Randomized Algorithm : Homework 1

Romain Markowitch (000540172)

24 October 2024

Contents

1 Introduction

For the course INFO-F413, Randomized algorithms, we were asked to implements two algorithms :

- LazySelect from the the pseudo-code in Chapter 3 of the textbook, Motwani and Raghava where they improved it.
- QuickSelect

The goal of the homework is to compare both algorithms in their time to complete their task and the number of comparisons they do to select, with justification, the one to use in practice. We also need to optimize them to respect a certain number of comparisons for LazySelect.

2 Experimental plan

2.1 Implementations

2.1.1 QuickSelect

To implement this algorithm, we were able to choose from where we would read its pseudo-code.
We decided to use Wikipedia as referenced in the code. See appendix A for the source and the code.

To be able to compare the number of comparisons, we had to slightly modify the code, by creating new variables as comparisons or creating a class to help keep the value of comparisons. It helped to test it and count the number of comparisons. So we created a class QuickSelect to help us count.

2.1.2 LazySelect

To implement LazySelect, as announced in the introduction, we had to base our code from the textbook from Motwani and Raghava where they improved the lazyselect to be able to run in time 2n + o(n) with high probability.
As for the QuickSelect we chose to implement it as a class to count the number of comparisons.
But to be able to reduce the number of comparisons as much as possible, a lot of changes were needed, as well as to have a 100% success rate for the algorithm.

First of all, to succeed at all time, we had to change some conditions from the pseudo-code :

- For the 4th step, we have to check if the length of p is greater than k - rank of a which is greater than 0. To be sure of our indexes.
- Then if the former condition (added to the code from the pseudo-code) was true, and if p was smaller than $\hat{n}(1/4)$ we return $p[k]$

With those conditions we can safely have a 100% success rate.

Secondly, to reduce the number of comparisons, we chose to calculate at once, rS_a, rS_b and the list p (see appendix B figure 14 to see the code). They are used to select the k-th element from the list S given. They were in the original version (see appendix C) calculated separately while each time going through S. As the list p change from the condition, too many comparisons would have been necessary to go through S each time, so we reduced it at only one list iteration.
But as much as it reduce the comparisons, it had to be implemented. So we analyzed the lists p to create, rS_a and rS_b and we conclude that with 2 ifs we could do it.

- rS_a, the number of elements lower than a (see figure 14 for a better understanding).
- rS_b, as rS_a but for elements lesser or equal to b.
- The lists to create p. 3 lists are needed for the 3 conditions to create p. The first one is for the element lesser or equal to b, the second one is from the elements greater than a and the last one is for the element in the interval $[a, b]$.

To do it, we just had to check if each element of S is lower than a, lower or equal to b or else. And then increase the counters and add the elements to the correct list(s).

Concerning rS_b, as in the pseudo-code, it is not used but as we need values equivalent to the comparisons of the rank of b, we keep calculating it. Those values are used in the creation of the list p.

2.2 Tests

2.2.1 Environment

All experiments were conducted on a MacBook Air (2024) with a M3 Apple chip (8-core CPU, 10-core GPU, and 16 GB unified memory), using Python 3.10.11 with libraries.

2.2.2 Values

To test the code and be as general as possible, the code will be run a certain number of times, depending on **n**, the size of the list **S** (list of elements given to both algorithms). Below this, we have the tests we ran for the code.

size of S (n)	**value of k**	**number of tests**
500	$[1, 500[$	100
5 000	$[1, 5000[$	100
10 000	$[1, 10000[$	100
100 000	$[1, 100000[$	100
1 000 000	$[1, 1000000[$	10
10 000 000	$[1, 10000000[$	10

2.2.3 Implementation

To implement the tests, we created a loop that did the $number\_of\_tests$ times each algorithm and that, for each k between 1 and and the size of S-1.

We used the results of those loops to create the graphs with the average value for each k-value and size of S.

The values that we computed were retrieved from different libraries as math, to calculate some logarithms (number of comparisons for a sort), time, to calculate the time taken for each select, random, to get a random order of numbers in S and matplotlib to create the graphs.
To calculate the time, we started a clock just after the initialization of class of the algorithm tested but still before the start of the selected function and we stopped the clock when the result was returned.
Concerning matplotlib, it helped us get a better idea and view of the different comparisons between both algoithms.

3 Results

3.1 Comparisons

Below are graphs comparing the number of comparisons made by QuickSelect and LazySelect, to find the k-th element for different sizes of S (500 to 10 million elements) and several different k.

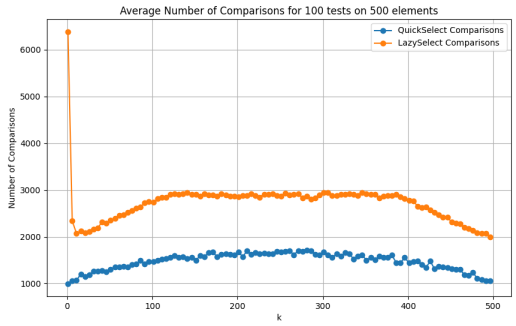


Figure 1: Number of comparisons to find the k-th element out of 500 between QuickSelect and Lazyselct

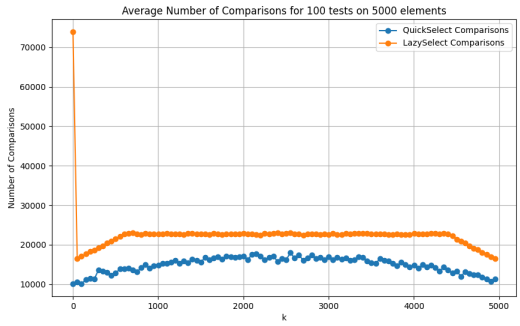


Figure 2: Number of comparisons to find the k-th element out of 5000 between QuickSelect and Lazyselct

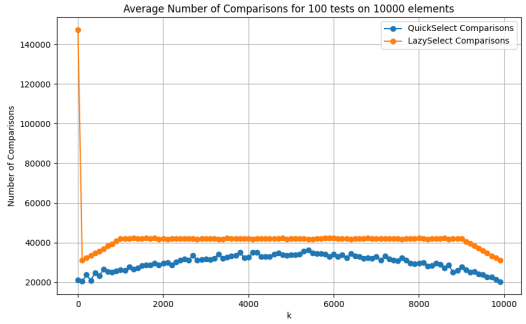


Figure 3: Number of comparisons to find the k-th element out of 10000 between QuickSelect and Lazyselct

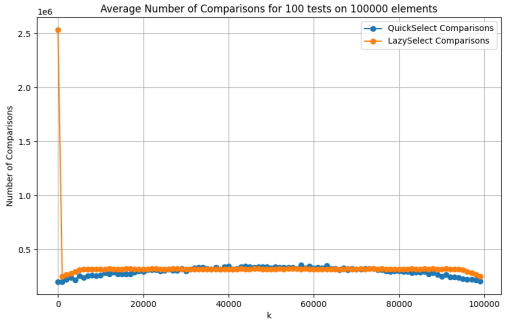


Figure 4: Number of comparisons to find the k-th element out of 100000 between QuickSelect and Lazyselct

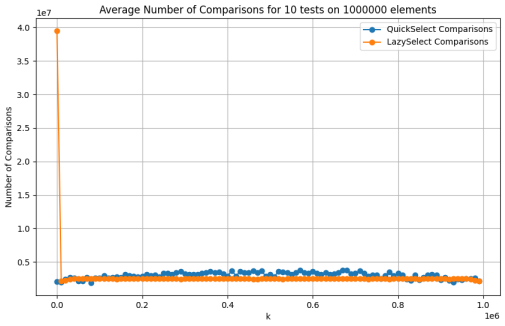


Figure 5: Number of comparisons to find the k-th element out of 1 million between QuickSelect and Lazyselct

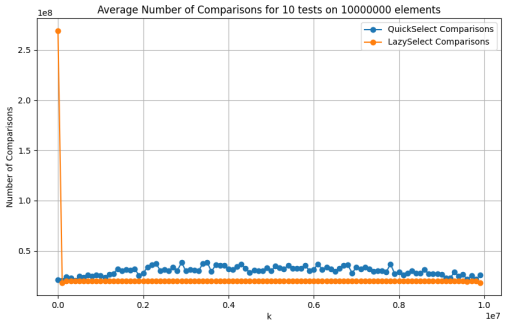


Figure 6: Number of comparisons to find the k-th element out of 10 million between QuickSelect and Lazyselct

3.2 Time

Below are graphs comparing the time taken for QuickSelect and LazySelect, to find the k-th element for different sizes of S (500 to 10 million elements) and several different k.

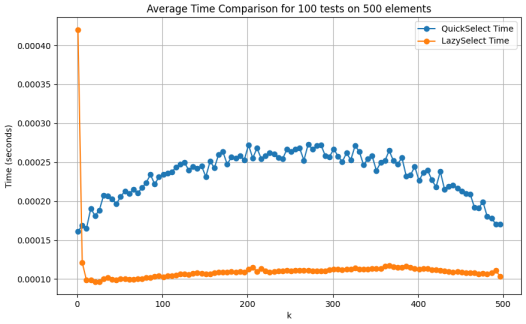


Figure 7: Time comparison between QuickSelect and Lazyselct for finding the k-th element out of 500

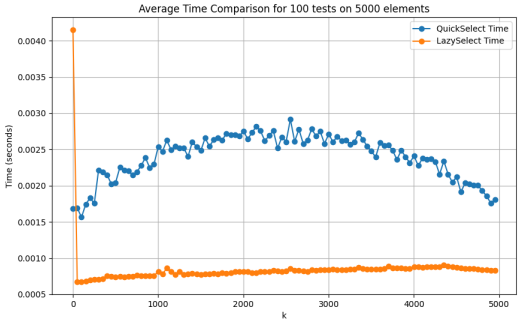


Figure 8: Time comparison between QuickSelect and Lazyselct for finding the k-th element out of 5000

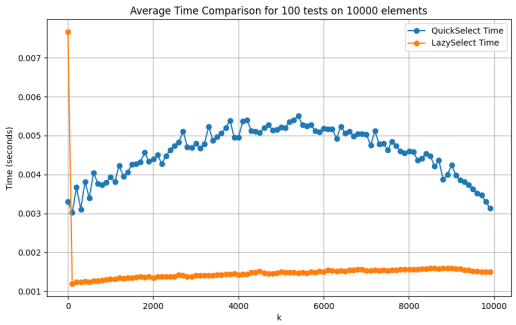


Figure 9: Time comparison between QuickSelect and Lazyselct for finding the k-th element out of 10000

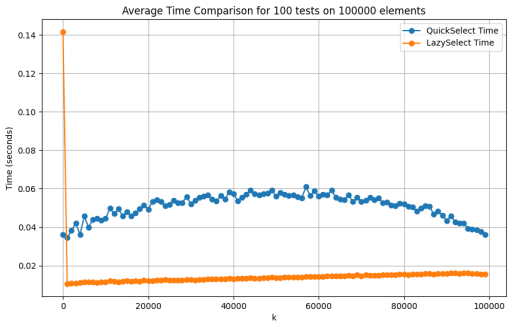


Figure 10: Time comparison between QuickSelect and Lazyselct for finding the k-th element out of 100000

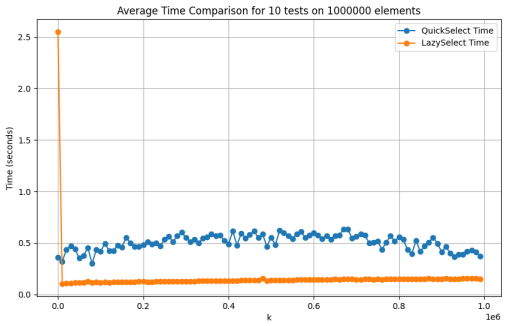


Figure 11: Time comparison between QuickSelect and Lazyselct for finding the k-th element out of 1 million

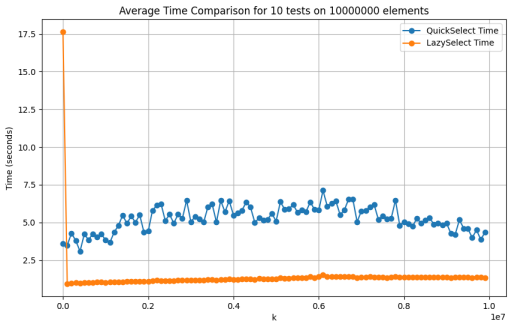


Figure 12: Time comparison between QuickSelect and Lazyselct for finding the k-th element out of 10 million

4 Analyze

Here we will compare the results (see section 3) from our tests and discuss them.

In this section, we will use **n** to compare the number of comparisons. **n** is the number of elements of **S**. S is the list where we have to find the k-th element.

4.1 Comparisons

As we can see in section 3.1, the number of comparisons for the QuickSelect is around 2n + o(n) and 4n. The more k is near the middle of S, the higher the number of comparisons will be. That observation is the same for lazySelect, but instead of staying between 2n and 4n, it can go to 6n for smaller size of S.

Another observation is that QuickSelect will not change its lower and upper bounds (2n and 4n) with the size of S while LazySelect will get better results (from 2n to maximum 3n) and have less comparisons with a bigger list S (we talk here for a list of 1 million elements or more). But for lists of 100 000 elements it is already better for k near $n/2$ as see in figure 4.
Moreover, for S of size 1 million and more, the curve of number of comparisons for LazySelect is flatter (the number of comparison is more constant, independently of k) than QuickSelect and with that it gets better results.
With those big lists S, LazySelect starts to stay in the bounds $2n + o(n)$ and 3,386n which is the most comparisons it should do.

We can also observe that each time $k = 0$ takes a lot of comparisons for LazySelect which is not the case for QuickSelect. The reason should be that it loops a lot for this case. This must be because the code has not been optimized enough for this peticular case.

4.2 Time

In contrast to section 4.1, we can see in section 3.2 that while comparing the time taken for each algorithm depending on the number of elements and the k-th element to find, each time, the lazySelect algorithm is faster. LazySelect is on average 2.5 times faster than QuickSelect. Except for the special case k=0 which is normal. A huge difference in comparison brings a difference in the time taken to find k-th element.

We can see that while QuickSelect has a curve similar to its number of comparisons, LazySelect stays mostly constant while slightly increasing.
But for QuickSelect this curve gets flatter for S between 100000 and 1 million elements but then returns to normal curve from before 100000 elements.

For LazySelect, the increase in time taken to find the k-th element is linearly similar to the size of S (if S is 10 times bigger, the time taken will be 10 times greater as well).

5 Conclusion

In conclusion, we did see some differences between both algorithms. LazysSelect is faster but needs more comparisons than QuickSelect for lists S smaller than 1 million elements. We can also see that LazySelect, in opposition to QuickSelect, is more constant in the time taken to find the k-th element. but they both have a curve (bigger on the middle where $k = n/2$) for their number of comparisons.

While QuickSelect should be the best choice for smaller lists if you need less comparisons, LazySelect stays better in the time taken and greatly improve its number of comparisons with a bigger size of S (100 000 and more where it reach and get better than QuickSelect and its number of comparisons).

So if we have to choose one algorithm to use in practice, we will use LazySelect who will be better most of the time.

Appendix

A QuickSelect

This is the code for the QuickSelect algorithm. It has been copied and adapted in python from the pseudo-code of Wikipedia : QuickSelect.
Source : `https://en.wikipedia.org/wiki/Quickselect`

```python
"""Code inspired from wikipedia : Quickselect -> https://en.wikipedia.org/wiki/Quickselect"""
import random
class QuickSelect:
    def __init__(self):
        self.comparisons = 0

    def partition(self,list, left, right, pivot_index):
        """ Partition the list between the left and the right around the pivotIndex """
        pivot_value = list[pivot_index]
        list[pivot_index], list[right] = list[right], list[pivot_index]  # Move pivot to the end
        store_index = left
        for i in range(left, right):
            self.comparisons += 1
            if list[i] < pivot_value:
                list[store_index], list[i] = list[i], list[store_index]
                store_index += 1
        list[right], list[store_index] = list[store_index], list[right]  # Move pivot to its final place
        return store_index

    def quickSelect(self,list, left, right, k):
        """ Return the k-th smallest element of list within [left:right+1] """
        if left == right:  # If the list contains only one element
            self.comparisons += 1
            return list[left]
        pivot_index = random.randint(left, right)  # select a pivotIndex between left and right
        pivot_index = self.partition(list, left, right, pivot_index)

        # The pivot is in its final sorted position
        self.comparisons += 1
        if k == pivot_index:
            return list[k]
        elif k < pivot_index:
            return self.quickSelect(list, left, pivot_index - 1, k)
        else:
            return self.quickSelect(list, pivot_index + 1, right, k)

```

Figure 13: QuickSelect code

B LazySelect

This is the code for the LazySelect algorithm. It is inspired and adapted from the pseudo-code present in the Chapter 3 of the textbook, Motwani and Raghava. See appendix C. Some personal modification have been done to reduce the number of comparisons and time to finish the algorithm and give the correct answer.

```python
import random
import math

class LazySelect:
    def __init__(self):
        self.comparisons = 0

    def rank_ab(self, S, a, b):
        """ Return the ranks of 'a' and 'b' in one pass with fewer comparisons
        and the different version of the list p """
        self.comparisons += len(S)
        rS_a = 0
        rS_b = 0
        greater_a = [] # all elements greater or equal to a
        between_ab = [] # all elements between a and b (inclusive)
        lesser_b = [] # all elements lesser or equal to b
        for elem in S:
            # Compare elem with 'a' and 'b', while avoiding duplicate comparisons
            if elem < a:
                rS_a += 1
                rS_b += 1  # Since elem < a, it is also <= b
                lesser_b.append(elem)
            elif elem <= b:
                rS_b += 1
                lesser_b.append(elem)
                greater_a.append(elem)
                between_ab.append(elem)
            else: # elem > b
                greater_a.append(elem)
        return rS_a, rS_b, lesser_b, greater_a, between_ab


```

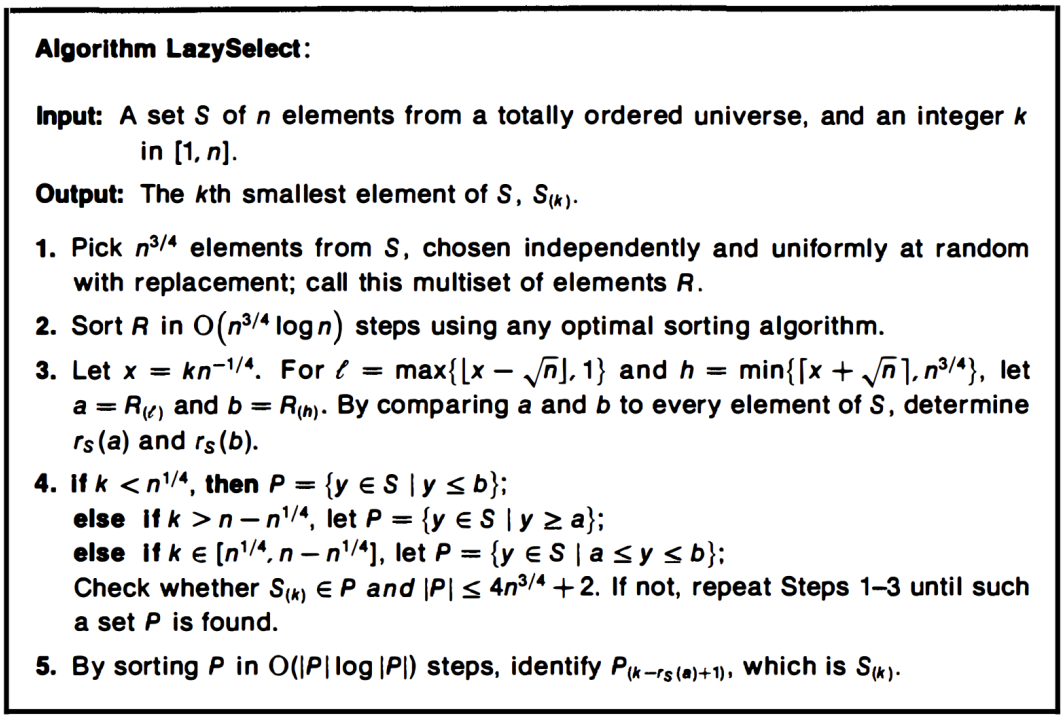
Figure 14: LazySelcect code rank A and B

```python
    def lazySelect(self, S, k)-> int:
        """ Return the k-th smallest element in S """
        n = len(S)
        if k < 1 or k > n:
            raise ValueError("k is out of bounds")
        while True:
            # Step 1: Pick n^(3/4) random elements from S with replacement
            R_size = int(n ** (3 / 4))
            R = [random.choice(S) for _ in range(R_size)]

            # Step 2: Sort R
            R.sort()
            self.comparisons += R_size * math.log2(R_size)

            # Step 3: Variables
            x = k * (n ** (-1 / 4))
            l = max(math.floor(x - (n ** 0.5)), 1)
            h = min(math.ceil(x + (n ** 0.5)), R_size)
            a = R[l-1]
            b = R[h-1]
            # Find the ranks of a and b
            rS_a, rS_b, lesser_b, greater_a, between_ab = self.rank_ab(S, a, b)

            # Step 4: Define P based on k
            if k < n ** (1 / 4):
                p = lesser_b
            elif k > n - n ** (1 / 4):
                p = greater_a
            else:
                p = between_ab

            # Check if the size of P is manageable
            if len(p) <= 4 * R_size + 2 and  len(p)> k-rS_a > 0:

                # Step 5: Sort P and find the (k - rS(a) + 1)-th element in P
                p.sort()
                self.comparisons += len(p) * math.log2(len(p))
                if k < n ** (1 / 4):
                    return p[k]
                return p[k - rS_a -1]  # Return the k-th smallest element
```

Figure 15: LazySelcect find k-th element

C Pseudo-code LazySelect

Here you will find the pseudo code present in the Chapter 3 of the textbook Randomized algorithms by Motwani and Raghava.
Source : `https://cibleplus.ulb.ac.be/discovery/fulldisplay?context=L&vid=32ULDB_U_INST:32ULB_VU1&search_scope=MyInst_and_CI&tab=Everything&docid=alma991006487889704066`



Algorithm LazySelect:

Input: A set S of n elements from a totally ordered universe, and an integer k in $[1, n]$.

Output: The kth smallest element of S, $S_{(k)}$.

1. Pick $n^{3/4}$ elements from S, chosen independently and uniformly at random with replacement; call this multiset of elements R.
2. Sort R in $O(n^{3/4} \log n)$ steps using any optimal sorting algorithm.
3. Let $x = kn^{-1/4}$. For $\ell = \max\{\lfloor x - \sqrt{n} \rfloor, 1\}$ and $h = \min\{\lceil x + \sqrt{n} \rceil, n^{3/4}\}$, let $a = R_{(\ell)}$ and $b = R_{(h)}$. By comparing a and b to every element of S, determine $r_S(a)$ and $r_S(b)$.
4. **if** $k < n^{1/4}$, **then** $P = \{y \in S \mid y \leq b\}$;
   else if $k > n - n^{1/4}$, let $P = \{y \in S \mid y \geq a\}$;
   else if $k \in [n^{1/4}, n - n^{1/4}]$, let $P = \{y \in S \mid a \leq y \leq b\}$;
   Check whether $S_{(k)} \in P$ *and* $|P| \leq 4n^{3/4} + 2$. If not, repeat Steps 1–3 until such a set P is found.
5. By sorting P in $O(|P| \log |P|)$ steps, identify $P_{(k - r_S(a)+1)}$, which is $S_{(k)}$.

Figure 16: Pseudo-code present in the Chapter 3 of the Randomized algorithms by Motwani and Raghava