**Buble Sort -** [**O**](http://rosettacode.org/wiki/O)**(*n*2) algorithm**

**Java**

**public** **static** <E **extends** Comparable<? **super** E>> **void** bubbleSort**(**E**[]** comparable**)** **{**  
 **boolean** changed = **false**;  
 **do** **{**  
 changed = **false**;  
 **for** **(int** a = **0**; a < comparable.**length** - **1**; a++**)** **{**  
 **if** **(**comparable**[**a**]**.**compareTo(**comparable**[**a + **1])** > **0)** **{**  
 E tmp = comparable**[**a**]**;  
 comparable**[**a**]** = comparable**[**a + **1]**;  
 comparable**[**a + **1]** = tmp;  
 changed = **true**;  
 **}**  
 **}**  
 **}** **while** **(**changed**)**;  
**}**

**Python**

**def** bubble\_sort**(**seq**)**:  
 """Inefficiently sort the mutable sequence (list) in place.  
 seq MUST BE A MUTABLE SEQUENCE.  
   
 As with list.sort() and random.shuffle this does NOT return   
 """  
 changed = **True**  
 **while** changed:  
 changed = **False**  
 **for** i **in** **xrange(len(**seq**)** - **1)**:  
 **if** seq**[**i**]** > seq**[**i+**1]**:  
 seq**[**i**]**, seq**[**i+**1]** = seq**[**i+**1]**, seq**[**i**]**  
 changed = **True**  
 **return** **None**  
   
**if** \_\_name\_\_ == "\_\_main\_\_":  
 """Sample usage and simple test suite"""  
   
 **from** **random** **import** shuffle  
   
 testset = **range(100)**  
 testcase = testset**[**:**]** *# make a copy*  
 shuffle**(**testcase**)**  
 **assert** testcase != testset *# we've shuffled it*  
 bubble\_sort**(**testcase**)**  
 **assert** testcase == testset *# we've unshuffled it back into a copy*

**Shell Sort -** [**O**](http://rosettacode.org/wiki/O)**(*n* log2*n*)  algorithm**

In this task, the goal is to sort an array of elements using the [Shell sort](http://en.wikipedia.org/wiki/Shell_sort) algorithm, a diminishing increment sort. The Shell sort is named after its inventor, Donald Shell, who published the algorithm in 1959. Shellsort is a sequence of interleaved insertion sorts based on an increment sequence. The increment size is reduced after each pass until the increment size is 1. With an increment size of 1, the sort is a basic insertion sort, but by this time the data is guaranteed to be almost sorted, which is insertion sort's "best case". Any sequence will sort the data as long as it ends in 1, but some work better than others. Empirical studies have shown a geometric increment sequence with a ratio of about 2.2 work well in practice.

**Java**

**public** **static** **void** shell**(int[]** a**)** **{**  
 **int** increment = a.**length** / **2**;  
 **while** **(**increment > **0)** **{**  
 **for** **(int** i = increment; i < a.**length**; i++**)** **{**  
 **int** j = i;  
 **int** temp = a**[**i**]**;  
 **while** **(**j >= increment && a**[**j - increment**]** > temp**)** **{**  
 a**[**j**]** = a**[**j - increment**]**;  
 j = j - increment;  
 **}**  
 a**[**j**]** = temp;  
 **}**  
 **if** **(**increment == **2)** **{**  
 increment = **1**;  
 **}** **else** **{**  
 increment \*= **(5.0** / **11)**;  
 **}**  
 **}**  
**}**

**Python**

**def** shell**(**seq**)**:  
 inc = **len(**seq**)** // **2**  
 **while** inc:  
 **for** i, el **in** **enumerate(**seq**)**:  
 **while** i >= inc **and** seq**[**i - inc**]** > el:  
 seq**[**i**]** = seq**[**i - inc**]**  
 i -= inc  
 seq**[**i**]** = el  
 inc = **1** **if** inc == **2** **else** **int(**inc \* **5.0** / **11)**  
   
data = **[22**, **7**, **2**, -**5**, **8**, **4]**  
shell**(**data**)**  
**print** data *# [-5, 2, 4, 7, 8, 22]*

**Insertion Sort -** [**O**](http://rosettacode.org/wiki/O)**(*n*2) algorithm**

**Java**

**public** **static** **void** insertSort**(int[]** A**){**  
 **for(int** i = **1**; i < A.**length**; i++**){**  
 **int** value = A**[**i**]**;  
 **int** j = i - **1**;  
 **while(**j >= **0** && A**[**j**]** > value**){**  
 A**[**j + **1]** = A**[**j**]**;  
 j = j - **1**;  
 **}**  
 A**[**j + **1]** = value;  
 **}**  
**}**

**Python**

**def** insertion\_sort**(**l**)**:  
 **for** i **in** **xrange(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

Counting Sort – O(n+k) algorithm

Java

**public** **static** **void** countingSort**(int[]** array, **int** min, **int** max**){**  
 **int[]** count= **new** **int[**max - min + **1]**;  
 **for(int** number : array**){**  
 count**[**number - min**]**++;  
 **}**  
 **int** z= **0**;  
 **for(int** i= min;i <= max;i++**){**  
 **while(**count**[**i - min**]** > **0){**  
 array**[**z**]**= i;  
 z++;  
 count**[**i - min**]**--;  
 **}**  
 **}**  
**}**

Python

>>> **from** **collections** **import** defaultdict  
>>> **def** countingSort**(array**, mn, mx**)**:  
 count = defaultdict**(int)**  
 **for** i **in** **array**:  
 count**[**i**]** += **1**  
 result = **[]**  
 **for** j **in** **range(**mn,mx+**1)**:  
 result += **[**j**]**\* count**[**j**]**  
 **return** result  
   
>>> data = **[9**, **7**, **10**, **2**, **9**, **7**, **4**, **3**, **10**, **2**, **7**, **10**, **2**, **1**, **3**, **8**, **7**, **3**, **9**, **5**, **8**, **5**, **1**, **6**, **3**, **7**, **5**, **4**, **6**, **9**, **9**, **6**, **6**, **10**, **2**, **4**, **5**, **2**, **8**, **2**, **2**, **5**, **2**, **9**, **3**, **3**, **5**, **7**, **8**, **4]**  
>>> mini,maxi = **1**,**10**  
>>> countingSort**(**data, mini, maxi**)** == **sorted(**data**)**  
**True**

Using a list:

**Works with**: [Python](http://rosettacode.org/wiki/Python) version 2.6

**def** countingSort**(**a, **min**, **max)**:  
 cnt = **[0]** \* **(max** - **min** + **1)**  
 **for** x **in** a:  
 cnt**[**x - **min]** += **1**  
   
 **return** **[**x **for** x, n **in** **enumerate(**cnt, start=**min)**  
 **for** i **in** **xrange(**n**)]**

Selection sort – O(N2) algorithm

In this task, the goal is to sort an [array](http://rosettacode.org/wiki/Array) (or list) of elements using the Selection sort algorithm. It works as follows:

First find the smallest element in the array and exchange it with the element in the first position, then find the second smallest element and exchange 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.

Java

**public** **static** **void** sort**(int[]** nums**){**  
 **for(int** currentPlace = **0**;currentPlace<nums.**length**-**1**;currentPlace++**){**  
 **int** smallest = [**Integer**](http://www.google.com/search?hl=en&q=allinurl%3Ainteger+java.sun.com&btnI=I%27m%20Feeling%20Lucky).**MAX\_VALUE**;  
 **int** smallestAt = currentPlace+**1**;  
 **for(int** check = currentPlace; check<nums.**length**;check++**){**  
 **if(**nums**[**check**]**<smallest**){**  
 smallestAt = check;  
 smallest = nums**[**check**]**;  
 **}**  
 **}**  
 **int** temp = nums**[**currentPlace**]**;  
 nums**[**currentPlace**]** = nums**[**smallestAt**]**;  
 nums**[**smallestAt**]** = temp;  
 **}**  
**}**

Python

**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

HeapSort – [**O**](http://rosettacode.org/wiki/O)**(*n* log*n*)**

[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.

Java

**public** **static** **void** heapSort**(int[]** a**){**  
 **int** count = a.**length**;  
   
 *//first place a in max-heap order*  
 heapify**(**a, count**)**;  
   
 **int** end = count - **1**;  
 **while(**end > **0){**  
 *//swap the root(maximum value) of the heap with the*  
 *//last element of the heap*  
 **int** tmp = a**[**end**]**;  
 a**[**end**]** = a**[0]**;  
 a**[0]** = tmp;  
 *//put the heap back in max-heap order*  
 siftDown**(**a, **0**, end - **1)**;  
 *//decrement the size of the heap so that the previous*  
 *//max value will stay in its proper place*  
 end--;  
 **}**  
**}**  
   
**public** **static** **void** heapify**(int[]** a, **int** count**){**  
 *//start is assigned the index in a of the last parent node*  
 **int** start = **(**count - **2)** / **2**; *//binary heap*  
   
 **while(**start >= **0){**  
 *//sift down the node at index start to the proper place*  
 *//such that all nodes below the start index are in heap*  
 *//order*  
 siftDown**(**a, start, count - **1)**;  
 start--;  
 **}**  
 *//after sifting down the root all nodes/elements are in heap order*  
**}**  
   
**public** **static** **void** siftDown**(int[]** a, **int** start, **int** end**){**  
 *//end represents the limit of how far down the heap to sift*  
 **int** root = start;  
   
 **while((**root \* **2** + **1)** <= end**){** *//While the root has at least one child*  
 **int** child = root \* **2** + **1**; *//root\*2+1 points to the left child*  
 *//if the child has a sibling and the child's value is less than its sibling's...*  
 **if(**child + **1** <= end && a**[**child**]** < a**[**child + **1])**  
 child = child + **1**; *//... then point to the right child instead*  
 **if(**a**[**root**]** < a**[**child**]){** *//out of max-heap order*  
 **int** tmp = a**[**root**]**;  
 a**[**root**]** = a**[**child**]**;  
 a**[**child**]** = tmp;  
 root = child; *//repeat to continue sifting down the child now*  
 **}else**  
 **return**;  
 **}**  
**}**

Python

**def** heapsort**(**lst**)**:  
 ''' Heapsort. Note: this function sorts in-place (it mutates the list). '''  
   
 *# in pseudo-code, heapify only called once, so inline it here*  
 **for** start **in** **range((len(**lst**)**-**2)**/**2**, -**1**, -**1)**:  
 siftdown**(**lst, start, **len(**lst**)**-**1)**  
   
 **for** end **in** **range(len(**lst**)**-**1**, **0**, -**1)**:  
 lst**[**end**]**, lst**[0]** = lst**[0]**, lst**[**end**]**  
 siftdown**(**lst, **0**, end - **1)**  
 **return** lst  
   
**def** siftdown**(**lst, start, end**)**:  
 root = start  
 **while** **True**:  
 child = root \* **2** + **1**  
 **if** child > end: **break**  
 **if** child + **1** <= end **and** lst**[**child**]** < lst**[**child + **1]**:  
 child += **1**  
 **if** lst**[**root**]** < lst**[**child**]**:  
 lst**[**root**]**, lst**[**child**]** = lst**[**child**]**, lst**[**root**]**  
 root = child  
 **else**:  
 **break**

Testing:

>>> ary = [7, 6, 5, 9, 8, 4, 3, 1, 2, 0]

>>> heapsort(ary)

[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

QuickSort - [**O**](http://rosettacode.org/wiki/O)**(*n* log*n*) algorithm**

The task is to sort an array (or list) elements using the *quicksort* algorithm. The elements must have a strict weak order and the index of the array can be of any discrete type. For languages where this is not possible, sort an array of integers.

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.

Java

**public** **static** <E **extends** [Comparable](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html)<? **super** E>> [List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> quickSort**(**[List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> arr**)** **{**  
**if** **(**!arr.**isEmpty())** **{**  
 E pivot = arr.**get(0)**; *//This pivot can change to get faster results*  
   
   
 [List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> less = **new** [LinkedList](http://java.sun.com/j2se/1.5.0/docs/api/java/util/LinkedList.html)<E>**()**;  
 [List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> pivotList = **new** [LinkedList](http://java.sun.com/j2se/1.5.0/docs/api/java/util/LinkedList.html)<E>**()**;  
 [List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> more = **new** [LinkedList](http://java.sun.com/j2se/1.5.0/docs/api/java/util/LinkedList.html)<E>**()**;  
   
 *// Partition*  
 **for** **(**E i: arr**)** **{**  
 **if** **(**i.**compareTo(**pivot**)** < **0)**  
 less.**add(**i**)**;  
 **else** **if** **(**i.**compareTo(**pivot**)** > **0)**  
 more.**add(**i**)**;  
 **else**  
 pivotList.**add(**i**)**;  
 **}**  
   
 *// Recursively sort sublists*  
 less = quickSort**(**less**)**;  
 more = quickSort**(**more**)**;  
   
 *// Concatenate results*  
 less.**addAll(**pivotList**)**;  
 less.**addAll(**more**)**;  
 **return** less;  
 **}**  
**return** arr;  
   
**}**

Python

**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  
   
a = **[4**, **65**, **2**, -**31**, **0**, **99**, **83**, **782**, **1]**  
a = quickSort**(**a**)**

In a Haskell fashion --

**def** qsort**(**L**)**:  
 **return** **(**qsort**([**y **for** y **in** L**[1**:**]** **if** y < L**[0]])** +   
 L**[**:**1]** +   
 qsort**([**y **for** y **in** L**[1**:**]** **if** y >= L**[0]]))** **if** **len(**L**)** > **1** **else** L

More readable, but still using list comprehensions:

**def** qsort**(list)**:  
 **if** **not** **list**:  
 **return** **[]**  
 **else**:  
 pivot = **list[0]**  
 less = **[**x **for** x **in** **list** **if** x < pivot**]**  
 more = **[**x **for** x **in** **list[1**:**]** **if** x >= pivot**]**  
 **return** qsort**(**less**)** + **[**pivot**]** + qsort**(**more**)**

More correctly in some tests:

**from** **random** **import** \*  
   
**def** qSort**(**a**)**:  
 **if** **len(**a**)** <= **1**:  
 **return** a  
 **else**:  
 q = choice**(**a**)**  
 **return** qSort**([**elem **for** elem **in** a **if** elem < q**])** + **[**q**]** \* a.**count(**q**)** + qSort**([**elem **for** elem **in** a **if** elem > q**])**

**def** quickSort**(**a**)**:  
 **if** **len(**a**)** <= **1**:  
 **return** a  
 **else**:  
 less = **[]**  
 more = **[]**  
 pivot = choice**(**a**)**  
 **for** i **in** a:  
 **if** i < pivot:  
 less.**append(**i**)**  
 **if** i > pivot:  
 more.**append(**i**)**  
 less = quickSort**(**less**)**  
 more = quickSort**(**more**)**  
 **return** less + **[**pivot**]** \* a.**count(**pivot**)** + more

Returning a new list:

**def** qsort**(array)**:  
 **if** **len(array)** < **2**:  
 **return** **array**  
 head, \*tail = **array**  
 less = qsort**([**i **for** i **in** tail **if** i < head**])**  
 more = qsort**([**i **for** i **in** tail **if** i >= head**])**  
 **return** less + **[**head**]** + more

Sorting a list in place:

**def** quicksort**(array)**:  
 \_quicksort**(array**, **0**, **len(array)** - **1)**  
   
**def** \_quicksort**(array**, start, stop**)**:  
 **if** stop - start > **0**:  
 pivot, left, right = **array[**start**]**, start, stop  
 **while** left <= right:  
 **while** **array[**left**]** < pivot:  
 left += **1**  
 **while** **array[**right**]** > pivot:  
 right -= **1**  
 **if** left <= right:  
 **array[**left**]**, **array[**right**]** = **array[**right**]**, **array[**left**]**  
 left += **1**  
 right -= **1**  
 \_quicksort**(array**, start, right**)**  
 \_quicksort**(array**, left, stop**)**

MergeSort - *O(n\*log(n)) algorithm*

The **merge sort** is a recursive sort of order n\*log(n).

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). 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.

*Java*

**import** ***java.util.List***;  
**import** ***java.util.ArrayList***;  
**import** ***java.util.Iterator***;  
   
**public** **class** Merge**{**  
 **public** **static** <E **extends** [Comparable](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html)<? **super** E>> [List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> mergeSort**(**[List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> m**){**  
 **if(**m.**size()** <= **1)** **return** m;  
   
 **int** middle = m.**size()** / **2**;  
 [List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> left = m.**subList(0**, middle**)**;  
 [List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> right = m.**subList(**middle, m.**size())**;  
   
 right = mergeSort**(**right**)**;  
 left = mergeSort**(**left**)**;  
 [List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> result = merge**(**left, right**)**;  
   
 **return** result;  
 **}**  
   
 **public** **static** <E **extends** [Comparable](http://java.sun.com/j2se/1.5.0/docs/api/java/lang/Comparable.html)<? **super** E>> [List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> merge**(**[List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> left, [List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> right**){**  
 [List](http://www.google.com/search?sitesearch=java.sun.com&q=allinurl%3Aj2se%2F1+5+0%2Fdocs%2Fapi+List)<E> result = **new** [ArrayList](http://java.sun.com/j2se/1.5.0/docs/api/java/util/ArrayList.html)<E>**()**;  
 [Iterator](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterator.html)<E> it1 = left.**iterator()**;  
 [Iterator](http://java.sun.com/j2se/1.5.0/docs/api/java/util/Iterator.html)<E> it2 = right.**iterator()**;  
   
 E x = it1.**next()**;  
 E y = it2.**next()**;  
 **while** **(true){**  
 *//change the direction of this comparison to change the direction of the sort*  
 **if(**x.**compareTo(**y**)** <= **0){**  
 result.**add(**x**)**;  
 **if(**it1.**hasNext()){**  
 x = it1.**next()**;  
 **}else{**  
 result.**add(**y**)**;  
 **while(**it2.**hasNext()){**  
 result.**add(**it2.**next())**;  
 **}**  
 **break**;  
 **}**  
 **}else{**  
 result.**add(**y**)**;  
 **if(**it2.**hasNext()){**  
 y = it2.**next()**;  
 **}else{**  
 result.**add(**x**)**;  
 **while** **(**it1.**hasNext()){**  
 result.**add(**it1.**next())**;  
 **}**  
 **break**;  
 **}**  
 **}**  
 **}**  
 **return** result;  
 **}**  
**}**

Python

**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**))**

Binary Search

Linear Search

Sequential Search