

**Algorithm**

[](http://en.wikipedia.org/wiki/File:Insertion-sort-example-300px.gif)

[http://bits.wikimedia.org/static-1.21wmf10/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Insertion-sort-example-300px.gif)

An example of insertion sort being done in-place. March up the array, checking each element. If larger (than what's in previous position checked), leave it (as happens with the 8). If smaller then march back down, shifting larger elements up until encounter a smaller element. Insert there.

Insertion sort iterates, consuming one input element each repetition, and growing a sorted output list. On a repetition, insertion sort removes one element from the input data, finds the location it belongs within the sorted list, and inserts it there. It repeats until no input elements remain.

Sorting is typically done in-place, by iterating up the array, growing the sorted list behind it. At each array-position, it checks the value there against the largest value in the sorted list (which happens to be next to it, in the previous array-position checked). If larger, it leaves the element in place and moves to the next. If smaller, it finds the correct position within the sorted list, shifts all the larger values up to make a space, and inserts into that correct position.

The resulting array after *k* iterations has the property where the first *k* + 1 entries are sorted ("+1" because the first entry is skipped). In each iteration the first remaining entry of the input is removed, and inserted into the result at the correct position, thus extending the result:

[Array prior to the insertion of x](http://en.wikipedia.org/wiki/File:Insertionsort-before.png)

becomes

[Array after the insertion of x](http://en.wikipedia.org/wiki/File:Insertionsort-after.png)

with each element greater than *x* copied to the right as it is compared against *x*.

The most common variant of insertion sort, which operates on arrays, can be described as follows:

1. Suppose there exists a function called *Insert* designed to insert a value into a sorted sequence at the beginning of an array. It operates by beginning at the end of the sequence and shifting each element one place to the right until a suitable position is found for the new element. The function has the side effect of overwriting the value stored immediately after the sorted sequence in the array.
2. To perform an insertion sort, begin at the left-most element of the array and invoke *Insert* to insert each element encountered into its correct position. The ordered sequence into which the element is inserted is stored at the beginning of the array in the set of indices already examined. Each insertion overwrites a single value: the value being inserted.

[Pseudocode](http://en.wikipedia.org/wiki/Pseudocode) of the complete algorithm follows, where the arrays are [zero-based](http://en.wikipedia.org/wiki/Zero-based_numbering):

for i ← 1 to i ← length(A)-1

{

//The values in A[ i ] are checked in-order, starting at the second one

// save A[i] to make a hole that will move as elements are shifted

// the value being checked will be inserted into the hole's final position

valueToInsert ← A[i]

holePos ← i

// keep moving the hole down until the value being checked is larger than

// what's just below the hole <!-- until A[holePos - 1] is <= item -->

while holePos > 0 and valueToInsert < A[holePos - 1]

{ //value to insert doesn't belong where the hole currently is, so shift

A[holePos] ← A[holePos - 1] //shift the larger value up

holePos ← holePos - 1 //move the hole position down

}

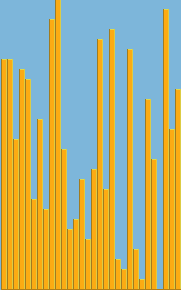
// hole is in the right position, so put value being checked into the hole

A[holePos] ← valueToInsert

}

Note that although the common practice is to implement in-place, which requires checking the elements in-order, the order of checking (and removing) input elements is actually arbitrary. The choice can be made using almost any pattern, as long as all input elements are eventually checked (and removed from the input).

**Best, worst, and average cases**

[](http://en.wikipedia.org/wiki/File:Insertion_sort.gif)

[http://bits.wikimedia.org/static-1.21wmf10/skins/common/images/magnify-clip.png](http://en.wikipedia.org/wiki/File:Insertion_sort.gif)

Animation of the insertion sort sorting a 30 element array.

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., [Θ](http://en.wikipedia.org/wiki/Big_Theta_notation)(*n*)). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

The simplest worst case input is an array sorted in reverse order. The set of all worst case inputs consists of all arrays where each element is the smallest or second-smallest of the elements before it. In these cases every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. This gives insertion sort a quadratic running time (i.e., O(*n*2)).

The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than [quicksort](http://en.wikipedia.org/wiki/Quicksort); indeed, good [quicksort](http://en.wikipedia.org/wiki/Quicksort) implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.

Example: The following table shows the steps for sorting the sequence {3, 7, 4, 9, 5, 2, 6, 1}. In each step, the item under consideration is underlined. The item that was moved (or left in place because it was biggest yet considered) in the previous step is shown in bold.

3 7 4 9 5 2 6 1

**3** 7 4 9 5 2 6 1

3 **7** 4 9 5 2 6 1

3 **4** 7 9 5 2 6 1

3 4 7 **9** 5 2 6 1

3 4 **5** 7 9 2 6 1

**2** 3 4 5 7 9 6 1

2 3 4 5 **6** 7 9 1

**1** 2 3 4 5 6 7 9

**Comparisons to other sorting algorithms**

Insertion sort is very similar to [selection sort](http://en.wikipedia.org/wiki/Selection_sort). As in selection sort, after *k* passes through the array, the first *k* elements are in sorted order. For selection sort these are the *k* smallest elements, while in insertion sort they are whatever the first *k* elements were in the unsorted array. Insertion sort's advantage is that it only scans as many elements as needed to determine the correct location of the *k*+1st element, while selection sort must scan all remaining elements to find the absolute smallest element.

Calculations show that insertion sort will usually perform about half as many comparisons as selection sort. Assuming the *k*+1st element's rank is random, insertion sort will on average require shifting half of the previous *k* elements, while selection sort always requires scanning all unplaced elements. If the input array is reverse-sorted, insertion sort performs as many comparisons as selection sort. If the input array is already sorted, insertion sort performs as few as *n*-1 comparisons, thus making insertion sort more efficient when given sorted or "nearly sorted" arrays.

While insertion sort typically makes fewer comparisons than [selection sort](http://en.wikipedia.org/wiki/Selection_sort), it requires more writes because the inner loop can require shifting large sections of the sorted portion of the array. In general, insertion sort will write to the array O(*n*2) times, whereas selection sort will write only O(*n*) times. For this reason selection sort may be preferable in cases where writing to memory is significantly more expensive than reading, such as with [EEPROM](http://en.wikipedia.org/wiki/EEPROM) or [flash memory](http://en.wikipedia.org/wiki/Flash_memory).

Some [divide-and-conquer algorithms](http://en.wikipedia.org/wiki/Divide-and-conquer_algorithm) such as [quicksort](http://en.wikipedia.org/wiki/Quicksort) and [mergesort](http://en.wikipedia.org/wiki/Mergesort) sort by recursively dividing the list into smaller sublists which are then sorted. A useful optimization in practice for these algorithms is to use insertion sort for sorting small sublists, where insertion sort outperforms these more complex algorithms. The size of list for which insertion sort has the advantage varies by environment and implementation, but is typically between eight and twenty elements.

### List insertion sort code in C

If the items are stored in a linked list, then the list can be sorted with O(1) additional space. The algorithm starts with an initially empty (and therefore trivially sorted) list. The input items are taken off the list one at a time, and then inserted in the proper place in the sorted list. When the input list is empty, the sorted list has the desired result.

The algorithm below uses a trailing pointer[[3]](http://en.wikipedia.org/wiki/Insertion_sort" \l "cite_note-3) for the insertion into the sorted list. A simpler recursive method rebuilds the list each time (rather than splicing) and can use O(*n*) stack space.

struct LIST

{

struct LIST \* pNext;

int iValue;

};

struct LIST \* SortList(struct LIST \* pList)

{

/\* build up the sorted array from the empty list \*/

struct LIST \* pSorted = NULL;

/\* take items off the input list one by one until empty \*/

while (pList != NULL)

{

/\* remember the head \*/

struct LIST \* pHead = pList;

/\* trailing pointer for efficient splice \*/

struct LIST \*\* ppTrail = &pSorted;

/\* pop head off list \*/

pList = pList->pNext;

/\* splice head into sorted list at proper place \*/

while (TRUE)

{

/\* does head belong here? \*/

if (\*ppTrail == NULL || pHead->iValue < (\*ppTrail)->iValue)

{

/\* yes \*/

pHead->pNext = \*ppTrail;

\*ppTrail = pHead;

break;

}

else

{

/\* no - continue down the list \*/

ppTrail = & (\*ppTrail)->pNext;

}

}

}

return pSorted;

}