



WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here
Related changes
Upload file
Special pages
Permanent link
Page information
Wikidata item
Cite this page

Print/export
Create a book
Download as PDF
Printable version

Languages
العربية
Български
Čeština
Dansk
Deutsch
Eesti
Ελληνικά
Español
فارسی
Français
한국어
Հայերեն
Italiano
עברית
Қазақша
Lietuvių
Nederlands
日本語
Polski
Português
Русский
Slovenčina
Slovenščina
Српски / srpski
Suomi
Svenska
Tagalog
தமிழ்
Türkçe

Create account Log in

Article [Talk](#)

[Read](#) [Edit](#) [View history](#)

Selection sort

From Wikipedia, the free encyclopedia

In [computer science](#), **selection sort** is a [sorting algorithm](#), specifically an [in-place comparison sort](#). It has $O(n^2)$ time complexity, making it inefficient on large lists, and generally performs worse than the similar [insertion sort](#). Selection sort is noted for its simplicity, and it has performance advantages over more complicated algorithms in certain situations, particularly where auxiliary memory is limited.

The algorithm divides the input list into two parts: the sublist of items already sorted, which is built up from left to right at the front (left) of the list, and the sublist of items remaining to be sorted that occupy the rest of the list. Initially, the sorted sublist is empty and the unsorted sublist is the entire input list. The algorithm proceeds by finding the smallest (or largest, depending on sorting order) element in the unsorted sublist, exchanging (swapping) it with the leftmost unsorted element (putting it in sorted order), and moving the sublist boundaries one element to the right.

Contents [\[hide\]](#)

1

Example

2

Implementation

3

Analysis

4

Comparison to other sorting algorithms

5

Variants

6

See also

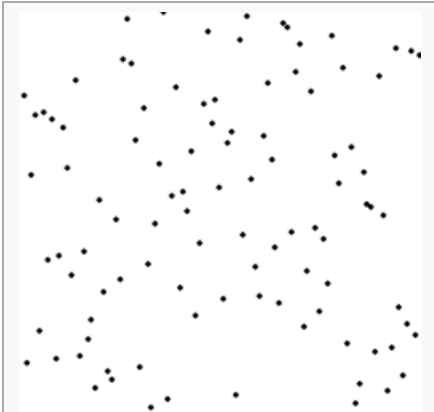
7

References

8

External links

Selection sort



Selection sort animation

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

Example [\[edit\]](#)

Here is an example of this sort algorithm sorting five elements:

```
64 25 12 22 11 // this is the initial, starting state of the array
11 25 12 22 64 // sorted sublist = {11}
11 12 25 22 64 // sorted sublist = {11, 12}
11 12 22 25 64 // sorted sublist = {11, 12, 22}
11 12 22 25 64 // sorted sublist = {11, 12, 22, 25}
11 12 22 25 64 // sorted sublist = {11, 12, 22, 25, 64}
```

(Nothing appears changed on these last two lines because the last two numbers were already in order)

Selection sort can also be used on list structures that make add and remove efficient, such as a [linked list](#). In this case it is more common to *remove* the minimum element from the remainder of the list, and then *insert* it at the end of the values sorted so far. For example:

```
64 25 12 22 11
11 64 25 12 22
11 12 64 25 22
11 12 22 64 25
11 12 22 25 64
```

Implementation [\[edit\]](#)


```
/* a[0] to a[n-1] is the array to sort */
int i,j;
int iMin;

/* advance the position through the entire array */
/* (could do j < n-1 because single element is also min element)
*/
for (j = 0; j < n-1; j++) {
    /* find the min element in the unsorted a[j .. n-1] */

    /* assume the min is the first element */
    iMin = j;
    /* test against elements after j to find the smallest */
    for ( i = j+1; i < n; i++) {
        /* if this element is less, then it is the new minimum */
        if (a[i] < a[iMin]) {
            /* found new minimum; remember its index */
            iMin = i;
        }
    }

    if(iMin != j) {
        swap(a[j], a[iMin]);
    }
}
```

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

Selection sort 
animation. Red
is current min.
Yellow is sorted
list. Blue is
current item.

Analysis [\[edit\]](#)

Selection sort is not difficult to analyze compared to other sorting algorithms since none of the loops depend on the data in the array. Selecting the lowest element requires scanning all n elements (this takes $n - 1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n - 1$ elements and so on, for $(n - 1) + (n - 2) + \dots + 2 + 1 = n(n - 1) / 2 \in \Theta(n^2)$ comparisons (see [arithmetic progression](#)). Each of these scans requires one swap for $n - 1$ elements (the final element is already in place).

Comparison to other sorting algorithms [\[edit\]](#)

Among simple average-case $\Theta(n^2)$ algorithms, selection sort almost always outperforms [bubble sort](#) and [gnome sort](#). [Insertion sort](#) is very similar in that after the k th iteration, the first k elements in the array are in sorted order. Insertion sort's advantage is that it only scans as many elements as it needs in order to place the $k + 1$ st element, while selection sort must scan all remaining elements to find the $k + 1$ st element.

Simple calculation shows that insertion sort will therefore usually perform about half as many comparisons as selection sort, although it can perform just as many or far fewer depending on the order the array was in prior to sorting. It can be seen as an advantage for some [real-time](#) applications that selection sort will perform identically regardless of the order of the array, while insertion sort's running time can vary considerably. However, this is more often an advantage for insertion sort in that it runs much more efficiently if the array is already sorted or "close to sorted."

While selection sort is preferable to insertion sort in terms of number of writes ($\Theta(n)$ swaps versus $O(n^2)$ swaps), it almost always far exceeds (and never beats) the number of writes that [cycle sort](#) makes, as cycle sort

is theoretically optimal in the number of writes. This can be important if writes are significantly more expensive than reads, such as with [EEPROM](#) or [Flash](#) memory, where every write lessens the lifespan of the memory.

Finally, selection sort is greatly outperformed on larger arrays by $\Theta(n \log n)$ [divide-and-conquer algorithms](#) such as [mergesort](#). However, insertion sort or selection sort are both typically faster for small arrays (i.e. fewer than 10–20 elements). A useful optimization in practice for the recursive algorithms is to switch to insertion sort or selection sort for "small enough" sublists.

Variants [\[edit\]](#)

[Heapsort](#) greatly improves the basic algorithm by using an [implicit heap data structure](#) to speed up finding and removing the lowest datum. If implemented correctly, the heap will allow finding the next lowest element in $\Theta(\log n)$ time instead of $\Theta(n)$ for the inner loop in normal selection sort, reducing the total running time to $\Theta(n \log n)$.

A bidirectional variant of selection sort, called **cocktail sort**, is an algorithm which finds both the minimum and maximum values in the list in every pass. This reduces the number of scans of the list by a factor of 2, eliminating some loop overhead but not actually decreasing the number of comparisons or swaps. Note, however, that [cocktail sort](#) more often refers to a bidirectional variant of bubble sort.

Selection sort can be implemented as a [stable sort](#). If, rather than swapping in step 2, the minimum value is inserted into the first position (that is, all intervening items moved down), the algorithm is stable. However, this modification either requires a data structure that supports efficient insertions or deletions, such as a linked list, or it leads to performing $\Theta(n^2)$ writes.

In the **bingo sort** variant, items are ordered by repeatedly looking through the remaining items to find the greatest value and moving all items with that value to their final location.^[1] Like [counting sort](#), this is an efficient variant if there are many duplicate values. Indeed, selection sort does one pass through the remaining items for each item moved. Bingo sort does one pass for each value (not item): after an initial pass to find the biggest value, the next passes can move every item with that value to its final location while finding the next value as in the following [pseudocode](#) (arrays are **zero-based** and the **for-loop** includes both the **top** and **bottom limits**, as in [Pascal](#)):

```
bingo(array A)

{ This procedure sorts in ascending order. }
begin
    max := length(A)-1;

    { The first iteration is written to look very similar to the subsequent ones,
    but
    without swaps. }
    nextValue := A[max];
    for i := max - 1 downto 0 do
        if A[i] > nextValue then
            nextValue := A[i];
    while (max > 0) and (A[max] = nextValue) do
        max := max - 1;

    while max > 0 do begin
        value := nextValue;
        nextValue := A[max];
        for i := max - 1 downto 0 do
            if A[i] = value then begin
                swap(A[i], A[max]);
                max := max - 1;
            end else if A[i] > nextValue then
                nextValue := A[i];
        while (max > 0) and (A[max] = nextValue) do
            max := max - 1;
        end;
    end;
```

Thus, if on average there are more than two items with the same value, bingo sort can be expected to be faster because it executes the inner loop fewer times than selection sort.

See also [\[edit\]](#)

- [Selection algorithm](#)

References [edit]

1. [^] Black, Paul E. "Bingo sort". *Dictionary of Algorithms and Data Structures*. NIST.

- Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison–Wesley, 1997. ISBN 0-201-89685-0. Pages 138–141 of Section 5.2.3: Sorting by Selection.
- Anany Levitin. *Introduction to the Design & Analysis of Algorithms*, 2nd Edition. ISBN 0-321-35828-7. Section 3.1: Selection Sort, pp 98–100.
- Robert Sedgewick. *Algorithms in C++, Parts 1–4: Fundamentals, Data Structure, Sorting, Searching: Fundamentals, Data Structures, Sorting, Searching Pts. 1–4*, Second Edition. Addison–Wesley Longman, 1998. ISBN 0-201-35088-2. Pages 273–274

External links [edit]

- [Animated Sorting Algorithms: Selection Sort](#) – graphical demonstration and discussion of selection sort

The Wikibook *Algorithm implementation* has a page on the topic of: **Selection sort**

v · t · e	Sorting algorithms	[hide]
Theory	Computational complexity theory · Big O notation · Total order · Lists · Inplacement · Stability · Comparison sort · Adaptive sort · Sorting network · Integer sorting	
Exchange sorts	Bubble sort · Cocktail sort · Odd–even sort · Comb sort · Gnome sort · Quicksort · Stooge sort · Bogosort	
Selection sorts	Selection sort · Heapsort · Smoothsort · Cartesian tree sort · Tournament sort · Cycle sort	
Insertion sorts	Insertion sort · Shellsort · Splaysort · Tree sort · Library sort · Patience sorting	
Merge sorts	Merge sort · Cascade merge sort · Oscillating merge sort · Polyphase merge sort · Strand sort	
Distribution sorts	American flag sort · Bead sort · Bucket sort · Burstsrt · Counting sort · Pigeonhole sort · Proxmap sort · Radix sort · Flashsort	
Concurrent sorts	Bitonic sorter · Batcher odd–even mergesort · Pairwise sorting network	
Hybrid sorts	Block sort · Timsort · Introsort · Spreadsort · JSort	
Other	Topological sorting · Pancake sorting · Spaghetti sort	

Categories: [Sorting algorithms](#) | [Comparison sorts](#)