Comparison of selection algorithms

Cristina Luengo Agulló

Randomized Algorithms

6 November 2015

Contents

Selection algorithms

Selection algorithms are algorithms for finding the k-th smallest element in a list. This includes the case of finding the median element, which is the case we are going to study in this assignment. The following sections describe two main algorithms used for this purpose: Quick select and Monte Carlo.

1.1 Quickselect

This algorithm is related to the Quicksort algorithm [1]. It is efficient in practice and has good average-case performance, but performs poorly in worst-case scenarios.

The algorithm works as follows:

Quickselect: Given a list A of size n and an integer $1 \leq k \leq n$

1. Pick a pivot element p from A.
2. Split A into subarrays LESSER and GREATER by comparing each element to p. While we are at it, count the number L of elements going in to LESSER.
3.
   - If $L = k{-}1$, then output p.
   - If $L > k{-}1$, output $QuickSelect(LESSER, k)$.
   - If $L < k{-}1$, output $QuickSelect(GREATER, k{-}L{-}1)$.

This algorithm constitutes the main body of Quickselect. Nevertheless, its performance can change depending on how the pivot is selected. We focus on two ways of selecting it: choosing a deterministic pivot or choosing it at random. Moreover, we describe how much the selection of the pivot affects performance in terms of running time.

However, in the average case this algorithm has the same performance for the different methods to select the pivot we present. Let $T(n)$ be the time Quickselect takes to execute in the average case. Also, assume that the pivot is not the desired element until it is the only element, and the pivot is chosen somewhere in the middle. Therefore, assume that the recursion is performed on a list half the size of the previous list, since the expected value of the pivot if we consider it randomly is to fall in the middle.

$$T(n) = 1 + n + T(\tfrac{n}{2})$$

The n term comes from performing the partition. Dropping the constant term and following the recursion produces the following series of equations

$$\begin{array}{rcl} T(n) & = & n + T(\frac{n}{2}) \\ T(\frac{n}{2}) & = & \frac{n}{2} + T(\frac{n}{4}) \\ T(\frac{n}{4}) & = & \frac{n}{2} + T(\frac{n}{8}) \\ & ... & \\ T(2) & = & 2 + T(1) \\ & ... & \end{array}$$

Summing the equations and cross-canceling like values produces a linear result.

$$\begin{array}{rcl} T(n) & = & \frac{n}{2} + \frac{n}{4} + ... + 2 + 1 + ... \\ & = & \sum_{i=0}^{\infty} \frac{n}{2^i} \\ & = & n \times \sum_{i=0}^{\infty} (\frac{1}{2})^i \\ & = & n \times (\frac{1}{1-\frac{1}{2}}) \\ & = & 2 \times n \\ & = & O(n) \end{array}$$

1.1.1 Deterministic pivot

One straight-forward way to choose a pivot is to always select the same one (e.g. we could choose the last element of the list). This makes the implementation much easier but it comes at a cost, since it will make the algorithm perform poorly in worst-case scenarios. One of such scenarios could be when the pivot always happens to be at an edge of the list (i.e. a minimum or a maximum). If that happens, the partition will be unbalanced and one of the sides will always hold all the elements of the list except for the pivot. Hence, the algorithm will go through the list for each element, which will take quadratic time.

Let $T(n)$ be the time Quickselect takes to run on an input of size n. In the worst case we will have:

$$T(n) = 1 + n + T(n-1)$$

The n term comes from performing the partition. Dropping the constant term and following the recursion produces n equations.

$$\begin{array}{rcl} T(n) & = & n + T(n-1) \\ T(n-1) & = & n - 1 + T(n-2) \\ T(n-2) & = & n - 2 + T(n-3) \\ & ... & \\ T(2) & = & 2 + T(1) \\ T(1) & = & 1 + T(0) \end{array}$$

Summing the equations and cross-canceling like values produces a polynomial result.

$$\begin{aligned} T(n) &= n + (n-1) + (n-2) + ... + 2 + 1 \\ &= \sum_{i=1}^{n} i \\ &= \frac{n \times (n-1)}{2} \\ &= O(n^2) \end{aligned}$$

Thus, in order to avoid the quadratic time in worst-case cases, other deterministic algorithms can be used, such as the median of Medians algorithm.

Median of medians

The Median of medians algorithm [2] allows us to achieve linear time in the worst case. It works as follows:

1. Divide the list into sublists of length five. (Note that the last sublist may have length less than five).
2. Sort each sublist and determine its median using a sorting algorithm (e.g. insertion sort).
3. Use the median of medians algorithm to recursively determine the median of the set of all medians from the previous step.
4. Use the median of the medians from step 3 as the pivot.

Although this algorithm is more complex than always choosing the same pivot, it allows us to have worst-case linear time complexity. This is due to the fact that the list will always be partitioned in balanced parts, thereby making recursive calls with practically the same input size.

Therefore, let x be the median of medians and n the number of elements of the list. Then either the list analyzed at a certain moment in the function doesn't contain any items greater than x, or it doesn't contain any items less than x. However, there are lots of elements greater than x and there are lots of elements less than x. In particular, each (non-remainder) group whose median is less than x contains at least three elements less than x, and each (non-remainder) group whose median is greater than x contains at least three elements greater than x. Moreover, there are at least $\lceil n/5 \rceil - 1$ non-remainder groups, of which at least $\lfloor 1/2 \times \lceil n/5 \rceil - 2 \rfloor$ have median less than x and at least $\lfloor 1/2 \times \lceil n/5 \rceil - 2 \rfloor$ have mean greater than x. Finally, the group with x as its median contains two elements greater than x and two elements less than x. Thus:

$$\begin{gathered} (\text{Number of elements in the list} < \text{than } x) \geq 3 \times (\lfloor 1/2 \times \lceil n/5 \rceil - 2 \rfloor) + 2 \\ \geq 3 \times (\lfloor 1/2 \times \lceil n/5 \rceil - 2 - 1/2 \rfloor) + 2 > 3/10 \times n - 6 \end{gathered}$$

Therefore, the current list analyzed at a certain moment in the function must exclude at least $3/10 \times n - 6$ elements of the original list, which means that the current list contains at most $n - (3/10 \times n - 6) = 7/10 \times n + 6$ elements. Hence, the following recurrence expresses the execution time of the algorithm:

$$T(n) \leq \begin{cases} O(1) & if \quad n \leq 140 \\ T(\lceil n/5 \rceil) + T(7/10 \times n + 6) + O(n) & otherwise \end{cases}$$

In order to deal with the base case of an inductive argument, it is often useful to separate out the small values of n. The inductive argument is as follows:

Base case:

$$\begin{aligned} T(140) &\leq T(\lceil 140/5 rceil) + T(7/10 \times 140 + 6) + a \times n \\ &\leq b + b + a \times n \\ &= 2 \times b + a \times n \\ &= 4 \times b + 140 \times a \end{aligned}$$

The base case holds if and only if $2 \times b + 140 \leq c \times 140$.

Inductive case: $n \geq 140$.

$$\begin{aligned} T(n) &\leq c \times \lceil n/5 \rceil + c \times (7/10 \times n + 6) + a \\ &\leq c \times (n/5 + 1) + c \times (7/10 \times n + 6) + a \\ &= 9/10 \times \times n + 7 \times c + a \times n \\ &= c \times n + (\tfrac{-c \times n}{10} + 7 \times c + a \times n) \end{aligned}$$

Thus all we need is for the second term of the last sum to be non positive. This will happen if and only if $c \geq \frac{10an}{n-70}$. On the other hand, since $n \geq 140$ we know that $\frac{n}{n-70} \leq 2$. Hence, the second term will be non positive whenever $c \geq 20 \times a$.

Therefore, we find that the base case and the inductive step will both be satisfied as long as c is greater than both $a + b/70$ and $20 \times a$. If we set $c = max(20a, b/70)$, then we have $T(n) = c \times n$, which gives $O(n)$ time.

1.1.2 Random pivot

As explained in the previous section, choosing a deterministic pivot can be either dangerous for worst-case scenarios or more costly. Hence, one way to partially avoid such problems is choosing a random pivot. This will be definitely less costly than performing the median of medians algorithm, and will also provide freedom when partitioning the list. Nevertheless, if the pivot always falls close to any of the edges of the list, we will also have quadratic time, as in the first deterministic case. But since we choose it uniformly at random, the probability that it always falls on an edge is small.

1.2 Monte Carlo

This section provides an overview on a randomized algorithm to find the median of a list of numbers. It implements the Monte Carlo algorithm [3], and it is significantly simpler than the

Quickselect algorithm we previously explained. We first introduce the algorithm and then comment on the changes we can make in order to allow it to return the k-th element of a list, rather than the median. The algorithm works as follows:

MonteCarlo: Given a set S of n distinct elements over a totally ordered universe

1. Pick a (multi-)set R of $\lceil n^{3/4} \rceil$ elements in S. chosen independently and uniformly at random with replacement.
2. Sort the set R.
3. Let d be the $(\lfloor 1/2 \times n^{3/4} - \sqrt{n} \rfloor)$th smallest element in the sorted set R.
4. Let u be the $(\lfloor 1/2 \times n^{3/4} + \sqrt{n} \rfloor)$th smallest element in the sorted set R.
5. By comparing every element in S to d and u, compute the set $C = \{x \in S : d \leq x \leq u\}$ and the numbers $l_d = |\{x \in S : x < d\}$ and $l_u = |\{x \in S : x > u\}|$.
6. If $l_d > n/2$ or $l_u > n/2$ then FAIL.
7. If $|C| \leq 4 \times n^{3/4}$ then sort the set C, otherwise FAIL.
8. Output the $(\lfloor n/2 \rfloor - l_d + 1)$th element in the sorted order of C.

As can be seen, the algorithm involves sampling, and it does not always provide a valid answer. Correctness follows because the algorithm could only give an incorrect answer if the median were not in the set C. But if that was the case, then either $l_d > n/2$ or $l_u > n/2$ and hence step 6 of the algorithm guarantees that the algorithm outputs FAIL. Similarly, as long as C is sufficiently small, the total work is only linear in the size of S. Therefore, step 7 of the algorithm guarantees that it does not take more than linear time, since it outputs FAIL without sorting if the sorting might take too long.

Hence, the execution time of this algorithm is linear, since eachs step takes either constant or linear time:

- **Step 1**: Takes $O(n^{3/4})$
- **Step 2**: Takes $O(n^{3/4} \times log(n^{3/4})) = O(3/4 \times n^{3/4} \times log(n)) = O(n)$
- **Step 3**: Takes $O(1)$
- **Step 4**: Takes $O(1)$
- **Step 5**: Takes $O(1)$
- **Step 6**: Takes $O(1)$
- **Step 7**: Because we have not failed, we know that $|C| \leq 4 \times n^{3/4}$ and hence we have $O(4 \times n^{3/4} \times log(4 \times n^{3/4})) = O(n)$
- **Step 8**: Takes $O(1)$

If we sum all steps, the total running time is $O(n)$.

The algorithm presented so far only computes the median of a list, but does not allow us to obtain the k-th element of it. In order to do so, we need to do some modifications on the original algorithm:

MonteCarloK: Given a set S of n distinct elements over a totally ordered universe and an integer number $1 \leq k \leq |S|$

1. Pick a (multi-)set R of $\lceil n^{3/4} \rceil$ elements in S. chosen independently and uniformly at random with replacement.
2. Sort the set R.
3. Let d be the $(\lfloor (k/n) \times n^{3/4} - \sqrt{n} \rfloor)$th smallest element in the sorted set R.
4. Let u be the $(\lfloor (k/n) \times n^{3/4} + \sqrt{n} \rfloor)$th smallest element in the sorted set R.
5. By comparing every element in S to d and u, compute the set $C = \{x \in S : d \leq x \leq u\}$ and the numbers $l_d = |\{x \in S : x < d\}$ and $l_u = |\{x \in S : x > u\}|$.
6. If $l_d > k$ or $l_u > (n - k)$ then FAIL.
7. If $|C| \leq 4 \times n^{3/4}$ then sort the set C, otherwise FAIL.
8. Output the $(k - l_d + 1)$th element in the sorted order of C.

Nevertheless, this algorithm fails more often with a k different than $n/2$, since the expected value if we pick a number at random will always be the median. This happens specially when k is close to the minimum or maximum of the list, since it is less probable that these numbers will fall into the sample set R.

Therefore, we decided to use the different algorithms presented to find the median of a list, rather than the k-th element. This will make the experimentation an easier task. The following sections explain the experiments carried out and the results obtained from them.

Experiments

The purpose of the experiments we performed is to assess the efficiency of the different algorithms we presented in Section 1.1, in terms of execution time. The implementation of all the algorithms can be found in [4], along with an explanation on how to generate input samples and execute them.

The following sections describe the experimental setup we used to carry out the experiments and the results obtained.

2.1 Experimental setup

The experiments were conducted in the following environment:

- **Operating system**: Ubuntu 14.04
- **RAM**: 8GB
- **Processor**: Intel Core i7-4710HQ at 2.50GHz
- **Language**: C++
- **Compiler**: g++-4.8

To generate the input samples we made use of the C++ random number generator, and we created lists of distinct elements with different sizes. The fact that we decided to generate non repeated elements is because we need this premise for the Monte Carlo algorithm.

We chose large sizes in order to see better how execution time changed as the size increased. We also chose odd size numbers, so there would be no ties when choosing the median. Therefore, we executed the different algorithms for each input size using the exact same input for all of them.

Moreover, we measured time by using the C++ time library (chrono). We only measured the execution time for each algorithm, and since system's time is typically not very accurate, we executed 5 times each experiment and calculated the average time.

2.1.1 Experimental cases

We defined two experimental cases:

- **Randomly generated input**: We generated different random inputs of increasing size in order to assess average execution times.

- **Already sorted input**: In order to study how Quickselect behaves in worst-case scenarios, we generated an already sorted input, which should help us show the quadratic behavior of the algorithm when choosing as pivot the last element of the list. This case also simulates when the pivot falls close to the edges if we pick it randomly, which is why we did not include results for that method.

2.2 Results

This section provides an overview on the results obtained from the experiments we carried out. Table 2.1 shows the results for randomly generated input of increasing size by a factor of 2 and a factor of 5. All the times shown are an average of 5 executions on the same input.

As can be seen, for each algorithm, the execution time increases approximately by the same factor (by 5 or by 2) as the input size does. Therefore, this matches the theoretical analyses we made in Section 1.1, which stated that the selection algorithms presented work in linear time. Moreover, Figure 2.1 shows a linear diagram of the execution times for all the algorithms, where it can be seen that the algorithm that takes longer to execute is Quickselect with a Median of medians pivot. This is due to the fact that more work is carried out in each recursive call than in the other Quickselect methods, since it has to estimate the median of the list. Also, note that the Monte Carlo algorithm takes approximately as much time as the median of medians algorithm, since sampling and sorting on the original list takes more time than partitioning the list in smaller inputs lists every time (as Quickselect does).

On the other hand, the algorithm that has the best performance in terms of execution time in Quickselect with a randomized pivot.

Size	Det. pivot	Rand. pivot	MOM pivot	Monte Carlo
10001	0.000461	0.000659	0.002231	0.017083
50001	0.001593	0.001857	0.011580	0.057667
100001	0.004231	0.003944	0.022424	0.096686
500001	0.020005	0.020529	0.112145	0.328974
1000001	0.045419	0.034039	0.227739	0.558126
5000001	0.195831	0.181818	1.106728	1.946066
10000001	0.352124	0.385493	2.313629	3.348377
50000001	2.044349	1.761923	11.776725	11.863786
100000001	3.680774	4.303980	23.230659	20.487782

Table 2.1: Comparison of execution times for Quickselect with deterministic pivot, Random pivot, Median of medians pivot and for Monte Carlo algorithm

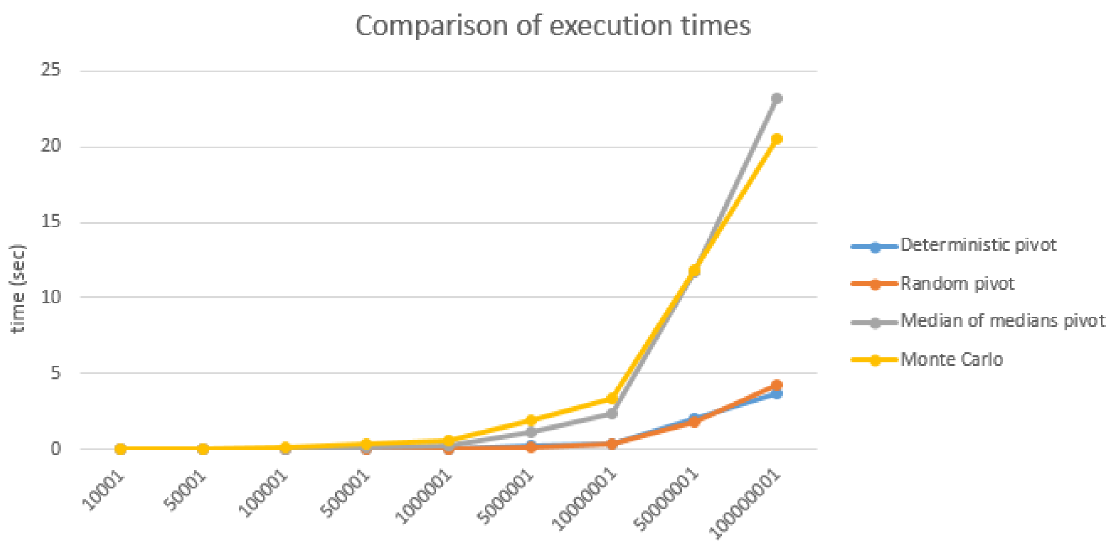

Figure 2.1: Comparison of execution times of the different algorithms

On the other hand, Table 2.2 shows the results for the worst case scenario. We only included executions up to size 500001, since they are enough to see the behavior of the algorithms as size increases when the input list is sorted. Moreover, we had to increase the stack size in order to execute the Quickselect with a deterministic pivot algorithm for size 500001 (the number of recursive calls that were being made was too big). All the times shown are also an average of 5 executions on the same input.

As can be seen, for the algorithm with deterministic pivot, the running times increase approximately by a factor of 25 when the input size is increased by a factor of 5, and they also increase by approximately a factor of 4 when the input size doubles. This confirms the theoretical analysis made in Section 1.1, which stated that the algorithm runs in quadratic time in worst-case scenarios. This also applies when choosing a random pivot which always falls in the edges of the list.

Furthermore, running times for Quickselect with a Median of medians pivot and for Monte Carlo remain linear as the input size increases (i.e. they increase approximately by the same factor as the size). Figure 2.2 shows a linear diagram of the execution times, where the quadratic shape for the algorithm with deterministic pivot can be seen.

Size	**Det. pivot**	**MOM pivot**	**Monte Carlo**
10001	0.098867	0.001505	0.016957
50001	2.431103	0.007384	0.056700
100001	9.716421	0.015214	0.095421
500001	323.995239	0.081684	0.321833

Table 2.2: Comparison of execution times in worst case scenario for Quick Select with deterministic pivot, with Median of medians pivot and for Monte Carlo algorithm

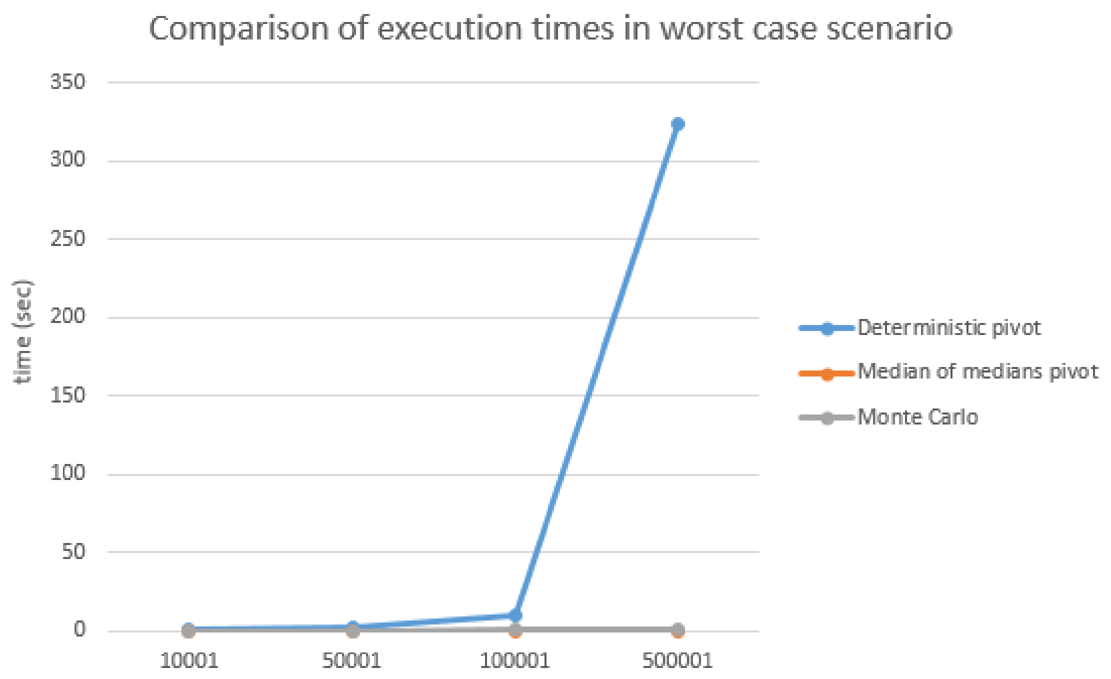

Figure 2.2: Comparison of execution times in a worst case scenario

Conclusions

As seen in Section 2.2, the algorithm that has worst perfomance in terms of running time is Quickselect with a Median of medians pivot, and the one with the best performance is Quickselect with a randomized pivot.

Depending on the application, we might want to use one algorithm or another. If we need to ensure worst-case linear time, then we should pick the Median of medians method for Quickselect or the Monte Carlo algorithm. Nevertheless, if we need to always get a correct result we should avoid using Monte Carlo.

On the other hand, if we only need good average performance, the randomized pivot method for Quickselect might be the best option, since it has proven to be the most efficient.

Finally, picking as pivot the last element of the list is a cheap method in terms of time, but it is bound to perform poorly in worst-case scenarios.

References

[1] Thomas Cormen and Devin Balkcom. Overview of quicksort. Lecture notes. URL `https://www.khanacademy.org/computing/computer-science/algorithms/quick-sort/a/overview-of-quicksort`. Last visited 2015-11-5.

[2] Introduction and median finding. URL `http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-046j-design-and-analysis-of-algorithms-spring-2012/lecture-notes/MIT6_046JS12_lec01.pdf`. Last visited 2015-11-5.

[3] Michael Mitzenmacher and Eli Upfal. *Probability and computing: Randomized algorithms and probabilistic analysis*. Cambridge University Press, 2005.

[4] Selection of the k-th element of a list. URL `https://github.com/cristinaluengoagullo/selection`.