Sorting algorithm

In computer science and mathematics , a **sorting algorithm** is an algorithm that puts elements of a list in a certain order . The most used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms (such as search and merge algorithms) that require sorted lists to work correctly; it is also often useful for canonicalizing data and for producing human-readable output. More formally, the output must satisfy two conditions:

1. The output is in nondecreasing order (each element is no smaller than the previous element according to the desired total order);
2. The output is a permutation, or reordering, of the input.

Since the dawn of computing, the sorting problem has attracted a great deal of research, perhaps due to the complexity of solving it efficiently despite its simple, familiar statement. For example, bubble sort was analyzed as early as 1956. Although many consider it a solved problem, useful new sorting algorithms are still being invented to this day (for example, library sort was first published in 2004). Sorting algorithms are prevalent in introductory computer science classes, where the abundance of algorithms for the problem provides a gentle introduction to a variety of core algorithm concepts, such as big O notation, divide-and-conquer algorithms, data structures, randomized algorithms, worst-case, average-case, and best-case analysis, time-space tradeoffs, and lower bounds.

Classification

Sorting algorithms used in computer science are often classified by:

- Computational complexity (worst, average and best behaviour) in terms of the size of the list (n). For typical sorting algorithms good behaviour is O(n log n) and bad behaviour is O(n^2). Ideal behaviour for a sort is O(n). Sort algorithms which only use an abstract key comparison operation always need at least O(n log n) comparisons on average;
- Memory usage (and use of other computer resources). In particular, some sorting algorithms are "in place", such that little memory is needed beyond the items being sorted, while others need to create auxiliary locations for data to be temporarily stored.
- Stability: **stable sorting algorithms** maintain the relative order of records with equal keys (*i.e.* values). That is, a sorting algorithm is *stable* if whenever there are two records R and S with the same key and with R appearing before S in the original list, R will appear before S in the sorted list.

- Whether or not they are a comparison sort. A comparison sort examines the data only by comparing two elements with a comparison operator.
- General method: insertion, exchange, selection, merging, *etc*. Exchange sorts include bubble sort and quicksort. Selection sorts include shaker sort and heapsort.

When equal elements are indistinguishable, such as with integers, stability is not an issue. However, assume that the following pairs of numbers are to be sorted by their first coordinate:

```
(4, 1)  (3, 1)  (3, 7)  (5, 6)
```

In this case, two different results are possible, one which maintains the relative order of records with equal keys, and one which does not:

```
(3, 1)  (3, 7)  (4, 1)  (5, 6)   (order maintained)
(3, 7)  (3, 1)  (4, 1)  (5, 6)   (order changed)
```

Unstable sorting algorithms may change the relative order of records with equal keys, but stable sorting algorithms never do so. Unstable sorting algorithms can be specially implemented to be stable. One way of doing this is to artificially extend the key comparison, so that comparisons between two objects with otherwise equal keys are decided using the order of the entries in the original data order as a tie-breaker. Remembering this order, however, often involves an additional space cost.

In this table, n is the number of records to be sorted and k is the number of distinct keys. The columns "Best", "Average", and "Worst" give the time complexity in each case; estimates that do not use k assume k to be constant. "Memory" denotes the amount of auxiliary storage needed beyond that used by the list itself. "Cmp" indicates whether the sort is a comparison sort.

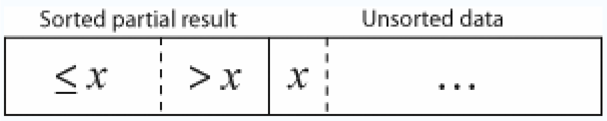
Name	Best	Average	Worst	Memory	Stable	Cmp	Method
Bubble sort	$O(n)$	—	$O(n^2)$	$O(1)$	Yes	Yes	Exchanging
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	No	Yes	Selection
Insertion sort	$O(n)$	—	$O(n^2)$	$O(1)$	Yes	Yes	Insertion
Binary tree sort	$O(n\log(n))$	—	$O(n\log(n))$	$O(1)$	Yes	Yes	Insertion
Merge sort	$O(n\log(n))$	—	$O(n\log(n))$	$O(n)$	Yes	Yes	Merging
In-place merge sort	$O(n\log(n))$	—	$O(n\log(n))$	$O(1)$	Yes	Yes	Merging
Heapsort	$O(n\log(n))$	—	$O(n\log(n))$	$O(1)$	No	Yes	Selection
Quicksort	$O(n\log(n))$	$O(n\log(n))$	$O(n^2)$	$O(\log n)$	No	Yes	Partitioning

Insertion sort is a simple sort algorithm, a comparison sort in which the sorted array (or list) is built one entry at a time. It is much less efficient on large lists than the more advanced algorithms such as quicksort, heapsort, or merge sort, but it has various advantages:

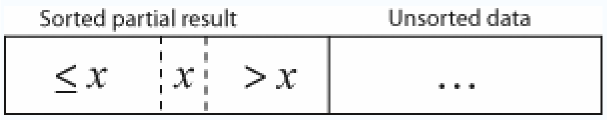
- Simple to implement
- Efficient on (quite) small data sets
- Efficient on data sets which are already substantially sorted
- More efficient in practice than most other simple O(n^2) algorithms such as selection sort or bubble sort: the average time is $n^2/4$ and it is linear in the best case
- Stable (does not change the relative order of elements with equal keys)
- In-place (only requires a constant amount O(1) of extra memory space)
- It is an online algorithm, in that it can sort a list as it receives it.

In abstract terms, each iteration of an insertion sort removes an element from the input data, inserting it at the correct position in the already sorted list, until no elements are left in the input. The choice of which element to remove from the input is arbitrary and can be made using almost any choice algorithm.

Sorting is typically done in-place. The result array after *k* iterations contains the first *k* entries of the input array and is sorted. In each step, the first remaining entry of the input is removed, inserted into the result at the right position, thus extending the result:



becomes:



with each element > x copied to the right as it is compared against x.

The most common variant, which operates on arrays, can be described as:

1. Suppose we have a method called *insert* designed to insert a value into a sorted sequence at the beginning of an array. It operates by starting at the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element. It has the side effect of overwriting the value stored immediately after the sorted sequence in the array.

2. To perform insertion sort, start at the left end of the array and invoke *insert* to insert each element encountered into its correct position. The ordered sequence into which we insert it is stored at the beginning of the array in the set of indexes already examined. Each insertion overwrites a single value, but this is okay because it's the value we're inserting.

A simple pseudocode version of the complete algorithm follows, where the arrays are zero-based:

```
insert(array a, int length, value) {
    int i := length - 1;
    while (i ≥ 0 and a[i] > value) {
        a[i + 1] := a[i];
        i := i - 1;
    }
    a[i + 1] := value;
}

insertionSort(array a, int length) {
    int i := 1;
    while (i < length) {
        insert(a, i, a[i]);
        i := i + 1;
    }
}
```

Java Implementation :

```
static void insertionSort (int[] A) {
       int j;
       for (int i = 1; i < A.length; i++) {
         int a = A[i];
         for (j = i - 1; j >=0 && A[j] > a; j--)
           A[j + 1] = A[j];
         A[j + 1] = a;
       }
   }
```

Good and bad input cases

In the best case of an already sorted array, this implementation of insertion sort takes O(n) time: in each iteration, the first remaining element of the input is only compared with the last element of the result. It takes O(n^2) time in the average and worst cases, which makes it impractical for sorting large numbers of elements. However, insertion sort's inner loop is very fast, which often makes it one of the fastest algorithms for sorting small numbers of elements, typically less than 10 or so.

Comparisons to other sorts

Insertion sort is very similar to bubble sort. In bubble sort, after *k* passes through the array, the *k* largest elements have bubbled to the top. (Or the *k* smallest elements have bubbled to the bottom, depending on which way you do it.) In insertion sort, after *k* passes through the array, you have a run of *k* sorted elements at the bottom of the array. Each pass inserts another element into the sorted run. So with bubble sort, each pass takes less time than the previous one, but with insertion sort, each pass may take more time than the previous one.

Some divide-and-conquer algorithms such as quicksort and mergesort sort by recursively dividing the list into smaller sublists which are then sorted. A useful optimization in practice for these algorithms is to switch to insertion sort for "small enough" sublists on which insertion sort outperforms the more complex algorithms. The size of list for which insertion sort has the advantage varies by environment and implementation, but is typically around 8 to 20 elements.

Merge sort

In computer science, **merge sort** or **mergesort** is a sorting algorithm for rearranging lists (or any other data structure that can only be accessed sequentially, e.g. file streams) into a specified order. It is a particularly good example of the divide and conquer algorithmic paradigm. It is a comparison sort.

Conceptually, merge sort works as follows:

1. Divide the unsorted list into two sublists of about half the size
2. Sort each of the two sublists
3. Merge the two sorted sublists back into one sorted list.

The algorithm was invented by John von Neumann in 1945.

In simple pseudocode, the algorithm could look something like this:

```
function mergesort(m)
    var list left, right
    if length(m) ≤ 1
        return m
    else
        middle = length(m) / 2
        for each x in m up to middle
            add x to left
        for each x in m after middle
            add x to right
        left = mergesort(left)
        right = mergesort(right)
        result = merge(left, right)
        return result
```

There are several variants for the merge() function, the simplest variant could look like this:

```
function merge(left,right)
    var list result
    while length(left) > 0 or length(right) > 0
        if length(left) > 0 and first(left) ≥ first(right)
            append first(left) to result
            left = rest(left)
        else
            append first(right) to result
            right = rest(right)
    return result
```

Analysis

In sorting n items, merge sort has an average and worst-case performance of O($n \log n$). If the running time of merge sort for a list of length n is $T(n)$, then the recurrence $T(n) = 2T(n/2) + n$ follows from the definition of the algorithm (apply the algorithm to two lists of half the size of the original list, and add the n steps taken to merge the resulting two lists). The closed form follows from the master theorem.

In the worst case, merge sort does exactly ($n \lceil \log n \rceil - 2^{\lceil \log n \rceil} + 1$) comparisons, which is between ($n \log n - n + 1$) and
($n \log n - 0.9139 \cdot n + 1$) [logs are base 2]. Note, the worst case number given here does not agree with that given in the "bible", Knuth's Art of Computer Programming, Vol 3; the discrepancy is due to Knuth analyzing a variant implementation of merge sort that is slightly sub-optimal.

For large n and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches $\alpha \cdot n$ fewer than the worst case, where $\alpha = -1 + \sum 1/(2^k + 1)$, $k = 0 \rightarrow \infty$, $\alpha \approx 0.2645$.

In the *worst* case, merge sort does about 39% fewer comparisons than quicksort does in the *average* case; merge sort always makes fewer comparisons than quicksort, except in extremely rare cases, when they tie, where merge sort's *worst* case is found simultaneously with quicksort's *best* case. In terms of moves, merge sort's worst case complexity is O($n \log n$)--the same complexity as quicksort's best case, and merge sort's best case takes about half as many iterations as the worst case.

Recursive implementations of merge sort make $2n - 1$ method calls in the worst case, compared to quicksort's n, thus has roughly twice as much recursive overhead as quicksort. However, iterative, non-recursive, implementations of merge sort, avoiding method call overhead, are not difficult to code. Merge sort's most common implementation does not sort in place, meaning memory the size of the input must be allocated for the sorted output to be stored in. Sorting in-place is possible but requires an extremely complicated implementation.

Merge sort is much more efficient than quicksort if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as Lisp, where sequentially accessed data structures are very common. Unlike some optimized implementations of quicksort, merge sort is a stable sort as long as the merge operation is implemented properly.

Merge sorting tape drives

Merge sort is so inherently sequential that it's practical to run it using slow tape drives as input and output devices. It requires very little memory, and the memory required does not change with the number of data elements. If you have four tape drives, it works as follows:

1. divide the data to be sorted in half and put half on each of two tapes
2. merge individual pairs of records from the two tapes; write two-record chunks alternately to each of the two output tapes
3. merge the two-record chunks from the two output tapes into four-record chunks; write these alternately to the original two input tapes
4. merge the four-record chunks into eight-record chunks; write these alternately to the original two output tapes
5. repeat until you have one chunk containing all the data, sorted --- that is, for log *n* passes, where *n* is the number of records.

On tape drives that can run both backwards and forwards, you can run merge passes in both directions, avoiding rewind time. For the same reason it is also very useful for sorting data on disk that is too large to fit entirely into primary memory.

online sorting

Mergesort's merge operation is useful in online sorting, where the list to be sorted is received a piece at a time, instead of all at the beginning (see online algorithm). In this application, we sort each new piece that is received using any sorting algorithm, and then merge it into our sorted list so far using the merge operation. However, this approach can be expensive in time and space if the received pieces are small compared to the sorted list — a better approach in this case is to store the list in a self-balancing binary search tree and add elements to it as they are received.

Comparison with other sort algorithms

Although heap sort has the same time bounds as merge sort, it requires only Ω(1) auxiliary space instead of merge sort's Ω(n), and is consequently often faster in practical implementations. Quicksort, however, is considered by many to be the fastest general-purpose sort algorithm in practice. Its average-case complexity is O(*n* log *n*), with a much smaller coefficient, in good implementations, than merge sort's, even though it is quadratic in the worst case. On the plus side, merge sort is a stable sort, parallelizes better, and is more efficient at handling slow-to-access sequential media. Merge sort is often the best choice for sorting a linked list: in this situation it is relatively easy to implement a merge sort in such a way that it does not require Ω(n) auxiliary space (instead only Ω(1)), and the slow random-access performance of a linked list makes some other algorithms (such as quick sort) perform poorly, and others (such as heapsort) completely impossible.

As of Perl 5.8, merge sort is its default sorting algorithm (it was quicksort in previous versions of Perl). In Java , the Arrays.sort() methods use mergesort and a tuned quicksort depending on the datatypes.

external sort

Definition: Any sort algorithm that uses external memory, such as tape or disk, during the sort. Since most common sort algorithms assume high-speed random access to all intermediate memory, they are unsuitable if the values to be sorted don't fit in main memory.

Java Implementation :

```
public int[] mergeSort(int array[])
// pre: array is full, all elements are valid integers (not null)
// post: array is sorted in ascending order (lowest to highest)
{
        // if the array has more than 1 element, we need to split it and merge the sorted halves
        if(array.length > 1)
        {
                // number of elements in sub-array 1
                // if odd, sub-array 1 has the smaller half of the elements
                // e.g. if 7 elements total, sub-array 1 will have 3, and sub-array 2 will have 4
                int elementsInA1 = array.length/2;
                // since we want an even split, we initialize the length of sub-array 2 to
                // equal the length of sub-array 1
                int elementsInA2 = elementsInA1;
                // if the array has an odd number of elements, let the second half take the extra one
                // see note (1)
                if((array.length % 2) == 1)
                        elementsInA2 += 1;
                // declare and initialize the two arrays once we've determined their sizes
                int arr1[] = new int[elementsInA1];
                int arr2[] = new int[elementsInA2];
                // copy the first part of 'array' into 'arr1', causing arr1 to become full
                for(int i = 0; i < elementsInA1; i++)
                        arr1[i] = array[i];
                // copy the remaining elements of 'array' into 'arr2', causing arr2 to become full
                for(int i = elementsInA1; i < elementsInA1 + elementsInA2; i++)
                        arr2[i - elementsInA1] = array[i];
                // recursively call mergeSort on each of the two sub-arrays that we've just created
                // note: when mergeSort returns, arr1 and arr2 will both be sorted!
                // it's not magic, the merging is done below, that's how mergesort works :)
                arr1 = mergeSort(arr1);
                arr2 = mergeSort(arr2);

                // the three variables below are indexes that we'll need for merging
                // [i] stores the index of the main array. it will be used to let us
                // know where to place the smallest element from the two sub-arrays.
                // [j] stores the index of which element from arr1 is currently being compared
                // [k] stores the index of which element from arr2 is currently being compared
                int i = 0, j = 0, k = 0;
                // the below loop will run until one of the sub-arrays becomes empty
                // in my implementation, it means until the index equals the length of the sub-array
                while(arr1.length != j && arr2.length != k)
                {
                        // if the current element of arr1 is less than current element of arr2
                        if(arr1[j] < arr2[k])
                        {
                            // copy the current element of arr1 into the final array
                            array[i] = arr1[j];
                            // increase the index of the final array to avoid replacing the element
                            // which we've just added
                            i++;
                            // increase the index of arr1 to avoid comparing the element
                            // which we've just added
                            j++;
                        }
                        // if the current element of arr2 is less than current element of arr1
                        else
                        {
                            // copy the current element of arr1 into the final array
                            array[i] = arr2[k];
                            // increase the index of the final array to avoid replacing the element
                            // which we've just added
                            i++;
                            // increase the index of arr2 to avoid comparing the element
                            // which we've just added
                            k++;
                        }
                }
                // at this point, one of the sub-arrays has been exhausted and there are no more
                // elements in it to compare. this means that all the elements in the remaining
                // array are the highest (and sorted), so it's safe to copy them all into the
                // final array.
                while(arr1.length != j)
                {
                        array[i] = arr1[j];
                        i++;
                        j++;
                }
                while(arr2.length != k)
                {
                        array[i] = arr2[k];
                        i++;
                        k++;
                }
        }
        // return the sorted array to the caller of the function
        return array;
}
```

Quicksort

Quicksort is a well-known sorting algorithm developed by C. A. R. Hoare that, on average, makes **Θ**(n log n) comparisons to sort n items. However, in the worst case, it makes **Θ**(n^2) comparisons. Typically, quicksort is significantly faster in practice than other **Θ**(n log n) algorithms, because its inner loop can be efficiently implemented on most architectures, and in most real-world data it is possible to make design choices which minimize the possibility of requiring quadratic time. Quicksort is a comparison sort and, in efficient implementations, is not a stable sort.

Quicksort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

The steps are:

1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements 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 sort the sub-list of lesser elements and the sub-list of greater elements.

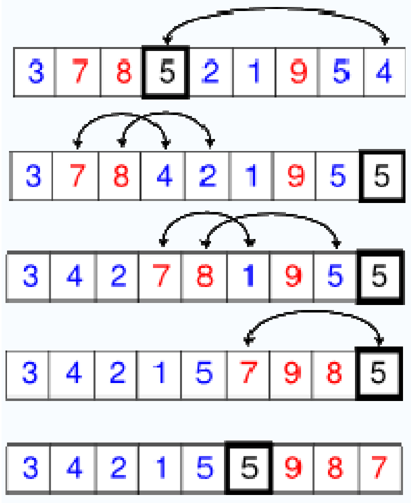
The base case of the recursion are lists of size zero or one, which are always sorted. The algorithm always terminates because it puts at least one element in its final place on each iteration.

In simple pseudocode, the algorithm might be expressed as:

```
function quicksort(q)
    var list less, pivotList, greater
    if length(q) ≤ 1
        return q
    select a pivot value pivot from q
    for each x in q except the pivot element
        if x < pivot then add x to less
        if x ≥ pivot then add x to greater
    add pivot to pivotList
    return concatenate(quicksort(less), pivotList, quicksort(greater))
```

Notice that we only examine elements by comparing them to other elements. This makes quicksort a comparison sort.

Version with in-place partition



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.

The disadvantage of the simple version above is that it requires Ω(n) extra storage space, which is as bad as mergesort (see big-O notation for the meaning of Ω). The additional memory allocations required can also drastically impact speed and cache performance in practical implementations. There is a more complicated version which uses an in-place partition algorithm and can achieve O(log *n*) space use on average for good pivot choices:

```
function partition(a, left, right, pivotIndex)
    pivotValue := a[pivotIndex]
    swap(a[pivotIndex], a[right]) // Move pivot to end
    storeIndex := left
    for i from left to right-1
        if a[i] <= pivotValue
            swap(a[storeIndex], a[i])
            storeIndex := storeIndex + 1
    swap(a[right], a[storeIndex]) // 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 to `a[pivotIndex]` to the beginning of the subarray, leaving all the greater elements following them. 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.

Once we have this, writing quicksort itself is easy:

```
function quicksort(a, left, right)
    if right > left
```

```
        select a pivot value a[pivotIndex]
        pivotNewIndex := partition(a, left, right, pivotIndex)
        quicksort(a, left, pivotNewIndex-1)
        quicksort(a, pivotNewIndex+1, right)
```

This version is the one more typically used in imperative languages such as C.

Performance details

Quicksort's inner loop, which performs the partition, is amenable to optimization on typical modern machines for two main reasons:

- All comparisons are being done with a single pivot value, which can be stored in a register.
- The list is being traversed sequentially, which produces very good locality of reference and cache behavior for arrays.

This close fit with modern architectures makes quicksort one of the fastest sorting algorithms on average. Because of this excellent average performance and simple implementation, quicksort has become one of the most popular sorting algorithms in practical use.

Although it is often believed to work in-place, quicksort uses $\mathbf{O}(\log n)$ additional stack space on average in recursive implementations and $\mathbf{O}(n)$ stack space in the worst case.

The most crucial concern of a quicksort implementation is the choosing of a good pivot element. A naïve pivot choice, such as always choosing the first element, will be terribly inefficient for certain inputs. For example, if the input is already sorted, a common practical case, always choosing the first element as the pivot causes quicksort to degenerate into a selection sort with $\mathbf{O}(n^2)$ running time. When using the simplest variant, the recursion depth also becomes linear, requiring $\mathbf{O}(n)$ extra stack space.

This worst-case performance is much worse than comparable sorting algorithms such as heapsort or merge sort. However, if pivots are chosen randomly, most poor choices of pivots are unlikely; the worst-case, for example, has only probability $1/n!$ of occurring. This variation, called *randomized quicksort*, can be shown to use $\mathbf{O}(n \log n)$ comparisons on any input with very high probability.

Note that the `partition` procedure only requires the ability to traverse the list sequentially; therefore, quicksort is not confined to operating on arrays (it can be used, for example, on linked lists). Choosing a good pivot, however, benefits from random access, as we will see.

Choosing a better pivot

The worst-case behavior of quicksort is not merely a theoretical problem. When quicksort is used in web services, for example, it is possible for an attacker to deliberately exploit the worst case performance and choose data which will cause a slow running time or maximize the chance of running out of stack space. See competitive analysis for more discussion of this issue.

Sorted or partially sorted data is quite common in practice and a naïve implementation which selects the first element as the pivot does poorly with such data. To avoid this problem the middle element may be chosen. This works well in practice, but attacks can still cause worst-case performance.

Another common choice is to randomly choose a pivot index, typically using a pseudorandom number generator. If the numbers are truly random, it can be proven that the resulting algorithm, called *randomized quicksort*, runs in an expected time of $\Theta(n \log n)$. Unfortunately, although simple pseudorandom number generators usually suffice for choosing good pivots, they are little defense against an attacker who can run the same generator to predict the values. One defense against this is to use highly unpredictable random seeds, such as numbers generated from hardware random number generators or entropy pools, and to avoid exposing partial information that might reveal information about the random seed or sequence.

Another choice is to select the median of the first, middle and last elements as the pivot. Adding two randomly selected elements resists chosen data attacks. The use of the fixed elements reduces the chance of bad luck causing a poor pivot selection for partially sorted data when not under attack. These steps increase overhead, so it may be worth skipping them once the partitions grow small and the penalty for poor pivot selection drops.

Interestingly, since finding the true median value to use as the pivot can be done in $\mathbf{O}(n)$ time (see selection algorithm), the worst-case time can be reduced to $\mathbf{O}(n \log n)$. Because the median algorithm is relatively slow in practice, however, this algorithm is rarely useful; if worst-case time is a concern, other sort algorithms are generally preferable.

The way that partitioning is done determines whether or not a quicksort implementation is a stable sort. Typically, in-place partitions are unstable, while not-in-place partitions are stable

Other optimizations

Using different algorithms for small lists

Another optimization is to switch to a different sorting algorithm once the list becomes small, perhaps ten or less elements. Insertion sort or selection sort might be inefficient for large data sets, but they are often faster than quicksort on small lists, requiring less setup time and less stack manipulation.

One widely used implementation of quicksort, found in the 1997 Microsoft C library, used a cutoff of 8 elements before switching to insertion sort, asserting that testing had shown that to be a good choice. It used the middle element for the partition value, asserting that testing had shown that the median of three algorithm did not, in general, increase performance.

Sedgewick (1978) suggested an enhancement to the use of simple sorts for small numbers of elements, which reduced the number of instructions required by postponing the simple sorts until the quicksort had finished, then running an insertion sort over the whole array. This is effective because insertion sort requires only O(*kn*) time to sort an array where every element is less than *k* places from its final position.

LaMarca and Ladner (1997) consider "The Influence of Caches on the Performance of Sorting", a significant issue in microprocessor systems with multi-level caches and high cache miss times. They conclude that while the Sedgewick optimization decreases the number of instructions, it also decreases locality of cache references and worsens performance compared to doing the simple sort when the need for it is first encountered. However, the effect was not dramatic and they suggested that it was starting to become more significant with more than 4 million 64 bit float elements. Greater improvement was shown for other sorting types.

Another variation on this theme which is becoming widely used is *introspective sort*, often called introsort. This starts with quicksort and switches to heapsort when the recursion depth exceeds a preset value. This overcomes the overhead of increasingly complex pivot selection techniques while ensuring O(*n* log *n*) worst-case performance. Musser reported that on a median-of-3 killer sequence of 100,000 elements running time was 1/200th that of median-of-3 quicksort. Musser also considered the effect of Sedgewick's delayed small sorting on caches, reporting that it could double the number of cache misses when used on arrays, but its performance with double-ended queues was significantly better.

Because recursion requires additional memory, quicksort has been implemented in a non-recursive, iterative form. This in-place variant has the advantage of predictable memory use regardless of input, and the disadvantage of considerably greater code complexity. A simpler in-place sort of competitive speed is heapsort.

A simple alternative for reducing quicksort's memory consumption is to use true recursion only on the smaller of the two sublists and tail recursion on the larger. This limits the additional storage of quicksort to Θ(log *n*); even for huge lists, this variant typically requires no more than roughly 100 words of memory. The procedure *quicksort* in the pseudocode with the in-place partition would be rewritten as

```
function quicksort(a, left, right)
    while right > left
```

```
        select a pivot value a[pivotIndex]
        pivotNewIndex := partition(a, left, right, pivotIndex)
        if (pivotNewIndex-1) - left < right - (pivotNewIndex+1)
            quicksort(a, left, pivotNewIndex-1)
            left  := pivotNewIndex+1
        else
            quicksort(a, pivotNewIndex+1, right)
            right := pivotNewIndex-1
```

Competitive sorting 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.

The most direct competitor of quicksort is heapsort. Heapsort is typically somewhat slower than quicksort, but the worst-case running time is always O(n log n). Quicksort is usually faster, though there remains the chance of worst case performance except in the introsort variant. If it's known in advance that heapsort is going to be necessary, using it directly will be faster than waiting for introsort to switch to it. Heapsort also has the important advantage of using only constant additional space (heapsort is in-place), whereas even the best variant of quicksort uses Θ(log n) space. However, heapsort requires efficient random access to be practical.

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 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 written to operate on linked lists, it will often suffer from poor pivot choices without random access. The main disadvantage of mergesort is that it requires Ω(n) auxiliary space in the best case, whereas the variant of quicksort with in-place partitioning and tail recursion uses only O(log n) space

Formal analysis

From the initial description it's not obvious that quicksort takes O(n log n) time on average. It's not hard to see that the partition operation, which simply loops over the elements of the array once, uses Θ(n) time. In versions that perform concatenation, this operation is also Θ(n).

In the best 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 n nested calls before we reach a list of size 1. This means that the depth of the call tree is O(log 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.

An alternate approach is to set up a recurrence relation for T(*n*), the time needed to sort a list of size *n*. Because a single quicksort call involves O(*n*) work plus two recursive calls on lists of size *n/2* in the best case, the relation would be:

$$T(n) = O(n) + 2T(n/2)$$

Standard mathematical induction techniques for solving this type of relation tell us that T(*n*) = Θ(*n* log *n*).

In fact, it's not necessary to divide the list this precisely; even if each pivot splits the elements with 99% on one side and 1% on the other, the call depth is still limited to 100log *n*, so the total running time is still O(*n* log *n*).

In the worst case, however, the two sublists have size 1 and *n*-1, and the call tree becomes a linear chain of *n* nested calls. The *i*th call does O(n-i) work, and $\sum_{i=0}^{n}(n-i) = O(n^2)$. The recurrence relation is:

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

This is the same relation as for insertion sort and selection sort, and it solves to T(*n*) = Θ(n^2).

Randomized quicksort expected complexity

Randomized quicksort has the desirable property that it requires only O(*n* log *n*) expected time, regardless of the input. But what makes random pivots a good choice?

Suppose we sort the list and then divide it into four parts. The two parts in the middle will contain the best pivots; each of them is larger than at least 25% of the elements and smaller than at least 25% of the elements. If we could consistently choose an element from these two middle parts, we would only have to split the list at most $2\log_2 n$ times before reaching lists of size 1, yielding an O(*n* log *n*) algorithm.

Unfortunately, a random choice will only choose from these middle parts half the time. The surprising fact is that this is good enough. 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 100*k* flips is astronomically small. By the same argument, quicksort's recursion will terminate on average at a call depth of only $2(2\log_2 n)$. 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*).

Average complexity

Even if we aren't able to choose pivots randomly, quicksort still requires only O(*n* log *n*) time over all possible permutations of its input. Because this average is simply the sum of the times over all permutations of the input divided by *n* factorial, it's equivalent to choosing a random permutation of the input. When we do this, the pivot choices are essentially random, leading to an algorithm with the same running time as randomized quicksort.

More precisely, the average number of comparisons over all permutations of the input sequence can be estimated accurately by solving the recurrence relation:

$$C(n) = n - 1 + \frac{1}{n}\sum_{i=0}^{n-1}(C(i) + C(n-i-1)) = 2n\ln n = 1.39n\log_2 n.$$

Here, *n - 1* is the number of comparisons the partition uses. Since the pivot is equally likely to fall anywhere in the sorted list order, the sum is averaging over all possible splits.

This means that, on average, quicksort performs only about 39% worse than the ideal number of comparisons, which is its best case. In this sense it is closer to the best case than the worst case. This fast average runtime is another reason for quicksort's practical dominance over other sorting algorithms.

Space complexity

The space used by quicksort depends on the version used. The version of quicksort with in-place partitioning uses only constant additional space before making any recursive call. However, if it has made O(log *n*) nested recursive calls, it needs to store a constant amount of information from each of them. Since the best case makes at most O(log *n*) nested recursive calls, it uses O(log *n*) space. The worst case makes O(*n*) nested recursive calls, and so needs O(*n*) space.

We are eliding a small detail here, however. If we consider sorting arbitrarily large lists, we have to keep in mind that our variables like *left* and *right* can no longer be considered to occupy constant space; it takes O(log *n*) bits to index into a list of *n* items. Because we have variables like this in every stack frame, in reality quicksort requires O($\log^2$ *n*) bits of space in the best and average case and O(*n* log *n*) space in the worst case. This isn't too terrible, though, since if the list contains mostly distinct elements, the list itself will also occupy O(*n* log *n*) bits of space.

The not-in-place version of quicksort uses O(n) space before it even makes any recursive calls. In the best case its space is still limited to O(n), because each level of the recursion uses half as much space as the last, and

$$\sum_{i=0}^{\infty} \frac{n}{2^i} = 2n.$$

Its worst case is dismal, requiring

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

space, far more than the list itself. If the list elements are not themselves constant size, the problem grows even larger; for example, if most of the list elements are distinct, each would require about O(log n) bits, leading to a best-case O(n log n) and worst-case O(n^2 log n) space requirement.

Relationship to selection

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, except 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 small change lowers the average complexity to linear or Θ(n) time, and makes it an in-place algorithm. A variation on this algorithm brings the worst-case time down to O(n) (see selection algorithm for more information).

Conversely, once we know a worst-case O(n) selection algorithm is available, we 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, however, this variant is considerably slower on average.

The QuickSort Algorithm

As one of the more advanced sorting algorithms, you might think that the Quicksort Algorithm is steeped in complicated theoretical background, but this is not so. Like Insertion Sort, this algorithm has a fairly simple concept at the core, but is made complicated by the constraints of the array structure.

The basic concept is to pick one of the elements in the array as a pivot value around which the other elements will be rearranged. Everything less than the pivot is moved left of the pivot (into the left partition). Similarly, everything greater than the pivot goes into the right partition. At this point each partition is recursively quicksorted.

The Quicksort algorithm is fastest when the median of the array is chosen as the pivot value. That is because the resulting partitions are of very similar size. Each partition splits itself in two and thus the base case is reached very quickly.

In practice, the Quicksort algorithm becomes very slow when the array passed to it is already close to being sorted. Because there is no efficient way for the computer to find the median element to use as the pivot, the first element of the array is used as the pivot. So when the array is almost sorted, Quicksort doesn't partition it equally. Instead, the partitions are lopsided like in Figure 2. This means that one of the recursion branches is much deeper than the other, and causes execution time to go up. Thus, it is said that the more random the arrangement of the array, the faster the Quicksort Algorithm finishes.

Figure 1: The ideal Quicksort on a random array.

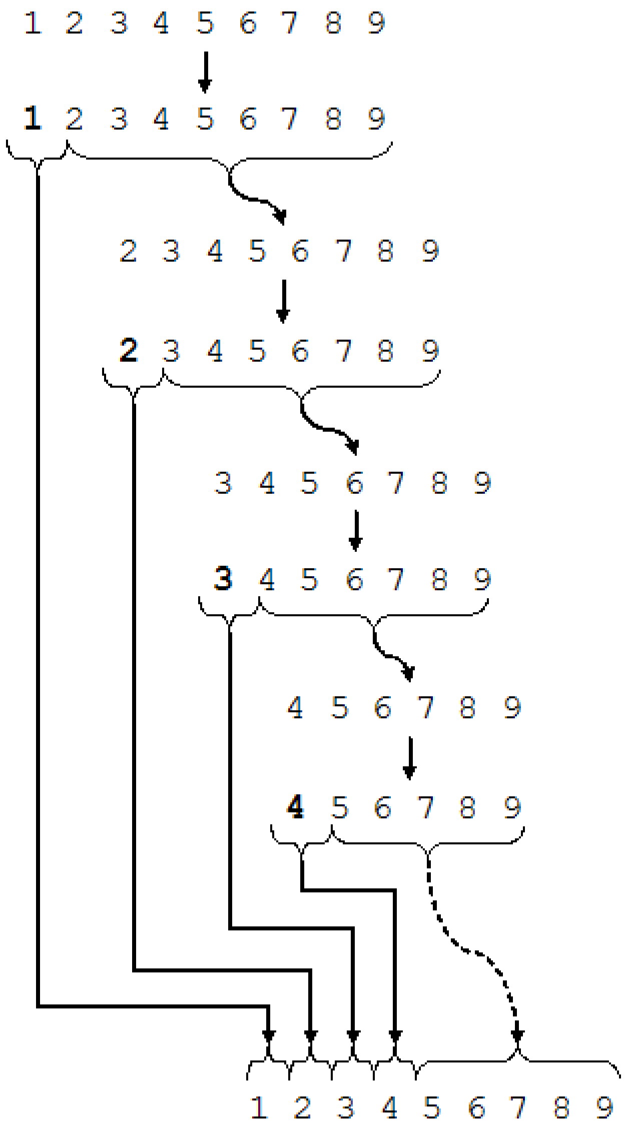


Figure 2: Quicksort on an already sorted array.

These are the steps taken to sort an array using QuickSort:

```
public void quickSort(int array[])
// pre: array is full, all elements are non-null integers
// post: the array is sorted in ascending order
{
         quickSort(array, 0, array.length - 1);
            // quicksort all the elements in the array
}


public void quickSort(int array[], int start, int end)
{
        int i = start;                          // index of left-to-right scan
        int k = end;                            // index of right-to-left scan

        if (end - start >= 1) // check that there are at least two elements to sort
        {
           int pivot = array[start];
                    // set the pivot as the first element in the partition

           while (k > i) // while the scan indices from left and right have not met,
                {
                        while (array[i] <= pivot && i <= end && k > i)
                              // from the left, look for the first
                                i++;              // element greater than the pivot
                        while (array[k] > pivot && k >= start && k >= i)
                                     // from the right, look for the first
                            k--;              // element not greater than the pivot
                        if (k > i)   // if the left seekindex is still smaller than
                                swap(array, i, k);
                            // the right index, swap the corresponding elements
                }
                swap(array, start, k);          // after the indices have crossed
                                                //  swap the last element in
                                                // the left partition with the pivot
                quickSort(array, start, k - 1); // quicksort the left partition
                quickSort(array, k + 1, end);   // quicksort the right partition
        }
        else // if there is only one element in the partition, do not do any sorting
        {
                return;                     // the array is sorted, so exit
        }
}

public void swap(int array[], int index1, int index2)
// pre: array is full and index1, index2 < array.length
// post: the values at indices 1 and 2 have been swapped
{
         int temp = array[index1];           // store the first value in a temp
         array[index1] = array[index2];  // copy the value of the second into the first
         array[index2] = temp;         // copy the value of the temp into the second
}
```