Merge Sort:

In sorting *n* objects, merge sort has an [average](https://en.wikipedia.org/wiki/Average_performance) and [worst-case performance](https://en.wikipedia.org/wiki/Worst-case_performance) of [O](https://en.wikipedia.org/wiki/Big_O_notation)(*n* log *n*). If the running time of merge sort for a list of length *n* is *T*(*n*), then the recurrence *T*(*n*) = 2*T*(*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](https://en.wikipedia.org/wiki/Master_theorem).

In the worst case, the number of comparisons merge sort makes is equal to or slightly smaller than (*n* ⌈[lg](https://en.wikipedia.org/wiki/Binary_logarithm" \o "Binary logarithm) *n*⌉ - 2⌈lg*n*⌉ + 1), which is between (*n* lg *n* - *n* + 1) and (*n* lg *n* + *n* + O(lg *n*)).[[4]](https://en.wikipedia.org/wiki/Merge_sort#cite_note-4)

For large *n* and a randomly ordered input list, merge sort's expected (average) number of comparisons approaches *α*·*n* fewer than the worst case where {\displaystyle \alpha =-1+\sum \_{k=0}^{\infty }{\frac {1}{2^{k}+1}}\approx 0.2645.}

In the *worst* case, merge sort does about 39% fewer comparisons than [quicksort](https://en.wikipedia.org/wiki/Quicksort) does in the *average* case. In terms of moves, merge sort's worst case complexity is [O](https://en.wikipedia.org/wiki/Big_O_notation)(*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.[[*citation needed*](https://en.wikipedia.org/wiki/Wikipedia:Citation_needed)]

Merge sort is more efficient than quicksort for some types of lists if the data to be sorted can only be efficiently accessed sequentially, and is thus popular in languages such as [Lisp](https://en.wikipedia.org/wiki/Lisp_programming_language), where sequentially accessed data structures are very common. Unlike some (efficient) implementations of quicksort, merge sort is a stable sort.

Merge sort's most common implementation does not sort in place;[[5]](https://en.wikipedia.org/wiki/Merge_sort#cite_note-5) therefore, the memory size of the input must be allocated for the sorted output to be stored in (see below for versions that need only *n*/2 extra spaces).

Insertion Sort

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., O(*n*2)).

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](https://en.wikipedia.org/wiki/Quicksort); indeed, good [quicksort](https://en.wikipedia.org/wiki/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.

Quick Sort

**Worst-case analysis**[[edit](https://en.wikipedia.org/w/index.php?title=Quicksort&action=edit&section=11)]

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](https://en.wikipedia.org/wiki/Call_stack) 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**[[edit](https://en.wikipedia.org/w/index.php?title=Quicksort&action=edit&section=12)]

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**[[edit](https://en.wikipedia.org/w/index.php?title=Quicksort&action=edit&section=13)]

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](https://en.wikipedia.org/wiki/Uniform_distribution_(discrete)). We list here three common proofs to this claim providing different insights into quicksort's workings.

Selection Sort

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all *n* elements (this takes *n* − 1 comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining *n* − 1 elements and so on, for (*n* − 1) + (*n* − 2) + ... + 2 + 1 = *n*(*n* - 1) / 2 ∈ Θ(*n*2) comparisons (see [arithmetic progression](https://en.wikipedia.org/wiki/Arithmetic_progression)).[[1]](https://en.wikipedia.org/wiki/Selection_sort#cite_note-1) Each of these scans requires one swap for *n* − 1 elements (the final element is already in place).

Heap Sort

Heapsort primarily competes with [quicksort](https://en.wikipedia.org/wiki/Quicksort), another very efficient general purpose nearly-in-place comparison-based sort algorithm.

Quicksort is typically somewhat faster due to some factors, but the worst-case running time for quicksort is [O](https://en.wikipedia.org/wiki/Big_O_notation)(*n*2), which is unacceptable for large data sets and can be deliberately triggered given enough knowledge of the implementation, creating a security risk. See [quicksort](https://en.wikipedia.org/wiki/Quicksort) for a detailed discussion of this problem and possible solutions.

Thus, because of the O(*n* log *n*) upper bound on heapsort's running time and constant upper bound on its auxiliary storage, embedded systems with real-time constraints or systems concerned with security often use heapsort, such as the Linux kernel.[[20]](https://en.wikipedia.org/wiki/Heapsort#cite_note-20)