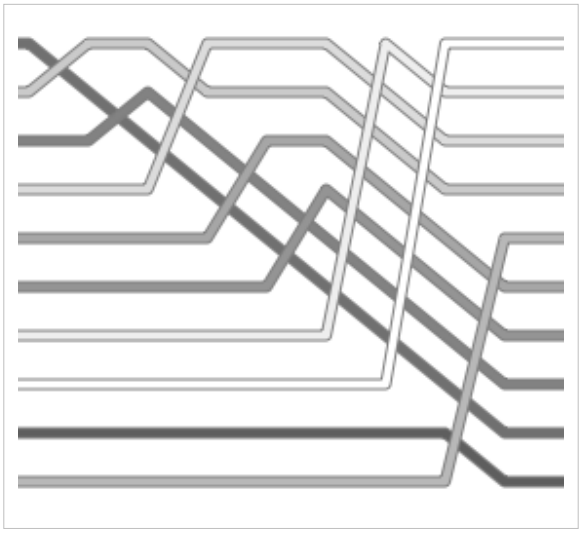
Insertion sort



Example of insertion sort sorting a list of random numbers.

Class	Sorting algorithm
Data structure	Array
Worst case performance	$O(n^2)$
Best case performance	$O(n)$
Average case performance	$O(n^2)$
Worst case space complexity	$O(n)$ total, $O(1)$ auxiliary

Insertion sort is a simple sorting algorithm: a comparison sort in which the sorted array (or list) is built one entry at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort provides several advantages:

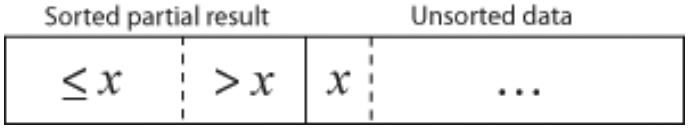
- Simple implementation
- Efficient for (quite) small data sets
- Adaptive, i.e. efficient for data sets that are already substantially sorted: the time complexity is $O(n + d)$, where d is the number of inversions
- More efficient in practice than most other simple quadratic, i.e. $O(n^2)$ algorithms such as selection sort or bubble sort; the best case (nearly sorted input) is $O(n)$
- Stable, i.e. does not change the relative order of elements with equal keys
- In-place, i.e. only requires a constant amount O(1) of additional memory space
- Online, i.e. can sort a list as it receives it

Most humans when sorting—ordering a deck of cards, for example—use a method that is similar to insertion sort.[1]

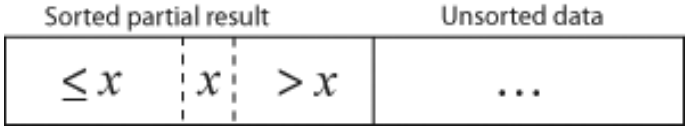
Algorithm

Every repetition of insertion sort removes an element from the input data, inserting it into the correct position in the already-sorted list, until no input elements remain. The choice of which element to remove from the input is arbitrary, and can be made using almost any choice algorithm.

Sorting is typically done in-place. The resulting array after k iterations has the property where the first $k + 1$ entries are sorted. In each iteration the first remaining entry of the input is removed, inserted into the result at the correct position, thus extending the result:



becomes



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 of the complete algorithm follows, where the arrays are **zero-based and the for-loop includes both the top and bottom limits** (as in Pascal):

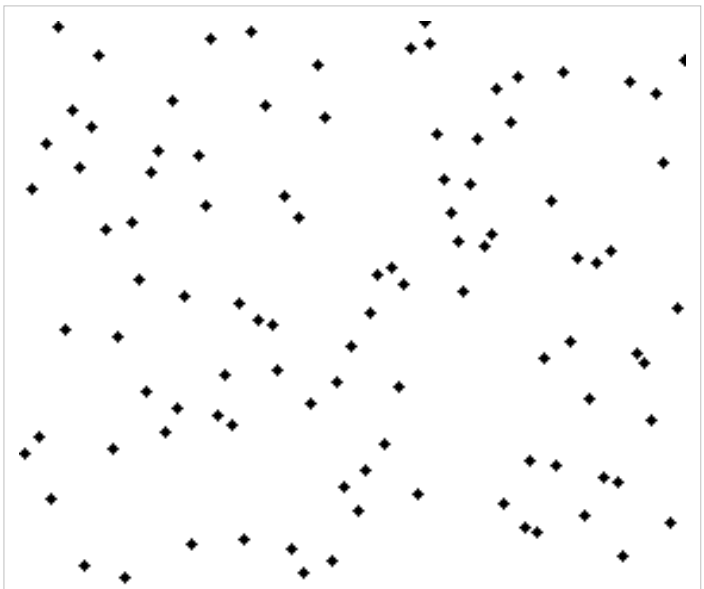
```
insertionSort(array A)

{ This procedure sorts in ascending order. }
begin
    for i := 1 to length[A]-1 do
    begin
        value := A[i];
        j := i - 1;
        done := false;
        repeat
            { To sort in descending order simply reverse
              the operator i.e. A[j] < value }
            if A[j] > value then
            begin
                A[j + 1] := A[j];
                j := j - 1;
                if j < 0 then
                    done := true;
            end
```

```
            else
                done := true;
        until done;
        A[j + 1] := value;
    end;
end;
```

Below is the pseudocode for insertion sort for a zero-based array (as in C):

```
1. for j ←1 to length(A)-1
2.     key ← A[ j ]
3.     > A[ j ] is added in the sorted sequence A[1, .. j-1]
4.     i ← j - 1
5.     while i >= 0 and A [ i ] > key
6.         A[ i +1 ] ← A[ i ]
7.         i ← i -1
8.     A [i +1] ← key
```

Best, worst, and average cases



Graphical example.

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., $\Theta(n)$). 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 worst case input is an array sorted in reverse order. In this case every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. For this case insertion sort has 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 quick sort; indeed, good quick sort 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 {5, 7, 0, 3, 4, 2, 6, 1}. For each iteration, the number of positions the inserted element has moved is shown in parentheses. Altogether this amounts to 17 steps.

5 7 0 3 4 2 6 1 (0)

5 7 *0* 3 4 2 6 1 (0)

0 5 7 *3* 4 2 6 1 (2)

0 **3** 5 7 *4* 2 6 1 (2)

0 3 **4** 5 7 *2* 6 1 (2)

0 **2** 3 4 5 7 *6* 1 (4)

0 2 3 4 5 **6** 7 *1* (1)

0 **1** 2 3 4 5 6 7 (6)

Comparisons to other sorting algorithms

Insertion sort is very similar to 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+1th 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+1th 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, 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 or flash memory.

Some divide-and-conquer algorithms such as quicksort and 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.

Variants

D.L. Shell made substantial improvements to the algorithm; the modified version is called Shell sort. The sorting algorithm compares elements separated by a distance that decreases on each pass. Shell sort has distinctly improved running times in practical work, with two simple variants requiring O($n^{3/2}$) and O($n^{4/3}$) running time.

If the cost of comparisons exceeds the cost of swaps, as is the case for example with string keys stored by reference or with human interaction (such as choosing one of a pair displayed side-by-side), then using *binary insertion sort* may yield better performance. Binary insertion sort employs a binary search to determine the correct location to insert new elements, and therefore performs $\lceil \log_2(n) \rceil$ comparisons in the worst case, which is $\Theta(n \log n)$. The algorithm as a whole still has a running time of $\Theta(n^2)$ on average because of the series of swaps required for each insertion.

The number of swaps can be reduced by calculating the position of multiple elements before moving them. For example, if the target position of two elements is calculated before they are moved into the right position, the number of swaps can be reduced by about 25% for random data. In the extreme case, this variant works similar to merge sort.

To avoid having to make a series of swaps for each insertion, the input could be stored in a linked list, which allows elements to be inserted and deleted in constant-time. However, performing a binary search on a linked list is impossible because a linked list does not support random access to its elements; therefore, the running time required for searching is O(n^2). If a more sophisticated data structure (e.g., heap or binary tree) is used, the time required for searching and insertion can be reduced significantly; this is the essence of heap sort and binary tree sort.

In 2004 Bender, Farach-Colton, and Mosteiro published a new variant of insertion sort called *library sort* or *gapped insertion sort* that leaves a small number of unused spaces (i.e., "gaps") spread throughout the array. The benefit is that insertions need only shift elements over until a gap is reached. The authors show that this sorting algorithm runs with high probability in O(n log n) time.[2]

References

[1] Robert Sedgewick, *Algorithms*, Addison-Wesley 1983 (chapter 8 p. 95)

[2] http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.9.3665

- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Second Edition. Addison-Wesley, 1998. ISBN 0-201-89685-0. Section 5.2.1: Sorting by Insertion, pp. 80–105.
- Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*, Second Edition. MIT Press and McGraw-Hill, 2001. ISBN 0-262-03293-7. Section 2.1: Insertion sort, pp. 15–21.
- Bender, Michael A.; Farach-Colton, Martín; Mosteiro, Miguel, *Insertion Sort is O(n log n)* (http://www.cs.sunysb.edu/~bender/pub/TOCS06-librarysort.pdf)

External links

- Binary Insertion Sort - Scoreboard (http://www.pathcom.com/~vadco/binary.html) – Complete Investigation and C Implementation – By JohnPaul Adamovsky
- Insertion Sort in C with demo (http://electrofriends.com/source-codes/software-programs/c/sorting-programs/program-to-sort-the-numbers-using-insertion-sort/) - Insertion Sort in C with demo
- Insertion Sort (http://corewar.co.uk/assembly/insertion.htm) - a comparison with other O(n^2) sorting algorithms
- Animated Sorting Algorithms: Insertion Sort (http://www.sorting-algorithms.com/insertion-sort) – graphical demonstration and discussion of insertion sort
- Category:Insertion Sort - LiteratePrograms (http://literateprograms.org/Category:Insertion_sort) – implementations of insertion sort in various programming languages
- InsertionSort (http://coderaptors.com/?InsertionSort) – colored, graphical Java applet that allows experimentation with the initial input and provides statistics
- Sorting Algorithms Demo (http://www.cs.ubc.ca/spider/harrison/Java/sorting-demo.html) – visual demonstrations of sorting algorithms (implemented in Java)
- Insertion sort illustrated explanation. Java and C++ implementations. (http://www.algolist.net/Algorithms/Sorting/Insertion_sort)

Article Sources and Contributors

Insertion sort *Source*: http://en.wikipedia.org/w/index.php?oldid=405035678 *Contributors*: 1qaz-pl, 24.252.226.xxx, Ahy1, Alansohn, Aleks80, Alex.atkins, Alex.mccarthy, Alexius08, Allan McInnes, Altenmann, Andre Engels, Andres, Apanag, Arthena, Arthur Rubin, Asdquefty, AxelBoldt, Baby123412, Baltar, Gaius, Barras, Baruneju, BiT, Billylikeswikis, Black Falcon, Blaisorblade, Bobo192, Booyabazooka, Brambleclawx, BrokenSegue, Buddhikaeport, CJLL Wright, Carey Evans, Ceros, Chet Gray, Colinb, Conversion script, Crashmatrix, Cyde, DFRussia, Damian Yerrick, Daniel Brockman, Daniel Quinlan, Dardasavta, David Eppstein, DavidGrayson, Dcoetzee, DevastatorIIC, Doradus, E.ruzi, Eequor, El C, Emdtechnology, Eric119, Faizbash, Fangyuan1st, Freakingtips, Fredrik, Frozenport, Garyzx, Giftlite, Glrx, Gmazeroff, Goutamrocks, GregorB, Groffles, H.ehsaan, HJ Mitchell, Hannes Hirzel, Hardmath, Hari, Harrisonmetz, HereToHelp, Hydrogen Iodide, I do not exist, Int19h, Ivan Pozdeev, Jarajapu, Jashar, Jesin, Jonas Kölker, Jordanbray, JosephMDecock, Josh Kehn, Jpmelos, KBKarma, Kangaroosrule, Karl Dickman, Keynell, Killiondude, Kiwi137, Klrste, Knuckles, Knutux, Kostmo, Kpjas, Kragen, Kri, LOL, Lawlzlawlz, Lidden, Looxix, Magicbronson, Merendoglu, Mess, Michael Hardy, Michael Slone, Mike1242, Mike1341, Minchenko Michael, Mollmerx, MorganGreen, Nasradu8, Neel basu, Nillerdk, Ninjakannon, Nixdorf, Nmnogueira, Nordald, Nuggetboy, Nuxnut, Ohnoitsjamie, Oli Filth, Oskar Sigvardsson, Oxaric, Oğuz Ergin, Pcp071098, Pdvyas, Pharaoh of the Wizards, Phil Boswell, PhilippWeissenbacher, Piet Delport, Pinball22, Pion, Pipedreambomb, Player 03, Ponggr, Pred, Pulveriser, Quentonamos, Qz, Ratheesh nan, Reidhoch, Rhanekom, Rlneumiller, Ruud Koot, SLi, Sapeur, Sfan00 IMG, SheldonYoung, Silly rabbit, Simetrical, SiobhanHansa, Sligocki, Smalljim, Smjg, Steven Zhang, Svick, Swift, Tamer ih, Ted Longstaffe, ThomasMueller, Tiddly Tom, Timwi, Tjwood, Tryptophan4, VTBassMatt, Vasil', Virtualblackfox, Vpshastry, Werdna, WookieInHeat, Ww, Ww2censor, XP1, XreDuex, Yandman, Zaradaqaw, ZeroOne, Zhou Yu, Zowayix, Zvar, 326 anonymous edits

Image Sources, Licenses and Contributors

File:Insertionsort-edited.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Insertionsort-edited.png *License*: Public Domain *Contributors*: User:Crashmatrix

Image:insertionsort-before.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Insertionsort-before.png *License*: Public Domain *Contributors*: Original uploader was Dcoetzee at en.wikipedia

Image:insertionsort-after.png *Source*: http://en.wikipedia.org/w/index.php?title=File:Insertionsort-after.png *License*: Public Domain *Contributors*: Original uploader was Dcoetzee at en.wikipedia

Image:Insertion sort animation.gif *Source*: http://en.wikipedia.org/w/index.php?title=File:Insertion_sort_animation.gif *License*: Creative Commons Attribution-Sharealike 2.5 *Contributors*: Original uploader was Nmnogueira at en.wikipedia

License