Why Java uses two kinds of Sort Merge Sort as well as Quick Sort

Highly likely from Josh Bloch [§](http://stackoverflow.com/revisions/15154287/2):

I did write these methods, so I suppose I'm qualified to answer. It is true that there is no single best sorting algorithm. QuickSort has two major deficiencies when compared to mergesort:

1. It's not stable (as parsifal noted).
2. It doesn't guarantee n log n performance; it can degrade to quadratic performance on pathological inputs.

Stability is a non-issue for primitive types, as there is no notion of identity as distinct from (value) equality. And the possibility of quadratic behavior was deemed not to be a problem in practice for Bentely and McIlroy's implementation (or subsequently for [Dual Pivot Quicksort](http://iaroslavski.narod.ru/quicksort/DualPivotQuicksort.pdf)), which is why these QuickSort variants were used for the primitive sorts.

Stability is a big deal when sorting arbitrary objects. For example, suppose you have objects representing email messages, and you sort them first by date, then by sender. You expect them to be sorted by date within each sender, but that will only be true if the sort is stable. That's why we elected to provide a stable sort (Merge Sort) to sort object references. (Techincally speaking, multiple sequential stable sorts result in a lexicographic ordering on the keys in the reverse order of the sorts: the final sort determines the most significant subkey.)

It's a nice side benefit that Merge Sort guarantees n log n (time) performance no matter what the input. Of course there is a down side: quick sort is an "in place" sort: it requies only log n external space (to maintain the call stack). Merge, sort, on the other hand, requires O(n) external space. The TimSort variant (introduced in Java SE 6) requires substantially less space (O(k)) if the input array is nearly sorted.

We know that quick sort is the best sorting algorithm.

Also, the [following](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6804124) is relevant:

The algorithm used by java.util.Arrays.sort and (indirectly) by java.util.Collections.sort to sort object references is a "modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist)." It is a reasonably fast stable sort that guarantees O(n log n) performance and requires O(n) extra space. In its day (it was written in 1997 by Joshua Bloch), it was a fine choice, but today but we can do much better.

Since 2003, Python's list sort has used an algorithm known as timsort (after Tim Peters, who wrote it). It is a stable, adaptive, iterative mergesort that requires far fewer than n log(n) comparisons when running on partially sorted arrays, while offering performance comparable to a traditional mergesort when run on random arrays. Like all proper mergesorts timsort is stable and runs in O(n log n) time (worst case). In the worst case, timsort requires temporary storage space for n/2 object references; in the best case, it requires only a small constant amount of space. Contrast this with the current implementation, which always requires extra space for n object references, and beats n log n only on nearly sorted lists.

Timsort is described in detail here: <http://svn.python.org/projects/python/trunk/Objects/listsort.txt>.

Tim Peters's original implementation is written in C. Joshua Bloch ported it from C to Java and end tested, benchmarked, and tuned the resulting code extensively. The resulting code is a drop-in replacement for java.util.Arrays.sort. On highly ordered data, this code can run up to 25 times as fast as the current implementation (on the HotSpot server VM). On random data, the speeds of the old and new implementations are comparable. For very short lists, the new implementation is substantially faster that the old even on random data (because it avoids unnecessary data copying).

Also, see [Is Java 7 using Tim Sort for the Method Arrays.Sort?](http://stackoverflow.com/questions/4018332/is-java-7-using-tim-sort-for-the-method-arrays-sort).

There isn't a single "best" choice. As with many other things, it's about tradeoffs.

**Answer given by Joshua Bloch**

We know that quick sort is the best sorting algorithm.

There isn't a single "best" choice. As with many other things, it's about tradeoffs.

As to the "why" question, only the original author can answer it with any certainty. However, the[following](http://bugs.sun.com/bugdatabase/view_bug.do?bug_id=6804124) is relevant:

The algorithm used by java.util.Arrays.sort and (indirectly) by java.util.Collections.sort to sort object references is a "modified mergesort (in which the merge is omitted if the highest element in the low sublist is less than the lowest element in the high sublist)." It is a reasonably fast stable sort that guarantees O(n log n) performance and requires O(n) extra space. In its day (it was written in 1997 by Joshua Bloch), it was a fine choice, but today but we can do much better.

Since 2003, Python's list sort has used an algorithm known as timsort (after Tim Peters, who wrote it). It is a stable, adaptive, iterative mergesort that requires far fewer than n log(n) comparisons when running on partially sorted arrays, while offering performance comparable to a traditional mergesort when run on random arrays. Like all proper mergesorts timsort is stable and runs in O(n log n) time (worst case). In the worst case, timsort requires temporary storage space for n/2 object references; in the best case, it requires only a small constant amount of space. Contrast this with the current implementation, which always requires extra space for n object references, and beats n log n only on nearly sorted lists.

Timsort is described in detail here: <http://svn.python.org/projects/python/trunk/Objects/listsort.txt>.

Tim Peters's original implementation is written in C. Joshua Bloch ported it from C to Java and end tested, benchmarked, and tuned the resulting code extensively. The resulting code is a drop-in replacement for java.util.Arrays.sort. On highly ordered data, this code can run up to 25 times as fast as the current implementation (on the HotSpot server VM). On random data, the speeds of the old and new implementations are comparable. For very short lists, the new implementation is substantially faster that the old even on random data (because it avoids unnecessary data copying).

Also, see [Is Java 7 using Tim Sort for the Method Arrays.Sort?](http://stackoverflow.com/questions/4018332/is-java-7-using-tim-sort-for-the-method-arrays-sort)

I did write these methods, so I suppose I'm qualified to answer. It is true that there is no single best sorting algorithm. QuickSort has two major deficiencies when compared to mergesort (1) It's not stable (as parsifal noted), and (2) It doesn't *guarantee* n log n performance; it can degrade to quadratic performance on pathological inputs. Stability is a non-issue for primitive types, as there is no notion of identity as distinct from (value) equality. And the possibility of quadratic behavior was deemed not to be a problem in practice for Bentely and McIlroy's implementation (or subsequently for Dual Pivot Quicksort), which is why these QuickSort variants were used for the primitive sorts.

Stability is a big deal when sorting arbitrary objects. For example, suppose you have objects representing email messages, and you sort them first by date, then by sender. You expect them to be sorted by date within each sender, but that will only be true if the sort is stable. That's why we elected to provide a stable sort (Merge Sort) to sort object references. (Techincally speaking, multiple sequential stable sorts result in a lexicographic ordering on the keys in the reverse order of the sorts: the final sort determines the most significant subkey.)

It's a nice side benefit that Merge Sort *guarantees* n log n (time) performance no matter what the input. Of course there is a down side: quick sort is an "in place" sort: it requies only log n external space (to maintain the call stack). Merge, sort, on the other hand, requires O(n) external space. The TimSort variant (introduced in Java SE 6) requires substantially less space (O(k)) if the input array is nearly sorted.

--Josh