Selection by sorting

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 kth smallest (resp., kth 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 kth 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 kth element can also be done relatively cheaply.

Unordered partial sorting

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 kth 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 kth element. If this results in fewer than $k - 1$ elements, any remaining elements equal the kth 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 kth element, as elements greater than the kth element may also be equal to it.

Thus unordered partial sorting (lowest k elements, but not ordered) and selection of the kth 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 kth element).

Partial selection sort

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 kth 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

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 kth 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 introselect 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

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