algorithms for the solution of geometric problems in the Euclidean space [5]. The convex hull problem belongs to this class of problems, and has a wide range of applications across several disciplines (e.g. data classification, collision avoidance, image processing and recognition). It is defined as follows: given a set of points, find the smallest convex polygon containing all the points [2]. For the purpose of this project, we will consider only the planar convex hull, with 2D Cartesian coordinates.

5.1 The Quickhull algorithm

The Quickhull algorithm is a variation of Quicksort for the solution of the convex hull problem. The algorithm works as follows: an initial partitioning step is done by picking the leftmost and rightmost points. Let these two points be respectively p_1 and p_2 . The line connecting p_1 and p_2 splits the set in two parts. For each subset, we search for the farthest point from the line: let this point be p_{max} . If it exists, the points lying inside the triangle $p_1p_2p_{max}$ are removed from the set, and the recursive step is applied on the points that lie outside the lines p_1p_{max} and p_2p_{max} . Again, let p_1 and p_2 be the end points of the line in the recursive step. The stop condition occurs when p_{max} does not exist, as there are no points outside p_1p_2 . The very first steps of the algorithm are illustrated in Figure 4. The pseudocode is given in Algorithms 4 and 5.

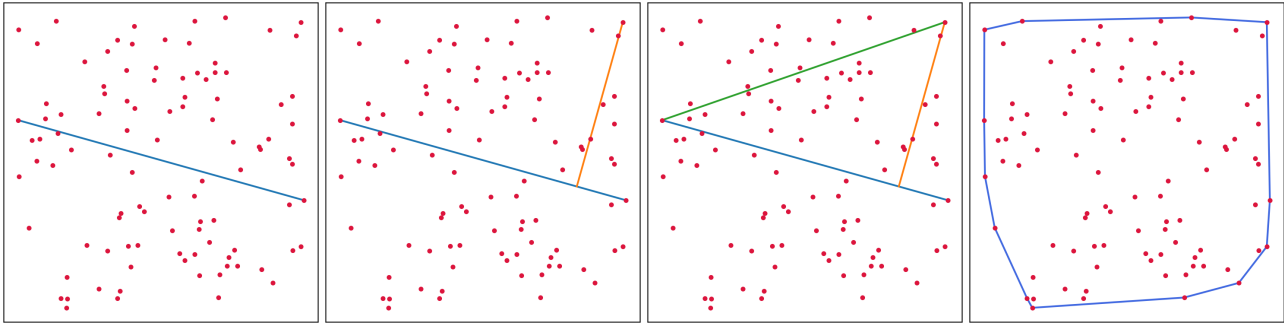


Figure 4. Three steps of the Quickhull algorithm: the line p_1p_2 partitions the set of points in two subsets (1); the farthest point p_{max} from the line is found (2); the recursive step considers just the outer points for the solution (3); at the end, the point set P is just the convex hull (4).

Algorithm 4: Quickhull (P)

- 1 P is a set of points with coordinates (x, y)
 - 2 Find the points p_1 and p_2 with x -coordinates x_{min} and x_{max}
 - 3 Quickhull_onside($P, p_1, p_2, 1$)
 - 4 Quickhull_onside($P, p_1, p_2, -1$)
-

Algorithm 5: Quickhull_onside (P, p_1, p_2, s)

- 1 Let p_{max} be the point that is farthest from the line p_1p_2 , on side s
 - 2 **if** p_{max} does not exist **then**
 - 3 return
 - 4 **end**
 - 5 Remove from P all points that lie in the triangle $p_1p_2p_{max}$
 - 6 Let s_1 and s_2 be the outer sides of lines p_1p_{max} and p_2p_{max}
 - 7 Quickhull_onside(P, p_{max}, p_1, s_1)
 - 8 Quickhull_onside(P, p_{max}, p_2, s_2)
-

We can observe how Quickhull, just like Quicksort, is both a divide-and-conquer and a sorting algorithm. The former approach is implemented by partitioning, splitting the initial problem into smaller subproblems in a recursive fashion, whereas the latter is applied by comparing Euclidean distances between points. Again, the complexity is $O(n^2)$ in the worst case and $O(n \log n)$ in the average case.

5.2 Distributed Quickhull

The Quickhull algorithm is sequential by design, as each recursive call depends on the previous ones for the computation of the solution set [6]. Therefore, parallelizing the algorithm by multithreading is not feasible, as it is not possible to independently compute the solution to subproblems. However, the set of points can be distributed over multiple processes, and inter-process communication can be used to combine the partial results.

The procedure of the distributed version of Quickhull is given in Algorithm 6.

Algorithm 6: Distributed Quickhull (P)

- 1 Each process computes the points with minimum and maximum x-coordinate among their local dataset, then the global minimum and maximum are computed, and broadcast to all processes (*allreduce*).
 - 2 Each process computes the farthest point from the line joining the two points, and the global result is found (*allreduce*).
 - 3 Once each process has all three vertices of the triangle, they can remove the points that lie inside the triangle from their dataset.
 - 4 The recursive step is repeated just like in the sequential version.
 - 5 At the end, the solution is scattered over all processes, so it must be gathered on the root process (*gather*).
-

The procedure is almost identical to the sequential version. The key difference is the use of inter-process communication in the partitioning steps: the computation performed by each process to find the leftmost and rightmost points, as well as the farthest point at every iteration, is limited to a subset of points, so the results of all processes must be combined, in order to obtain the global result. Note that communication is used just for the aforementioned purpose, and to combine the final solution on the root process, keeping the communication cost low.

5.2.1 Scaling analysis

Running the algorithm for large input sizes with different numbers of processes, we can observe significant gains in performance. A scaling chart is reported in Figure 5.

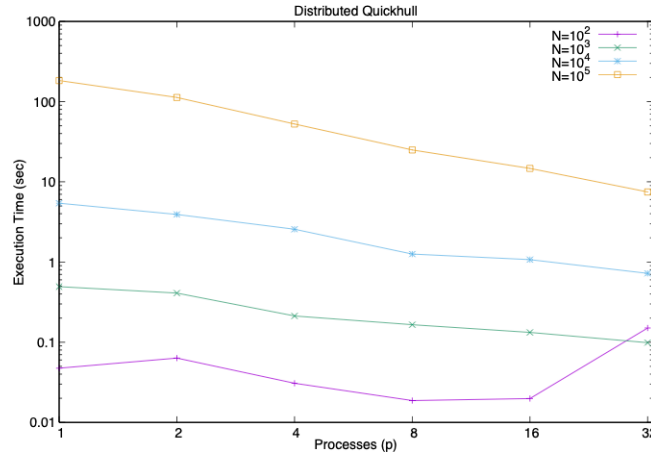


Figure 5. Performance scaling of distributed Quickhull.

We can observe the gain in performance obtained by the distributed version of the algorithm: by doubling the number of processes, the execution time decrease by a factor 2, if the problem size is large enough (e.g. $N = 10^5$). However, for small problem sizes, we can observe the effect of over-parallelizing, as the execution time decreases by smaller and smaller factors, or does not decrease at all (e.g. for $N = 10^2$, 32 processes perform worse than 16).

References

- [1] https://en.wikipedia.org/wiki/Total_order.
- [2] https://en.wikipedia.org/wiki/Computational_geometry.
- [3] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.
- [4] C. A. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.
- [5] J. JaJa. A perspective on quicksort. *Computing in Science & Engineering*, 2(1):43–49, 2000.
- [6] J. Ramesh and S. Suresha. Convex hull-parallel and distributed algorithms.