First, a definition, since it's pretty important: A **stable sort** is one that's guaranteed not to reorder elements with identical keys.

Recommendations:

**Quick sort:** When you don't need a stable sort and average case performance matters more than worst case performance. A quick sort is O(N log N) on average, O(N^2) in the worst case. A good implementation uses O(log N) auxiliary storage in the form of stack space for recursion.

**Merge sort:** When you need a stable, O(N log N) sort, this is about your only option. The only downsides to it are that it uses O(N) auxiliary space and has a slightly larger constant than a quick sort. There are some in-place merge sorts, but AFAIK they are all either not stable or worse than O(N log N). Even the O(N log N) in place sorts have so much larger a constant than the plain old merge sort that they're more theoretical curiosities than useful algorithms.

**Heap sort:** When you don't need a stable sort and you care more about worst case performance than average case performance. It's guaranteed to be O(N log N), and uses O(1) auxiliary space, meaning that you won't unexpectedly run out of heap or stack space on very large inputs.

**Introsort:** This is a quick sort that switches to a heap sort after a certain recursion depth to get around quick sort's O(N^2) worst case. It's almost always better than a plain old quick sort, since you get the average case of a quick sort, with guaranteed O(N log N) performance. Probably the only reason to use a heap sort instead of this is in severely memory constrained systems where O(log N) stack space is practically significant.

**Insertion sort**: When N is guaranteed to be small, including as the base case of a quick sort or merge sort. While this is O(N^2), it has a very small constant and is a stable sort.

**Bubble sort, selection sort**: When you're doing something quick and dirty and for some reason you can't just use the standard library's sorting algorithm. The only advantage these have over insertion sort is being slightly easier to implement.

**Non-comparison sorts:** Under some fairly limited conditions it's possible to break the O(N log N) barrier and sort in O(N). Here are some cases where that's worth a try:

**Counting sort:** When you are sorting integers with a limited range.

**Radix sort:** When log(N) is significantly larger than K, where K is the number of radix digits.

**Bucket sort:** When you can guarantee that your input is approximately uniformly distributed.

**selection sort**: Greedy algorithm

**Merge sort:** Divide and conquer