1.Algorithms

In mathematics and computer science, an algorithm is a self-contained sequence of actions to be performed. Algorithms can perform calculation, data processing and automated reasoning tasks.

An algorithm is an effective method that can be expressed within a finite amount of space and time and in a well-defined formal language for calculating a function. Starting from an initial state and initial input (perhaps empty) the instructions describe a computation that, when executed, proceeds through a finite number of well-defined successive states, eventually producing "output" and terminating at a final ending state. The transition from one state to the next is not necessarily deterministic; some algorithms, known as randomized algorithms, incorporate random input. [1]

1.1 Sorting algorithms

Sorting is the process of rearranging a sequence of objects so as to put them in some logical order. Sorting plays a major role in commercial data processing and in modern scientific computing. Applications abound in transaction processing, combinatorial optimization, astrophysics, molecular dynamics, linguistics, genomics, weather prediction, and many other fields.

In this chapter, we consider several classical sorting methods and an efficient implementation of a fundamental data type known as the priority queue. We discuss the theoretical basis for comparing sorting algorithms and conclude the chapter with a survey of applications of sorting and priority-queue algorithms. [2]

Popular sorting algorithms:

1. Simple sorts

Two of the simplest sorts are insertion sort and selection sort, both of which are efficient on small data, due to low overhead, but not efficient on large data. Insertion sort is generally faster than selection sort in practice, due to fewer comparisons and good performance on almost-sorted data, and thus is preferred in practice, but selection sort uses fewer writes, and thus is used when write performance is a limiting factor.

* Insertion sort: It is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and is often used as part of more sophisticated algorithms. It works by taking elements from the list one by one and inserting them in their correct position into a new sorted list. In arrays, the new list and the remaining elements can share the array's space, but insertion is expensive, requiring shifting all following elements over by one. Shell sort (see below) is a variant of insertion sort that is more efficient for larger lists.
* Selection sort: It is an in-place comparison sort. It has O(n2) complexity, making it inefficient on large lists, and generally performs worse than the similar insertion sort. Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations.

The algorithm finds the minimum value, swaps it with the value in the first position, and repeats these steps for the remainder of the list. It does no more than n swaps, and thus is useful where swapping is very expensive.

1. Efficient sorts

Practical general sorting algorithms are almost always based on an algorithm with average time complexity (and generally worst-case complexity) O(n log n), of which the most common are heap sort, merge sort, and quicksort. Each has advantages and drawbacks, with the most significant being that simple implementation of merge sort uses O(n) additional space, and simple implementation of quicksort has O(n2) worst-case complexity. These problems can be solved or ameliorated at the cost of a more complex algorithm.

While these algorithms are asymptotically efficient on random data, for practical efficiency on real-world data various modifications are used. First, the overhead of these algorithms becomes significant on smaller data, so often a hybrid algorithm is used, commonly switching to insertion sort once the data is small enough. Second, the algorithms often perform poorly on already sorted data or almost sorted data – these are common in real-world data, and can be sorted in O(n) time by appropriate algorithms. Finally, they may also be unstable, and stability is often a desirable property in a sort. Thus more sophisticated algorithms are often employed, such as Timsort (based on merge sort) or introsort (based on quicksort, falling back to heap sort).

* Merge sort: It takes advantage of the ease of merging already sorted lists into a new sorted list. It starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting lists of two into lists of four, then merges those lists of four, and so on; until at last two lists are merged into the final sorted list. Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O(n log n). It is also easily applied to lists, not only arrays, as it only requires sequential access, not random access. However, it has additional O(n) space complexity, and involves a large number of copies in simple implementations.

Merge sort has seen a relatively recent surge in popularity for practical implementations, due to its use in the sophisticated algorithm Timsort, which is used for the standard sort routine in the programming languages Python and Java (as of JDK7). Merge sort itself is the standard routine in Perl, among others, and has been used in Java at least since 2000 in JDK1.3.

* Heap sort: It is a much more efficient version of selection sort. It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a heap, a special type of binary tree. Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes O(log n) time, instead of O(n) for a linear scan as in simple selection sort. This allows Heapsort to run in O(n log n) time, and this is also the worst case complexity.
* Quick sort: It is a divide and conquer algorithm which relies on a partition operation: to partition an array an element called a pivot is selected. All elements smaller than the pivot are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and in-place. The lesser and greater sublists are then recursively sorted. This yields average time complexity of O(n log n), with low overhead, and thus this is a popular algorithm. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log n) space usage, quicksort is one of the most popular sorting algorithms and is available in many standard programming libraries.

The important caveat about quicksort is that its worst-case performance is O(n2); while this is rare, in naive implementations (choosing the first or last element as pivot) this occurs for sorted data, which is a common case. The most complex issue in quicksort is thus choosing a good pivot element, as consistently poor choices of pivots can result in drastically slower O(n2) performance, but good choice of pivots yields O(n log n) performance, which is asymptotically optimal. For example, if at each step the median is chosen as the pivot then the algorithm works in O(n log n). Finding the median, such as by the median of medians selection algorithm is however an O(n) operation on unsorted lists and therefore exacts significant overhead with sorting. In practice choosing a random pivot almost certainly yields O(n log n) performance.

1. Bubble sort and variants

Bubble sort, and variants such as the cocktail sort, are simple but highly inefficient sorts. They are thus frequently seen in introductory texts, and are of some theoretical interest due to ease of analysis, but they are rarely used in practice, and primarily of recreational interest. Some variants, such as the Shell sort, have open questions about their behavior.

* Bubble sort: Bubble sort is a simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. This algorithm's average time and worst-case performance is O(n2), so it is rarely used to sort large, unordered data sets. Bubble sort can be used to sort a small number of items (where its asymptotic inefficiency is not a high penalty). Bubble sort can also be used efficiently on a list of any length that is nearly sorted (that is, the elements are not significantly out of place). For example, if any number of elements are out of place by only one position (e.g. 0123546789 and 1032547698), bubble sort's exchange will get them in order on the first pass, the second pass will find all elements in order, so the sort will take only 2n time.
* Shell sort: It was invented by Donald Shell in 1959. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. The concept behind Shell sort is that both of these algorithms perform in O(kn) time, where k is the greatest distance between two out-of-place elements. This means that generally, they perform in O(n2), but for data that is mostly sorted, with only a few elements out of place, they perform faster. So, by first sorting elements far away, and progressively shrinking the gap between the elements to sort, the final sort computes much faster. One implementation can be described as arranging the data sequence in a two-dimensional array and then sorting the columns of the array using insertion sort.

The worst-case time complexity of Shell sort largely depends on the gap sequence used, and can range from O(n2) to Also, unlike efficient sorting algorithms, Shell sort does not require use of the call stack, making it useful in embedded systems where memory is at a premium.

* Comb sort: It is a relatively simple sorting algorithm originally designed by Wlodzimierz Dobosiewicz in 1980. It was later rediscovered and popularized by Stephen Lacey and Richard Box with a Byte Magazine article published in April 1991. Comb sort improves on bubble sort. The basic idea is to eliminate turtles, or small values near the end of the list, since in a bubble sort these slow the sorting down tremendously. (Rabbits, large values around the beginning of the list, do not pose a problem in bubble sort)

1. Distribution sort

Distribution sort refers to any sorting algorithm where data are distributed from their input to multiple intermediate structures which are then gathered and placed on the output. For example, both bucket sort and flash sort are distribution based sorting algorithms. Distribution sorting algorithms can be used on a single processor, or they can be a distributed algorithm, where individual subsets are separately sorted on different processors, then combined. This allows external sorting of data too large to fit into a single computer's memory.

* Counting sort: Counting sort is applicable when each input is known to belong to a particular set, S, of possibilities. The algorithm runs in O(|S| + n) time and O(|S|) memory where n is the length of the input. It works by creating an integer array of size |S| and using the ith bin to count the occurrences of the ith member of S in the input. Each input is then counted by incrementing the value of its corresponding bin. Afterward, the counting array is looped through to arrange all of the inputs in order. This sorting algorithm often cannot be used because S needs to be reasonably small for the algorithm to be efficient, but it is extremely fast and demonstrates great asymptotic behavior as n increases. It also can be modified to provide stable behavior.
* Bucket sort: Bucket sort is a divide and conquer sorting algorithm that generalizes counting sort by partitioning an array into a finite number of buckets. Each bucket is then sorted individually, either using a different sorting algorithm, or by recursively applying the bucket sorting algorithm.

A bucket sort works best when the elements of the data set are evenly distributed across all buckets.

* Radix sort: Radix sort is an algorithm that sorts numbers by processing individual digits. n numbers consisting of k digits each are sorted in O(n · k) time. Radix sort can process digits of each number either starting from the least significant digit (LSD) or starting from the most significant digit (MSD). The LSD algorithm first sorts the list by the least significant digit while preserving their relative order using a stable sort. Then it sorts them by the next digit, and so on from the least significant to the most significant, ending up with a sorted list. While the LSD radix sort requires the use of a stable sort, the MSD radix sort algorithm does not (unless stable sorting is desired). In-place MSD radix sort is not stable. It is common for the counting sort algorithm to be used internally by the radix sort.

A hybrid sorting approach, such as using insertion sort for small bins improves performance of radix sort significantly. [3]

1.1.1 Quick Sort

Quicksort (sometimes called partition-exchange sort) is an efficient sorting algorithm, serving as a systematic method for placing the elements of an array in order. Developed by Tony Hoare in 1959 and published in 1961, 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.

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. In efficient implementations it is 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.

Mathematical analysis of quicksort shows that, on average, the algorithm takes  comparisons to sort n items. In the worst case, it makes ) comparisons, though this behavior is rare.

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). This "median-of-three" rule counters the case of sorted (or reverse-sorted) input, and gives a better estimate of the optimal pivot (the true median) than selecting any single element, when no information about the ordering of the input is known.

Specifically, the expected number of comparisons needed to sort n elements with random pivot selection is. Median-of-three pivoting brings this down to , at the expense of a three-percent increase in the expected number of swaps. An even stronger pivoting rule, for larger arrays, is to pick the ninther, a recursive median-of-three (Mo3), defined as

ninther(a) = median(Mo3(first ⅓ of a), Mo3(middle ⅓ of a), Mo3(final ⅓ 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.

1. Formal analysis

* Worst-case analysis

The most unbalanced partition occurs when the pivot divides the list into two sublists of sizes 0 and *n* − 1. 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 *i*th call does *O*(*n* − *i*) work to do the partition, and {\displaystyle \textstyle \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 log2 *n* nested calls before we reach a list of size 1. This means that the depth of the [call tree](https://en.wikipedia.org/wiki/Call_stack) is log2 *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  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 . 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.

* 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(n²).

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(n/2)

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:

Solving the recurrence gives C(n) = 2n ln n ≈ 1.39n log₂ 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 log₂(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 log₂(n!) ≈ n(log₂ n − log₂ 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.[4]

* Elimination of recurrences

I learned about quicksort in my engineering classes, one of the few algorithm which I managed to understand then. Since it's a divide-and-conquer algorithm, you choose a pivot and divide the array. Unlike merge sort, which is also a divide-and-conquer algorithm and where all important work happens on combine steps, In quicksort, the real work happens in divide step and the combining step does nothing important.

By the way, the working of the algorithm will remain same whether you implement an iterative solution or a recursion solution. In iterative solution, we'll use Stack instead of recursion. Here are the steps to implement iterative quicksort in Java:

* Push the range (0...n) into the Stack
* Partition the given array with a pivot
* Pop the top element.
* Push the partitions ( index range ) into a stack if the range has more than one element
* Do the above 3 steps, till the stack, is empty

You might know that even though writing recursive algorithms are easy they are always slower than their Iterative counterpart. So, when Interviewer will ask you to choose a method in terms of  time complexity where memory is not a concern, which version will you choose?  
  
Well, both recursive and iterative quicksort are O(n log n) average case and O(n^2) in the worst case but the recursive version is shorter and clearer. Iterative is faster and makes you simulate the recursion call stack. [5]

* 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)

1. 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 quick sorts 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. 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 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. (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 *k*th 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 quick select. 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 quick select, 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 multi quicksort) 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. A 1999 assessment of a multi quicksort 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. A version of dual-pivot quicksort developed by Yaroslavskiy in 2009 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). 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.

* 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 quick sort

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() or at least O() as for standard quicksort, given for unique keys , 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() behaviours of standard quicksort and radix quicksort, and will be faster even in the best case of those comparison algorithms under these conditions of uniqueprefix(K) >> logN. See Powers for further discussion of the hidden overheads in comparison, radix and parallel sorting.

* 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 logn + O(n) comparisons (close to the information theoretic lower bound) and  operations; at worst they perform 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).[5]

1.1.2 Insertion sort

Insertion sort is a simple sorting algorithm that builds the final sorted array (or list) one item at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

Simple implementation: Jon Bentley shows a three-line C version, and a five-line optimized version

Efficient for (quite) small data sets, much like other quadratic sorting algorithms

More efficient in practice than most other simple quadratic (i.e., ) algorithms such as selection sort or bubble sort

Adaptive, i.e., efficient for data sets that are already substantially sorted: the time complexity is O(nk) when each element in the input is no more than k places away from its sorted position

Stable; i.e., does not change the relative order of elements with equal keys

In-place; i.e., only requires a constant amount O(1) of additional memory space

Online; i.e., can sort a list as it receives it

When people manually sort cards in a bridge hand, most use a method that is similar to insertion sort.

1. Algorithm for insertion sort

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. Each iteration, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

The resulting array after k iterations has the property where the first k + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:

https://upload.wikimedia.org/wikipedia/commons/3/32/Insertionsort-before.png

becomes

https://upload.wikimedia.org/wikipedia/commons/d/d9/Insertionsort-after.png

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

The most common variant of insertion sort, which operates on arrays, can be described as follows:

1. Suppose there exists a function called Insert designed to insert a value into a sorted sequence at the beginning of an array. It operates by beginning 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. The function has the side effect of overwriting the value stored immediately after the sorted sequence in the array.
2. To perform an insertion sort, begin at the left-most element of the array and invoke Insert to insert each element encountered into its correct position. The ordered sequence into which the element is inserted is stored at the beginning of the array in the set of indices already examined. Each insertion overwrites a single value: the value being inserted.

Pseudocode of the complete algorithm follows, where the arrays are zero-based:

**for** i = 1 **to** length(A)

j ← i

**while** j > 0 and A[j-1] > A[j]

**swap** A[j] and A[j-1]

j ← j - 1

**end while**

**end for**

The outer loop runs over all the elements except the first one, because the single-element prefix A[0:1] is trivially sorted, so the invariant that the first i+1 entries are sorted is true from the start. The inner loop moves element A[i] to its correct place so that after the loop, the first i+2 elements are sorted.

After expanding the "swap" operation in-place as x ← A[j]; A[j] ← A[j-1]; A[j-1] ← x (where x is a temporary variable), a slightly faster version can be produced that moves A[i] to its position in one go and only performs one assignment in the inner loop body:

**for** i = 1 **to** length(A)

x = A[i]

j = i - 1

**while** j >= 0 and A[j] > x

A[j+1] ← A[j]

j ← j – 1

**end while**

A[j+1] = x[[4]](https://en.wikipedia.org/wiki/Insertion_sort#cite_note-4)

**end for**

The new inner loop shifts elements to the right to clear a spot for x = A[i].

Note that although the common practice is to implement in-place, which requires checking the elements in-order, the order of checking (and removing) input elements is actually arbitrary. The choice can be made using almost any pattern, as long as all input elements are eventually checked (and removed from the input).

The algorithm can also be implemented in a recursive way. The idea is that to sort an array of n elements, A[0..n-1], one first orders the subarray A[0..n-2] and then inserts the last element (with index n-1).

**function** insertionSortR(array A, int n)

**if** n>1

insertionSortR(A,n-1)

x = A[n]

j = n-1

**while** j >= 0 **and** A[j] > x

A[j+1] ← A[j]

j ← j-1

**end while**

A[j+1] = x

**end if**

**end function**

1. Best, worst, and average cases

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., O(n)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., ).

The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quicksort; indeed, good quicksort implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.

Example: The following table shows the steps for sorting the sequence {3, 7, 4, 9, 5, 2, 6, 1}. In each step, the key under consideration is underlined. The key that was moved (or left in place because it was biggest yet considered) in the previous step is shown in bold.

3 7 4 9 5 2 6 1

**3** 7 4 9 5 2 6 1

3 **7** 4 9 5 2 6 1

3 **4** 7 9 5 2 6 1

3 4 7 **9** 5 2 6 1

3 4 **5** 7 9 2 6 1

**2** 3 4 5 7 9 6 1

2 3 4 5 **6** 7 9 1

**1** 2 3 4 5 6 7 9

[6]

References

[1]: <https://en.wikipedia.org/wiki/Algorithm>

[2]: <http://algs4.cs.princeton.edu/20sorting/>

[3]: <https://en.wikipedia.org/wiki/Sorting_algorithm>

[4]: <https://en.wikipedia.org/wiki/Quicksort>

[5]: <http://javarevisited.blogspot.com.tr/2016/09/iterative-quicksort-example-in-java-without-recursion.html>

[6]: <https://en.wikipedia.org/wiki/Quicksort>

[7]: <https://en.wikipedia.org/wiki/Insertion_sort>