Language/ Library	Method	In-place or Return a Copy	Argument to Switch Sort Order to Descending	Type of Sort	Other Arguments	Notes		Default Axis
Python (vanilla)	my_list.sort()	In-place	reverse = True	timsort	key	For lists only.		n/a
Python (vanilla)	my_list = sorted(my_iterable)	Copy	reverse = True	timsort	key	For any iterable.		n/a
Numpy	my_ndarray.sort()	In-place	n/a - do outside with [::-1]	Default is quicksort. Alternative options: *kind* : 'mergesort', 'heapsort', 'stable'.	*axis*=-1, *kind*='quicksort', *order*=None.	*Quicksort* is now introsort, which becomes heapsort if slow. *Stable* is mapped to mergesort. Mergesort doesn't use a mergesort. It uses timsort or radix sort under the hood, depening upon the data type.		last axis
Numpy	np.sort(my_array)	Copy	n/a - do outside with [::-1]	Default is quicksort Alternative options: *kind* : 'mergesort', 'heapsort', 'stable'.	*axis*=-1, *kind*='quicksort', *order*=None.	*Quicksort* is now introsort, which becomes heapsort if slow. *Stable* is mapped to mergesort. Mergesort doesn't use a mergesort. It uses timsort or radix sort under the hood, depening upon the data type.		last axis
Pandas	df = df.sort_values(by='my_column')	Copy (unless *inplace*=True)	ascending = False	Default is numpy quicksort. Alternative options: *kind* : 'mergesort', 'heapsort', 'stable'.	*by*, *axis*=0, *inplace*=False, *kind*='quicksort', *na_position*='last.	For DataFrame and Series. Uses Numpy for sorting under the hood.		last axis
TensorFlow	tf.sort(my_values)	Copy	direction = 'DESCENDING'	Finds the largest value in the tensor using *top_k()*. Uses CUB Cuda library, which wraps thrust, for parallel sort with GPU for large cases; algorithms vary - e.g. merge sort, radix sort.				last axis
PyTorch	torch.sort(my_values)	Copy	descending = True	Uses thrust CUDA library for parallel sort with GPU for large cases; algorithms vary - e.g. merge sort, radix sort.				last axis
SQL	ORDER BY my_column	Copy	DESC	Varies. Postgres uses a disk merge sort, quick sort, or heap sort depending upon the situation.				n/a

Sorting Algorithms		mostly from	http://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html https://johnderinger.wordpress.com/2012/12/28/quadratic-and-linearithmic-comparison-based-sorting-algorithms/										
				(BigO)									
	Name	When?	How?	Worst	Average	Best	Memory Required	Stable	InPlace	Recursive	Type	Notes	
1	Bubble	Never	Swap adjacent pairs. Bubble up.	n^2	n^2	n					Comparison	Don't use	
2	Insertion	Almost never		n^2	n^2	n^2		N	Y		Comparison	Insertion Sort - this one is actually pretty good on nearly sorted collections. If you have reliable apriori statistics that cofirm this to be the case, or if you are sorting a very small collection, insertion sort is the way to go. It's also commonly used to implement the base case for quicksort because it has very low overhead, it's stable and doesn't require aditional memory.	Insertion sort: this sort is useful if the data are already almost sorted, or if the list is very short (roughly, 50 items or less).
3	Selection	Almost never		n^2				N	Y		Comparison	Selection Sort - also a quadratic algorithm, performs poorly on any but the smallest collections. But unlike Insertion Sort which does fairly well on almost sorted collections, Selection Sort performs the same. It's only advantage is that it performs fewer swaps than Insertion Sort so if writes are expensive, it might be preferable.	
4	Heap	When concerned about worst case	nlogn version of selection sort leverages a heap data structure to improve time complexity. organize the data set into a heap data structure determine the largest (or smallest) element of the dataset place at the end (or beginning) of the set continue with the rest of the dataset root node is guaranteed to be the largest (or smallest) element. When root is removed and placed at the end of the datset, heap is rearranged so the largest element remaining moves to the root	n(log n)	n(log n)	n(log n)		N	Y	N	Binary	Many consider best sorting algorithm for guaranteed O(n log n) time complexity.	
5	Merge	When need stable	zip. comparison-based, stable, divide and conquer algorithm that requires additional memory, in fact each pass through the data doubles the size of the sorted subsections. The algorithm starts by comparing every two elements (i.e., 1 with 2, then 3 with 4...) and swapping them if the first should come after the second. It then merges each of the resulting datasets of two into sets of four, then merges those sets of four, and so on; until at last two sets are merged into the final sorted dataset.	n(log n)	n(log n)	n(log n)		Y				average slower than quicksort b/c copy collection	Merge Sort is the only guaranteed O(n log n) even in the worst case. The cost is that merge sort uses more memory.
6	Quick	For speed	merge with a pivot. all values smaller than the pivot are moved before it and all greater are moved after it. This is done in linear time O(n) and in-place. The lesser and greater subsets are then recursively sorted, with each recursion selecting a new pivot.	n^2	n(log n)	n(log n)		Can be, but slower	Y	Y		Mergesort with a pivot	
7	Tim	Python default	A stable, adaptive, hybrid sorting algorithm, derived from Merge sort and Insertion sort. The algorithm finds subsets of the data that are already ordered, and uses the subsets to sort the data more efficiently. This is done by merging an identified subset, called a run, with existing runs until certain criteria are fulfilled. Timsort was designed to take advantage of partial orderings that already exist in most real-world data.	n(log n)	n(log n)	n		Y			hybrid		
8	Shell		A shell sort improves on the insertion sort by sorting incremental sublists. It falls between O(n) and O(n2)										
9	Intro	combo											

A merge sort is O(nlogn), but requires additional space for the merging process.

A quick sort is O(nlogn), but may degrade to O(n2) if the split points are not near the middle of the list. It does not require additional space.

Bubble Sort

Bubble sort - compare one to another down the list and swap each one as needed. takes lots of passes. Short bubble is a modification to bubble so that it can stop early if everything is in place - don't think any other sort algos can do that. So even though n2, can be good with short lists with most items in order.

Selection sort walks the list to find the highest number and swaps the it with the largest position - or next largest. Repeat.

You may see that the selection sort makes the same number of comparisons as the bubble sort and is therefore also O(n2)O(n2). However, due to the reduction in the number of exchanges, the selection sort typically executes faster in benchmark studies. In fact, for our list, the bubble sort makes 20 exchanges, while the selection sort makes only 8.

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheSelectionSort.html

Insertion sort:

sort as you go through the list - make mini sorted lists. when the next item needs put into the sorted list you shift all the larger items to the right

n2

One note about shifting versus exchanging is also important. In general, a shift operation requires approximately a third of the processing work of an exchange since only one assignment is performed. In benchmark studies, insertion sort will show very good performance.

Shell Sort

gaps and then an insertion sort on last pass

Although this list is not completely sorted, something very interesting has happened. By sorting the sublists, we have moved the items closer to where they actually belong.

http://interactivepython.org/runestone/static/pythonds/SortSearch/TheShellSort.html

merge sort

Merge sort is a recursive algorithm that continually splits a list in half.

to analyze the mergeSort function, we need to consider the two distinct processes that make up its implementation. First, the list is split into halves. We already computed (in a binary search) that we can divide a list in half lognlogn times where n is the length of the list. The second process is the merge. Each item in the list will eventually be processed and placed on the sorted list. So the merge operation which results in a list of size n requires n operations. The result of this analysis is that lognlogn splits, each of which costs nn for a total of nlognnlogn operations. A merge sort is an O(nlogn)O(nlogn) algorithm.

Recall that the slicing operator is O(k)O(k) where k is the size of the slice. In order to guarantee that mergeSort will be O(nlogn)O(nlogn) we will need to remove the slice operator. Again, this is possible if we simply pass the starting and ending indices along with the list when we make the recursive call. We leave this as an exercise.

It is important to notice that the mergeSort function requires extra space to hold the two halves as they are extracted with the slicing operations. This additional space can be a critical factor if the list is large and can make this sort problematic when working on large data sets.

Quicksort

Like merge sort, quicksort uses divide-and-conquer, and so it's a recursive algorithm. The way that quicksort uses divide-and-conquer is a little different from how merge sort does. In merge sort, the divide step does hardly anything, and all the real work happens in the combine step. Quicksort is the opposite: all the real work happens in the divide step. In fact, the combine step in quicksort does absolutely nothing.

Quicksort has a couple of other differences from merge sort. Quicksort works in place. And its worst-case running time is as bad as selection sort's and insertion sort's: \Theta(n^2)Θ(n2). But its average-case running time is as good as merge sort's: \Theta(n \log_2 n)Θ(nlog2n). So why think about quicksort when merge sort is at least as good? That's because the constant factor hidden in the big-O notation for quicksort is quite good. In practice, quicksort outperforms merge sort, and it significantly outperforms selection sort and insertion sort.

Divide by choosing any element in the subarray array[p..r]. Call this element the pivot. Rearrange the elements in array[p..r] so that all elements in array[p..r] that are less than or equal to the pivot are to its left and all elements that are greater than the pivot are to its right. We call this procedure partitioning. At this point, it doesn't matter what order the elements to the left of the pivot are in relation to each other, and the same holds for the elements to the right of the pivot. We just care that each element is somewhere on the correct side of the pivot.

As a matter of practice, we'll always choose the rightmost element in the subarray, array[r], as the pivot. So, for example, if the subarray consists of [9, 7, 5, 11, 12, 2, 14, 3, 10, 6], then we choose 6 as the pivot. After partitioning, the subarray might look like [5, 2, 3, 6, 12, 7, 14, 9, 10, 11]. Let q be the index of where the pivot ends up.

Conquer by recursively sorting the subarrays array[p..q-1] (all elements to the left of the pivot, which must be less than or equal to the pivot) and array[q+1..r] (all elements to the right of the pivot, which must be greater than the pivot).

khan academy:

2048 max sort in place		rows	columns
4,096	110,000,000,000	1,100,000	100,000
2,048	50,000,000,000	500000	100000