**Comparison chart**

The following chart compares several sorting algorithms on the criteria outlined above. You can use this chart as a rough guide to help you pick between sorts of the same efficiency. Note that this chart lists more sorting algorithms than were covered in this module, but it isn't an exhaustive list of all existing sorting algorithms.

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| **Sort** | **Average time** | **Best time** | **Worst time** | **Space** | **Stability** | **Remarks** |
| Bubble | O(n²) | O(n) | O(n²) | Constant | Stable | Stops after reaching a sorted array. |
| Selection | O(n²) | O(n²) | O(n²) | Constant | Stable | Even a perfectly sorted input requires scanning the entire array. |
| Insertion | O(n²) | O(n) | O(n²) | Constant | Stable | In the best case (already sorted), every insert requires constant time. |
| Heap | O(n log n) | O(n log n) | O(n log n) | Constant | Not stable | By using an input array as storage for the heap, it is possible to achieve constant space. |
| Merge | O(n log n) | O(n log n) | O(n log n) | Depends | Stable | On arrays, merge sort requires O(n) space. But on linked lists, merge sort requires constant space. |
| Quick | O(n log n) | O(n log n) | O(n²) | Constant | Not stable | To avoid worst-case scenarios, it can help to pick a pivot value at random or shuffle the array before running the quicksort algorithm. |

The ideal sorting algorithm would have the following properties:

* Stable, meaning that elements with equal values aren't reordered.
* Operates in place, requiring O(1) extra space
* O(n log n) comparisons in the worst case
* O(n) swaps in the worst case
* Adaptive, with speeds up to O(n) when data is nearly sorted or when there are few unique values.