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[1] and published in 1961,[2] 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.[3]

Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. Efficient implementations of Quicksort are not a stable sort, meaning that the relative order of equal sort items is not preserved. Quicksort can operate in-place on an array, requiring small additional amounts of memory to perform the sorting. It is very similar to selection sort, except that it does not always choose worst-case partition.

Mathematical analysis of quicksort shows that, on average, the algorithm takes O(n log n) comparisons to sort n items. In the worst case, it makes O(n^2) comparisons, though this behavior is rare.
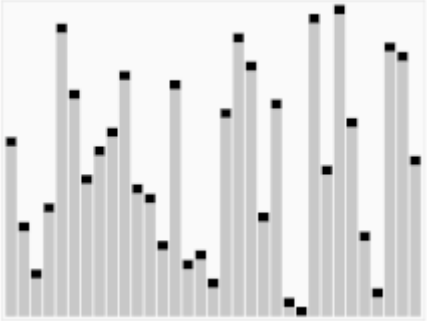
Quicksort	
Animated visualization of the quicksort algorithm. The horizontal lines are pivot values.	
Class	Sorting algorithm
Worst-case performance	O(n^2)
Best-case performance	O(n log n) (simple partition) or O(n) (three-way partition and equal keys)
Average performance	O(n log n)
Worst-case space complexity	O(n) auxiliary (naive) O(log n) auxiliary (Sedgewick 1978)

Contents

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 on a project on machine translation for the National Physical Laboratory. As a part of the translation process, he needed to sort the words in Russian sentences prior to looking them up in a Russian-English dictionary that 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 could not 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 did not. His boss ultimately accepted that he had lost the bet. Later, Hoare learned about ALGOL and its ability to do recursion that enabled him to publish the code in *Communications of the Association for Computing Machinery*, the premier computer science journal of the time.[2][5]

Quicksort gained widespread adoption, appearing, for example, in Unix as the default library sort subroutine. Hence, it lent its name to the C standard library subroutine `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 nine elements is divided into groups of three and then the median of the three medians from three 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 three times more swaps on average and degrades to $O(n^2)$ runtime when all elements are equal.[9]

In 2009, Vladimir 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[12] after extensive empirical performance tests.[13]

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 are in order by definition, so they 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*[14] and Cormen *et al.* in their book *Introduction to Algorithms*.[15] 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 at `lo` through `i-1` (inclusive) are less than the pivot, and the elements at `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.[16] This scheme degrades to $O(n^2)$ when the array is already in order.[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 at `lo` through `hi` (inclusive) of an array A can be expressed as:[15]

```
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
```

Sorting the entire array is accomplished by `quicksort(A, 0, length(A) - 1)`.

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 or equal to the pivot, one lesser or equal, that are in the wrong order relative to each other. The inverted elements are then swapped.[17] When the indices meet, the algorithm stops and returns the final index. 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's partitioning also would cause Quicksort to degrade to $O(n^2)$ for already sorted input, if the pivot was chosen as the first or the last element. With the middle element as the pivot, however, sorted data results with (almost) no swaps in equally sized partitions leading to best case behavior of Quicksort, i.e. $O(n \log(n))$. Like others, Hoare's partitioning doesn't produce a stable sort. 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+1..hi)` as opposed to `(lo..p-1)` and `(p+1..hi)` as in Lomuto's scheme. However, the partitioning algorithm guarantees `lo ≤ p < hi` which implies both resulting partitions are non-empty, hence there's no risk of infinite recursion. In pseudocode,[15]

```
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 + (hi - lo) / 2]
    i := lo - 1
    j := hi + 1
    loop forever
        do
            i := i + 1
        while A[i] < pivot
        do
            j := j - 1
        while A[j] > pivot
        if i >= j then
            return j
        swap A[i] with A[j]
```
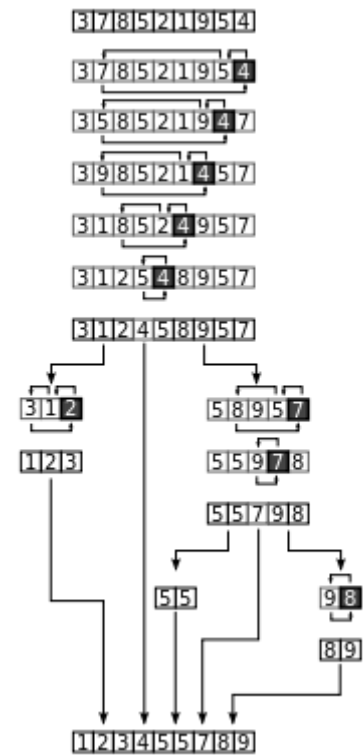

Full example of quicksort on a random set of numbers. The shaded element is the pivot. It is always chosen as the last element of the partition. However, always choosing the last element in the partition as the pivot in this way results in poor performance ($O(n^2)$) on *already sorted* arrays, or arrays 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 that choose the pivot as the middle element run much more quickly than the algorithm described in this diagram on large sets of numbers.

The entire array is sorted by `quicksort(A, 0, length(A) - 1)`.

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).[18] 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.

Median-of-three code snippet for Lomuto partition:

```
mid := (lo + hi) / 2
if A[mid] < A[lo]
    swap A[lo] with A[mid]
if A[hi] < A[lo]
    swap A[lo] with A[hi]
if A[mid] < A[hi]
    swap A[mid] with A[hi]
pivot := A[hi]
```

It puts a median into `A[hi]` first, then that new value of `A[hi]` is used for a pivot, as in a basic algorithm presented above.

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 (Mo3), defined as[6]

$$\text{ninther}(a) = \text{median}(\text{Mo3}(\text{first } \tfrac{1}{3} \text{ of } a), \text{Mo3}(\text{middle } \tfrac{1}{3} \text{ of } a), \text{Mo3}(\text{final } \tfrac{1}{3} \text{ of } a))$$

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 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). In the case where all elements are equal, Hoare partition scheme needlessly swaps elements, but the partitioning itself is best case, as noted in the Hoare partition section above.

To solve the Lomuto partition scheme 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 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 - 1)
        quicksort(A, right + 1, hi)
```

The `partition` algorithm returns indices to the first ('leftmost') and to the last ('rightmost') item of the middle partition. Every item of the partition is equal to `p` and is therefore sorted. Consequently, the items of the partition need not be included in the recursive calls to `quicksort`.

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 only two recursive calls on empty subarrays and thus finish in linear time (assuming the `partition` subroutine takes no longer than linear time).

Optimizations

Two other important optimizations, also suggested by Sedgewick and widely used in practice, are:[19][20]

- To make sure at most $O(\log n)$ space is used, recur first into the smaller side of the partition, then use a tail call to recur into the other, or update the parameters to no longer include the now sorted smaller side, and iterate to sort the larger side.
- When the number of elements is below some threshold (perhaps ten elements), switch to a non-recursive sorting algorithm such as insertion sort that performs fewer swaps, comparisons or other operations on such small arrays. The ideal 'threshold' will vary based on the details of the specific implementation.
- An older variant of the previous optimization: when the number of elements is less than the threshold k, simply stop; then after the whole array has been processed, perform insertion sort on it. Stopping the recursion early leaves the array k-sorted, meaning that each element is at most k positions away from its final sorted position. In this case, insertion sort takes $O(kn)$ time to finish the sort, which is linear if k is a constant.[21][14]:117 Compared to the "many small sorts" optimization, this version may execute fewer instructions, but it makes suboptimal use of the cache memories in modern computers.[22]

Parallelization

Quicksort's divide-and-conquer formulation makes it amenable to parallelization using task parallelism. The partitioning step is accomplished through the use of a parallel prefix sum algorithm to compute an index for each array element in its section of the partitioned array.[23][24] Given an array of size n, the partitioning step performs $O(n)$ work in $O(\log n)$ time and requires $O(n)$ additional scratch space. After the array has been partitioned, the two partitions can be sorted recursively in parallel. Assuming an ideal choice of pivots, parallel quicksort sorts an array of size n in $O(n \log n)$ work in $O(\log^2 n)$ time using $O(n)$ additional space.

Quicksort has some disadvantages when compared to alternative sorting algorithms, like merge sort, which complicate its efficient parallelization. The depth of quicksort's divide-and-conquer tree directly impacts the algorithm's scalability, and this depth is highly dependent on the algorithm's choice of pivot. Additionally, it is difficult to parallelize the partitioning step efficiently in-place. The use of scratch space simplifies the partitioning step, but increases the algorithm's memory footprint and constant overheads.

Other more sophisticated parallel sorting algorithms can achieve even better time bounds.[25] For example, in 1991 David Powers described a parallelized quicksort (and a related radix sort) that can operate in $O(\log n)$ time on a CRCW (concurrent read and concurrent write) PRAM (parallel random-access machine) with n processors by performing partitioning implicitly.[26]

Formal analysis

Worst-case analysis

The most unbalanced partition occurs when one of the sublists returned by the partitioning routine is of size $n - 1$.[27] 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 $\sum_{i=0}^{n}(n - i) = O(n^2)$, so in that case Quicksort takes $O(n^2)$ time.

Best-case analysis

In the most balanced case, each time we perform a partition we divide the list into two nearly equal pieces. This means each recursive call processes a list of half the size. Consequently, we can make only $\log_2 n$ nested calls before we reach a list of size 1. This means that the depth of the call tree is $\log_2 n$. But no two calls at the same level of the call tree process the same part of the original list; thus, each level of calls needs only $O(n)$ time all together (each call has some constant overhead, but since there are only $O(n)$ calls at each level, this is subsumed in the $O(n)$ factor). The result is that the algorithm uses only $O(n \log n)$ time.

Average-case analysis

To sort an array of n distinct elements, quicksort takes $O(n \log n)$ time in expectation, averaged over all $n!$ permutations of n elements with equal probability. We list here three common proofs to this claim providing different insights into quicksort's workings.

Using percentiles

If each pivot has rank somewhere in the middle 50 percent, that is, between the 25th percentile and the 75th percentile, then it splits the elements with at least 25% and at most 75% on each side. If we could consistently choose such pivots, we would only have to split the list at most $\log_{4/3} n$ times before reaching lists of size 1, yielding an $O(n \log n)$ algorithm.

When the input is a random permutation, the pivot has a random rank, and so it is not guaranteed to be in the middle 50 percent. However, when we start from a random permutation, in each recursive call the pivot has a random rank in its list, and so it is in the middle 50 percent about half the time. That is good enough. Imagine that you flip a coin: heads means that the rank of the pivot is in the middle 50 percent, tail means that it isn't. Imagine that you are flipping a coin over and over until you get k heads. Although this could take a long time, on average only $2k$ flips are required, and the chance that you won't get k heads after $100k$ flips is highly improbable (this can be made rigorous using Chernoff bounds). By the same argument, Quicksort's recursion will terminate on average at a call depth of only $2 \log_{4/3} n$. But if its average call depth is $O(\log n)$, and each level of the call tree processes at most n elements, the total amount of work done on average is the product, $O(n \log n)$. The algorithm does not have to verify that the pivot is in the middle half—if we hit it any constant fraction of the times, that is enough for the desired complexity.

Using recurrences

An alternative approach is to set up a recurrence relation for the $T(n)$ factor, the time needed to sort a list of size n. In the most unbalanced case, a single quicksort call involves $O(n)$ work plus two recursive calls on lists of size 0 and $n-1$, so the recurrence relation is

$$T(n) = O(n) + T(0) + T(n-1) = O(n) + T(n-1).$$

This is the same relation as for insertion sort and selection sort, and it solves to worst case $T(n) = O(n^2)$.

In the most balanced case, a single quicksort call involves $O(n)$ work plus two recursive calls on lists of size $n/2$, so the recurrence relation is

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

The master theorem for divide-and-conquer recurrences tells us that $T(n) = O(n \log n)$.

The outline of a formal proof of the $O(n \log n)$ expected time complexity follows. Assume that there are no duplicates as duplicates could be handled with linear time pre- and post-processing, or considered cases easier than the analyzed. When the input is a random permutation, the rank of the pivot is uniform random from 0 to $n - 1$. Then the resulting parts of the partition have sizes i and $n - i - 1$, and i is uniform random from 0 to $n - 1$. So, averaging over all possible splits and noting that the number of comparisons for the partition is $n - 1$, the average number of comparisons over all permutations of the input sequence can be estimated accurately by solving the recurrence relation:

$$C(n) = n - 1 + \frac{1}{n} \sum_{i=0}^{n-1} (C(i) + C(n-i-1)) = n - 1 + \frac{2}{n} \sum_{i=0}^{n-1} C(i)$$

$$nC(n) = n(n-1) + 2 \sum_{i=0}^{n-1} C(i)$$

$$nC(n) - (n-1)C(n-1) = n(n-1) - (n-1)(n-2) + 2C(n-1)$$

$$nC(n) = (n+1)C(n-1) + 2n - 2$$

$$\frac{C(n)}{n+1} = \frac{C(n-1)}{n} + \frac{2}{n+1} - \frac{2}{n(n+1)} \leq \frac{C(n-1)}{n} + \frac{2}{n+1}$$

$$= \frac{C(n-2)}{n-1} + \frac{2}{n} - \frac{2}{(n-1)n} + \frac{2}{n+1} \leq \frac{C(n-2)}{n-1} + \frac{2}{n} + \frac{2}{n+1}$$

$$\vdots$$

$$= \frac{C(1)}{2} + \sum_{i=2}^{n} \frac{2}{i+1} \leq 2 \sum_{i=1}^{n-1} \frac{1}{i} \approx 2 \int_{1}^{n} \frac{1}{x} \mathrm{d}x = 2 \ln n$$

Solving the recurrence gives $C(n) = 2n \ln n \approx 1.39 n \log_2 n$.

This means that, on average, quicksort performs only about 39% worse than in its best case. In this sense, it is closer to the best case than the worst case. A comparison sort cannot use less than $\log_2(n!)$ comparisons on average to sort n items (as explained in the article Comparison sort) and in case of large n, Stirling's approximation yields $\log_2(n!) \approx n(\log_2 n - \log_2 e)$, so quicksort is not much worse than an ideal comparison sort. This fast average runtime is another reason for quicksort's practical dominance over other sorting algorithms.

Using a binary search tree

To each execution of quicksort corresponds the following binary search tree (BST): the initial pivot is the root node; the pivot of the left half is the root of the left subtree, the pivot of the right half is the root of the right subtree, and so on. The number of comparisons of the execution of quicksort equals the number of comparisons during the construction of the BST by a sequence of insertions. So, the average number of comparisons for randomized quicksort equals the average cost of constructing a BST when the values inserted $(x_1, x_2, \ldots, x_n)$ form a random permutation.

Consider a BST created by insertion of a sequence $(x_1, x_2, \ldots, x_n)$ of values forming a random permutation. Let C denote the cost of creation of the BST. We have $C = \sum_i \sum_{j<i} c_{i,j}$, where $c_{i,j}$ is an binary random variable expressing whether during the insertion of x_i there was a comparison to x_j.

By linearity of expectation, the expected value $\mathbf{E}[C]$ of C is $\mathbf{E}[C] = \sum_i \sum_{j<i} \Pr(c_{i,j})$.

Fix i and $j<i$. The values $x_1, x_2, \ldots, x_j$, once sorted, define $j+1$ intervals. The core structural observation is that x_i is compared to x_j in the algorithm if and only if x_i falls inside one of the two intervals adjacent to x_j.

Observe that since $(x_1, x_2, \ldots, x_n)$ is a random permutation, $(x_1, x_2, \ldots, x_j, x_i)$ is also a random permutation, so the probability that x_i is adjacent to x_j is exactly $\dfrac{2}{j+1}$.

We end with a short calculation:

$$\mathrm{E}[C] = \sum_i \sum_{j<i} \frac{2}{j+1} = O\left(\sum_i \log i\right) = O(n \log n).$$

Space complexity

The space used by quicksort depends on the version used.

The in-place version of quicksort has a space complexity of $O(\log n)$, even in the worst case, when it is carefully implemented using the following strategies:

- in-place partitioning is used. This unstable partition requires $O(1)$ space.
- After partitioning, the partition with the fewest elements is (recursively) sorted first, requiring at most $O(\log n)$ space. Then the other partition is sorted using tail recursion or iteration, which doesn't add to the call stack. This idea, as discussed above, was described by R. Sedgewick, and keeps the stack depth bounded by $O(\log n)$.[18][21]

Quicksort with in-place and unstable partitioning uses only constant additional space before making any recursive call. Quicksort must store a constant amount of information for each nested recursive call. Since the best case makes at most $O(\log n)$ nested recursive calls, it uses $O(\log n)$ space. However, without Sedgewick's trick to limit the recursive calls, in the worst case quicksort could make $O(n)$ nested recursive calls and need $O(n)$ auxiliary space.

From a bit complexity viewpoint, variables such as *lo* and *hi* do not use constant space; it takes $O(\log n)$ bits to index into a list of n items. Because there are such variables in every stack frame, quicksort using Sedgewick's trick requires $O((\log n)^2)$ bits of space. This space requirement isn't too terrible, though, since if the list contained distinct elements, it would need at least $O(n \log n)$ bits of space.

Another, less common, not-in-place, version of quicksort uses $O(n)$ space for working storage and can implement a stable sort. The working storage allows the input array to be easily partitioned in a stable manner and then copied back to the input array for successive recursive calls. Sedgewick's optimization is still appropriate.

Relation to other algorithms

Quicksort is a space-optimized version of the binary tree sort. Instead of inserting items sequentially into an explicit tree, quicksort organizes them concurrently into a tree that is implied by the recursive calls. The algorithms make exactly the same comparisons, but in a different order. An often desirable property of a sorting algorithm is stability – that is the order of elements that compare equal is not changed, allowing controlling order of multikey tables (e.g. directory or folder listings) in a natural way. This property is hard to maintain for in situ (or in place) quicksort (that uses only constant additional space for pointers and buffers, and $O(\log n)$ additional space for the management of explicit or implicit

recursion). For variant quicksorts involving extra memory due to representations using pointers (e.g. lists or trees) or files (effectively lists), it is trivial to maintain stability. The more complex, or disk-bound, data structures tend to increase time cost, in general making increasing use of virtual memory or disk.

The most direct competitor of quicksort is heapsort. Heapsort's running time is $O(n \log n)$, but heapsort's average running time is usually considered slower than in-place quicksort. This result is debatable; some publications indicate the opposite.[28][29] Introsort is a variant of quicksort that switches to heapsort when a bad case is detected to avoid quicksort's worst-case running time.

Quicksort also competes with merge sort, another $O(n \log n)$ sorting algorithm. Mergesort is a stable sort, unlike standard in-place quicksort and heapsort, and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media such as disk storage or network-attached storage. Although quicksort can be implemented as a stable sort using linked lists, it will often suffer from poor pivot choices without random access. The main disadvantage of mergesort is that, when operating on arrays, efficient implementations require $O(n)$ auxiliary space, whereas the variant of quicksort with in-place partitioning and tail recursion uses only $O(\log n)$ space. (When operating on linked lists, mergesort only requires a small, constant amount of auxiliary storage.)

Bucket sort with two buckets is very similar to quicksort; the pivot in this case is effectively the value in the middle of the value range, which does well on average for uniformly distributed inputs.

Selection-based pivoting

A selection algorithm chooses the kth smallest of a list of numbers; this is an easier problem in general than sorting. One simple but effective selection algorithm works nearly in the same manner as quicksort, and is accordingly known as quickselect. The difference is that instead of making recursive calls on both sublists, it only makes a single tail-recursive call on the sublist that contains the desired element. This change lowers the average complexity to linear or $O(n)$ time, which is optimal for selection, but the sorting algorithm is still $O(n^2)$.

A variant of quickselect, the median of medians algorithm, chooses pivots more carefully, ensuring that the pivots are near the middle of the data (between the 30th and 70th percentiles), and thus has guaranteed linear time – $O(n)$. This same pivot strategy can be used to construct a variant of quicksort (median of medians quicksort) with $O(n \log n)$ time. However, the overhead of choosing the pivot is significant, so this is generally not used in practice.

More abstractly, given an $O(n)$ selection algorithm, one can use it to find the ideal pivot (the median) at every step of quicksort and thus produce a sorting algorithm with $O(n \log n)$ running time. Practical implementations this variant are considerably slower on average, but they are of theoretical interest because they show an optimal selection algorithm can yield an optimal sorting algorithm.

Variants

Multi-pivot quicksort

Instead of partitioning into two subarrays using a single pivot, multi-pivot quicksort (also multiquicksort[22]) partitions its input into some s number of subarrays using $s - 1$ pivots. While the dual-pivot case ($s = 3$) was considered by Sedgewick and others already in the mid-1970s, the resulting

algorithms were not faster in practice than the "classical" quicksort.[30] A 1999 assessment of a multiquicksort with a variable number of pivots, tuned to make efficient use of processor caches, found it to increase the instruction count by some 20%, but simulation results suggested that it would be more efficient on very large inputs.[22] A version of dual-pivot quicksort developed by Yaroslavskiy in 2009[10] turned out to be fast enough to warrant implementation in Java 7, as the standard algorithm to sort arrays of primitives (sorting arrays of objects is done using Timsort).[31] The performance benefit of this algorithm was subsequently found to be mostly related to cache performance, and experimental results indicate that the three-pivot variant may perform even better on modern machines.[32]

External quicksort

For magnetic tape files is the same as regular quicksort except the pivot is replaced by a buffer. First, the $M/2$ first and last elements are read into the buffer and sorted, then the next element from the beginning or end is read to balance writing. If the next element is less than the least of the buffer, write it to available space at the beginning. If greater than the greatest, write it to the end. Otherwise write the greatest or least of the buffer, and put the next element in the buffer. Keep the maximum lower and minimum upper keys written to avoid resorting middle elements that are in order. When done, write the buffer. Recursively sort the smaller partition, and loop to sort the remaining partition. This is a kind of three-way quicksort in which the middle partition (buffer) represents a sorted subarray of elements that are approximately equal to the pivot.

Three-way radix quicksort

This algorithm is a combination of radix sort and quicksort. Pick an element from the array (the pivot) and consider the first character (key) of the string (multikey). Partition the remaining elements into three sets: those whose corresponding character is less than, equal to, and greater than the pivot's character. Recursively sort the "less than" and "greater than" partitions on the same character. Recursively sort the "equal to" partition by the next character (key). Given we sort using bytes or words of length W bits, the best case is $O(KN)$ and the worst case $O(2^K N)$ or at least $O(N^2)$ as for standard quicksort, given for unique keys $N < 2^K$, and K is a hidden constant in all standard comparison sort algorithms including quicksort. This is a kind of three-way quicksort in which the middle partition represents a (trivially) sorted subarray of elements that are *exactly* equal to the pivot.

Quick radix sort

Also developed by Powers as an $o(K)$ parallel PRAM algorithm. This is again a combination of radix sort and quicksort but the quicksort left/right partition decision is made on successive bits of the key, and is thus $O(KN)$ for N K-bit keys. All comparison sort algorithms impliclty assume the transdichotomous model with K in $\Theta(\log N)$, as if K is smaller we can sort in $O(N)$ time using a hash table or integer sorting. If $K \gg \log N$ but elements are unique within $O(\log N)$ bits, the remaining bits will not be looked at by either quicksort or quick radix sort. Failing that, all comparison sorting algorithms will also have the same overhead of looking through $O(K)$ relatively useless bits but quick radix sort will avoid the worst case $O(N^2)$ behaviours of standard quicksort and radix quicksort, and will be faster even in the best case of those comparison algorithms under these conditions of $\text{uniqueprefix}(K) \gg \log N$. See Powers[33] for further discussion of the hidden overheads in comparison, radix and parallel sorting.

BlockQuicksort

In any comparison-based sorting algorithm, minimizing the number of comparisons requires maximizing the amount of information gained from each comparison, meaning that the comparison results are unpredictable. This causes frequent [branch mispredictions](), limiting performance.[34] BlockQuicksort[35] rearranges the computations of quicksort to convert unpredictable branches to [data dependencies](). When partitioning, the input is divided into moderate-sized [blocks]() (which fit easily into the [data cache]()), and two arrays are filled with the positions of elements to swap. (To avoid conditional branches, the position is unconditionally stored at the end of the array, and the index of the end is incremented if a swap is needed.) A second pass exchanges the elements at the positions indicated in the arrays. Both loops have only one conditional branch, a test for termination, which is usually taken.

Partial and incremental quicksort

Several variants of quicksort exist that separate the k smallest or largest elements from the rest of the input.

Generalization

[Richard Cole]() and David C. Kandathil, in 2004, discovered a one-parameter family of sorting algorithms, called partition sorts, which on average (with all input orderings equally likely) perform at most $n \log n + O(n)$ comparisons (close to the information theoretic lower bound) and $\Theta(n \log n)$ operations; at worst they perform $\Theta(n \log^2 n)$ comparisons (and also operations); these are in-place, requiring only additional $O(\log n)$ space. Practical efficiency and smaller variance in performance were demonstrated against optimised quicksorts (of [Sedgewick]() and [Bentley-McIlroy]()).[36]

See also

- [Introsort]() – Hybrid sorting algorithm

Notes

1. "Sir Antony Hoare" (https://web.archive.org/web/20150403184558/http://www.computerhistory.org/fellowawards/hall/bios/Antony%2CHoare/). Computer History Museum. Archived from the original (http://www.computerhistory.org/fellowawards/hall/bios/Antony,Hoare/) on 3 April 2015. Retrieved 22 April 2015.
2. Hoare, C. A. R. (1961). "Algorithm 64: Quicksort". *Comm. ACM*. **4** (7): 321. doi:10.1145/366622.366644 (https://doi.org/10.1145%2F366622.366644).
3. Skiena, Steven S. (2008). *The Algorithm Design Manual* (https://books.google.com/books?id=7XUSn0IKQEgC). Springer. p. 129. ISBN 978-1-84800-069-8.
4. Shustek, L. (2009). "Interview: An interview with C.A.R. Hoare". *Comm. ACM*. **52** (3): 38–41. doi:10.1145/1467247.1467261 (https://doi.org/10.1145%2F1467247.1467261).
5. "My Quickshort interview with Sir Tony Hoare, the inventor of Quicksort" (http://anothercasualcoder.blogspot.com/2015/03/my-quickshort-interview-with-sir-tony.html). Marcelo M De Barros. 15 March 2015.
6. Bentley, Jon L.; McIlroy, M. Douglas (1993). "Engineering a sort function" (http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.8162). *Software—Practice and Experience*. **23** (11): 1249–1265. CiteSeerX 10.1.1.14.8162 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.8162). doi:10.1002/spe.4380231105 (https://doi.org/10.1002%2Fspe.4380231105).

7. Van Emden, M. H. (1 November 1970). "Algorithms 402: Increasing the Efficiency of Quicksort". *Commun. ACM*. **13** (11): 693–694. doi:10.1145/362790.362803 (https://doi.org/10.1145%2F362790.362803). ISSN 0001-0782 (https://www.worldcat.org/issn/0001-0782).

8. Bentley, Jon (2007). "The most beautiful code I never wrote". In Oram, Andy; Wilson, Greg (eds.). *Beautiful Code: Leading Programmers Explain How They Think*. O'Reilly Media. p. 30. ISBN 978-0-596-51004-6.

9. "Quicksort Partitioning: Hoare vs. Lomuto" (https://cs.stackexchange.com/q/11550). *cs.stackexchange.com*. Retrieved 3 August 2015.

10. Yaroslavskiy, Vladimir (2009). "Dual-Pivot Quicksort" (https://web.archive.org/web/20151002230717/http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf) (PDF). Archived from the original (http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf) (PDF) on 2 October 2015.

11. "Replacement of Quicksort in java.util.Arrays with new Dual-Pivot Quick" (http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628). *permalink.gmane.org*. Retrieved 3 August 2015.

12. "Java 7 Arrays API documentation" (https://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#sort(int%5b%5d)). Oracle. Retrieved 23 July 2018.

13. Wild, S.; Nebel, M.; Reitzig, R.; Laube, U. (7 January 2013). *Engineering Java 7's Dual Pivot Quicksort Using MaLiJAn*. Proceedings. Society for Industrial and Applied Mathematics. pp. 55–69. doi:10.1137/1.9781611972931.5 (https://doi.org/10.1137%2F1.9781611972931.5). ISBN 978-1-61197-253-5.

14. Jon Bentley (1999). *Programming Pearls*. Addison-Wesley Professional.

15. Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford (2009) [1990]. "Quicksort". *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. pp. 170–190. ISBN 0-262-03384-4.

16. Wild, Sebastian (2012). "Java 7's Dual Pivot Quicksort" (https://kluedo.ub.uni-kl.de/frontdoor/index/index/docId/3463). Technische Universität Kaiserslautern.

17. Hoare, C. A. R. (1 January 1962). "Quicksort" (http://comjnl.oxfordjournals.org/content/5/1/10). *The Computer Journal*. **5** (1): 10–16. doi:10.1093/comjnl/5.1.10 (https://doi.org/10.1093%2Fcomjnl%2F5.1.10). ISSN 0010-4620 (https://www.worldcat.org/issn/0010-4620).

18. Sedgewick, Robert (1 September 1998). *Algorithms in C: Fundamentals, Data Structures, Sorting, Searching, Parts 1–4* (https://books.google.com/books?id=ylAETlep0CwC) (3 ed.). Pearson Education. ISBN 978-81-317-1291-7. Retrieved 27 November 2012.

19. qsort.c in GNU libc: [1] (https://www.cs.columbia.edu/~hgs/teaching/isp/hw/qsort.c), [2] (http://repo.or.cz/w/glibc.git/blob/HEAD:/stdlib/qsort.c)

20. http://www.ugrad.cs.ubc.ca/~cs260/chnotes/ch6/Ch6CovCompiled.html

21. Sedgewick, R. (1978). "Implementing Quicksort programs". *Comm. ACM*. **21** (10): 847–857. doi:10.1145/359619.359631 (https://doi.org/10.1145%2F359619.359631).

22. LaMarca, Anthony; Ladner, Richard E. (1999). "The Influence of Caches on the Performance of Sorting". *Journal of Algorithms*. **31** (1): 66–104. CiteSeerX 10.1.1.27.1788 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.27.1788). doi:10.1006/jagm.1998.0985 (https://doi.org/10.1006%2Fjagm.1998.0985). "Although saving small subarrays until the end makes sense from an instruction count perspective, it is exactly the wrong thing to do from a cache performance perspective."

23. Umut A. Acar, Guy E Blelloch, Margaret Reid-Miller, and Kanat Tangwongsan, Quicksort and Sorting Lower Bounds (https://www.cs.cmu.edu/afs/cs/academic/class/15210-s13/www/lectures/lecture19.pdf), *Parallel and Sequential Data Structures and Algorithms*. 2013.

24. Breshears, Clay (2012). "Quicksort Partition via Prefix Scan" (http://www.drdobbs.com/parallel/quicksort-partition-via-prefix-scan/240003109). *Dr. Dobb's*.

25. Miller, Russ; Boxer, Laurence (2000). *Algorithms sequential & parallel: a unified approach* (https://books.google.com/books?id=dZoZAQAAIAAJ). Prentice Hall. ISBN 978-0-13-086373-7. Retrieved 27 November 2012.

26. Powers, David M. W. (1991). *Parallelized Quicksort and Radixsort with Optimal Speedup*. Proc. Int'l Conf. on Parallel Computing Technologies. CiteSeerX 10.1.1.57.9071 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.9071).

27. The other one may either have 1 element or be empty (have 0 elements), depending on whether the pivot is included in one of subpartitions, as in the Hoare's partitioning routine, or is excluded from both of them, like in the Lomuto's routine.

28. Hsieh, Paul (2004). "Sorting revisited" (http://www.azillionmonkeys.com/qed/sort.html). www.azillionmonkeys.com. Retrieved 26 April 2010.

29. MacKay, David (1 December 2005). "Heapsort, Quicksort, and Entropy" (https://web.archive.org/web/20090401163041/http://users.aims.ac.za/~mackay/sorting/sorting.html). users.aims.ac.za/~mackay. Archived from the original (http://users.aims.ac.za/~mackay/sorting/sorting.html) on 1 April 2009. Retrieved 26 April 2010.

30. Wild, Sebastian; Nebel, Markus E. (2012). *Average case analysis of Java 7's dual pivot quicksort*. European Symposium on Algorithms. arXiv:1310.7409 (https://arxiv.org/abs/1310.7409). Bibcode:2013arXiv1310.7409W (https://ui.adsabs.harvard.edu/abs/2013arXiv1310.7409W).

31. "Arrays" (http://docs.oracle.com/javase/7/docs/api/java/util/Arrays.html#sort%28byte%5B%5D%29). *Java Platform SE 7*. Oracle. Retrieved 4 September 2014.

32. Kushagra, Shrinu; López-Ortiz, Alejandro; Qiao, Aurick; Munro, J. Ian (2014). *Multi-Pivot Quicksort: Theory and Experiments*. Proc. Workshop on Algorithm Engineering and Experiments (ALENEX). doi:10.1137/1.9781611973198.6 (https://doi.org/10.1137%2F1.9781611973198.6).

33. David M. W. Powers, Parallel Unification: Practical Complexity (http://david.wardpowers.info/Research/AI/papers/199501-ACAW-PUPC.pdf), Australasian Computer Architecture Workshop, Flinders University, January 1995

34. Kaligosi, Kanela; Sanders, Peter (11–13 September 2006). *How Branch Mispredictions Affect Quicksort* (https://www.cs.auckland.ac.nz/~mcw/Teaching/refs/sorting/quicksort-branch-prediction.pdf) (PDF). ESA 2006: 14th Annual European Symposium on Algorithms. Zurich. doi:10.1007/11841036_69 (https://doi.org/10.1007%2F11841036_69).

35. Edelkamp, Stefan; Weiß, Armin (22 April 2016). "BlockQuicksort: How Branch Mispredictions don't affect Quicksort". arXiv:1604.06697 (https://arxiv.org/abs/1604.06697) [cs.DS (https://arxiv.org/archive/cs.DS)].

36. Richard Cole, David C. Kandathil: "The average case analysis of Partition sorts" (http://www.cs.nyu.edu/cole/papers/part-sort.pdf), European Symposium on Algorithms, 14–17 September 2004, Bergen, Norway. Published: *Lecture Notes in Computer Science* 3221, Springer Verlag, pp. 240–251.

References

- Sedgewick, R. (1978). "Implementing Quicksort programs". *Comm. ACM*. **21** (10): 847–857. doi:10.1145/359619.359631 (https://doi.org/10.1145%2F359619.359631).

- Dean, B. C. (2006). "A simple expected running time analysis for randomized 'divide and conquer' algorithms". *Discrete Applied Mathematics*. **154**: 1–5. doi:10.1016/j.dam.2005.07.005 (https://doi.org/10.1016%2Fj.dam.2005.07.005).

- Hoare, C. A. R. (1961). "Algorithm 63: Partition". *Comm. ACM*. **4** (7): 321. doi:10.1145/366622.366642 (https://doi.org/10.1145%2F366622.366642).

- Hoare, C. A. R. (1961). "Algorithm 65: Find". *Comm. ACM*. **4** (7): 321–322. doi:10.1145/366622.366647 (https://doi.org/10.1145%2F366622.366647).

- Hoare, C. A. R. (1962). "Quicksort". *Comput. J.* **5** (1): 10–16. doi:10.1093/comjnl/5.1.10 (https://doi.org/10.1093%2Fcomjnl%2F5.1.10). (Reprinted in Hoare and Jones: *Essays in computing science* (http://portal.acm.org/citation.cfm?id=SERIES11430.63445), 1989.)
- Musser, David R. (1997). "Introspective Sorting and Selection Algorithms" (http://www.cs.rpi.edu/~musser/gp/introsort.ps). *Software: Practice and Experience*. **27** (8): 983–993. doi:10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-# (https://doi.org/10.1002%2F%28SICI%291097-024X%28199708%2927%3A8%3C983%3A%3AAID-SPE117%3E3.0.CO%3B2-%23).
- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Pages 113–122 of section 5.2.2: Sorting by Exchanging.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Chapter 7: Quicksort, pp. 145–164.
- Faron Moller. Analysis of Quicksort (http://www.cs.swan.ac.uk/~csfm/Courses/CS_332/quicksort.pdf). CS 332: Designing Algorithms. Department of Computer Science, Swansea University.
- Martínez, C.; Roura, S. (2001). "Optimal Sampling Strategies in Quicksort and Quickselect". *SIAM J. Comput.* **31** (3): 683–705. CiteSeerX 10.1.1.17.4954 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.17.4954). doi:10.1137/S0097539700382108 (https://doi.org/10.1137%2FS0097539700382108).
- Bentley, J. L.; McIlroy, M. D. (1993). "Engineering a sort function". *Software: Practice and Experience*. **23** (11): 1249–1265. CiteSeerX 10.1.1.14.8162 (https://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.14.8162). doi:10.1002/spe.4380231105 (https://doi.org/10.1002%2Fspe.4380231105).

External links

- "Animated Sorting Algorithms: Quick Sort" (https://web.archive.org/web/20150302145415/http://www.sorting-algorithms.com/quick-sort). Archived from the original on 2 March 2015. Retrieved 25 November 2008. – graphical demonstration
- "Animated Sorting Algorithms: Quick Sort (3-way partition)" (https://web.archive.org/web/20150306071949/http://www.sorting-algorithms.com/quick-sort-3-way). Archived from the original on 6 March 2015. Retrieved 25 November 2008.
- Open Data Structures – Section 11.1.2 – Quicksort (http://opendatastructures.org/versions/edition-0.1e/ods-java/11_1_Comparison_Based_Sorti.html#SECTION001412000000000000000), Pat Morin
- Interactive illustration of Quicksort (https://web.archive.org/web/20180629183103/http://www.tomgsmith.com/quicksort/content/illustration/), with code walkthrough