TEST
TEST TEST
TEST TEST TEST

2025/26

Table of Contents

Quicksort

Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of a random access file or an array in order. Developed by British computer scientist Tony Hoare in 1959 and published in 1961 it is still a commonly used algorithm for sorting. When implemented well, it can be about two or three times faster than its main competitors, merge sort and heapsort.

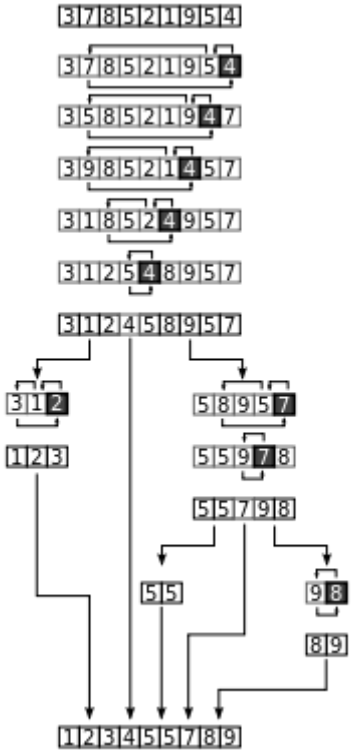


Figure 1 Full example of quicksort on a random set of numbers. The shaded element is the pivot. It is always chosen as the last element of the partition.

Algorithm

Quicksort is a divide and conquer algorithm. Quicksort first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps are:

Pick an element, called a pivot, from the array.

Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.

Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

Lomuto partition scheme

This scheme is attributed to Nico Lomuto and popularized by Bentley in his book Programming Pearls and Cormen et al. in their book Introduction to Algorithms. This scheme chooses a pivot that is typically the last element in the array. The algorithm maintains index i as it scans the array using another index j such that the elements lo through i-1 (inclusive) are less than the pivot, and the elements i through j (inclusive) are equal to or greater than the pivot. As this scheme is more compact and easy to understand, it is frequently used in introductory material, although it is less efficient than Hoare's original scheme. This scheme degrades to $O(n^2)$ when the array is already in order.

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := partition(A, lo, hi)
        quicksort(A, lo, p - 1)
        quicksort(A, p + 1, hi)

algorithm partition(A, lo, hi) is
    pivot := A[hi]
    i := lo
    for j := lo to hi do
        if A[j] < pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Implementation issues

Choice of pivot

In the very early versions of quicksort, the leftmost element of the partition would often be chosen as the pivot element. Unfortunately, this causes worst-case behavior on already sorted arrays, which is a rather common use-case. The problem was easily solved by choosing either a random index for the pivot, choosing the middle index of the partition or (especially for longer partitions) choosing the median of the first, middle and last element of the partition for the pivot (as recommended by Sedgewick). This "median-of-three" rule counters the case of sorted (or reverse-sorted) input, and gives a better estimate of the optimal pivot (the true median) than selecting any single element, when no information about the ordering of the input is known.

Repeated elements

With a partitioning algorithm such as the Lomuto partition scheme described above (even one that chooses good pivot values), quicksort exhibits poor performance for inputs that contain many repeated elements. The problem is clearly apparent when all the input elements are equal: at each recursion, the left partition is empty (no input values are less than the pivot), and the right partition has only decreased by one element (the pivot is removed). Consequently, the Lomuto partition scheme takes quadratic time to sort an array of equal values. However, with a partitioning algorithm such as the Hoare partition scheme, repeated elements generally results in better partitioning, and although needless swaps of elements equal to the pivot may occur, the running time generally decreases as the number of repeated elements increases (with memory cache reducing the swap overhead).

Formal analysis

Worst-case analysis

The most unbalanced partition occurs when one of the sublists returned by the partitioning routine is of size $n - 1$. This may occur if the pivot happens to be the smallest or largest element in the list, or in some implementations (e.g., the Lomuto partition scheme as described above) when all the elements are equal.

If this happens repeatedly in every partition, then each recursive call processes a list of size one less than the previous list. Consequently, we can make $n - 1$ nested calls before we reach a list of size 1. This means that the call tree is a linear chain of $n - 1$ nested calls. The ith call does $O(n - i)$ work to do the partition, and $\sum_{i=0}^{n}(n - i) = O(n^2)$

Insertion sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

Simple implementation

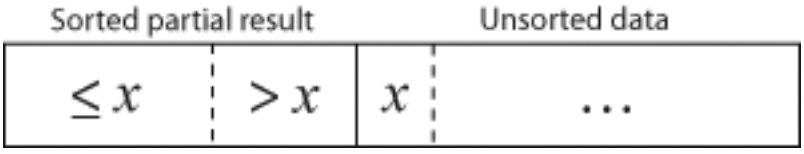
Efficient for (quite) small data sets

More efficient in practice than most other simple quadratic algorithms such as selection sort or bubble sort

Algorithm

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. At each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

The resulting array after k iterations has the property where the first k + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:



becomes