# Sorting Algorithm Comparison

By Alex Allain

Sorting algorithms are an important part of managing data. At Cprogramming.com, we offer tutorials for understanding the most important and common sorting techniques. Each algorithm has particular strengths and weaknesses and in many cases the best thing to do is just use the built-in sorting function qsort. For times when this isn't an option or you just need a quick and dirty sorting algorithm, there are a variety of choices.

Most sorting algorithms work by comparing the data being sorted. In some cases, it may be desirable to sort a large chunk of data (for instance, a struct containing a name and address) based on only a portion of that data. The piece of data actually used to determine the sorted order is called the key.

Sorting algorithms are usually judged by their efficiency. In this case, efficiency refers to the algorithmic efficiency as the size of the input grows large and is generally based on the number of elements to sort. Most of the algorithms in use have an algorithmic efficiency of either O(n^2) or O(n*log(n)). A few special case algorithms (one example is mentioned in Programming Pearls) can sort certain data sets faster than O(n*log(n)). These algorithms are not based on comparing the items being sorted and rely on tricks. It has been shown that no key-comparison algorithm can perform better than O(n*log(n)).

Many algorithms that have the same efficiency do not have the same speed on the same input. First, algorithms must be judged based on their average case, best case, and worst case efficiency. Some algorithms, such as quick sort, perform exceptionally well for some inputs, but horribly for others. Other algorithms, such as merge sort, are unaffected by the order of input data. Even a modified version of bubble sort can finish in O(n) for the most favorable inputs.

A second factor is the "constant term". As big-O notation abstracts away many of the details of a process, it is quite useful for looking at the big picture. But one thing that gets dropped out is the constant in front of the expression: for instance, O(c*n) is just O(n). In the real world, the constant, c, will vary across different algorithms. A well-implemented quicksort should have a much smaller constant multiplier than heap sort.

A second criterion for judging algorithms is their space requirement -- do they require scratch space or can the array be sorted in place (without additional memory beyond a few variables)? Some algorithms never require extra space, whereas some are most easily understood when implemented with extra space (heap sort, for instance, can be done in place, but conceptually it is much easier to think of a separate heap). Space requirements may even depend on the data structure used (merge sort on arrays versus merge sort on linked lists, for instance).

A third criterion is stability -- does the sort preserve the order of keys with equal values? Most simple sorts do just this, but some sorts, such as heap sort, do not.

The following chart compares sorting algorithms on the various criteria outlined above; the algorithms with higher constant terms appear first, though this is clearly an implementation-dependent concept and should only be taken as a rough guide when picking between sorts of the same big-O efficiency.

| | Time | | | | | |
|---|---|---|---|---|---|---|
| Sort | Average | Best | Worst | Space | Stability | Remarks |
| Bubble sort | O(n^2) | O(n^2) | O(n^2) | Constant | Stable | Always use a modified bubble sort |
| Modified Bubble sort | O(n^2) | O(n) | O(n^2) | Constant | Stable | Stops after reaching a sorted array |
| Selection Sort | O(n^2) | O(n^2) | O(n^2) | Constant | Stable | Even a perfectly sorted input requires scanning the entire array |
| Insertion Sort | O(n^2) | O(n) | O(n^2) | Constant | Stable | In the best case (already sorted), every insert requires constant time |
| Heap Sort | O(n*log(n)) | O(n*log(n)) | O(n*log(n)) | Constant | Instable | By using input array as storage for the heap, it is possible to achieve constant space |
| Merge Sort | O(n*log(n)) | O(n*log(n)) | O(n*log(n)) | Depends | Stable | On arrays, merge sort requires O(n) space; on linked lists, merge sort requires constant space |
| Quicksort | O(n*log(n)) | O(n*log(n)) | O(n^2) | Constant | Stable | Randomly picking a pivot value (or shuffling the array prior to sorting) can help avoid worst case scenarios such as a perfectly sorted array. |