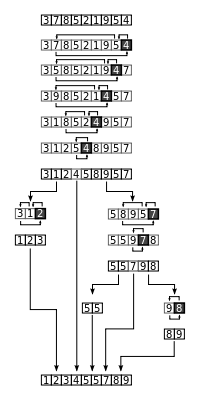
**Implementation Problems in Quick Sort**

**Algorithm**

[](http://en.wikipedia.org/wiki/File:Quicksort-diagram.svg)

[http://bits.wikimedia.org/static-1.24wmf10/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Quicksort-diagram.svg)

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.

Quicksort is a [divide and conquer algorithm](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm). Quicksort first divides a large array into two smaller sub-array: 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. 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](http://en.wikipedia.org/wiki/Recursion_(computer_science)) 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](http://en.wikipedia.org/wiki/Recursion_(computer_science)) of the recursion is arrays of size zero or one, which never need to be sorted. In [pseudocode](http://en.wikipedia.org/wiki/Pseudocode), a quicksort that sorts elements *i* through *k* (inclusive) of an array *A* can be expressed compactly as[[5]](http://en.wikipedia.org/wiki/Quicksort" \l "cite_note-5):171

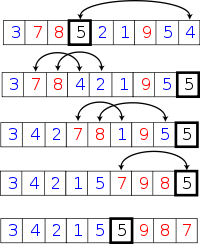
quicksort(A, i, k):

if i < k:

p := partition(A, i, k)

quicksort(A, i, p - 1)

quicksort(A, p + 1, k)

[](http://en.wikipedia.org/wiki/File:Partition_example.svg)

[http://bits.wikimedia.org/static-1.24wmf10/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Partition_example.svg)

In-place partition in action on a small list. The boxed element is the pivot element, blue elements are less or equal, and red elements are larger.

Sorting the entire array is accomplished by calling quicksort(A, 1, length(A)). The partition operation is step 2 from the description in English, above. It can be defined as:

*// left is the index of the leftmost element of the subarray*

*// right is the index of the rightmost element of the subarray (inclusive)*

*// number of elements in subarray = right-left+1*

partition(array, left, right)

pivotIndex := choose-pivot(array, left, right)

pivotValue := array[pivotIndex]

swap array[pivotIndex] and array[right]

storeIndex := left

**for** i **from** left **to** right - 1

**if** array[i] ≤ pivotValue

swap array[i] and array[storeIndex]

storeIndex := storeIndex + 1

swap array[storeIndex] and array[right] // *Move pivot to its final place*

**return** storeIndex

This is the in-place partition algorithm. It partitions the portion of the array between indexes *left* and *right*, inclusively, by moving all elements less than or equal array[pivotIndex] before the pivot, and the greater elements after it. In the process it also finds the final position for the pivot element, which it returns. It temporarily moves the pivot element to the end of the subarray, so that it doesn't get in the way. Because it only uses exchanges, the final list has the same elements as the original list. Notice that an element may be exchanged multiple times before reaching its final place. Also, in case of pivot duplicates in the input array, they can be spread across the right subarray, in any order. This doesn't represent a partitioning failure, as further sorting will reposition and finally "glue" them together.

This form of the partition algorithm is not the original form; multiple variations can be found in various textbooks, such as versions not having the storeIndex. However, this form is probably the easiest to understand.

Each recursive call to the combined *quicksort* function reduces the size of the array being sorted by at least one element, since in each invocation the element at *pivotNewIndex* is placed in its final position. Therefore, this algorithm is guaranteed to terminate after at most *n* recursive calls. However, since *partition* reorders elements within a partition, this version of quicksort is not a stable sort.

**Implementation issues[[edit](http://en.wikipedia.org/w/index.php?title=Quicksort&action=edit&section=3" \o "Edit section: Implementation issues)]**

**Choice of pivot[[edit](http://en.wikipedia.org/w/index.php?title=Quicksort&action=edit&section=4" \o "Edit section: Choice of pivot)]**

In 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](http://en.wikipedia.org/wiki/Median) of the first, middle and last element of the partition for the pivot (as recommended by [Sedgewick](http://en.wikipedia.org/wiki/Robert_Sedgewick_(computer_scientist))).[[6]](http://en.wikipedia.org/wiki/Quicksort#cite_note-sedgewickBook-6) 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.[[*citation needed*](http://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

Selecting a pivot element is also complicated by the existence of [integer overflow](http://en.wikipedia.org/wiki/Integer_overflow). If the boundary indices of the subarray being sorted are sufficiently large, the naïve expression for the middle index, *(left + right)/2*, will cause overflow and provide an invalid pivot index. This can be overcome by using, for example, *left + (right-left)/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[[edit](http://en.wikipedia.org/w/index.php?title=Quicksort&action=edit&section=5" \o "Edit section: Repeated elements)]**

With a partitioning algorithm such as the one 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 quicksort equivalent of the [Dutch national flag problem](http://en.wikipedia.org/wiki/Dutch_national_flag_problem),[[4]](http://en.wikipedia.org/wiki/Quicksort" \l "cite_note-engineering-4) 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](http://en.wikipedia.org/wiki/Qsort) of [Version 7 Unix](http://en.wikipedia.org/wiki/Version_7_Unix).[[4]](http://en.wikipedia.org/wiki/Quicksort#cite_note-engineering-4)) 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

**function** quicksort(A, lo, hi)

if lo < hi

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* ≪ *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[[edit](http://en.wikipedia.org/w/index.php?title=Quicksort&action=edit&section=6" \o "Edit section: Optimizations)]**

Two other important optimizations, also suggested by Sedgewick and widely used in practice are:[[7]](http://en.wikipedia.org/wiki/Quicksort" \l "cite_note-glibc_qsort-7)[[8]](http://en.wikipedia.org/wiki/Quicksort#cite_note-8)

* To make sure at most O(log N) space is used, [recurse](http://en.wiktionary.org/wiki/recurse) first into the smaller side of the partition, then use a [tail call](http://en.wikipedia.org/wiki/Tail_call) to recurse into the other.
* Use [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort), which has a smaller constant factor and is thus faster on small arrays, for invocations on small arrays (i.e. where the length is less than a threshold *k* determined experimentally). This can be implemented by simply stopping the recursion when less than *k* elements are left, leaving the entire array *k*-sorted: each element will be at most *k* positions away from its final position. Then, a single [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) pass finishes the sort in O(*k*×*n*) time.[[9]](http://en.wikipedia.org/wiki/Quicksort#cite_note-9) A separate insertion sort of each small segment as they are identified adds the overhead of starting and stopping many small sorts, but avoids wasting effort comparing keys across the many segment boundaries, which keys will be in order due to the workings of the quicksort process.

**Parallelization[[edit](http://en.wikipedia.org/w/index.php?title=Quicksort&action=edit&section=7" \o "Edit section: Parallelization)]**

Like [merge sort](http://en.wikipedia.org/wiki/Merge_sort), quicksort can also be [parallelized](http://en.wikipedia.org/wiki/Parallel_algorithm) due to its divide-and-conquer nature. Individual in-place partition operations are difficult to parallelize, but once divided, different sections of the list can be sorted in parallel. The following is a straightforward approach: If we have pprocessors, we can divide a list of nelements into psublists in *O*(*n*) average time, then sort each of these in \textstyle O\left(\frac{n}{p} \log\frac{n}{p}\right)average time. Ignoring the *O*(*n*) preprocessing and merge times, this is [linear speedup](http://en.wikipedia.org/wiki/Linear_speedup). If the split is blind, ignoring the values, the merge naïvely costs *O*(*n*). If the split partitions based on a succession of pivots, it is tricky to parallelize and naïvely costs *O*(*n*). Given *O*(log *n*) or more processors, only *O*(*n*) time is required overall, whereas an approach with [linear speedup](http://en.wikipedia.org/wiki/Linear_speedup) would achieve *O*(log *n*) time for overall.

One advantage of this simple parallel quicksort over other parallel sort algorithms is that no synchronization is required, but the disadvantage is that sorting is still *O*(*n*) and only a sublinear speedup of *O*(log *n*) is achieved. A new thread is started as soon as a sublist is available for it to work on and it does not communicate with other threads. When all threads complete, the sort is done.

Other more sophisticated parallel sorting algorithms can achieve even better time bounds.[[10]](http://en.wikipedia.org/wiki/Quicksort#cite_note-10) For example, in 1991 David Powers described a parallelized quicksort (and a related [radix sort](http://en.wikipedia.org/wiki/Radix_sort)) that can operate in *O*(log *n*) time on a CRCW [PRAM](http://en.wikipedia.org/wiki/Parallel_Random_Access_Machine) with *n* processors by performing partitioning implicitly.[[11]](http://en.wikipedia.org/wiki/Quicksort#cite_note-11)