# Median of medians

From Wikipedia, the free encyclopedia

In computer science, the **median of medians** algorithm is a selection algorithm based on the quickselect algorithm, and is optimal, having worst-case linear time complexity for selecting the *k*th largest element. The algorithm consists of an algorithm to find an approximate median in linear time – this is the key step – which is then used as a pivot in quickselect. In other words, it uses an (asymptotically) optimal approximate median-selection algorithm to build an (asymptotically) optimal general selection algorithm.

**Median of Medians**

| | |
|---|---|
| **Class** | Selection algorithm |
| **Data structure** | Array |
| **Worst case performance** | $O(n)$ |
| **Best case performance** | $O(n)$ |
| **Worst case space complexity** | $O(1)$ auxiliary |

The approximate median-selection algorithm can also be used as a pivot strategy in quicksort, yielding an optimal algorithm, with worst-case complexity O(*n* log *n*). Although this approach optimizes quite well, it is typically outperformed in practice by instead choosing random pivots, which has average linear time for selection and average linearithmic time for sorting, and avoids the overhead of computing the pivot. Median of medians is used in the hybrid introselect algorithm as a fallback, to ensure worst-case linear performance: introselect starts with quickselect, to obtain good average performance, and then falls back to median of medians if progress is too slow.

The algorithm was published in Blum et al. (Tarjan), and thus is sometimes called **BFPRT** after the last names of the authors. In the original paper the algorithm was referred to as **PICK**, referring to quickselect as "FIND".

## Contents


- 1 Outline
- 2 Algorithm
- 3 Properties of pivot
- 4 Proof of O(*n*) running time
- 5 Analysis
- 6 References
- 7 External links


## Outline

Quickselect is linear-time on average, but it can require quadratic time with poor pivot choices. This is because quickselect is a decrease and conquer algorithm, with each step taking O(*n*) time in the size of the remaining search set. If the search set decreases exponentially quickly in size (by a fixed proportion), this yields a geometric series times the O(*n*) factor of a single step, and thus linear overall time. However, if the search set decreases slowly in size, such as linearly (by a fixed number of elements, in the worst case only reducing by one element each time), then a linear sum of linear steps yields quadratic overall time (formally, triangular numbers grow quadratically). For example, the worst case occurs when pivoting on the smallest element at each step,

such as applying quickselect for the maximum element to already sorted data and taking the first element as pivot each time.

If one instead consistently chooses "good" pivots, this is avoided and one always gets linear performance even in the worst case. A "good" pivot is one for which we can establish that a constant proportion of elements fall both below and above it, as then the search set decreases at least by a constant proportion at each step, hence exponentially quickly, and the overall time remains linear. The median is a good pivot – the best for sorting, and the best overall choice for selection – decreasing the search set by half at each step. Thus if one can compute the median in linear time, this only adds linear time to each step, and thus the overall complexity of the algorithm remains linear.

The median-of-medians algorithm does not actually compute the exact median, but computes an approximate median, namely a point that is guaranteed to be between the 30th and 70th percentiles (in the middle 4 deciles). Thus the search set decreases by a fixed proportion at each step, namely at least 30% (so at most 70% left). Lastly, the overhead of computing the pivot consists of recursing in a set of size 20% the size of the original search set, plus a linear factor, so at linear cost at each step, the problem is reduced to 90% (20% and 70%) the original size, which is a fixed proportion smaller, and thus by induction the overall complexity is linear in size.

# Algorithm

The *Select* algorithm divides the list into groups of five elements. (Left over elements are ignored for now.) Then, for each group of five, the median is calculated. *Select* is then called recursively on this sublist of $n/5$ elements to find their true median. Finally, the "median of medians" is chosen to be the pivot.

```
// returns the index of the median of medians.
// requires a variant of select, "selectIdx"
// which returns the index of the selected item rather than the value
function medianOfMedians(list, left, right)
   numMedians = ceil((right - left) / 5)
   for i from 0 to numMedians
       // get the median of the five-element subgroup
       subLeft  := left + i*5
       subRight  := subLeft + 4
       if (subRight > right) subRight  := right
       // alternatively, use a faster method that works on lists of size 5
       medianIdx  := selectIdx(list, subLeft, subRight, (subRight - subLeft) / 2)
       // move the median to a contiguous block at the beginning of the list
       swap list[left+i] and list[medianIdx]
    // select the median from the contiguous block
    return selectIdx(list, left, left + numMedians - 1, numMedians / 2)
```

# Properties of pivot

Of the $n/5$ groups, half the number of groups have their median less than the pivot(Median of Medians). Also, another half the number of groups($\frac{1}{2} \times n/5 = n/10$) have their median greater than the pivot. In each of these $n/10$ groups, there are two elements that are smaller than their respective medians, which are smaller than the pivot. Thus, each of the $n/10$ groups have at least 3 elements that are smaller than the pivot. Similarly, in each of the $n/10$ groups with median greater than the pivot, there are two elements that are greater than their respective medians, which are greater than the pivot. Thus, each of the $n/10$ groups have at least 3 elements that are greater than the pivot. Hence, the pivot is less than $3(n/10)$ elements and greater than another $3(n/10)$ elements. Thus the chosen median splits the elements somewhere between 30%/70% and 70%/30%, which assures worst-case linear behavior of the algorithm. To visualize:

**One iteration on a randomized set of 100 elements from 0 to 99**

| | | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 12 | 15 | 11 | 2 | 9 | 5 | 0 | 7 | 3 | 21 | 44 | 40 | 1 | 18 | 20 | 32 | 19 | 35 | 37 | 39 |
| | 13 | 16 | 14 | 8 | 10 | 26 | 6 | 33 | 4 | 27 | 49 | 46 | 52 | 25 | 51 | 34 | 43 | 56 | 72 | 79 |
| **Medians** | 17 | 23 | 24 | 28 | 29 | 30 | 31 | 36 | 42 | 47 | 50 | 55 | 58 | 60 | 63 | 65 | 66 | 67 | 81 | 83 |
| | 22 | 45 | 38 | 53 | 61 | 41 | 62 | 82 | 54 | 48 | 59 | 57 | 71 | 78 | 64 | 80 | 70 | 76 | 85 | 87 |
| | 96 | 95 | 94 | 86 | 89 | 69 | 68 | 97 | 73 | 92 | 74 | 88 | 99 | 84 | 75 | 90 | 77 | 93 | 98 | 91 |

(red = "(one of the two possible) median of medians", gray = "number < red", white = "number > red")

5-tuples are shown here sorted by median, for clarity. Sorting the tuples is not necessary because we only need the median for use as pivot element.

Note that all elements above/left of the red (30% of the 100 elements) are less, and all elements below/right of the red (another 30% of the 100 elements) are greater.

# Proof of O(*n*) running time

The median-calculating recursive call does not exceed worst-case linear behavior because the list of medians is 20% of the size of the list, while the other recursive call recurses on at most 70% of the list. Let *T(n)* be the time it takes to run a median-of-medians Quickselect algorithm on an array of size *n*. Then we know this time is:

$$T(n) \leq T(n \cdot 2/10) + T(n \cdot 7/10) + c \cdot n.$$

where

- the *T(n·2/10)* part is for finding the *true* median of the *n/5* medians, by running an (independent) Quickselect on them (since finding the median is just a special case of selecting a *k*-largest element)
- the O(*n*) term *c·n* is for the partitioning work to create the two sides, one of which our Quickselect will recurse (we visited each element a constant number of times, in order to form them into n/5 groups and take each median in O(1) time).
- the *T(n·7/10)* part is for the actual Quickselect recursion (for the worst case, in which the *k*-th element is in the bigger partition that can be of size n·7/10 maximally)

From this, using induction one can easily show that

$$T(n) \leq 10 \cdot c \cdot n \in O(n).$$

# Analysis

The key step is reducing the problem to selecting in two lists whose total length is shorter than the original list, plus a linear factor for the reduction step. This allows a simple induction to show that the overall running time is linear.

The specific choice of groups of five elements is explained as follows. Firstly, computing median of an odd list is faster and simpler; while one could use an even list, this requires taking the average of the two middle elements,

which is slower than simply selecting the single exact middle element. Secondly, five is the smallest odd number such that median of medians works. With groups of only three elements, the resulting list of medians to search in is length $n/3$ (33 1/3%), and reduces the list to recurse into to length $2n/3$ (66 2/3%), since it is greater than $1/2 \times 2/3 = 1/3$ of the elements and less than $1/2 \times 2/3 = 1/3$ of the elements. Thus this still leaves $n$ elements to search in, not reducing the problem sufficiently. The individual lists are shorter, however, and one can bound the resulting complexity by the Akra–Bazzi method, but it does not prove linearity.

Conversely, one may instead group by $g$ = seven, nine, or more elements, and this does work. This reduces the size of the list of medians to $n/g$, and the size of the list to recurse into asymptotes at $3n/4$ (75%), as the quadrants in the above table approximate 25%, as the size of the overlapping lines decreases proportionally. This reduces the scaling factor from 10 asymptotically to 4, but accordingly raises the $c$ term for the partitioning work. Finding the median of a larger group takes longer, but is a constant factor (depending only on $g$), and thus does not change the overall performance as $n$ grows.

If one instead groups the other way, say dividing the $n$ element list into 5 lists, computing the median of each, and then computing the median of these – i.e., grouping by a constant fraction, not a constant number – one does not as clearly reduce the problem, since it requires computing 5 medians, each in a list of $n/5$ elements, and then recursing on a list of length at most $7n/10$. As with grouping by 3, the individual lists are shorter, but the overall length is no shorter – in fact longer – and thus one can only prove superlinear bounds. Grouping into a square of $\sqrt{n}$ lists of length $\sqrt{n}$ is similarly complicated.

# References


- Blum, M.; Floyd, R. W.; Pratt, V. R.; Rivest, R. L.; Tarjan, R. E. (August 1973). "Time bounds for selection" (http://people.csail.mit.edu/rivest/pubs/BFPRT73.pdf). *Journal of Computer and System Sciences* **7** (4): 448–461. doi:10.1016/S0022-0000(73)80033-9 (https://dx.doi.org/10.1016%2FS0022-0000%2873%2980033-9).


# External links

- "Lecture notes for January 30, 1996: Deterministic selection (http://www.ics.uci.edu/~eppstein/161/960130.html)", *ICS 161: Design and Analysis of Algorithms*, David Eppstein



Categories: Selection algorithms

---