

Date:20/12/2011

Project4

Sorting

Name : Areej Arar

ID:1091820

Dr: Iyad Jaber

# Sorting :

In [computer science](http://en.wikipedia.org/wiki/Computer_science), a **sorting algorithm** is an [algorithm](http://en.wikipedia.org/wiki/Algorithm) that puts elements of a [list](http://en.wikipedia.org/wiki/List_(computing)) in a certain [order](http://en.wikipedia.org/wiki/Total_order). The most-used orders are numerical order and [lexicographical order](http://en.wikipedia.org/wiki/Lexicographical_order). Efficient [sorting](http://en.wikipedia.org/wiki/Sorting) is important for optimizing the use of other algorithms (such as [search](http://en.wikipedia.org/wiki/Search_algorithm) and [merge](http://en.wikipedia.org/wiki/Merge_algorithm) algorithms) that require sorted lists to work correctly; it is also often useful for [canonicalizing](http://en.wikipedia.org/wiki/Canonicalization) 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](http://en.wikipedia.org/wiki/Total_order));
2. The output is a [permutation](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Bubble_sort) was analyzed as early as 1956.[[1]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-0) Although many consider it a solved problem, useful new sorting algorithms are still being invented (for example, [library sort](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Big_O_notation), [divide and conquer algorithms](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm), [data structures](http://en.wikipedia.org/wiki/Data_structure), [randomized algorithms](http://en.wikipedia.org/wiki/Randomized_algorithm), [best, worst and average case](http://en.wikipedia.org/wiki/Best,_worst_and_average_case) analysis, [time-space tradeoffs](http://en.wikipedia.org/wiki/Time-space_tradeoff), and lower bounds.

## Contents

* [1 Classification](http://en.wikipedia.org/wiki/Sorting_algorithm#Classification)
* [2 Summaries of popular sorting algorithms](http://en.wikipedia.org/wiki/Sorting_algorithm#Summaries_of_popular_sorting_algorithms)
  + [2.1 Bubble sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Bubble_sort)
  + [2.2 Selection sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Selection_sort)
  + [2.3 Insertion sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Insertion_sort)
  + [2.4 Shell sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Shell_sort)
  + [2.5 Comb sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Comb_sort)
  + [2.6 Merge sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Merge_sort)
  + [2.7 Heapsort](http://en.wikipedia.org/wiki/Sorting_algorithm#Heapsort)
  + [2.8 Quicksort](http://en.wikipedia.org/wiki/Sorting_algorithm#Quicksort)
  + [2.9 Counting sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Counting_sort)
  + [2.10 Radix sort](http://en.wikipedia.org/wiki/Sorting_algorithm#Radix_sort)
  + [2.11 Timsort](http://en.wikipedia.org/wiki/Sorting_algorithm#Timsort)
* [3 Memory usage patterns and index sorting](http://en.wikipedia.org/wiki/Sorting_algorithm#Memory_usage_patterns_and_index_sorting)
* [4 Inefficient/humorous sorts](http://en.wikipedia.org/wiki/Sorting_algorithm#Inefficient.2Fhumorous_sorts)

## Classification

Sorting algorithms used in [computer science](http://en.wikipedia.org/wiki/Computer_science) are often classified by:

* [Computational complexity](http://en.wikipedia.org/wiki/Computational_complexity_theory) ([worst](http://en.wikipedia.org/wiki/Worst-case_performance), [average](http://en.wikipedia.org/wiki/Average_performance) and [best](http://en.wikipedia.org/wiki/Best-case_performance) behavior) of element comparisons in terms of the size of the list \left( n \right). For typical sorting algorithms good behavior is [\mathcal{O}](http://en.wikipedia.org/wiki/Big_O_notation)\left( n \log n\right)and bad behavior is \mathcal{O}\left( n^2 \right). (See [Big O notation](http://en.wikipedia.org/wiki/Big_O_notation).) Ideal behavior for a sort is \mathcal{O}\left( n \right), but this is not possible in the average case. [Comparison-based sorting algorithms](http://en.wikipedia.org/wiki/Comparison_sort), which evaluate the elements of the list via an abstract key comparison operation, need at least \mathcal{O}\left( n \log n\right)comparisons for most inputs.
* [Computational complexity](http://en.wikipedia.org/wiki/Computational_complexity_theory) of swaps (for "in place" algorithms).
* Memory usage (and use of other computer resources). In particular, some sorting algorithms are "[in place](http://en.wikipedia.org/wiki/In-place_algorithm)". This means that they need only \mathcal{O}(1)or \mathcal{O}(\log n)memory beyond the items being sorted and they don't need to create auxiliary locations for data to be temporarily stored, as in other sorting algorithms.
* Recursion. Some algorithms are either recursive or non-recursive, while others may be both (e.g., merge sort).
* Stability: **stable sorting algorithms** maintain the relative order of records with equal keys (i.e., values).
* Whether or not they are a [comparison sort](http://en.wikipedia.org/wiki/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.
* Adaptability: Whether or not the presortedness of the input affects the running time. Algorithms that take this into account are known to be [adaptive](http://en.wikipedia.org/wiki/Adaptive_sort).

[http://bits.wikimedia.org/skins-1.18/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:SortingAlgoComp.png)

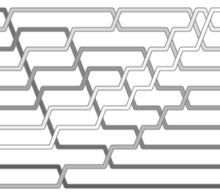
The complexity of different algorithms in a specific situation.

In this table, *n* is the number of records to be sorted. The columns "Average" and "Worst" give the time complexity in each case, under the assumption that the length of each key is constant, and that therefore all comparisons, swaps, and other needed operations can proceed in constant time. "Memory" denotes the amount of auxiliary storage needed beyond that used by the list itself, under the same assumption. These are all [comparison sorts](http://en.wikipedia.org/wiki/Comparison_sort). The run time and the memory of algorithms could be measured using various notations like theta, sigma, Big-O, small-o, etc. The memory and the run times below are applicable for all the 5 notations.

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| [Comparison sorts](http://en.wikipedia.org/wiki/Comparison_sort) | | | | | | | | |
| **Name** | | **Best** | **Average** | **Worst** | **Memory** | **Stable** | **Method** | **Other notes** |
| [Quicksort](http://en.wikipedia.org/wiki/Quicksort) | | 20 !\mathcal{} n \log n | 20 !\mathcal{} n \log n | 25 !\mathcal{} n^2 | 05 !\mathcal{} \log n | Depends | Partitioning | Quicksort can be done in place with O(log(*n*)) stack space, but this sort is unstable. variants use an O(*n*) space array to store the partition. An O(*n*) space implementation can be stable. |
| [Merge sort](http://en.wikipedia.org/wiki/Merge_sort) | | 20 !\mathcal{} {n \log n} | 20 !\mathcal{} {n \log n} | 20 !\mathcal{} {n \log n} | 15 !Depends; worst case is  \mathcal{} n | Yes | Merging | Used to sort this table in Firefox [[2]](http://mxr.mozilla.org/seamonkey/source/js/src/jsarray.c). |
| [Heapsort](http://en.wikipedia.org/wiki/Heapsort) | | 20 !\mathcal{} {n \log n} | 20 !\mathcal{} {n \log n} | 20 !\mathcal{} {n \log n} | 00 !\mathcal{} {1} | No | Selection |  |
| [Insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) | | 15 ! \mathcal{} n | 25 ! \mathcal{} n^2 | 25 ! \mathcal{} n^2 | 00 ! \mathcal{} {1} | Yes | Insertion | Average case is also \mathcal{O}\left( {n + d} \right), where *d* is the number of [inversions](http://en.wikipedia.org/wiki/Permutation_groups#Transpositions.2C_simple_transpositions.2C_inversions_and_sorting) |
| [Selection sort](http://en.wikipedia.org/wiki/Selection_sort) | | 25 ! \mathcal{} n^2 | 25 ! \mathcal{} n^2 | 25 ! \mathcal{} n^2 | 00 ! \mathcal{} {1} | Depends [[5]](http://www.algolist.net/Algorithms/Sorting/Selection_sort). | Selection | Its stability depends on the implementation. Used to sort this table in Safari or other Webkit web browser [[6]](http://svn.webkit.org/repository/webkit/trunk/Source/JavaScriptCore/runtime/ArrayPrototype.cpp). |
| [Timsort](http://en.wikipedia.org/wiki/Timsort) | | 15 ! \mathcal{} {n} | 20 ! \mathcal{} {n \log n} | 20 ! \mathcal{} {n \log n} | 15 ! \mathcal{} n | Yes | Insertion & Merging | \mathcal{} {n} comparisons when the data is already sorted or reverse sorted. |
| [Shell sort](http://en.wikipedia.org/wiki/Shell_sort) | | 15 !\mathcal{} n | 23 !\mathcal{} n (\log n)^2  or  \mathcal{} n^{3/2} | 23 !Depends on gap sequence; best known is \mathcal{} n (\log n)^2 | 00 !\mathcal{} 1 | No | Insertion |  |
| [Bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) | | 15 !\mathcal{} n | 25 !\mathcal{} n^2 | 25 !\mathcal{} n^2 | 00 !\mathcal{} {1} | Yes | Exchanging | Tiny code size |
| [Binary tree sort](http://en.wikipedia.org/wiki/Binary_tree_sort) | | 15 !\mathcal{} n | 20 !\mathcal{} {n \log n} | 20 !\mathcal{} {n \log n} | 15 !\mathcal{} n | Yes | Insertion | When using a [self-balancing binary search tree](http://en.wikipedia.org/wiki/Self-balancing_binary_search_tree) |
| [Cycle sort](http://en.wikipedia.org/wiki/Cycle_sort) | | 50 !— | 25 ! \mathcal{} n^2 | 25 ! \mathcal{} n^2 | 00 !\mathcal{} {1} | No | Insertion | In-place with theoretically optimal number of writes |
| [Library sort](http://en.wikipedia.org/wiki/Library_sort) | | 50 !— | 20 ! \mathcal{} {n \log n} | 25 ! \mathcal{} n^2 | 15 ! \mathcal{} n | Yes | Insertion |  |
| [Smoothsort](http://en.wikipedia.org/wiki/Smoothsort) | | 15 !\mathcal{} {n} | 20 !\mathcal{} {n \log n} | 20 !\mathcal{} {n \log n} | 00 !\mathcal{} {1} | No | Selection | An [adaptive sort](http://en.wikipedia.org/wiki/Adaptive_sort) - \mathcal{} {n} comparisons when the data is already sorted, and 0 swaps. |
| [Strand sort](http://en.wikipedia.org/wiki/Strand_sort) | | 15 !\mathcal{} n | 25 !\mathcal{} n^2 | 25 !\mathcal{} n^2 | 15 !\mathcal{} n | Yes | Selection |  |
|  |
| [Cocktail sort](http://en.wikipedia.org/wiki/Cocktail_sort) | | 15 !\mathcal{} n | 25 !\mathcal{} n^2 | 25 ! \mathcal{} n^2 | 00 !\mathcal{} {1} | Yes | Exchanging |  |
| [Comb sort](http://en.wikipedia.org/wiki/Comb_sort) | | 50 !— | 50 !— | 25 ! \mathcal{} n^2 | 00 ! \mathcal{} {1} | No | Exchanging | Small code size |
| [Gnome sort](http://en.wikipedia.org/wiki/Gnome_sort) | | 15 ! \mathcal{} n | 25 ! \mathcal{} n^2 | 25 ! \mathcal{} n^2 | 00 ! \mathcal{} {1} | Yes | Exchanging | Tiny code size |
| [Bogosort](http://en.wikipedia.org/wiki/Bogosort) | | 15 ! \mathcal{} n | 45 ! \mathcal{} n \cdot n! | 45 ! \mathcal{} {n \cdot n! \to \infty} | 00 ! \mathcal{} {1} | No | Luck | Randomly permute the array and check if sorted. |

## Summaries of popular sorting algorithms

### Bubble sort

[](http://en.wikipedia.org/wiki/File:Bubblesort-edited.png)

[http://bits.wikimedia.org/skins-1.18/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Bubblesort-edited.png)

A bubble sort, a sorting algorithm that continuously steps through a list, [swapping](http://en.wikipedia.org/wiki/Swap_(computer_science)) items until they appear in the correct order.

*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 and worst case performance is O(*n*2), 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 inefficiency is not a high penalty). Bubble sort may also be efficiently used on a list that is already sorted except for a very small number of elements. For example, if only one element is not in order, bubble sort will take only *2n* time. If two elements are not in order, bubble sort will take only at most *3n* time.

Bubble sort average case and worst case are both O(*n*²).

### Selection sort

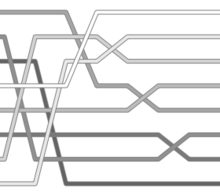
*Selection sort* is an [in-place](http://en.wikipedia.org/wiki/In-place_algorithm) [comparison sort](http://en.wikipedia.org/wiki/Comparison_sort). It has [O](http://en.wikipedia.org/wiki/Big_O_notation)(*n*2) complexity, making it inefficient on large lists, and generally performs worse than the similar [insertion sort](http://en.wikipedia.org/wiki/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.

### Insertion sort

*Insertion sort* is a simple sorting algorithm that is relatively efficient for small lists and mostly sorted lists, and often is 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](http://en.wikipedia.org/wiki/Shell_sort) (see below) is a variant of insertion sort that is more efficient for larger lists.

### Shell sort

[](http://en.wikipedia.org/wiki/File:Shellsort-edited.png)

[http://bits.wikimedia.org/skins-1.18/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Shellsort-edited.png)

A Shell sort, different from bubble sort in that it moves elements numerous positions [swapping](http://en.wikipedia.org/wiki/Swap_(computer_science))

*Shell sort* was invented by [Donald Shell](http://en.wikipedia.org/wiki/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. 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.

### Comb sort

*Comb sort* is a relatively simplistic sorting algorithm originally designed by [Wlodzimierz Dobosiewicz](http://en.wikipedia.org/w/index.php?title=Wlodzimierz_Dobosiewicz&action=edit&redlink=1) in 1980. Later it was rediscovered and popularized by [Stephen Lacey](http://en.wikipedia.org/w/index.php?title=Stephen_Lacey&action=edit&redlink=1) and [Richard Box](http://en.wikipedia.org/w/index.php?title=Richard_Box&action=edit&redlink=1) with a [Byte Magazine](http://en.wikipedia.org/wiki/Byte_Magazine) article published in April 1991. Comb sort improves on [bubble sort](http://en.wikipedia.org/wiki/Bubble_sort), and rivals algorithms like [Quicksort](http://en.wikipedia.org/wiki/Quicksort). 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.)

### Merge sort

*Merge sort* 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*). Merge sort has seen a relatively recent surge in popularity for practical implementations, being used for the standard sort routine in the programming languages [Perl](http://en.wikipedia.org/wiki/Perl),[[7]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-6) [Python](http://en.wikipedia.org/wiki/Python_(programming_language)) (as [timsort](http://en.wikipedia.org/wiki/Timsort)[[8]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-7)), and [Java](http://en.wikipedia.org/wiki/Java_(programming_language)) (also uses timsort as of [JDK7](http://en.wikipedia.org/wiki/JDK7)[[9]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-8)), among others. Merge sort has been used in Java at least since 2000 in JDK1.3.[[10]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-mergesort_in_jdk13-9)[[11]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-jdk13_since_2000-10)

### Heapsort

*Heapsort* is a much more efficient version of [selection sort](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Heap_(data_structure)), a special type of [binary tree](http://en.wikipedia.org/wiki/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.

### Quicksort

*Quicksort* is a [divide and conquer](http://en.wikipedia.org/wiki/Divide_and_conquer_algorithm) [algorithm](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/In-place_algorithm). The lesser and greater sublists are then recursively sorted. 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 most complex issue in quicksort is choosing a good pivot element; consistently poor choices of pivots can result in drastically slower O(*n*²) performance, if at each step the [median](http://en.wikipedia.org/wiki/Median) is chosen as the pivot then the algorithm works in O(*n* log *n*). Finding the median however, is an O(n) operation on unsorted lists and therefore exacts its own penalty with sorting.

### 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 *i*th bin to count the occurrences of the *i*th 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 cannot often be used because *S* needs to be reasonably small for it to be efficient, but the algorithm is extremely fast and demonstrates great asymptotic behavior as *n* increases. It also can be modified to provide stable behavior.

### 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](http://en.wikipedia.org/wiki/Least_significant_digit) (LSD) or starting from the [most significant digit](http://en.wikipedia.org/wiki/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](http://en.wikipedia.org/wiki/Counting_sort) algorithm to be used internally by the radix sort. Hybrid sorting approach, such as using [insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) for small bins improves performance of radix sort significantly.

### Timsort

*Timsort* finds runs in the data, creates runs with insertion sort if necessary, and then uses merge sort to create the final sorted list. It has the same complexity (O(nlogn)) in the average and worst cases, but with pre-sorted data it goes down to O(n).

## Memory usage patterns and index sorting

When the size of the array to be sorted approaches or exceeds the available primary memory, so that (much slower) disk or swap space must be employed, the memory usage pattern of a sorting algorithm becomes important, and an algorithm that might have been fairly efficient when the array fit easily in RAM may become impractical. In this scenario, the total number of comparisons becomes (relatively) less important, and the number of times sections of memory must be copied or swapped to and from the disk can dominate the performance characteristics of an algorithm. Thus, the number of passes and the localization of comparisons can be more important than the raw number of comparisons, since comparisons of nearby elements to one another happen at [system bus](http://en.wikipedia.org/wiki/Computer_bus) speed (or, with caching, even at [CPU](http://en.wikipedia.org/wiki/Central_Processing_Unit) speed), which, compared to disk speed, is virtually instantaneous.

For example, the popular recursive [quicksort](http://en.wikipedia.org/wiki/Quicksort) algorithm provides quite reasonable performance with adequate RAM, but due to the recursive way that it copies portions of the array it becomes much less practical when the array does not fit in RAM, because it may cause a number of slow copy or move operations to and from disk. In that scenario, another algorithm may be preferable even if it requires more total comparisons.

One way to work around this problem, which works well when complex records (such as in a [relational database](http://en.wikipedia.org/wiki/Relational_database)) are being sorted by a relatively small key field, is to create an index into the array and then sort the index, rather than the entire array. (A sorted version of the entire array can then be produced with one pass, reading from the index, but often even that is unnecessary, as having the sorted index is adequate.) Because the index is much smaller than the entire array, it may fit easily in memory where the entire array would not, effectively eliminating the disk-swapping problem. This procedure is sometimes called "tag sort".[[12]](http://en.wikipedia.org/wiki/Sorting_algorithm#cite_note-11)

Another technique for overcoming the memory-size problem is to combine two algorithms in a way that takes advantages of the strength of each to improve overall performance. For instance, the array might be subdivided into chunks of a size that will fit easily in RAM (say, a few thousand elements), the chunks sorted using an efficient algorithm (such as [quicksort](http://en.wikipedia.org/wiki/Quicksort) or [heapsort](http://en.wikipedia.org/wiki/Heapsort)), and the results merged as per [mergesort](http://en.wikipedia.org/wiki/Mergesort). This is less efficient than just doing mergesort in the first place, but it requires less physical RAM (to be practical) than a full quicksort on the whole array.

Techniques can also be combined. For sorting very large sets of data that vastly exceed system memory, even the index may need to be sorted using an algorithm or combination of algorithms designed to perform reasonably with [virtual memory](http://en.wikipedia.org/wiki/Virtual_memory), i.e., to reduce the amount of swapping required.

## Inefficient/humorous sorts

These are algorithms that are extremely slow compared to those discussed above — [Bogosort](http://en.wikipedia.org/wiki/Bogosort) O(n\cdot n!), [Stooge sort](http://en.wikipedia.org/wiki/Stooge_sort) *O*(*n*2.7).

|  |  |
| --- | --- |
| **Sorting algorithms** | |
|  |  |
| **Theory** | * [Computational complexity theory](http://en.wikipedia.org/wiki/Computational_complexity_theory) * [Big O notation](http://en.wikipedia.org/wiki/Big_O_notation) * [Total order](http://en.wikipedia.org/wiki/Total_order) * [Lists](http://en.wikipedia.org/wiki/List_(computing)) * [Stability](http://en.wikipedia.org/wiki/Sorting_algorithm#Stability) * [Comparison sort](http://en.wikipedia.org/wiki/Comparison_sort) * [Adaptive sort](http://en.wikipedia.org/wiki/Adaptive_sort) * [Sorting network](http://en.wikipedia.org/wiki/Sorting_network) * [Integer sorting](http://en.wikipedia.org/wiki/Integer_sorting) |
|  |  |
| **Exchange sorts** | * [Bubble sort](http://en.wikipedia.org/wiki/Bubble_sort) * [Cocktail sort](http://en.wikipedia.org/wiki/Cocktail_sort) * [Odd–even sort](http://en.wikipedia.org/wiki/Odd%E2%80%93even_sort) * [Comb sort](http://en.wikipedia.org/wiki/Comb_sort) * [Gnome sort](http://en.wikipedia.org/wiki/Gnome_sort) * [Quicksort](http://en.wikipedia.org/wiki/Quicksort) * [Stooge sort](http://en.wikipedia.org/wiki/Stooge_sort) * [Bogosort](http://en.wikipedia.org/wiki/Bogosort) |
|  |  |
| **Selection sorts** | * [Selection sort](http://en.wikipedia.org/wiki/Selection_sort) * [Heapsort](http://en.wikipedia.org/wiki/Heapsort) * [Smoothsort](http://en.wikipedia.org/wiki/Smoothsort) * [Cartesian tree sort](http://en.wikipedia.org/wiki/Cartesian_tree#Application_in_sorting) * [Tournament sort](http://en.wikipedia.org/wiki/Tournament_sort) * [Cycle sort](http://en.wikipedia.org/wiki/Cycle_sort) |
|  |  |
| **Insertion sorts** | * [Insertion sort](http://en.wikipedia.org/wiki/Insertion_sort) * [Shellsort](http://en.wikipedia.org/wiki/Shellsort) * [Tree sort](http://en.wikipedia.org/wiki/Tree_sort) * [Library sort](http://en.wikipedia.org/wiki/Library_sort) * [Patience sorting](http://en.wikipedia.org/wiki/Patience_sorting) |
|  |  |
| **Merge sorts** | * [Merge sort](http://en.wikipedia.org/wiki/Merge_sort) * [Polyphase merge sort](http://en.wikipedia.org/wiki/Polyphase_merge_sort) * [Strand sort](http://en.wikipedia.org/wiki/Strand_sort) |
|  |  |
| **Distribution sorts** | * [American flag sort](http://en.wikipedia.org/wiki/American_flag_sort) * [Bead sort](http://en.wikipedia.org/wiki/Bead_sort) * [Bucket sort](http://en.wikipedia.org/wiki/Bucket_sort) * [Burstsort](http://en.wikipedia.org/wiki/Burstsort) * [Counting sort](http://en.wikipedia.org/wiki/Counting_sort) * [Pigeonhole sort](http://en.wikipedia.org/wiki/Pigeonhole_sort) * [Proxmap sort](http://en.wikipedia.org/wiki/Proxmap_sort) * [Radix sort](http://en.wikipedia.org/wiki/Radix_sort) * [Flashsort](http://en.wikipedia.org/wiki/Flashsort) |
|  |  |
| **Concurrent sorts** | * [Bitonic sorter](http://en.wikipedia.org/wiki/Bitonic_sorter) * [Batcher odd–even mergesort](http://en.wikipedia.org/wiki/Batcher_odd%E2%80%93even_mergesort) * [Pairwise sorting network](http://en.wikipedia.org/wiki/Pairwise_sorting_network) |
|  |  |
| **Hybrid sorts** | * [Timsort](http://en.wikipedia.org/wiki/Timsort) * [Introsort](http://en.wikipedia.org/wiki/Introsort) * [Spreadsort](http://en.wikipedia.org/wiki/Spreadsort) * [UnShuffle sort](http://en.wikipedia.org/wiki/UnShuffle_sort) * [JSort](http://en.wikipedia.org/wiki/JSort) |
|  |  |
| **Quantum sorts** | * [Spaghetti sort](http://en.wikipedia.org/wiki/Spaghetti_sort) |
|  |  |
| **Other** | * [Bogosort](http://en.wikipedia.org/wiki/Bogosort) * [Topological sorting](http://en.wikipedia.org/wiki/Topological_sorting) * [Pancake sorting](http://en.wikipedia.org/wiki/Pancake_sorting) |