1. **Partition of elements in the array** :  
   In the merge sort, the array is parted into just 2 halves (i.e. n/2).  
   whereas  
   In case of quick sort, the array is parted into any ratio. There is no compulsion of dividing the array of elements into equal parts in quick sort.
2. **Worst case complexity** :  
   The worst case complexity of quick sort is O(n2) as there is need of lot of comparisons in the worst condition.  
   whereas  
   In merge sort, worst case and average case has same complexities O(n log n).
3. **Usage with datasets** :  
   Merge sort can work well on any type of data sets irrespective of its size (either large or small).  
   whereas  
   The quick sort cannot work well with large datasets.
4. **Additional storage space requirement** :  
   Merge sort is not in place because it requires additional memory space to store the auxiliary arrays.  
   whereas  
   The quick sort is in place as it doesn’t require any additional storage.
5. **Efficiency** :  
   Merge sort is more efficient and works faster than quick sort in case of larger array size or datasets.  
   whereas  
   Quick sort is more efficient and works faster than merge sort in case of smaller array size or datasets.
6. **Sorting method** :  
   The quick sort is internal sorting method where the data is sorted in main memory.  
   whereas  
   The merge sort is external sorting method in which the data that is to be sorted cannot be accommodated in the memory and needed auxiliary memory for sorting.
7. **Stability** :  
   Merge sort is stable as two elements with equal value appear in the same order in sorted output as they were in the input unsorted array.  
   whereas  
   Quick sort is unstable in this scenario. But it can be made stable using some changes in code.
8. **Preferred for** :  
   Quick sort is preferred for arrays.  
   whereas  
   Merge sort is preferred for linked lists.
9. **Locality of reference** :  
   Quicksort exhibits good cache locality and this makes quicksort faster than merge sort (in many cases like in virtual memory environment).

There's no one algorithm that's clearly the "best" algorithm. It depends on a bunch of factors.

For starters, can you fit your data into main memory? If you can't, then you'd need to rely on an external sorting algorithm. These algorithms are often based on quicksort and mergesort.

Second, do you know anything about your input distribution? If it's mostly sorted, then something like Timsort might be a great option, since it's designed to work well on sorted data. If it's mostly random, Timsort is probably not a good choice.

Third, what kind of elements are you sorting? If you are sorting generic objects, then you're pretty much locked into comparison sorting. If not, perhaps you could use a non-comparison sort like counting sort or radix sort.

Fourth, how many cores do you have? Some sorting algorithms (quicksort, mergesort, MSD radix sort) parallelize really well, while others do not (heapsort).

Fifth, how are your data represented? If they're stored in an array, quicksort or a quicksort variant will likely do well because of locality of reference, while mergesort might be slow due to the extra memory needed. If they're in a linked list, though, the locality of reference from quicksort goes away and mergesort suddenly becomes competitive again.

The best option is probably to take a lot of different factors into account and then make a decision from there. One of the reason it's so fun to design and study algorithms is that there's rarely one single best choice; often, the best option depends a ton on your particular situation and changes based on what you're seeing.

(You mentioned a few details about quicksort, heapsort, and mergesort that I wanted to touch on before wrapping up this answer. While you're right that quicksort has a degenerate O(n2) worst case, there are many ways to avoid this. The introsort algorithm keeps track of the recursion depth and switches the algorithm to heapsort if it looks like the quicksort will degenerate. This guarantees O(n log n) worst-case behavior with low memory overhead and maximizes the amount of benefit you get from quicksort. Randomized quicksort, while still having an O(n2) worst case, has a vanishingly small probability of actually hitting that worst case.

Heapsort is a good algorithm in practice, but isn't as fast as the other algorithms in some cases because it doesn't have good locality of reference. That said, the fact that it never degenerates and needs only O(1) auxiliary space is a huge selling point.

Mergesort does need a lot of auxiliary memory, which is one reason why you might not want to use it if you have a huge amount of data to sort. It's worth knowing about, though, since its variants are widely used.)