Quickselect - Analysis of the expected number of comparisons

Alberto Parravicini

1 Abstract

The following report will analyze the number of comparisons performed by a **Rust** implementation of the **Quickselect** algorithm, a randomized algorithm used to find the $k-th$ smallest element in an unordered list.

The first section of the report presents the algorithm, and its **Rust implementation**.

The second section studies the number of **comparisons** between done by the algorithm, i.e. how many elements from the input list must be compared to each other to give the desired output. This numbers are then compared to the theoretical expected number of comparisons done by the algorithm.
The algorithm is tested against vectors with increasing size n and various ranges of k.

As addendum, an analysis of the **execution time** of *Quickselect* is reported, to show how the number of comparisons influences the running time of the algorithm.

2 Quickselect's algorithm implementation

2.1 *Theoretical background*

Given a list of $\mathbf{n}$ elements for which exists a total order under $\leqslant$, the **Quickselect** algorithm is used to find the $\mathbf{k-th}$ smallest element in the list (also referred as element of *rank k*), for $\mathbf{k} \in [\mathbf{1},\ \mathbf{n}]$.

To select the element of rank k (with $1 \leqslant k \leqslant n$), the algorithm works as follow:

```
quickselect(list, start, end):
    pick an element at random from the list,
        in the range [start, end], call it pivot;
    put all the elements < pivot on its left;
    put all the elements > pivot on its right;
    the pivot is now in its sorted position in the list:
        if position(pivot) == k, return it;
        if position(pivot) < k, return quickselect(start, position(pivot));
        if position(pivot) > k, return quickselect(position(pivot), end);
```

It can be seen how there is a close resemblance between the **Quickselect** and the **Quicksort** algorithms.
Indeed, **Quickselect** can be seen as a modified version of **Quicksort** in which the recursion is applied only to the portion of the list where the desired element of *rank k* is expected to be found.

2.2 *Rust implementation*

Rust was chosen as the programming language of the implementation as the inherent speed of the language, combined with the relative simplicity of the **Quickselect** algorithm, make it a good candidate for performing a large number of tests over many different parameter values.

What follows are snippets of code, to explain how the **Quickselect** algorithm was implemented. The following code is inspired to the pseudo-code of the algorithm found on *Wikipedia*. [2]

- **Quickselect**

```
fn quickselect(vec: &mut Vec<usize>,
        start: usize, end: usize,
        k: usize, num_of_comparisons: u32) -> (usize, u32) {
  let mut num_of_comparisons: u32 = num_of_comparisons;

  if k < start || k > end {
    panic!("INVALID VALUE OF K: {}", k)
  }
  // If just one element is present, return it;
  if start >= end {
    return (vec[end], num_of_comparisons)
  }
  let pivot_index = rand::thread_rng().gen_range(start, end + 1);
  // Put the current pivot in its correct place, and return its position;
  let (pivot_index, temp_num_of_comparisons) = partition(vec, start, end, pivot_index);
  num_of_comparisons += temp_num_of_comparisons;

  // Apply the sorting only where the desired element could be,
```

```
    // or return it if it was found;
    if k == pivot_index {
        return (vec[k], num_of_comparisons)
    }
    else if k < pivot_index {
        let result = quickselect(vec, start, pivot_index, k, num_of_comparisons);
        return result
    }
    else {
        let result = quickselect(vec, pivot_index + 1, end, k, num_of_comparisons);
        return result
    }
}
```

- **Select a pivot and put it into its sorted position**

```
fn partition(vec: &mut Vec<usize>,
             start: usize, end: usize,
             pivot_index: usize) -> (usize, u32) {
    let pivot_value = vec[pivot_index];
    let mut temp_num_of_comparisons: u32 = 0;

    // Temporarily put the pivot at the end of the vector;
    vec.swap(pivot_index, end);
    let mut store_index = start;
    for i in start..end {
        temp_num_of_comparisons += 1;
        if vec[i] < pivot_value {
            // If a value lower than the pivot is found, put it in the left part of the vector;
            vec.swap(store_index, i);
            // store_index keeps track of how many values lower that the pivot exist,
            // and where the pivot should be placed at the end;
            store_index += 1;
        }
    }
    // Put the pivot in its correct place;
    vec.swap(end, store_index);

    // Return the sorted position of the pivot and the number of comparisons performed;
    (store_index, temp_num_of_comparisons)
}
```

3 Analysis of the expected number of comparisons

3.1 *Theoretical background*

For any two elements X_i, X_j of the list, with $j > i$,

$$X_{i,j} = \begin{cases} 1 \text{ if } X_i,\ X_j \text{ are compared during the execution of the algorithm.} \\ 0 \text{ else.} \end{cases}$$

The complexity of **Quickselect** is related to the total number of comparisons X performed by the algorithm.
It can be shown that the expected number of comparisons $E[X]$ is given by:

$$E[X] = E\Big[\sum_{i<j} X_{i,j}\Big] = \left(2n + 2n \cdot \ln\left(\frac{n}{n-k}\right) + 2k \cdot \ln\left(\frac{n-k}{k}\right)\right) \cdot (1 + o(1)) \qquad (*)$$

A number of observations can be done on this formula:

- If k is chosen to be proportional to n, the number of comparisons becomes linear with respect to n.
  As an example, if $k = \frac{n}{2}$, $E[X] = n \cdot (2 + 2 \cdot \ln(2)) \cdot (1 + o(1))$.

- The function $(*)$ is not defined for $k = n$; that said, it can be seen that for $k = n$ the value of $(*)$ is very close to its value in $k = 1$.
  As such, the function will be approximated in this way in the following analyses.

- In the interval $k \in [1, n)$, it can be seen that $(*)$ has a single local maximum, in $k = \frac{n}{2}$. In fact, by putting the derivative of $(*)$ w.r.t. k equal to 0:

  $$\frac{\partial}{\partial k}\left(2n + 2n \cdot \ln\left(\frac{n}{n-k}\right) + 2k \cdot \ln\left(\frac{n-k}{k}\right)\right) = \ln\left(\frac{n}{k} - 1\right) = 0$$

  From which one can find $k = \frac{n}{2}$, which is a maximum as the second derivative is negative in the interval $k \in [1, n)$. Moreover, the local minima of $(*)$ are at the extremes of the range of k, and the function is concave, with no other extrema in the $k \in [1, n)$.

The **Quicksort** algorithm has been tested against lists of increasing size, while keeping the value of k proportional to the size of the list, and against different values of k, while keeping the list size fixed. Note that as the selection of the pivot is random, the permutation of the input list doesn't influence the number of comparisons performed by the algorithm; as such, it is possible to use already sorted list as input of the algorithm.

3.2 ***First test:*** *increasing size of the list, select the median*

The first test consisted in running **Quickselect** on lists with *increasing size* n *and* $k = \frac{n}{2}$. Using a value of k proportional to n is required to make the expected number of comparisons a function of only n.
It can be seen from $(*)$ that if $k = \frac{n}{2}$ (i.e. one wants to find the **median** of a list), the expected number of comparisons will be equal to

$$E[X] = n \cdot (2 + 2 \cdot \ln(2)) \cdot (1 + o(1)) \qquad (**)$$

linear with respect to n, with a constant slope of 3.38 (note from the following plot that $o(1)$ doesn't seem to have any meaningful impact on the number of comparisons).

In the test, the size of the lists ranged from 10000 to 1000000, with increments of 10000. For each list size, *Quickselect* was run 1000 times.

The empirical number of comparisons with respect to each list size has been compared to the theoretical number of comparisons given by $(**)$.

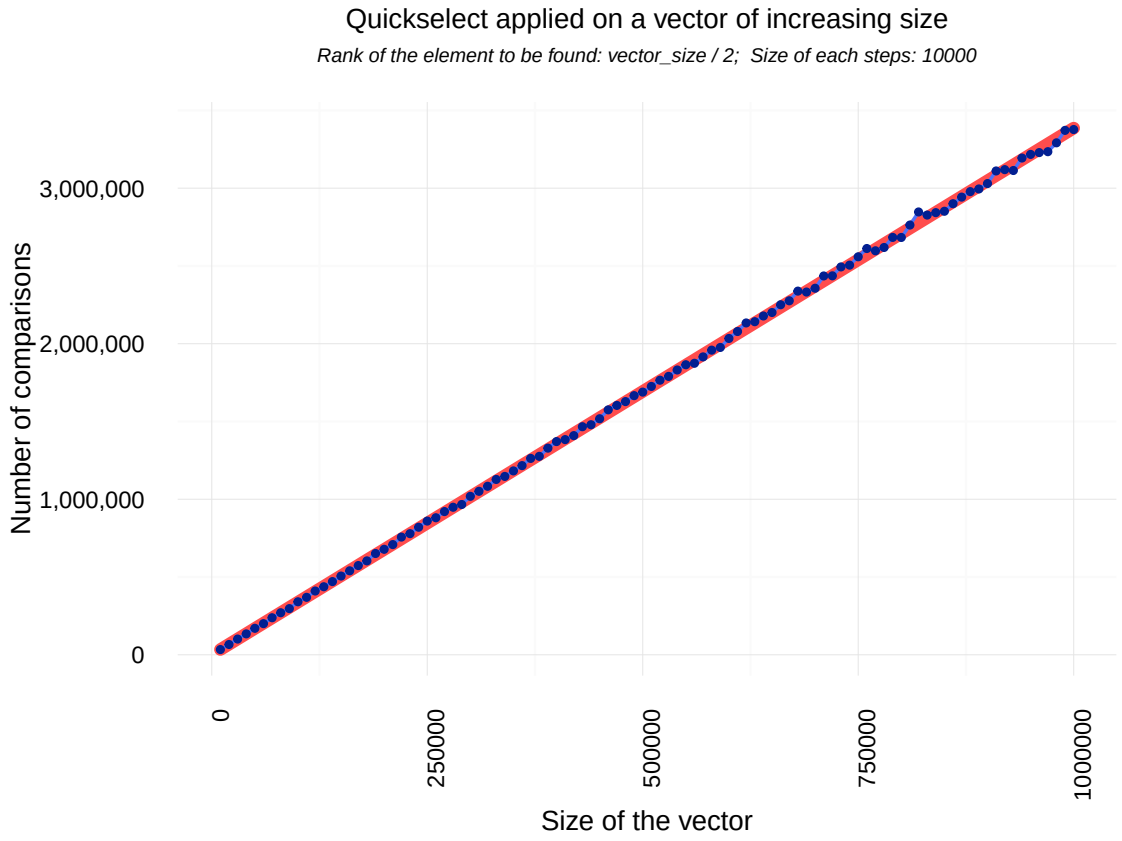

Figure 1: *In blue, the measured number of comparisons. In red, the theoretical number of comparisons.*

It can be seen that the empirical number of comparisons is perfectly in line with the theoretical numbers given by $(**)$. In fact, the *mean absolute percentage error (MAPE)* [1] is of just 0.6%. The *MAPE* is defined as $\frac{100}{m}\sum_{i=1}^{m}\left|1-\frac{F_i}{A_i}\right|$, with A the theoretical number of comparisons, F the observed number of comparisons, and m the size of A and F,

3.3 ***Second test:*** *increasing value of k, fixed list size*

As a second test, the size of the list was kept constant, and it was measured how changing the value of k can influence the expected number of comparisons. By fixing n, the number of comparisons becomes thus a function of only k, ranging from 1 to n. Note that $(*)$ isn't defined for $k = n$, but its value can be approximated by using $k = 1$.

The test was performed on a list of size 100000, with k ranging from 1 to 100000, with increments of 123 (this value is used so that it is possible to cover the entire range of k, as 123 is a divisor of $100000 - 1 = 99999$). For each value of k, the *Quickselect* algorithm was run 1000 times.

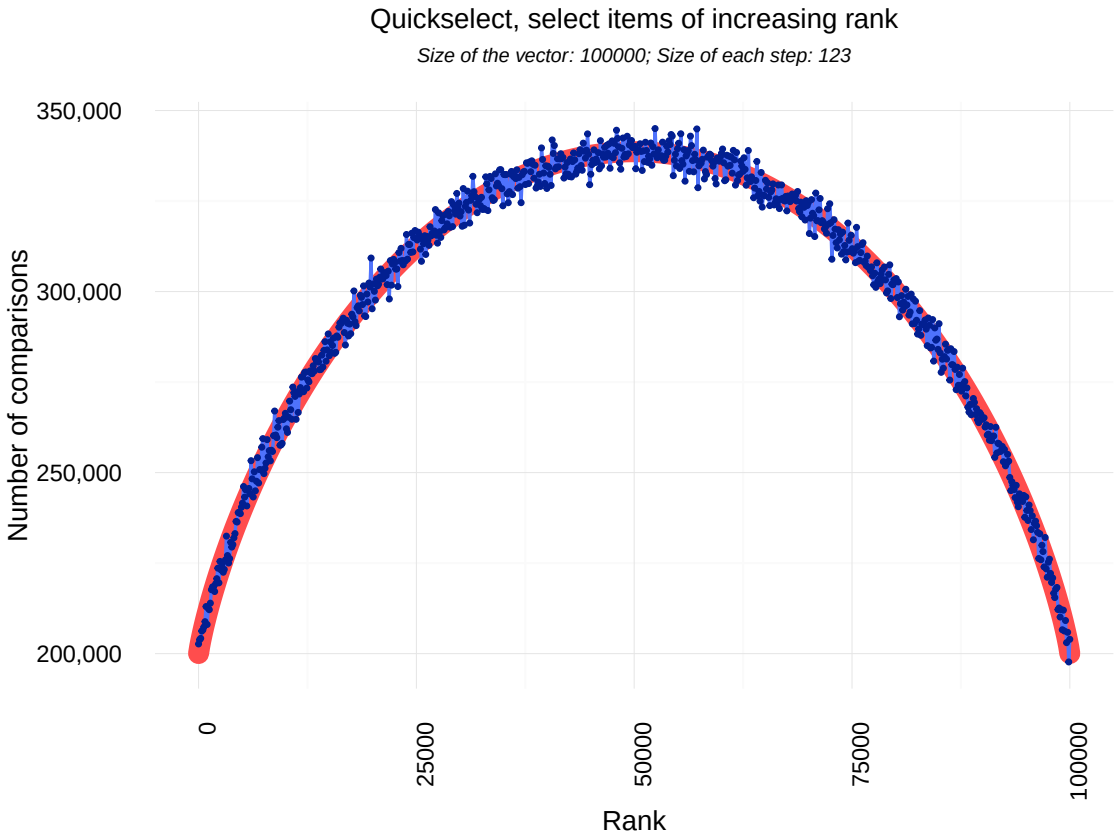

Figure 2: *In blue, the measured number of comparisons. In red, the theoretical number of comparisons.*

The number of comparisons seems to have a parabolic concave shape, with a unique local maximum in $k = \frac{n}{2}$, consistent with the previous theoretical results. Once again, the measured values are inline with the theoretical ones. In this case, the *MAPE* is 0.79%.

4 Addendum

4.1 *Execution time analysis*

While executing the previous tests, it was possible to gather the execution times of the algorithm.
It is interesting to measure how the number of comparisons is related to the effective execution time.

For both the previous tests, execution times were recorded and summarized in the following plots.

The tests were performed on a *Surface Pro 4* with a Intel Core i5-6300U CPU clocked at 2.95 Ghz, and 4 GB of DDR3 RAM at 1867Mhz.

In the case of increasing list size n, with $k = \frac{n}{2}$, it emerges once again a linear scaling with respect to n. A linear regression was performed on the measured data: the *slope* of the line has been found to be $4.39 \cdot 10^{-6}$ (note that the execution times are expressed in *milliseconds*), and the *intercept* is -29.02.

If n is kept constant and k ranges from 1 to n, it emerges a parabolic shape, which is coherent with the results found in the second test. The maximum execution time, not considering the obvious outliers, is found for $k = \frac{n}{2}$.
From the measured data, it was performed a order-2 polynomial regression. The parabola found by the regression has parameters $[-1164.60, -46.11, 364.06]$, and the measured *MAPE* is 4.39%.

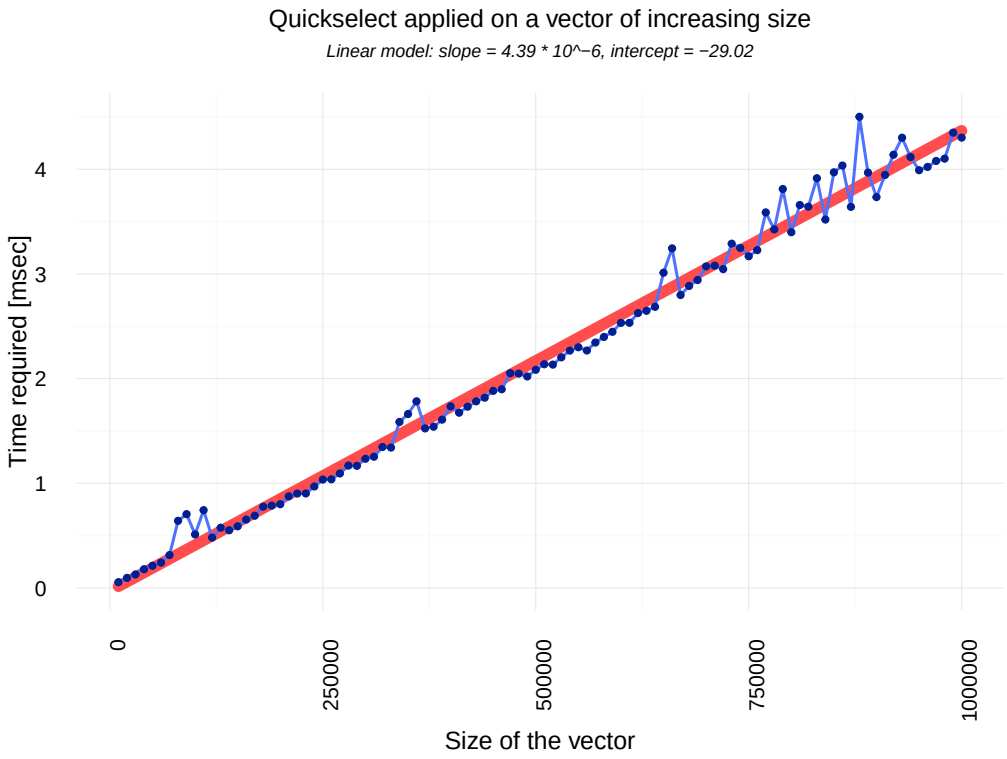

Figure 3: *In blue, the measured execution time. In red, the linear regression of the execution time.*

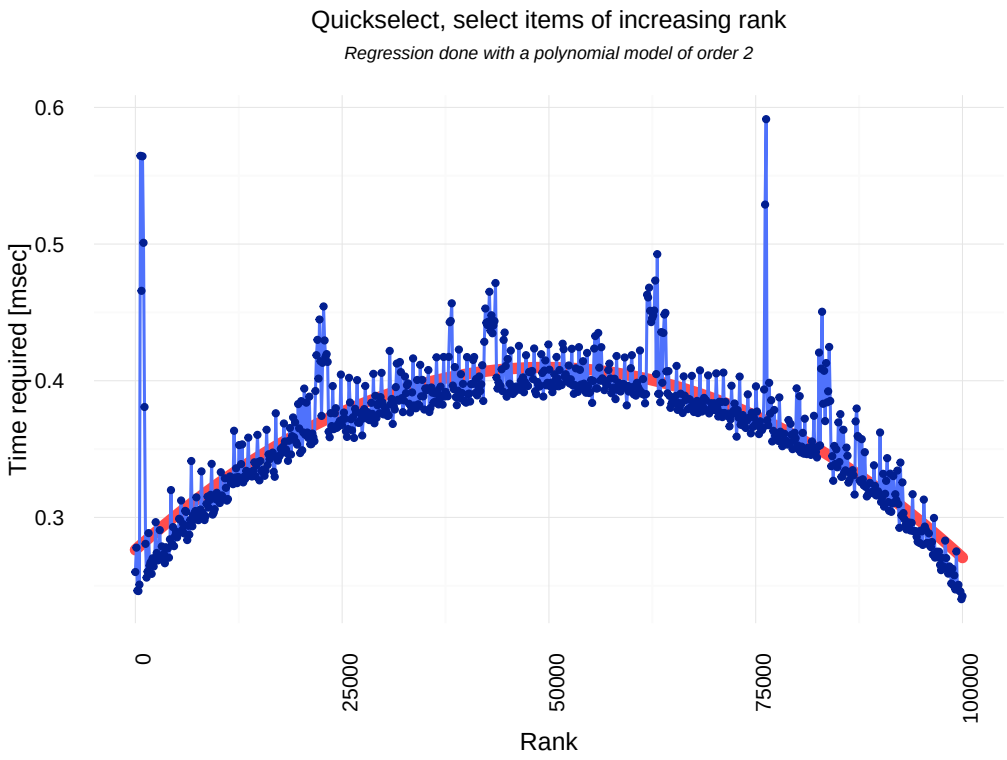

Figure 4: *In blue, the measured execution time. In red, the order 2 polynomial regression of the execution time.*

• **Full code (Rust implementations + R analysis) available at https://github.com/AlbertoParravicini/data_structures_and_algorithms**

References

[1] Mean absolute percentage error. URL: https://en.wikipedia.org/wiki/Mean_absolute_percentage_error.

[2] Quickselect. URL: https://en.wikipedia.org/wiki/Quickselect.