WIKIPEDIA

Selection sort

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.

Selection sort

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

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. Uniqueness of selection sort when compared to other sorting techniques:The time efficiency of selection sort is quadratic, so there exists a number of sorting techniques which have better time complexity than Selection Sort. Even then, considering the number of swaps made, the number of swaps will be n-1 both in worst as well as best case. That is, time efficiency of selection sort with respect to swaps is linear. This property distinguishes selection sort positively from many other sorting algorithms.

Contents

Example

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

Sorted sublist	Unsorted sublist	Least element in unsorted list
( )	(11, 25, 12, 22, 64)	11
(11)	(25, 12, 22, 64)	12
(11, 12)	(25, 22, 64)	22
(11, 12, 22)	(25, 64)	25
(11, 12, 22, 25)	(64)	64
(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:

```
arr[] = 64 25 12 22 11

// Find the minimum element in arr[0...4]
// and place it at beginning
11 25 12 22 64

// Find the minimum element in arr[1...4]
// and place it at beginning of arr[1...4]
11 12 25 22 64

// Find the minimum element in arr[2...4]
// and place it at beginning of arr[2...4]
11 12 22 25 64

// Find the minimum element in arr[3...4]
// and place it at beginning of arr[3...4]
11 12 22 25 64
```

Implementations

Below is an implementation in C. More implementations can be found on the talk page of this Wikipedia article.

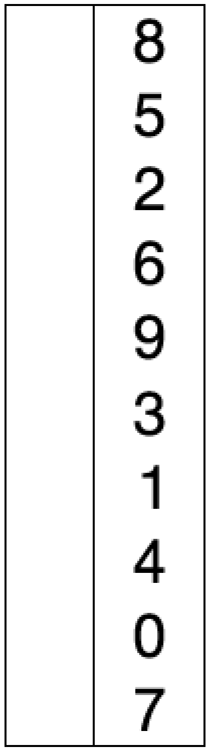
```
/* a[0] to a[aLength-1] is the array to sort */
int i,j;
int aLength; // initialise to a's length

```

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

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

    if (jMin != i)
    {
        swap(a[i], a[jMin]);
    }
}
```



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

Complexity

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 minimum requires scanning n elements (taking $n-1$ comparisons) and then swapping it into the first position. Finding the next lowest element requires scanning the remaining $n-1$ elements and so on. Therefore, the total number of comparisons is

$$(n-1)+(n-2)+\ldots+1=\sum_{i=1}^{n-1} i$$

By arithmetic progression,

$$\sum_{i=1}^{n-1} i=\frac{(n-1)+1}{2}(n-1)=\frac{1}{2}n(n-1)=\frac{1}{2}(n^2-n)$$

which is of complexity $O(n^2)$ in terms of number of comparisons. Each of these scans requires one swap for $n-1$ elements (the final element is already in place).

Comparison to other sorting algorithms

Among quadratic sorting algorithms (sorting algorithms with a simple average-case of $\Theta(n^2)$), selection sort almost always outperforms bubble sort and gnome sort. Insertion sort is very similar in that after the kth 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

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. Sometimes this is **double selection 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

- Selection algorithm

References

1. This article incorporates public domain material from the NIST document: Black, Paul E. "Bingo sort" (https://xlinux.nist.gov/dads/HTML/bingosort.html). *Dictionary of Algorithms and Data Structures*.

- 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

- Animated Sorting Algorithms: Selection Sort (https://web.archive.org/web/20150307110315/http://www.sorting-algorithms.com/selection-sort) at the Wayback Machine (archived 7 March 2015) – graphical demonstration

Retrieved from "https://en.wikipedia.org/w/index.php?title=Selection_sort&oldid=902219077"