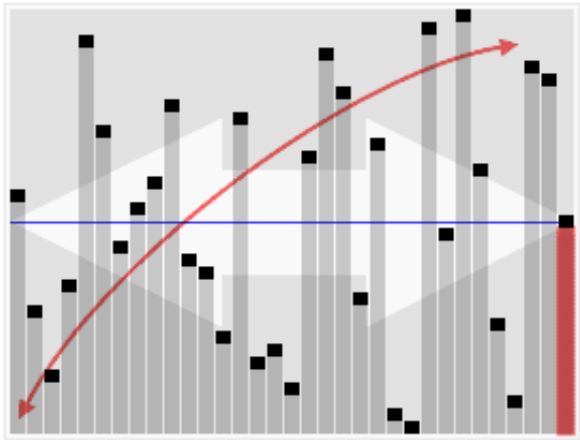# Quicksort

From Wikipedia, the free encyclopedia

Quicksort, or **partition-exchange sort**, is a sorting algorithm developed by Tony Hoare that, on average, makes 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 is often faster in practice than other O(n log n) algorithms.[1] Additionally, quicksort's sequential and localized memory references work well with a cache. Quicksort is a comparison sort and, in efficient implementations, is not a stable sort. Quicksort can be implemented with an in-place partitioning algorithm, so the entire sort can be done with only O(log n) additional space used by the stack during the recursion.[2]

Contents

- 1 History
- 2 Algorithm
    - 2.1 Implementation issues
        - 2.1.1 Choice of pivot
        - 2.1.2 Repeated elements
        - 2.1.3 Optimizations
        - 2.1.4 Parallelization
- 3 Formal analysis
    - 3.1 Average-case analysis using discrete probability
    - 3.2 Average-case analysis using recurrences
        - 3.2.1 Solving the recurrence using mathematical induction
    - 3.3 Analysis of randomized quicksort
    - 3.4 Space complexity
- 4 Relation to other algorithms
    - 4.1 Selection-based pivoting
    - 4.2 Variants
    - 4.3 Generalization

Quicksort

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 case performance	O(n log n)
Worst case space complexity	O(n) auxiliary (naive) O(log n) auxiliary (Sedgewick 1978)

History

The quicksort algorithm was developed in 1960 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. He developed the algorithm in order to sort the words to be translated, to make them more easily matched to an already-sorted Russian-to-English dictionary that was stored on magnetic tape.[3]

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`[4] and in the reference implementation of Java. It was analyzed extensively by Robert Sedgewick, who wrote his Ph.D. thesis about the algorithm and suggested several improvements.[4]

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. 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. In pseudocode, a quicksort that sorts elements i through k (inclusive) of an array A can be expressed compactly as[5]:171

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

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:

```
// lo is the index of the leftmost element of the subarray
// hi is the index of the rightmost element of the subarray (inclusi
partition(A, lo, hi)
    pivotIndex := choosePivot(A, lo, hi)
    pivotValue := A[pivotIndex]
    // put the chosen pivot at A[hi]
    swap A[pivotIndex] and A[hi]
    storeIndex := lo
    // Compare remaining array elements against pivotValue = A[hi]
    for i from lo to hi-1, inclusive
        if A[i] < pivotValue
            swap A[i] and A[storeIndex]
            storeIndex := storeIndex + 1
    swap A[storeIndex] and A[hi]  // Move pivot to its final place
    return storeIndex
```

This is the in-place partition algorithm. It partitions the portion of the array between indexes *lo* and *hi*, inclusively, by moving all elements less than or equal `A[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 does not 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.
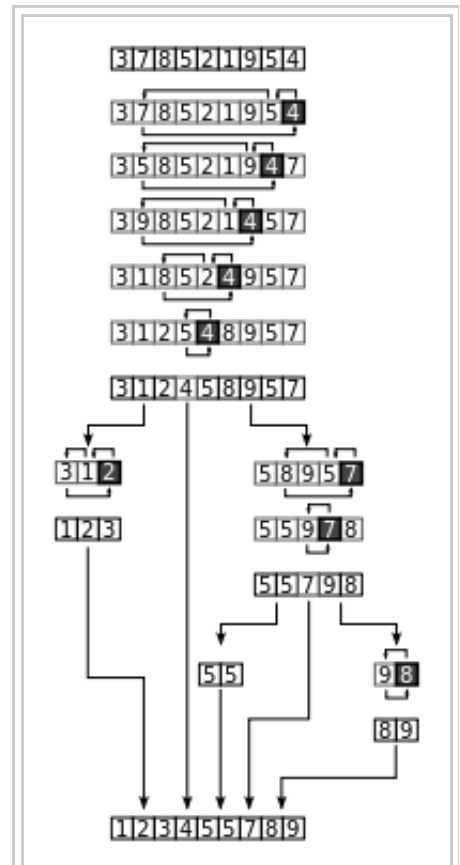
This form of the partition algorithm is not the original form; multiple variations can be found in various textbooks, such as versions not having the storeIndex. However, this form is probably the easiest to understand.

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

Choice of pivot

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.

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).[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.

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 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.

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.

To solve this problem (sometimes called the Dutch national flag problem[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` of Version 7 Unix.[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
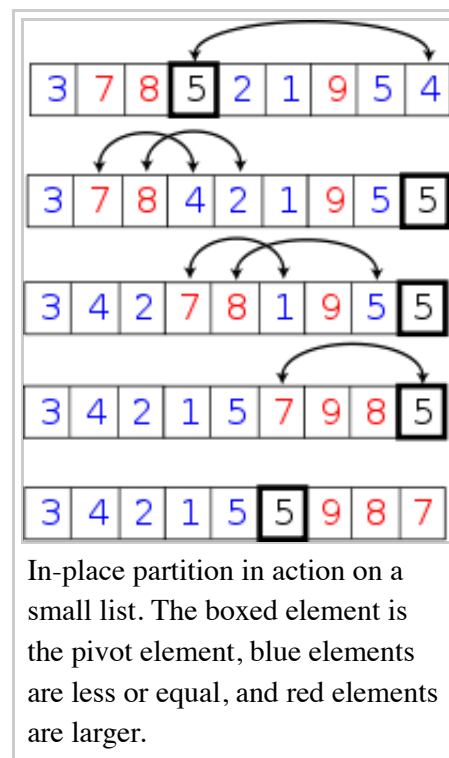
```
quicksort(A, lo, hi)
    if lo < hi
        p := pivot(A, lo, hi)
        left, right := partition(A, p, lo, hi)  // note: multiple return values
        quicksort(A, lo, left)
        quicksort(A, right, hi)
```

The best case for the algorithm now occurs when all elements are equal (or are chosen from a small set of $k \ll n$ 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:[7][8]

  - 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 other.

- Use 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 pass[9]:117 finishes the sort in $O(kn)$ time. 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, where keys will be in order due to the workings of the quicksort process.

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.[10] 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.[11] 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 PRAM with n processors by performing partitioning implicitly.[12]

Formal analysis

Average-case analysis using discrete probability

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. Why? For a start, it is not hard to see that the partition operation takes $O(n)$ time.

In the most unbalanced case, each time we perform a partition we divide the list into two sublists of size 0 and $n - 1$ (for example, if all elements of the array are equal). This means 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. That is

the worst case: given knowledge of which comparisons are performed by the sort, there are adaptive algorithms that are effective at generating worst-case input for quicksort on-the-fly, regardless of the pivot selection strategy.[13]

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.

In fact, it's not necessary to be perfectly balanced; even if each pivot splits the elements with 75% on one side and 25% on the other side (or any other fixed fraction), the call depth is still limited to $\log_{4/3} n$, so the total running time is still $O(n \log n)$.

So what happens on average? If the 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 a pivot from the two middle 50 percent, 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, $O(n \log n)$. Note that 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.

Average-case analysis 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 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:

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

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

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. Also note that 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.

Solving the recurrence using mathematical induction

$$T(1) = c$$

$$T(n) = \frac{1}{n} \sum_{i=1}^{n} (T(i - 1) + T(n - i)) + dn$$

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

Assertion A(m) : $T(m) \le am \log m + b$ for all $m \ge 1$
Base case A(0) : Holds for $b \ge c$
Induction step : Assuming $A(m)$ holds for all $m < n$, we have to prove $A(n)$.

$$T(n) \leq \frac{2}{n} \sum_{i=1}^{n-1} (ai \log i + b) + dn$$

$$\leq \frac{2}{n} \left(\sum_{i=1}^{n-1} ai \log i \right) + 2b + dn$$

$$= \frac{2}{n} \left(\sum_{i=1}^{n/2} ai \log i + \sum_{i=n/2+1}^{n-1} (ai \log i) \right) + 2b + dn$$

$$\leq \frac{2}{n} \left(\sum_{i=1}^{n/2} ai \log(n/2) + \sum_{i=n/2+1}^{n-1} ai \log n \right) + 2b + dn$$

$$= \frac{2}{n} \left(\sum_{i=1}^{n-1} ai \log n - \sum_{i=1}^{n/2} ai \right) + 2b + dn$$

$$= \frac{2}{n} \left(\frac{n(n-1)}{2} (a \log n) - \frac{n/2(n/2+1)}{2} (a) \right) + 2b + dn$$

$$\leq a(n-1) \log n - \frac{n}{4}(a) + 2b + dn$$

$$= an \log n + b - \frac{n}{4}(a) + b + dn$$

$$\leq an \log n + b \text{ for } a > 4(b+d)$$

Analysis of randomized quicksort

Using the same analysis, one can show that randomized quicksort has the desirable property that, for any input, it requires only $O(n \log n)$ expected time (averaged over all choices of pivots). However, there also exists a combinatorial proof.

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 $(x_1, x_2, ..., x_n)$ form a random permutation.

Consider a BST created by insertion of a sequence $(x_1, x_2, ..., x_n)$ 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 $\mathbb{E}[C]$ of C is $\mathbb{E}[C] = \sum_i \sum_{j<i} \Pr(c_{i,j})$.

Fix i and $j<i$. The values $x_1, x_2, ..., x_j$, 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 $(x_1, x_2, ..., x_n)$ is a random permutation, $(x_1, x_2, ..., x_j, x_i)$ 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: $\mathbb{E}[C] = \sum_i \sum_{j<i} \dfrac{2}{j+1} = O\left(\sum_i \log i\right) = O(n \log n)$.

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 $O(\log n)$.[6][14]

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 logN 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 worst-case running time is always $O(n \log n)$. But, heapsort is assumed to be on average somewhat slower than standard in-place quicksort. This is still debated and in research, with some publications indicating the opposite.[15][16] 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 mergesort, another recursive sort algorithm but with the benefit of worst-case $O(n \log n)$ running time. 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 easily 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. (Note that 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 which contains the desired element. This change lowers the average complexity to linear or $O(n)$ time, which is optimal for selection, but worst-case time 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 – worst-case $O(n)$. This same pivot strategy can be used to construct a variant of quickselect (median of medians quicksort) with worst-case $O(n)$ time. However, the overhead of choosing the pivot is significant, so this is generally not used in practice.

More abstractly, given a worst-case $O(n)$ selection algorithm, one can use it to find the ideal pivot (the median) at every step of quicksort, producing a variant with worst-case $O(n \log n)$ running time. In practical implementations this variant is considerably slower on average, but it is of theoretical interest, showing how an optimal selection algorithm can yield an optimal sorting algorithm.

Variants

Multi-pivot quicksort

Instead of partitioning into two subarrays using a single pivot, partition 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.[17] However, a version of dual-pivot quicksort developed by Yaroslavskiy in 2009[18] 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).[19]

Balanced quicksort

Choose a pivot likely to represent the middle of the values to be sorted, and then follow the regular quicksort algorithm.

External quicksort

The same as regular quicksort except the pivot is replaced by a buffer. First, read the M/2 first and last elements into the buffer and sort them. Read the next element from the beginning or end 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

Developed by Sedgewick and also known as **multikey quicksort**, it 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^KN) 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. Note that all comparison sort algorithms effectively assume an ideal K of O(logN) as if k is smaller we can sort in O(N) using a hash table or integer sorting, and if K >> logN but elements are unique within O(logN) bits, the remaining bits will not be looked at by either quicksort or quick radix sort, and otherwise 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(N²) behaviours of standard quicksort and quick radix sort, and will be faster even in the best case of those comparison algorithms under these conditions of uniqueprefix(K) >> logN. See Powers [20] for further discussion of the hidden overheads in comparison, radix and parallel sorting.

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).[21]

See also

- Introsort
- Flashsort

Notes

1. ^ Steven S. Skiena (27 April 2011). *The Algorithm Design Manual* (http://books.google.com/books?id=7XUSn0IKQEgC). Springer. p. 129. ISBN 978-1-84800-069-8. Retrieved 27 November 2012.
2. ^ "Data structures and algorithm: Quicksort" (http://www.cs.auckland.ac.nz/~jmor159/PLDS210/qsort1a.html). Auckland University.
3. ^ Shustek, L. (2009). "Interview: An interview with C.A.R. Hoare". *Comm. ACM* **52** (3): 38–41. doi:10.1145/1467247.1467261 (http://dx.doi.org/10.1145%2F1467247.1467261).
4. ^ *a b c d* 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. doi:10.1002/spe.4380231105 (http://dx.doi.org/10.1002%2Fspe.4380231105).
5. ^ Cormen, Thomas H.; Leiserson, Charles E., Rivest, Ronald L., Stein, Clifford (2009) [1990]. *Introduction to Algorithms* (3rd ed.). MIT Press and McGraw-Hill. ISBN 0-262-03384-4.
6. ^ *a b* Sedgewick, Robert (1 September 1998). *Algorithms In C: Fundamentals, Data Structures, Sorting, Searching, Parts 1-4* (http://books.google.com/books?id=ylAETlep0CwC) (3 ed.). Pearson Education. ISBN 978-81-317-1291-7. Retrieved 27 November 2012.

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

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

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

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

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

12. ^ David M. W. Powers, Parallelized Quicksort and Radixsort with Optimal Speedup (http://citeseer.ist.psu.edu/327487.html), *Proceedings of International Conference on Parallel Computing Technologies*. Novosibirsk. 1991.

13. ^ McIlroy, M. D. (1999). "A killer adversary for quicksort". *Software: Practice and Experience* **29** (4): 341–237. doi:10.1002/(SICI)1097-024X(19990410)29:4<341::AID-SPE237>3.3.CO;2-I (http://dx.doi.org/10.1002%2F%28SICI%291097-024X%2819990410%2929%3A4%3C341%3A%3AAID-SPE237%3E3.3.CO%3B2-I).

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

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

16. ^ MacKay, David (1 December 2005). "Heapsort, Quicksort, and Entropy" (http://users.aims.ac.za/~mackay/sorting/sorting.html). users.aims.ac.za/~mackay. Retrieved 26 April 2010.

17. ^ 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).

18. ^ http://permalink.gmane.org/gmane.comp.java.openjdk.core-libs.devel/2628

19. ^ "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.

20. ^ 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

21. ^ 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 (http://dx.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
(http://dx.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 (http://dx.doi.org/10.1145%2F366622.366642).
- Hoare, C. A. R. (1961). "Algorithm 64: Quicksort". *Comm. ACM* **4** (7): 321.
  doi:10.1145/366622.366644 (http://dx.doi.org/10.1145%2F366622.366644).
- Hoare, C. A. R. (1961). "Algorithm 65: Find". *Comm. ACM* **4** (7): 321–322.
  doi:10.1145/366622.366647 (http://dx.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
  (http://dx.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* (Wiley) **27** (8):
  983–993. doi:10.1002/(SICI)1097-024X(199708)27:8<983::AID-SPE117>3.0.CO;2-#
  (http://dx.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.
- A. LaMarca and R. E. Ladner. "The Influence of Caches on the Performance of Sorting." Proceedings
  of the Eighth Annual ACM-SIAM Symposium on Discrete Algorithms, 1997. pp. 370–379.
- 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. doi:10.1137/S0097539700382108
  (http://dx.doi.org/10.1137%2FS0097539700382108).
- Bentley, J. L.; McIlroy, M. D. (1993). "Engineering a sort function". *Software: Practice and
  Experience* **23** (11): 1249–1265. doi:10.1002/spe.4380231105
  (http://dx.doi.org/10.1002%2Fspe.4380231105).

External links

- Animated Sorting Algorithms: Quick Sort (http://www.sorting-
  algorithms.com/quick-sort) – graphical demonstration and

The Wikibook *Algorithm*

discussion of quick sort

- Animated Sorting Algorithms: 3-Way Partition Quick Sort
  (http://www.sorting-algorithms.com/quick-sort-3-way) –
  graphical demonstration and discussion of 3-way partition quick sort
- Interactive Tutorial for Quicksort (http://pages.stern.nyu.edu/~panos/java/Quicksort/index.html)
- Quicksort applet (http://www.yorku.ca/sychen/research/sorting/index.html) with "level-order"
  recursive calls to help improve algorithm analysis
- Open Data Structures - Section 11.1.2 - Quicksort (http://opendatastructures.org/versions/edition-
  0.1e/ods-java/11_1_Comparison_Based_Sorti.html#SECTION001412000000000000000)
- Literate implementations of Quicksort in various languages
  (http://en.literateprograms.org/Category:Quicksort) on LiteratePrograms
- A colored graphical Java applet (http://coderaptors.com/?QuickSort) which allows experimentation
  with initial state and shows statistics
- An in-place, stable Quicksort
  (http://h2database.googlecode.com/svn/trunk/h2/src/tools/org/h2/dev/sort/InPlaceStableQuicksort.java
) (Java) which runs in $O(n \log n \log n)$ time.

Retrieved from "http://en.wikipedia.org/w/index.php?title=Quicksort&oldid=638238833"

Categories: Sorting algorithms │ Comparison sorts │ 1961 in science

---