Quickselect

From Wikipedia, the free encyclopedia

In computer science, **quickselect** is a selection algorithm to find the kth smallest element in an unordered list. It is related to the quicksort sorting algorithm. Like quicksort, it was developed by Tony Hoare, and thus is also known as **Hoare's selection algorithm**.[1] Like quicksort, it is efficient in practice and has good average-case performance, but has poor worst-case performance. Quickselect and variants is the selection algorithm most often used in efficient real-world implementations.

Quickselect	
Class	Selection algorithm
Data structure	Array
Worst case performance	$O(n^2)$
Best case performance	$O(n)$
Average case performance	$O(n)$

Quickselect uses the same overall approach as quicksort, choosing one element as a pivot and partitioning the data in two based on the pivot, accordingly as less than or greater than the pivot. However, instead of recursing into both sides, as in quicksort, quickselect only recurses into one side – the side with the element it is searching for. This reduces the average complexity from O(n log n) (in quicksort) to O(n) (in quickselect).

As with quicksort, quickselect is generally implemented as an in-place algorithm, and beyond selecting the kth element, it also partially sorts the data. See selection algorithm for further discussion of the connection with sorting.

Contents

- 1 Algorithm
- 2 Time complexity
- 3 Variants
- 4 References

Algorithm

In quicksort, there is a subprocedure called partition that can, in linear time, group a list (ranging from indices `left` to `right`) into two parts, those less than a certain element, and those greater than or equal to the element. Here is pseudocode that performs a partition about the element `list[pivotIndex]`:

```
function partition(list, left, right, pivotIndex)
    pivotValue := list[pivotIndex]
    swap list[pivotIndex] and list[right]  // Move pivot to end
    storeIndex := left
    for i from left to right-1
        if list[i] < pivotValue
            swap list[storeIndex] and list[i]
            increment storeIndex
    swap list[right] and list[storeIndex]  // Move pivot to its final place
    return storeIndex
```

In quicksort, we recursively sort both branches, leading to best-case $\Omega(n \log n)$ time. However, when doing selection, we already know which partition our desired element lies in, since the pivot is in its final sorted position, with all those preceding it in an unsorted order and all those following it in an unsorted order. Therefore, a single recursive call locates the desired element in the correct partition, and we build upon this for quickselect:

```
// Returns the n-th smallest element of list within left..right inclusive (i.e. left <= n <= right).
// The size of the list is not changing with each recursion. Thus, n does not need to be
// updated with each round.
function select(list, left, right, n)
    if left = right        // If the list contains only one element
        return list[left]  // Return that element
    pivotIndex  := ...        // select a pivotIndex between left and right, e.g. left + Math.floor(Math.rand
    pivotIndex  := partition(list, left, right, pivotIndex)
    // The pivot is in its final sorted position
    if n = pivotIndex
        return list[n]
    else if n < pivotIndex
        return select(list, left, pivotIndex - 1, n)
    else
        return select(list, pivotIndex + 1, right, n)
```

Note the resemblance to quicksort: just as the minimum-based selection algorithm is a partial selection sort, this is a partial quicksort, generating and partitioning only $O(\log n)$ of its $O(n)$ partitions. This simple procedure has expected linear performance, and, like quicksort, has quite good performance in practice. It is also an in-place algorithm, requiring only constant memory overhead, since the tail recursion can be eliminated with a loop like this:

```
function select(list, left, right, n)
    if left = right
        return list[left]
    loop
        pivotIndex := ...      // select pivotIndex between left and right
        pivotIndex := partition(list, left, right, pivotIndex)
        if n = pivotIndex
            return list[n]
        else if n < pivotIndex
            right := pivotIndex - 1
        else
            left := pivotIndex + 1
```

Time complexity

Like quicksort, the quickselect has good average performance, but is sensitive to the pivot that is chosen. If good pivots are chosen, meaning ones that consistently decrease the search set by a given fraction, then the search set decreases in size exponentially and by induction (or summing the geometric series) one sees that performance is linear, as each step is linear and the overall time is a constant times this (depending on how quickly the search set reduces). However, if bad pivots are consistently chosen, such as decreasing by only a single element each time, then worst-case performance is quadratic: $O(n^2)$. This occurs for example in searching for the maximum element of a set, using the first element as the pivot, and having sorted data.

Variants

The easiest solution is to choose a random pivot, which yields almost certain linear time. Deterministically, one can use median-of-3 pivot strategy (as in the quicksort), which yields linear performance on partially sorted

data, as is common in the real world. However, contrived sequences can still cause worst-case complexity; David Musser describes a "median-of-3 killer" sequence that allows an attack against that strategy, which was one motivation for his introselect algorithm.

One can assure linear performance even in the worst case by using a more sophisticated pivot strategy; this is done in the median of medians algorithm. However, the overhead of computing the pivot is high, and thus this is generally not used in practice. One can combine basic quickselect with median of medians as fallback to get both fast average case performance and linear worst-case performance; this is done in introselect.

Finer computations of the average time complexity yield a worst case of $n(2 + 2\log 2 + o(1)) \leq 3.4n + o(n)$ for random pivots (in the case of the median; other k are faster).[2] The constant can be improved to 3/2 by a more complicated pivot strategy, yielding the Floyd–Rivest algorithm, which has average complexity of $1.5n + O(n^{1/2})$ for median, with other k being faster.

References

1. ^ Hoare, C. A. R. (1961). "Algorithm 65: Find". *Comm. ACM* **4** (7): 321–322. doi:10.1145/366622.366647 (https://dx.doi.org/10.1145%2F366622.366647).
2. ^ Blum-style analysis of Quickselect (http://11011110.livejournal.com/119720.html), David Eppstein, October 9, 2007.

Retrieved from "http://en.wikipedia.org/w/index.php?title=Quickselect&oldid=635595866"

Categories: Selection algorithms