



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
Català
Español
فارسی
Français
Bahasa Indonesia
Magyar
日本語
Русский
Українська  Edit links

Create account Log in

Article Talk

Read Edit

More ▾

Search 

Selection algorithm

From Wikipedia, the free encyclopedia

For simulated natural selection in genetic algorithms, see [Selection \(genetic algorithm\)](#).

In [computer science](#), a **selection algorithm** is an [algorithm](#) for finding the *k*th smallest number in a [list](#) or [array](#); such a number is called the *k*th [order statistic](#). This includes the cases of finding the [minimum](#), [maximum](#), and [median](#) elements. There are $O(n)$ (worst-case linear time) selection algorithms, and sublinear performance is possible for structured data; in the extreme, $O(1)$ for an array of sorted data. Selection is a subproblem of more complex problems like the [nearest neighbor](#) and [shortest path](#) problems. Many selection algorithms are derived by generalizing a [sorting algorithm](#), and conversely some sorting algorithms can be derived as repeated application of selection.

The simplest case of a selection algorithm is finding the minimum (or maximum) element by iterating through the list, keeping track of the running minimum – the minimum so far – (or maximum) and can be seen as related to the [selection sort](#). Conversely, the hardest case of a selection algorithm is finding the median, and this necessarily takes $n/2$ storage. In fact, a specialized median-selection algorithm can be used to build a general selection algorithm, as in [median of medians](#). The best-known selection algorithm is [quickselect](#), which is related to [quicksort](#); like quicksort, it has (asymptotically) optimal average performance, but poor worst-case performance, though it can be modified to give optimal worst-case performance as well.

Contents [\[hide\]](#)

- Selection by sorting
 - Unordered partial sorting
 - Partial selection sort
- Partition-based selection
 - Median selection as pivot strategy
- Incremental sorting by selection
- Using data structures to select in sublinear time
- Lower bounds
- Space complexity
- Online selection algorithm
- Related problems
- Language support
- See also
- References
- External links

Selection by sorting [\[edit\]](#)

By sorting the list or array then selecting the desired element, selection can be [reduced](#) to [sorting](#). This method is inefficient for selecting a single element, but is efficient when many selections need to be made from an array, in which case only one initial, expensive sort is needed, followed by many cheap selection operations – $O(1)$ for an array, though selection is $O(n)$ in a list, even if sorted, due to lack of [random access](#). In general, sorting requires $O(n \log n)$ time, where n is the length of the list, although a lower bound is possible with non-comparative sorting algorithms like [radix sort](#) and [counting sort](#).

Rather than sorting the whole list or array, one can instead use [partial sorting](#) to select the *k* smallest or *k* largest elements. The *k*th smallest (resp., *k*th largest element) is then the largest (resp., smallest element) of the partially sorted list – this then takes $O(1)$ to access in an array and $O(k)$ to access in a list. This is more efficient than full sorting, but less efficient than simply selecting, and takes $O(n + k \log k)$ time, due to the sorting of the *k* elements. Partial sorting algorithms can often be derived from (total) sorting algorithms. As with total sorting, partial sorting means that further selections (below the *k*th element) can be done in $O(1)$ time for an array and $O(k)$ time for a list. Further, if the partial sorting also partitions the original data into "sorted" and "unsorted", as with an in-place sort, the partial sort can be extended to a larger partial sort by only sorting the incremental portion, and if this is done, further selections above the *k*th element can also be done relatively cheaply.

Unordered partial sorting [\[edit\]](#)

If partial sorting is relaxed so that the k smallest elements are returned, but not in order, the factor of $O(k \log k)$ can be eliminated. An additional maximum selection (taking $O(k)$ time) is required, but since $k \leq n$, this still yields asymptotic complexity of $O(n)$. In fact, partition-based selection algorithms yield both the k th smallest element itself and the k smallest elements (with other elements not in order). This can be done in $O(n)$ time – average complexity of [quickselect](#), and worst-case complexity of refined partition-based selection algorithms.

Conversely, given a selection algorithm, one can easily get an unordered partial sort (k smallest elements, not in order) in $O(n)$ time by iterating through the list and recording all elements less than the k th element. If this results in fewer than $k - 1$ elements, any remaining elements equal the k th element. Care must be taken, due to the possibility of equality of elements: one must not include all elements less than *or equal to* the k th element, as elements greater than the k th element may also be equal to it.

Thus unordered partial sorting (lowest k elements, but not ordered) and selection of the k th element are very similar problems. Not only do they have the same asymptotic complexity, $O(n)$, but a solution to either one can be converted into a solution to the other by a straightforward algorithm (finding a max of k elements, or filtering elements of a list below a cutoff of the value of the k th element).

Partial selection sort [\[edit\]](#)

A simple example of selection by partial sorting is to use the partial [selection sort](#).

The obvious linear time algorithm to find the minimum (resp. maximum) – iterating over the list and keeping track of the minimum (resp. maximum) element so far – can be seen as a partial selection sort that selects the 1 smallest element. However, many other partial sorts also reduce to this algorithm for the case $k = 1$, such as a partial heap sort.

More generally, a partial selection sort yields a simple selection algorithm which takes $O(kn)$ time. This is asymptotically inefficient, but can be sufficiently efficient if k is small, and is easy to implement. Concretely, we simply find the minimum value and move it to the beginning, repeating on the remaining list until we have accumulated k elements, and then return the k th element. Here is partial selection sort-based algorithm:

```
function select(list[1..n], k)
    for i from 1 to k
        minIndex = i
        minValue = list[i]
        for j from i+1 to n
            if list[j] < minValue
                minIndex = j
                minValue = list[j]
        swap list[i] and list[minIndex]
    return list[k]
```

Partition-based selection [\[edit\]](#)

Further information: [Quickselect](#)

Linear performance can be achieved by a partition-based selection algorithm, most basically [quickselect](#). Quickselect is a variant of [quicksort](#) – in both one chooses a pivot and then partitions the data by it, but while Quicksort recurses on both sides of the partition, Quickselect only recurses on one side, namely the side on which the desired k th element is. As with Quicksort, this has optimal average performance, in this case linear, but poor worst-case performance, in this case quadratic. This occurs for instance by taking the first element as the pivot and searching for the maximum element, if the data is already sorted. In practice this can be avoided by choosing a random element as pivot, which yields [almost certain](#) linear performance. Alternatively, a more careful deterministic pivot strategy can be used, such as [median of medians](#). These are combined in the hybrid [introslect](#) algorithm (analogous to [introsort](#)), which starts with Quickselect but falls back to median of medians if progress is slow, resulting in both fast average performance and optimal worst-case performance. The average time complexity performance is $O(n)$.

The partition-based algorithms are generally done in place, which thus results in partially sorting the data. They can be done out of place, not changing the original data, at the cost of $O(n)$ additional space.

Median selection as pivot strategy [\[edit\]](#)

Further information: [Median of medians](#)

A median-selection algorithm can be used to yield a general selection algorithm or sorting algorithm, by applying it as the pivot strategy in Quickselect or Quicksort; if the median-selection algorithm is asymptotically optimal (linear-time), the resulting selection or sorting algorithm is as well. In fact, an exact median is not necessary – an approximate median is sufficient. In the [median of medians](#) selection algorithm, the pivot strategy computes an approximate median and uses this as pivot, recursing on a smaller set to compute this pivot. In practice the overhead of pivot computation is significant, so these algorithms are generally not used, but this technique is of theoretical interest in relating selection and sorting algorithms.

In detail, given a median-selection algorithm, one can use it as a pivot strategy in Quickselect, obtaining a selection algorithm. If the median-selection algorithm is optimal, meaning $O(n)$, then the resulting general selection algorithm is also optimal, again meaning linear. This is because Quickselect is a [decrease and conquer](#) algorithm, and using the median at each pivot means that at each step the search set decreases by half in size, so the overall complexity is a [geometric series](#) times the complexity of each step, and thus simply a constant times the complexity of a single step, in fact $2 = 1/(1 - (1/2))$ times (summing the series).

Similarly, given a median-selection algorithm or general selection algorithm applied to find the median, one can use it as a pivot strategy in Quicksort, obtaining a sorting algorithm. If the selection algorithm is optimal, meaning $O(n)$, then the resulting sorting algorithm is optimal, meaning $O(n \log n)$. The median is the best pivot for sorting, as it evenly divides the data, and thus guarantees optimal sorting, assuming the selection algorithm is optimal. A sorting analog to median of medians exists, using the pivot strategy (approximate median) in Quicksort, and similarly yields an optimal Quicksort.

Incremental sorting by selection [\[edit\]](#)

Converse to selection by sorting, one can incrementally sort by repeated selection. Abstractly, selection only yields a single element, the k th element. However, practical selection algorithms frequently involve partial sorting, or can be modified to do so. Selecting by partial sorting naturally does so, sorting the elements up to k , and selecting by partitioning also sorts some elements: the pivots are sorted to the correct positions, with the k th element being the final pivot, and the elements between the pivots have values between the pivot values. The difference between partition-based selection and partition-based sorting, as in quickselect versus quicksort, is that in selection one recurses on only one side of each pivot, sorting only the pivots (an average of $\log(n)$ pivots are used), rather than recursing on both sides of the pivot.

This can be used to speed up subsequent selections on the same data; in the extreme, a fully sorted array allows $O(1)$ selection. Further, compared with first doing a full sort, incrementally sorting by repeated selection [amortizes](#) the sorting cost over multiple selections.

For partially sorted data (up to k), so long as the partially sorted data and the index k up to which the data is sorted are recorded, subsequent selections of j less than or equal to k can simply select the j th element, as it is already sorted, while selections of j greater than k only need to sort the elements above the k th position.

For partitioned data, if the list of pivots is stored (for example, in a sorted list of the indices), then subsequent selections only need to select in the interval between two pivots (the nearest pivots below and above). The biggest gain is from the top-level pivots, which eliminate costly large partitions: a single pivot near the middle of the data cuts the time for future selections in half. The pivot list will grow over subsequent selections, as the data becomes more sorted, and can even be passed to a partition-based sort as the basis of a full sort.

Using data structures to select in sublinear time [\[edit\]](#)

Given an unorganized list of data, linear time ($\Omega(n)$) is required to find the minimum element, because we have to examine every element (otherwise, we might miss it). If we organize the list, for example by keeping it sorted at all times, then selecting the k th largest element is trivial, but then insertion requires linear time, as do other operations such as combining two lists.

The strategy to find an order statistic in [sublinear time](#) is to store the data in an organized fashion using suitable data structures that facilitate the selection. Two such data structures are tree-based structures and frequency tables.

When only the minimum (or maximum) is needed, a good approach is to use a [heap](#), which is able to find the minimum (or maximum) element in constant time, while all other operations, including insertion, are $O(\log n)$ or better. More generally, a [self-balancing binary search tree](#) can easily be augmented to make it possible to both insert an element and find the k th largest element in $O(\log n)$ time; this is called an [order statistic tree](#). We simply store in each node a count of how many descendants it has, and use this to determine which path to follow. The information can be updated efficiently since adding a node only affects the counts of its $O(\log n)$ ancestors, and tree rotations only affect the counts of the nodes involved in the rotation.

Another simple strategy is based on some of the same concepts as the [hash table](#). When we know the range of values beforehand, we can divide that range into h subintervals and assign these to h buckets. When we insert an element, we add it to the bucket corresponding to the interval it falls in. To find the minimum or maximum element, we scan from the beginning or end for the first nonempty bucket and find the minimum or maximum element in that bucket. In general, to find the k th element, we maintain a count of the number of elements in each bucket, then scan the buckets from left to right adding up counts until we find the bucket containing the desired element, then use the expected linear-time algorithm to find the correct element in that bucket.

If we choose h of size roughly \sqrt{n} , and the input is close to uniformly distributed, this scheme can perform selections in expected $O(\sqrt{n})$ time. Unfortunately, this strategy is also sensitive to clustering of elements in a narrow interval, which may result in buckets with large numbers of elements (clustering can be eliminated through a good hash function, but finding the element with the k th largest hash value isn't very useful). Additionally, like hash tables this structure requires table resizings to maintain efficiency as elements are added and n becomes much larger than h^2 . A useful case of this is finding an order statistic or extremum in a finite range of data. Using above table with bucket interval 1 and maintaining counts in each bucket is much superior to other methods. Such hash tables are like [frequency tables](#) used to classify the data in [descriptive statistics](#).

Lower bounds [\[edit\]](#)

In *The Art of Computer Programming*, Donald E. Knuth discussed a number of lower bounds for the number of comparisons required to locate the t smallest entries of an unorganized list of n items (using only comparisons). There is a trivial lower bound of $n - 1$ for the minimum or maximum entry. To see this, consider a tournament where each game represents one comparison. Since every player except the winner of the tournament must lose a game before we know the winner, we have a lower bound of $n - 1$ comparisons.

The story becomes more complex for other indexes. We define $W_t(n)$ as the minimum number of comparisons required to find the t smallest values. Knuth references a paper published by S. S. Kislitsyn, which shows an upper bound on this value:

$$W_t(n) \leq n - t + \sum_{n+1-t < j \leq n} \lceil \log_2 j \rceil \quad \text{for } n \geq t$$

This bound is achievable for $t=2$ but better, more complex bounds are known for larger t .

Space complexity [\[edit\]](#)

The required space complexity of selection is easily seen to be $k + O(1)$ (or $n - k$ if $k > n/2$), and in-place algorithms can select with only $O(1)$ additional storage. k storage is necessary as the following data illustrates: start with $1, 2, \dots, k$, then continue with $k + 1, k + 1, \dots, k + 1$, and finally finish with j copies of 0 , where j is from 0 to $k - 1$. In this case the k th smallest element is one of $1, 2, \dots, k$, depending on the number of 0 s, but this can only be determined at the end. One must store the initial k elements until near the end, since one cannot reduce the number of possibilities below the lowest k values until there are fewer than k elements left. Note that selecting the minimum (or maximum) by tracking the running minimum is a special case of this, with $k = 1$.

This space complexity is achieved by doing a progressive partial sort – tracking a sorted list of the lowest k elements so far, such as by the partial insertion sort above. Note however that selection by partial sorting, while space-efficient, has superlinear time complexity, and that time-efficient partition-based selection algorithms require $O(n)$ space.

This space complexity bound helps explain the close connection between selecting the k th element and selecting the (unordered) lowest k elements, as it shows that selecting the k th element effectively requires selecting the lowest k elements as an intermediate step.

Space complexity is particularly an issue when k is a fixed fraction of n , particularly for computing the median, where $k = n/2$, and in on-line algorithms. The space complexity can be reduced at the cost of only obtaining an approximate answer, or correct answer with certain probability; these are discussed below.

Online selection algorithm [\[edit\]](#)

[Online](#) selection may refer narrowly to computing the k th smallest element of a stream, in which case partial sorting algorithms (with $k + O(1)$ space for the k smallest elements so far) can be used, but partition-based algorithms cannot be.

Alternatively, selection itself may be required to be [online](#), that is, an element can only be selected from a sequential input at the instance of observation and each selection, respectively refusal, is irrevocable. The problem is to select, under these constraints, a specific element of the input sequence (as for example the

largest or the smallest value) with largest probability. This problem can be tackled by the [Odds algorithm](#), which yields the optimal under an independence condition; it is also optimal itself as an algorithm with the number of computations being linear in the length of input.

The simplest example is the [secretary problem](#) of choosing the maximum with high probability, in which case optimal strategy (on random data) is to track the running maximum of the first n/e elements and reject them, and then select the first element that is higher than this maximum.

Related problems [\[edit\]](#)

One may generalize the selection problem to apply to ranges within a list, yielding the problem of [range queries](#). The question of [range median queries](#) (computing the medians of multiple ranges) has been analyzed.

Language support [\[edit\]](#)

Very few languages have built-in support for general selection, although many provide facilities for finding the smallest or largest element of a list. A notable exception is **C++**, which provides a templated `nth_element` method with a guarantee of expected linear time, and also partitions the data, requiring that the n th element be sorted into its correct place, elements before the n th element are less than it, and elements after the n th element are greater than it. It is implied but not required that it is based on Hoare's algorithm (or some variant) by its requirement of expected linear time and partitioning of data.^{[1][2]}

C++ also provides an `nth_element` template function,^[3] which solves the problem of selecting the smallest k element in linear time while reordering its input sequence. No algorithm is provided for selecting the greatest k elements since this should be done by inverting the ordering [predicate](#).

For **Perl**, the module [Sort::Key::Top](#), available from [CPAN](#), provides a set of functions to select the top n elements from a list using several orderings and custom key extraction procedures. Furthermore, the [Statistics::CaseResampling](#) module provides a function to calculate quantiles using quickselect.

Python's standard library (since 2.4) includes `heapq.nsmallest()` and `nlargest()`, returning sorted lists, the former in $O(n + k \log n)$ time, the latter in $O(n \log k)$ time.

Because [language support for sorting](#) is more ubiquitous, the simplistic approach of sorting followed by indexing is preferred in many environments despite its disadvantage in speed. Indeed for [lazy languages](#), this simplistic approach can even achieve the best complexity possible for the k smallest/greatest sorted (with maximum/minimum as a special case) if the sort is lazy enough^{[\[citation needed\]](#)}.

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

- Ordinal optimization

References [\[edit\]](#)

- ↑ Section 25.3.2 of ISO/IEC 14882:2003(E) and 14882:1998(E)
- ↑ [nth_element](#), SGI STL
- ↑ ["std::nth_element"](#). *cppreference.com*. Retrieved 20 May 2014.
- ↑ Blum, M.; Floyd, R. W.; Pratt, V. R.; Rivest, R. L.; Tarjan, R. E. (August 1973). "Time bounds for selection" (PDF). *Journal of Computer and System Sciences* **7** (4): 448–461. doi:10.1016/S0022-0000(73)80033-9.
- ↑ Floyd, R. W.; Rivest, R. L. (March 1975). "Expected time bounds for selection". *Communications of the ACM* **18** (3): 165–172. doi:10.1145/360680.360691.
- ↑ Kiwiel, K. C. (2005). "On Floyd and Rivest's SELECT algorithm". *Theoretical Computer Science* **347**: 214–238. doi:10.1016/j.tcs.2005.06.032.
- ↑ Donald Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Section 5.3.3: Minimum-Comparison Selection, pp.207–219.
- ↑ 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. Chapter 9: Medians and Order Statistics, pp.183–196. Section 14.1: Dynamic order statistics, pp.302–308.
- ↑ Black, Paul E. "Select". *Dictionary of Algorithms and Data Structures*. NIST.

External links [\[edit\]](#)

- "Lecture notes for January 25, 1996: Selection and order statistics", ICS 161: *Design and Analysis of Algorithms*, David Eppstein

Categories: [Selection algorithms](#)

This page was last modified on 4 September 2015, at 07:50.

Text is available under the [Creative Commons Attribution-ShareAlike License](#); additional terms may apply. By using this site, you agree to the [Terms of Use](#) and [Privacy Policy](#). Wikipedia® is a registered trademark of the [Wikimedia Foundation, Inc.](#), a non-profit organization.

[Privacy policy](#) [About Wikipedia](#) [Disclaimers](#) [Contact Wikipedia](#) [Developers](#) [Mobile view](#)

