History

The quicksort algorithm was developed in 1959 by Tony Hoare while in the Soviet Union, as a visiting student at Moscow State University. At that time, Hoare worked in a project on machine translation for the National Physical Laboratory. As a part of translation process, he needed to sort the words of Russian sentences prior to looking them up in a Russian-English dictionary which was already sorted in alphabetic order on magnetic tape.[4] After recognizing that his first idea, insertion sort, would be slow, he quickly came up with a new idea that was Quicksort. He wrote a program in Mercury Autocode for the partition but couldn't write the program to account for the list of unsorted segments. On return to England, he was asked to write code for Shellsort as part of his new job. Hoare mentioned to his boss that he knew of a faster algorithm and his boss bet sixpence that he didn't. His boss ultimately accepted that he had lost the bet. Later, Hoare learned about ALGOL and its ability to do recursion which enabled him to publish the code in ACM.[5]

Quicksort gained widespread adoption, appearing, for example, in Unix as the default library sort function. Hence, it lent its name to the C standard library function `qsort`[6] and in the reference implementation of Java.

Robert Sedgewick's Ph.D. thesis in 1975 is considered a milestone in the study of Quicksort where he resolved many open problems related to the analysis of various pivot selection schemes including Samplesort, adaptive partitioning by Van Emden[7] as well as derivation of expected number of comparisons and swaps.[6] Bentley and McIlroy incorporated various improvements for use in programming libraries, including a technique to deal with equal elements and a pivot scheme known as *pseudomedian of nine,* where a sample of 9 elements is divided into groups of 3 and then the median of the 3 medians from 3 groups is chosen.[6] Jon Bentley described another simpler and compact partitioning scheme in his book Programming Pearls that he attributed to Nico Lomuto. Later Bentley wrote that he used Hoare's version for years but never really understood it but Lomuto's version was simple enough to prove correct.[8] Bentley described Quicksort as the "most beautiful code I had ever written" in the same essay. Lomuto's partition scheme was also popularized by the textbook Introduction to Algorithms although it is inferior to Hoare's scheme because it does 3 times more swaps on average and degrades to $O(n^2)$ runtime when all elements are equal.[9]

In 2009, Yaroslavskiy proposed the new dual pivot Quicksort implementation.[10] In the Java core library mailing lists, he initiated a discussion claiming his new algorithm to be superior to the runtime library's sorting method, which was at that time based on the widely used and carefully tuned variant of classic Quicksort by Bentley and McIlroy.[11] Yaroslavskiy's Quicksort has been chosen as the new default sorting algorithm in Oracle's Java 7 runtime library after extensive empirical performance tests.[12]

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:

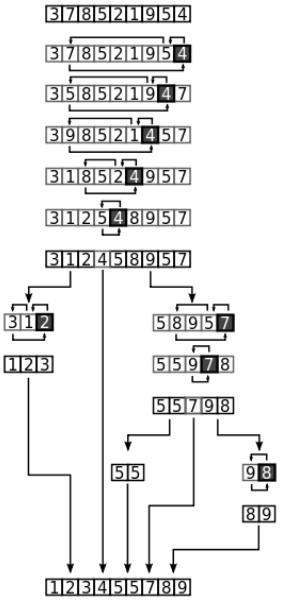
1. Pick an element, called a *pivot*, from the array.
2. *Partitioning*: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the *partition* operation.
3. 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.

The base case of the recursion is arrays of size zero or one, which never need to be sorted.

The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.

Lomuto partition scheme

This scheme is attributed to Nico Lomuto and popularized by Bentley in his book Programming Pearls[13] and Cormen *et al.* in their book *Introduction to Algorithms*.[14] This scheme chooses a pivot which is typically the last element in the array. The algorithm maintains the index to put the pivot in variable `i` and each time when it finds an element less than or equal to pivot, this index is incremented and that element would be placed before 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.[15] This scheme degrades to $O(n^2)$ when the array is already sorted as well as when the array has all equal elements.[9] There have been various variants proposed to boost performance including various ways to select pivot, deal with equal elements, use other sorting algorithms such as Insertion sort for small arrays and so on. In pseudocode, a quicksort that sorts elements `lo` through `hi` (inclusive) of an array A can be expressed as:[14]



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. However, always choosing the last element in the partition as the pivot in this way results in poor performance ($O(n^2)$) on *already sorted* arrays, or arrays of identical elements. Since sub-arrays of sorted / identical elements crop up a lot towards the end of a sorting procedure on a large set, versions of the quicksort algorithm which choose the pivot as the middle element run much more quickly than the algorithm described in this diagram on large sets of numbers.

```
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        // place for swapping
    for j := lo to hi - 1 do
        if A[j] ≤ pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[hi]
    return i
```

Sorting the entire array is accomplished by `quicksort(A, 1, length(A))`.

Hoare partition scheme

The original partition scheme described by C.A.R. Hoare uses two indices that start at the ends of the array being partitioned, then move toward each other, until they detect an inversion: a pair of elements, one greater than the pivot, one smaller, that are in the wrong order relative to each other. The inverted elements are then swapped.[16] When the indices meet, the algorithm stops and returns the final index. There are many variants of this algorithm, for example, selecting pivot from `A[hi]` instead of `A[lo]`. Hoare's scheme is more efficient than Lomuto's partition scheme because it does three times fewer swaps on average, and it creates efficient partitions even when all values are equal.[9] Like Lomuto's partition scheme, Hoare partitioning also causes Quicksort to degrade to $O(n^2)$ when the input array is already sorted; it also doesn't produce a stable sort. Note that in this scheme, the pivot's final location is not necessarily at the index that was returned, and the next two segments that the main algorithm recurs on are `[lo..p]` and `(p..hi]` as opposed to `[lo..p)` and `(p..hi]` as in Lomuto's scheme. In pseudocode,[14]

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

algorithm partition(A, lo, hi) is
    pivot := A[lo]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            j := j - 1
        while A[j] > pivot
        do
            i := i + 1
        while A[i] < pivot
        if i < j then
            swap A[i] with A[j]
        else
            return j
```

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).[17] 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.

Specifically, the expected number of comparisons needed to sort n elements (see § Analysis of randomized quicksort) with random pivot selection is $1.386\, n \log n$. Median-of-three pivoting brings this down to $C_{n,\,2} \approx 1.188\, n \log n$, at the expense of a three-percent increase in the expected number of swaps.[6] An even stronger pivoting rule, for larger arrays, is to pick the ninther, a recursive median-of-three, defined as[6]

$$\text{ninther}(a) = \text{median}(\text{median-of-three}(\text{first } \tfrac{1}{3} \text{ of } a), \text{median-of-three}(\text{middle } \tfrac{1}{3} \text{ of } a), \text{median-of-three}(\text{final } \tfrac{1}{3} \text{ of } a))$$

Selecting a pivot element is also complicated by the existence of integer overflow. If the boundary indices of the subarray being sorted are sufficiently large, the naïve expression for the middle index, $(lo + hi)/2$, will cause overflow and provide an invalid pivot index. This can be overcome by using, for example, $lo + (hi - lo)/2$ to index the middle element, at the cost of more complex arithmetic. Similar issues arise in some other methods of selecting the pivot element.

Repeated elements

With a partitioning algorithm such as the ones described above (even with 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 algorithm takes quadratic time to sort an array of equal values.

To solve this problem (sometimes called the Dutch national flag problem[6]), an alternative linear-time partition routine can be used that separates the values into three groups: values less than the pivot, values equal to the pivot, and values greater than the pivot. (Bentley and McIlroy call this a "fat partition" and note that it was already implemented in the `qsort` of Version 7 Unix.[6]) The values equal to the pivot are already sorted, so only the less-than and greater-than partitions need to be recursively sorted. In pseudocode, the quicksort algorithm becomes

```
algorithm quicksort(A, lo, hi) is
    if lo < hi then
        p := pivot(A, lo, hi)
        left, right := partition(A, p, lo, hi)  // note: multiple return values
        quicksort(A, lo, left)
        quicksort(A, right, hi)
```

The best case for the algorithm now occurs when all elements are equal (or are chosen from a small set of $k \ll n$ elements). In the case of all equal elements, the modified quicksort will perform at most two recursive calls on empty subarrays and thus finish in linear time.

Optimizations

Two other important optimizations, also suggested by Sedgewick and widely used in practice are:[18][19]

- To make sure at most $O(\log n)$ space is used, recurse first into the smaller side of the partition, then use a tail call to recurse into the