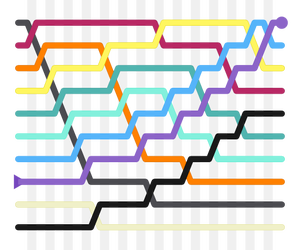
**Popular Sorting Algorithms - PYTHON**

[](https://i1.wp.com/python3.codes/wp-content/uploads/2015/08/Bubblesort-edited-color.svg_.png)

[Sorting](https://en.wikipedia.org/wiki/Sorting) is probably the most common type of [algorithm](https://en.wikipedia.org/wiki/Sorting_algorithm" \t "_blank)in programming. A sorting algorithm puts elements of a list in a certain order. The most-used orders are numerical, and alphabetical. It’s extremely useful in many applications, because sorted data can be searched or merged very quickly (e.g. by [binary search](http://python3.codes/binary-search/)). A sorted data set is one where every item is greater than its left neighbour (if any) and less than its right neighbour (if any), if it’s sorted into ascending order. In descending order swap ‘greater’ and ‘less’. You don’t normally need to write your own sorting routines in Python because they are already in the standard library. For example, list.sort() will sort a list in place, newlist=sorted(list) will return a sorted copy, the original list remains unchanged. However, an understanding of how they work and their performance characteristics is indispensible to any professional software developer.

The following is mostly adapted from [Wikipedia](https://en.wikipedia.org/wiki/Sorting_algorithm) and [Rosettacode](http://rosettacode.org/wiki/Sorting_algorithms" \t "_blank).

**Simple Sorts**

Three of the simplest sorts are bubble sort, insertion sort and selection sort, which are efficient on small data, due to low overhead, but not efficient on large data. Insertion sort is generally faster than selection sort in practice, due to fewer comparisons and good performance on almost-sorted data, and thus is preferred in practice, but selection sort uses fewer writes, and thus is used when write performance is a limiting factor.

[**Bubble Sort**](http://rosettacode.org/wiki/Sorting_algorithms/Bubble_sort)

[*Bubble sort*](https://en.wikipedia.org/wiki/Bubble_sort) is a very simple sorting algorithm. The algorithm starts at the beginning of the data set. It compares the first two elements, and if the first is greater than the second, it swaps them. It continues doing this for each pair of adjacent elements to the end of the data set. It then starts again with the first two elements, repeating until no swaps have occurred on the last pass. Over a number of passes, at most equal to the number of elements in the list, all of the values drift into their correct positions (large values “bubble” rapidly toward the end, pushing others down around them). Because each pass finds the maximum item and puts it at the end, the portion of the list to be sorted can be reduced at each pass. A boolean variable is used to track whether any changes have been made in the current pass; when a pass completes without changing anything, the algorithm exits.

This algorithm’s average and worst-case performance is O(*n*2), so it is rarely used to sort large, unordered data sets. Bubble sort can be used to sort a small number of items (where its asymptotic inefficiency is not a high penalty). Bubble sort can also be used efficiently on a list of any length that is nearly sorted (that is, the elements are not significantly out of place). For example, if any number of elements are out of place by only one position (e.g. 0123546789 and 1032547698), bubble sort’s exchange will get them in order on the first pass, the second pass will find all elements in order, so the sort will take only 2*n* time.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | def bubble\_sort(seq):      changed = True      while changed:          changed = False          for i in range(len(seq) - 1):              if seq[i] > seq[i+1]:                  seq[i], seq[i+1] = seq[i+1], seq[i]                  changed = True      return None |

[**Insertion sort**](http://rosettacode.org/wiki/Sorting_algorithms/Insertion_sort)

[*Insertion sort*](https://en.wikipedia.org/wiki/Insertion_sort) is an [O](http://rosettacode.org/wiki/O)(*n*2) sorting algorithm which moves elements one at a time into the correct position. The algorithm consists of inserting one element at a time into the previously sorted part of the array, moving higher ranked elements up as necessary. To start off, the first (or smallest, or any arbitrary) element of the unsorted array is considered to be the sorted part. It is relatively efficient for small lists and mostly sorted lists, and is often used as part of more sophisticated algorithms. In arrays, the new list and the remaining elements can share the array’s space, but insertion is expensive, requiring shifting all following elements over by one.

Although insertion sort is an [O](http://rosettacode.org/wiki/O)(*n*2) algorithm, its simplicity, low overhead, good locality of reference and efficiency make it a good choice in two cases:  
(i) small *n*,  
(ii) as the final finishing-off algorithm for [O](http://rosettacode.org/wiki/O)(*n* log*n*) algorithms such as [mergesort](http://rosettacode.org/wiki/Merge_sort" \o "Merge sort) and [quicksort](http://rosettacode.org/wiki/Quicksort).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | def insertion\_sort(l):      for i in range(1, len(l)):          j = i-1          key = l[i]          while (l[j] > key) and (j >= 0):             l[j+1] = l[j]             j -= 1          l[j+1] = key |

The insertion can be sped up by taking advantage of the already sorted part, and binary searching in it for the insertion point (or just after it if the new item is greater than any item in it).

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15 | def insertion\_sort\_bin(seq):      for i in range(1, len(seq)):          key = seq[i]          # invariant: ``seq[:i]`` is sorted          # find the least `low' such that ``seq[low]`` is not less then `key'.          #   Binary search in sorted sequence ``seq[low:up]``:          low, up = 0, i          while up > low:              middle = (low + up) // 2              if seq[middle] < key:                  low = middle + 1              else:                  up = middle          # insert key at position ``low``          seq[:] = seq[:low] + [key] + seq[low:i] + seq[i + 1:] |

Or, using bisect from the standard library:

|  |  |
| --- | --- |
| 1  2  3  4 | import bisect  def insertion\_sort\_bin(seq):      for i in range(1, len(seq)):          bisect.insort(seq, seq.pop(i), 0, i) |

[**Selection Sort**](http://rosettacode.org/wiki/Sorting_algorithms/Selection_sort#Python)

[*Selection sort*](https://en.wikipedia.org/wiki/Selection_sort) is an [in-place](https://en.wikipedia.org/wiki/In-place_algorithm) [comparison sort](https://en.wikipedia.org/wiki/Comparison_sort). It has [O](https://en.wikipedia.org/wiki/Big_O_notation)(*n*2) complexity, making it inefficient on large lists, and generally performs worse than the similar [insertion sort](https://en.wikipedia.org/wiki/Insertion_sort). Selection sort is noted for its simplicity, and also has performance advantages over more complicated algorithms in certain situations. First find the smallest element in the array and swap it with the element in the first position, then find the second smallest element and swap it with the element in the second position, and continue in this way until the entire array is sorted. Its asymptotic complexity is [O](http://rosettacode.org/wiki/O)(n2) making it inefficient on large arrays. Its primary purpose is for when writing data is very expensive (slow) when compared to reading, eg. writing to flash memory or EEPROM. No other sorting algorithm has less data movement. It does no more than *n* swaps, and thus is useful where swapping is very expensive.

|  |  |
| --- | --- |
| 1  2  3  4  5 | def selection\_sort(lst):      for i, e in enumerate(lst):          mn = min(range(i,len(lst)), key=lst.\_\_getitem\_\_)          lst[i], lst[mn] = lst[mn], e      return lst |

**Efficient Sorts**

Practical general sorting algorithms are almost always based on an algorithm with average complexity (and generally worst-case complexity) O(*n* log *n*), of which the most common are heap sort, merge sort, and quicksort. Each has advantages and drawbacks, with the most significant being that simple implementation of merge sort uses O(*n*) additional space, and simple implementation of quicksort has O(*n*2) worst-case complexity. These problems can be solved or ameliorated at the cost of a more complex algorithm.

While these algorithms are asymptotically efficient on random data, for practical efficiency on real-world data various modifications are used. First, the overhead of these algorithms becomes significant on smaller data, so often a hybrid algorithm is used, commonly switching to insertion sort once the data is small enough. Second, the algorithms often perform poorly on already sorted data or almost sorted data – these are common in real-world data, and can be sorted in O(*n*) time by appropriate algorithms. Finally, they may also be [unstable](https://en.wikipedia.org/wiki/Unstable_sort), and stability is often a desirable property in a sort. Thus more sophisticated algorithms are often employed, such as [Timsort](https://en.wikipedia.org/wiki/Timsort" \o "Timsort) (based on merge sort) or [introsort](https://en.wikipedia.org/wiki/Introsort" \o "Introsort) (based on quicksort, falling back to heap sort).

[**Merge Sort**](http://rosettacode.org/wiki/Sorting_algorithms/Merge_sort)

[*Merge sort*](https://en.wikipedia.org/wiki/Merge_sort) takes advantage of the ease of merging already sorted lists into a new sorted list. The basic idea is to split the collection into smaller groups by halving it until the groups only have one element or no elements (which are both entirely sorted groups). Then merge the groups back together so that their elements are in order. This is how the algorithm gets its “divide and conquer” description.

It is notable for having a worst case and average complexity of *O(n\*log(n))*, and a best case complexity of *O(n)* (for pre-sorted input).

Of the algorithms described here, this is the first that scales well to very large lists, because its worst-case running time is O(*n* log *n*). It is also easily applied to lists, not only arrays, as it only requires sequential access, not random access. However, it has additional O(*n*) space complexity, and involves a large number of copies in simple implementations.

Merge sort has seen a relatively recent surge in popularity for practical implementations, due to its use in the sophisticated algorithm [Timsort](https://en.wikipedia.org/wiki/Timsort" \o "Timsort), which is used for the standard sort routine in the programming languages [Python](https://en.wikipedia.org/wiki/Python_(programming_language)) and [Java](https://en.wikipedia.org/wiki/Java_(programming_language)) (as of [JDK7](https://en.wikipedia.org/wiki/JDK7)). Merge sort itself is the standard routine in [Perl](https://en.wikipedia.org/wiki/Perl), among others, and has been used in Java at least since 2000 in JDK1.3.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13 | from heapq import merge    def merge\_sort(m):      if len(m) <= 1:          return m        middle = len(m) // 2      left = m[:middle]      right = m[middle:]        left = merge\_sort(left)      right = merge\_sort(right)      return list(merge(left, right)) |

[**Heap Sort**](http://rosettacode.org/wiki/Sorting_algorithms/Heapsort)

[Heapsort](http://en.wikipedia.org/wiki/Heapsort) is an in-place sorting algorithm with worst case and average complexity of O(*n* log*n*).

The basic idea is to turn the array into a binary heap structure, which has the property that it allows efficient retrieval and removal of the maximal element. We repeatedly “remove” the maximal element from the heap, thus building the sorted list from back to front. Heapsort requires random access, so can only be used on an array-like data structure.

Heapsort is a much more efficient version of [selection sort](https://en.wikipedia.org/wiki/Selection_sort). It also works by determining the largest (or smallest) element of the list, placing that at the end (or beginning) of the list, then continuing with the rest of the list, but accomplishes this task efficiently by using a data structure called a [heap](https://en.wikipedia.org/wiki/Heap_(data_structure)), a special type of [binary tree](https://en.wikipedia.org/wiki/Binary_tree). Once the data list has been made into a heap, the root node is guaranteed to be the largest (or smallest) element. When it is removed and placed at the end of the list, the heap is rearranged so the largest element remaining moves to the root. Using the heap, finding the next largest element takes O(log *n*) time, instead of O(*n*) for a linear scan as in simple selection sort. This allows Heapsort to run in O(*n* log*n*) time, and this is also the worst case complexity.

[This code is from StackOverlow](http://stackoverflow.com/a/13980081):

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23 | def swap(i, j):      list[i], list[j] = list[j], list[i]    def heapify(end,i):      l=2 \* i + 1      r=2 \* (i + 1)      max=i      if l < end and list[i] < list[l]:          max = l      if r < end and list[max] < list[r]:          max = r      if max != i:          swap(i, max)          heapify(end, max)    def heap\_sort():      end = len(list)      start = end // 2 - 1      for i in range(start, -1, -1):          heapify(end, i)      for i in range(end-1, 0, -1):          swap(i, 0)          heapify(i, 0) |

[**Quicksort**](http://rosettacode.org/wiki/Sorting_algorithms/Quicksort)

*Quicksort* is a [divide and conquer](https://en.wikipedia.org/wiki/Divide_and_conquer_algorithm) [algorithm](https://en.wikipedia.org/wiki/Algorithm) which relies on a *partition* operation: to partition an array an element called a *pivot* is selected. All elements smaller than the pivot are moved before it and all greater elements are moved after it. This can be done efficiently in linear time and [in-place](https://en.wikipedia.org/wiki/In-place_algorithm). The lesser and greater sublists are then recursively sorted. This yields average time complexity of O(*n* log *n*), with low overhead, and thus this is a popular algorithm. Efficient implementations of quicksort (with in-place partitioning) are typically unstable sorts and somewhat complex, but are among the fastest sorting algorithms in practice. Together with its modest O(log *n*) space usage, quicksort is one of the most popular sorting algorithms and is available in many standard programming libraries.

The important caveat about quicksort is that its worst-case performance is O(*n*2); while this is rare, in naive implementations (choosing the first or last element as pivot) this occurs for sorted data, which is a common case. The most complex issue in quicksort is thus choosing a good pivot element, as consistently poor choices of pivots can result in drastically slower O(*n*2) performance, but good choice of pivots yields O(*n* log *n*) performance, which is asymptotically optimal. For example, if at each step the [median](https://en.wikipedia.org/wiki/Median) is chosen as the pivot then the algorithm works in O(*n* log *n*). Finding the median, such as by the [median of medians](https://en.wikipedia.org/wiki/Median_of_medians) [selection algorithm](https://en.wikipedia.org/wiki/Selection_algorithm) is however an O(*n*) operation on unsorted lists and therefore exacts significant overhead with sorting. In practice choosing a random pivot almost certainly yields O(*n* log *n*) performance.

Quicksort, also known as *partition-exchange sort*, uses these steps.

1. Choose any element of the array to be the pivot.
2. Divide all other elements (except the pivot) into two partitions.
   * All elements less than the pivot must be in the first partition.
   * All elements greater than the pivot must be in the second partition.
3. Use recursion to sort both partitions.
4. Join the first sorted partition, the pivot, and the second sorted partition.

The best pivot creates partitions of equal length (or lengths differing by 1). The worst pivot creates an empty partition (for example, if the pivot is the first or last element of a sorted array). The runtime of Quicksort ranges from [*O*](http://rosettacode.org/wiki/O)*(n*log*n)* with the best pivots, to [*O*](http://rosettacode.org/wiki/O)*(n2)* with the worst pivots, where *n* is the number of elements in the array.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | def quickSort(arr):      less = []      pivotList = []      more = []      if len(arr) <= 1:          return arr      else:          pivot = arr[0]          for i in arr:              if i < pivot:                  less.append(i)              elif i > pivot:                  more.append(i)              else:                  pivotList.append(i)          less = quickSort(less)          more = quickSort(more)          return less + pivotList + more |

[RosettaCode has several variants](http://rosettacode.org/wiki/Sorting_algorithms/Quicksort#Python).

**For Amusement Only**

A sorted array is a permutation of the unsorted array (or vice-versa). So if you shuffle the elements of the array, there is a chance that it may become sorted – especially if the array is quite small. This is called [Bogosort](https://en.wikipedia.org/wiki/Bogosort" \t "_blank), and is more suitable for programmer humour than actual use! It took from .5 seconds to 1 minute on my PC to sort only 10 elements.

“[Bogosort](http://rosettacode.org/wiki/Sorting_algorithms/Bogosort" \t "_blank)” is a perversely inefficient algorithm only used as an in-joke. Its average run-time is O(n!) because the chance that any given shuffle of a set will end up in sorted order is about one in *n* factorial, and the worst case is infinite since there’s no guarantee that a random shuffling will ever produce a sorted sequence. Its best case is O(n) since a single pass through the elements may suffice to order them.

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | import time  from random import shuffle    nitems=10    def inorder(numbs):      for i in range(len(numbs)-1):          if numbs[i] > numbs[i+1]: return False      return True    numbs=[i for i in range(nitems)]  shuffle(numbs)    start=time.clock()  while not inorder(numbs):      shuffle(numbs)  end = time.clock()  print(numbs, end-start, "seconds") |

**Further Information**

* [Sorting Algorithms Better Explained](http://betterexplained.com/articles/sorting-algorithms/)
* [Sorting Algorithm Animations](http://www.sorting-algorithms.com/)
* [15 Sorting Algorithms in 6 Minutes](https://www.youtube.com/watch?v=kPRA0W1kECg)
* [Comparison Sorting Algorithms](https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html)