Lazy Selection

Carlos Requena López

December 6, 2018

1 Introduction

This assignment tries to empirically verify Theorem 3.5 (here 1) from [1, p. 49], running the **LazySelect** algorithm for different values of n and k to establish the asymptotic behaviour of the number of comparisons performed.

The **LazySelect** algorithm finds the kth smallest element in a set S (assumed to have total order) of size n. It essentially outputs the element with rank k. Concerning notation, $A_{(i)}$ will refer to the element with rank i in set A and $r_A(j)$ will refer to the rank of element j in set A.

2 Analysis and expectations

We detail the steps of the aforementioned algorithm in Algorithm 1.

Algorithm 1: LazySelect

Input: An (unsorted) array S of size n and and integer $k < n$
Output: The element in S that has rank k

1 Take a random sample of $n^{3/4}$ elements of S. Call it R;
2 Sort R;
3 Define $x = kn^{-1/4}$. With $\ell = \max\{\lfloor x - \sqrt{n} \rfloor, 1\}$ and $h = \min\{\lceil x - \sqrt{n} \rceil, n^{3/4}\}$;
4 Assign $a = R_{(\ell)}$ and $b = R_{(h)}$;
5 Determine the rank of a and b in S. That is, find $r_S(a)$ and $r_S(b)$;
6 **if** $k < n^{1/4}$ **then**
  $P = \{y \in S \mid y \leq b\}$
else if $k > n - n^{1/4}$ **then**
  $P = \{y \in S \mid y \geq a\}$
else if $k \in [n^{1/4}\, n - n^{1/4}]$ **then**
  $P = \{y \in S \mid a \leq y \leq b\}$
end
7 Check whether $S_{(k)} \in P$ *and* $|P| \leq 4n^{3/4} + 2$. If not, repeat steps 1 through 6;
8 If the conditions are satisfied: sort P and look for $S_{(k)}$ in P;

We can make some observations about this algorithm:

- By taking R, we hope it will be a good representative of the original set S.

- Any optimal sorting algorithm works for step 2. The number of comparisons performed is sublinear: $\mathcal{O}(n^{3/4}\log(n))$
- x is a sort of rank scaling. Rank x is to R what k is to S (roughly).
- When $k < n^{1/4}$ and $S_{(k)} \in P$, the element is simply $P_{(k)}$.
- The elements are put back to simplify the analysis, since the random variables used become independent.

Also, as proved in [1, p. 49], we have:

Theorem 1 *With probability* $1 - \mathcal{O}(n^{-1/4})$, **LazySelect** *finds* $S_{(k)}$ *on the first pass, and thus performs only* $2n + o(n)$ *comparisons.*

Indeed, finding the rank of a and b in S takes linear time for both a and b, and the sorting or R and P takes sublinear time, hence little-o of n.

3 Implementation and results[1]

Having the version of the algorithm that reports the number of comparisons implemented, we can run a series of test to find the asymptotic behaviour of this number and the distribution of it, for different values of n and k.

For every n and k, we run **LazySelect** a significant number of times, starting at ten thousand times for $n = 100$ and finishing at fifty times for $n = 10^6$. Then we build a histogram to visualise the distribution of the number of comparisons performed.

To determine the asymptotic behaviour, we plot in a line chart the average number of comparisons against the number of elements.

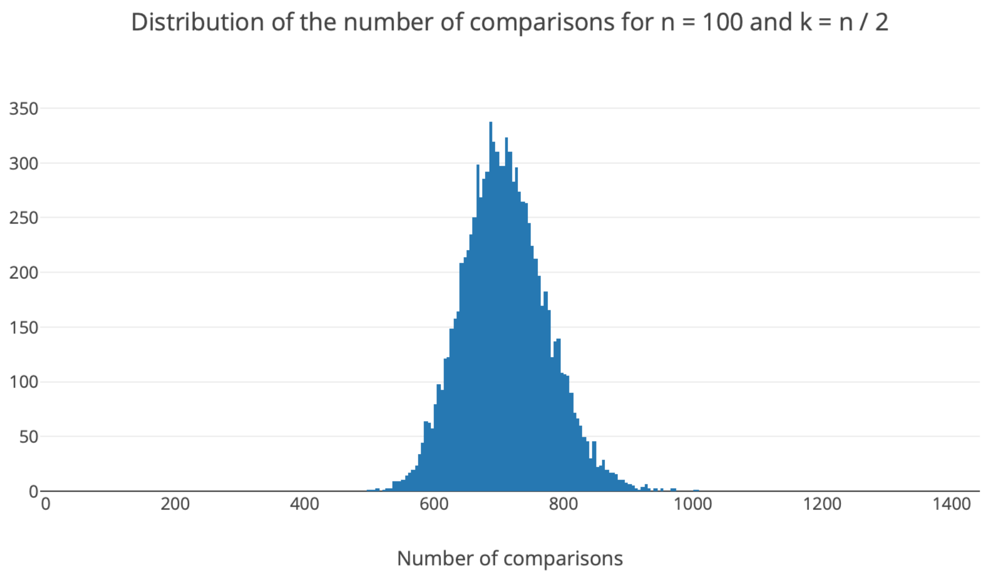

Figure 1

[1]Code can be found at `https://github.com/carlosgeos/lazy-selection`

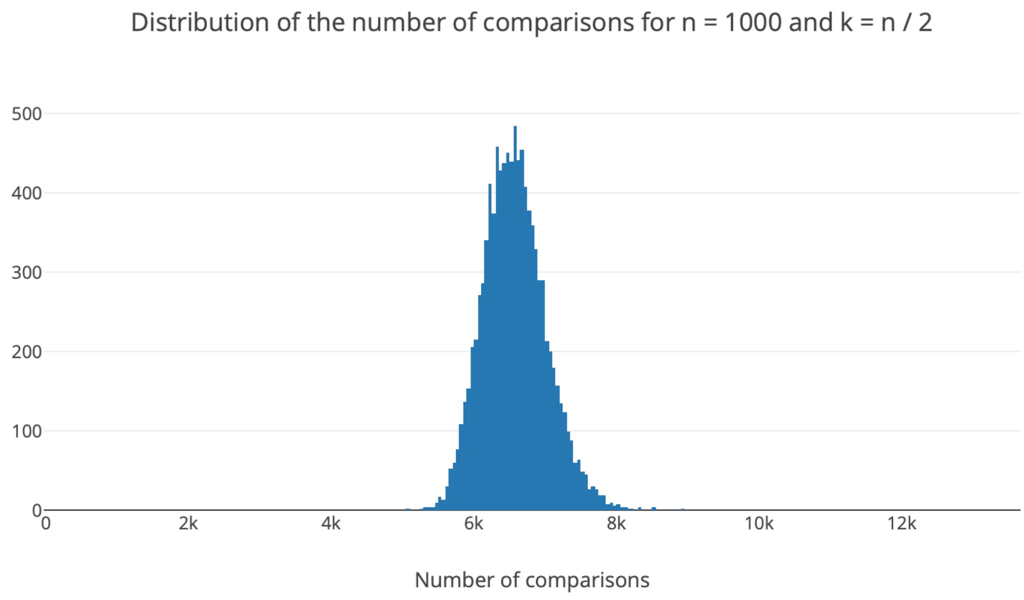

Figure 2

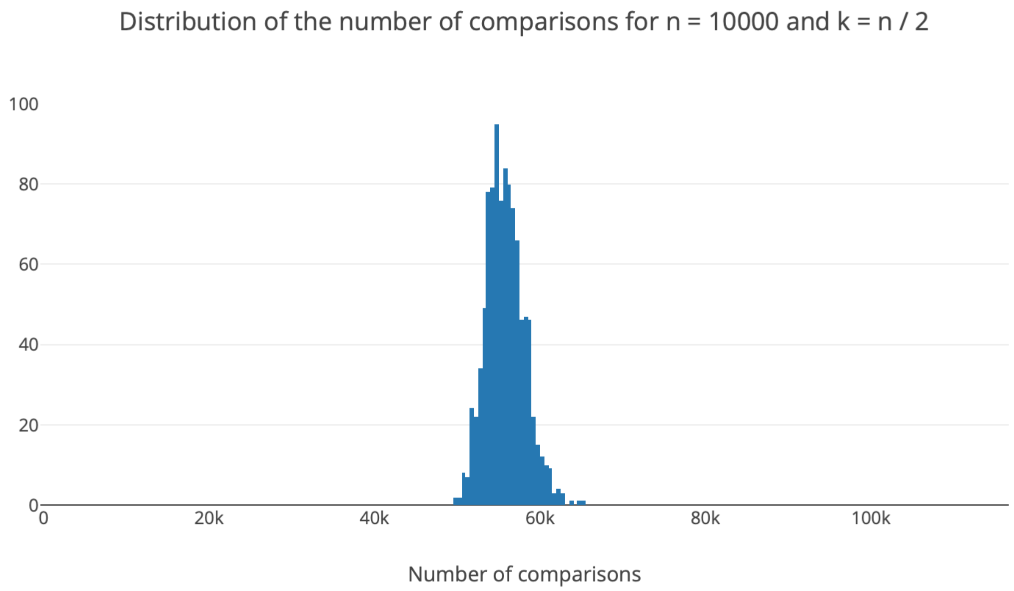

Figure 3

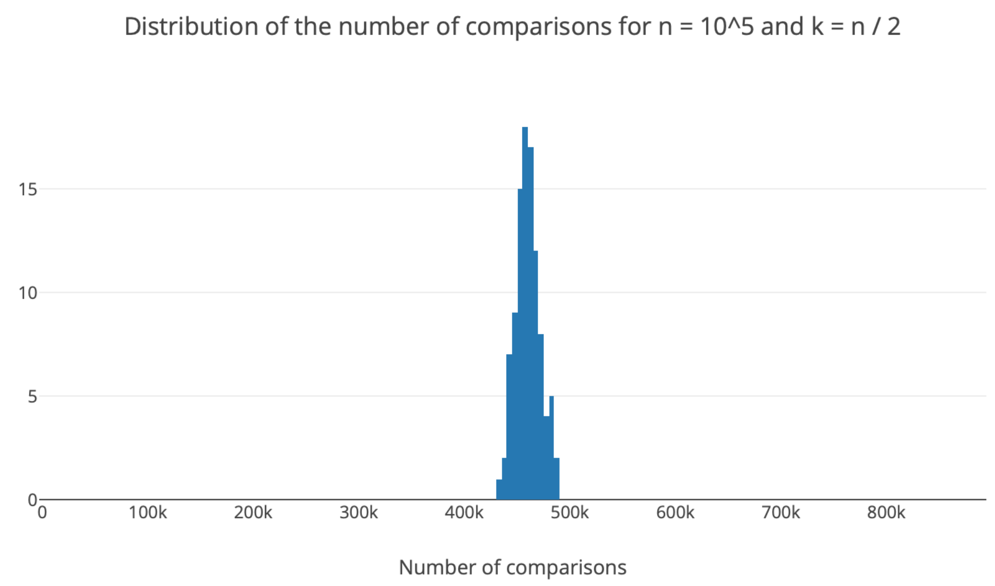

Figure 4

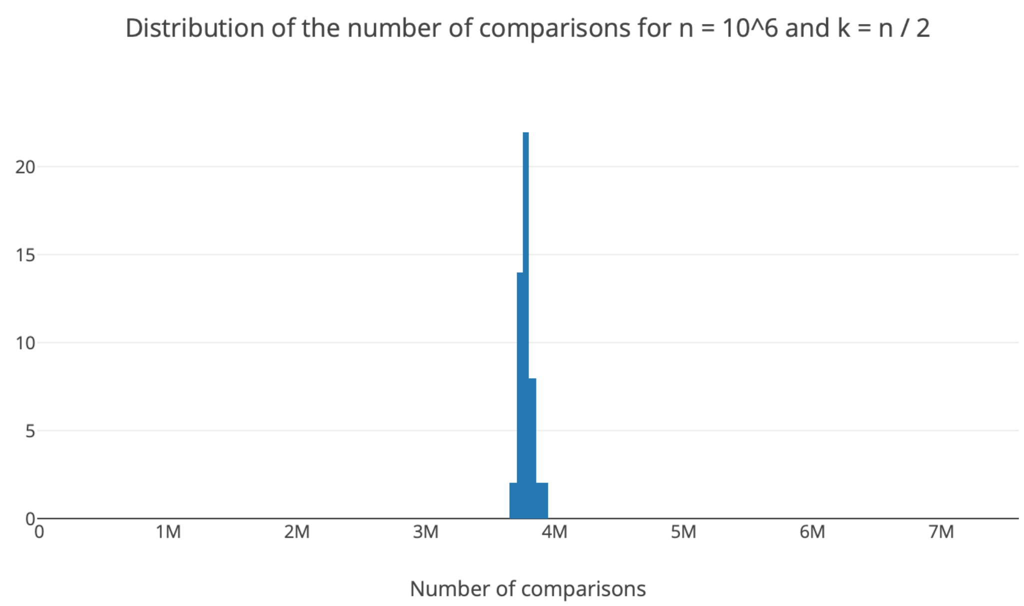

Figure 5

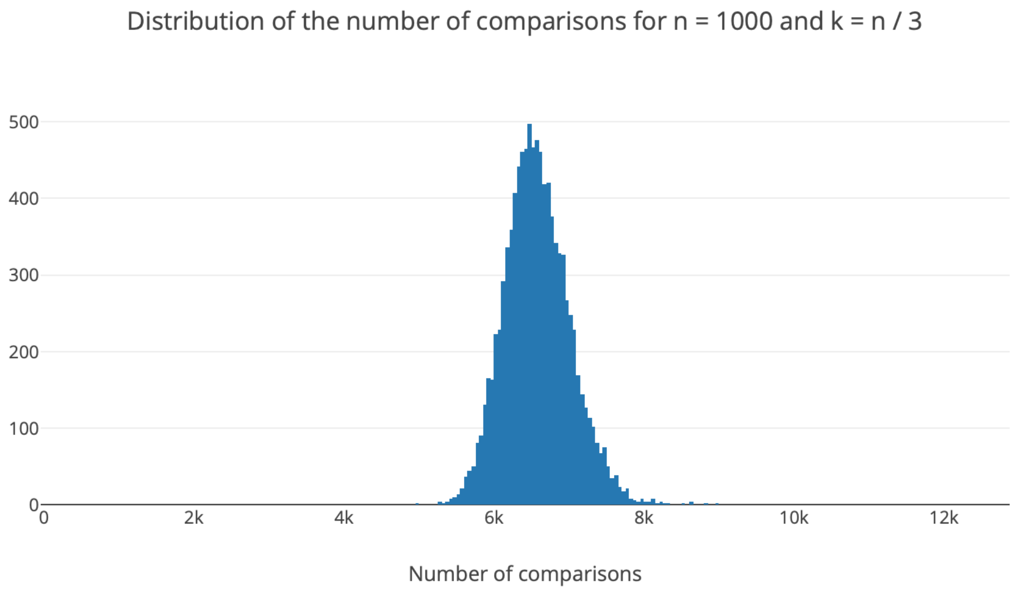

Figure 6

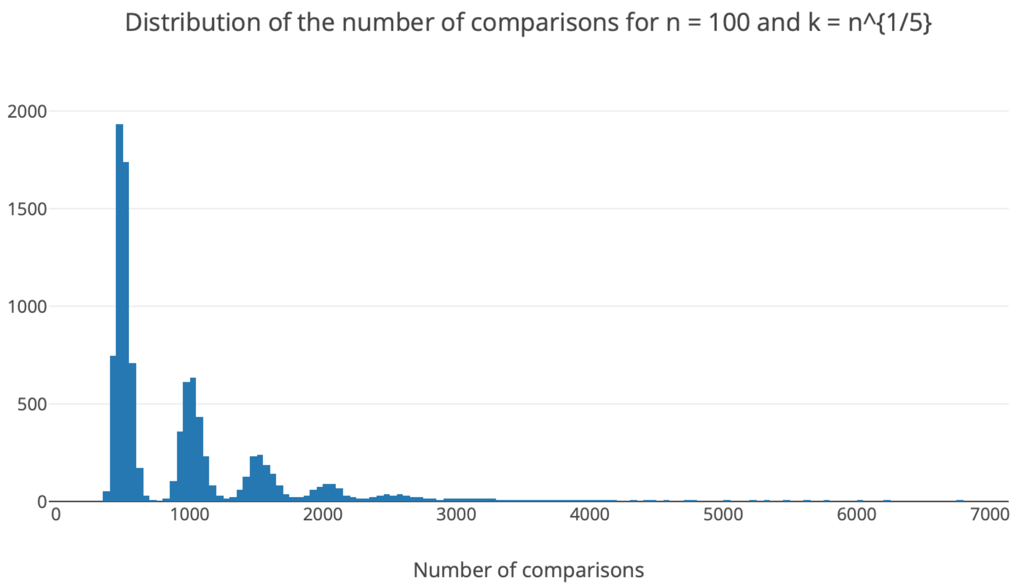

Figure 7

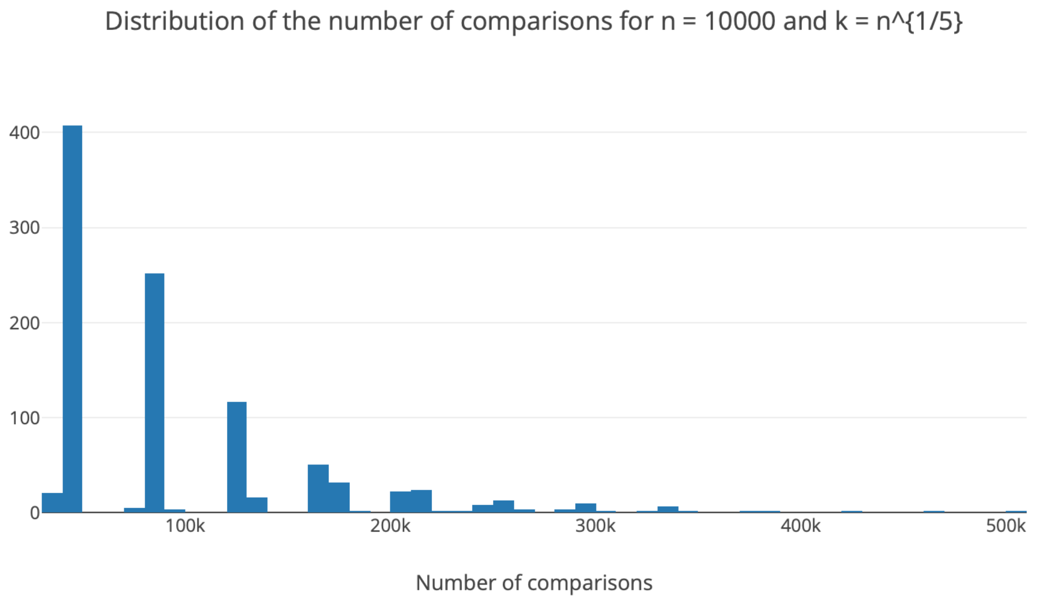

Figure 8

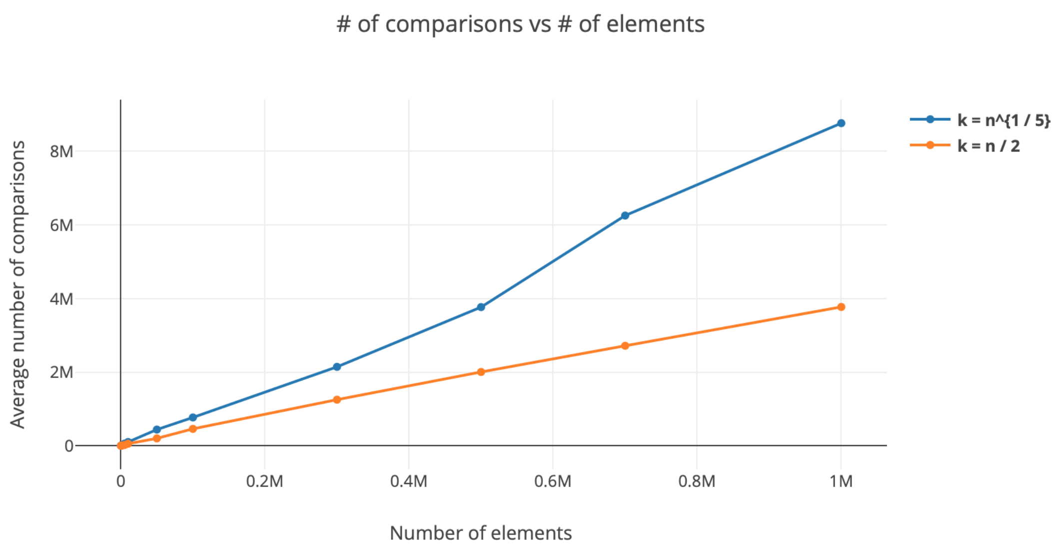

Figure 9

4 Conclusion

The algorithm behaves as described in Theorem 1. If k is set to the median, the histograms show how virtually all runs end up doing $2n + o(n)$ comparisons.

An interesting feature happens in figures 7 and 8. They show how, if k is sufficiently small or large, the algorithm does not satisfy the two conditions and iterates. The decreasing size clusters to the right of the first one show the negative exponential $\mathcal{O}(n^{1/4})$ probability of having to perform successive iterations.

Figure 6 shows that for a moderate k, results are no different than the ones obtained when setting k to the median. The result is the same for other n (not shown).

These tests were all done with a random dataset in which all numbers had the same probability of appearing. However, if we had a skewed dataset, we could guess the results would be similar to those found by tweaking k.

Finally, figure 9 shows the linearity of the number of comparisons as n grows (both axes are linear), and the extra cost of finding a very small or very large element in S.

References

[1] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, New York, NY, USA, 1995.

A Appendix - code listing

core.clj

```
(ns lazy-selection.core
  (:gen-class)
  (:require [clojure.math.numeric-tower :as math]))

(defn random_array
  "Returns a random permutation of an array of n unique elements. In the
  range 1..inf, element i is chosen with probability p (defaults to
  1/2)."
  [n & {:keys [p] :or {p 0.5}}]
  (shuffle (take n (random-sample p (range)))))

(defn rank_of
  "Returns the rank of element e in array a. Takes O(n). Assumes element
  is in array, otherwise it returns the rank it would have if it was"
  [i A]
  (reduce (fn [e1 e2] (if (< e2 i) (+ e1 1) e1)) 1 A))

(defn R
  "Returns R, which are the randomly sampled n^{3/4} elements of array
  S"
  [S]
```

```
  (let [n (count S)
        sample (math/ceil (math/expt n (/ 3 4)))]
    (take sample (shuffle S))))

(defn quick_sort
  "Modified version of quick sort that includes information about the
  number of comparisons"
  [[pivot & coll]]
  (when pivot
    (let [greater? #(> % pivot)]
      (lazy-cat (quick_sort (remove greater? coll))
                [pivot]
                [{:comp (count coll)}]
                (quick_sort (filter greater? coll))))))

(defn sorted
  "Returns only the list of sorted elements"
  [quicksort_output]
  (remove map? quicksort_output))

(defn count_comp
  "Returns the number of comparisons performed in a quick sort"
  [quicksort_output]
  (reduce (fn [e1 e2] (if (map? e2) (+ e1 (:comp e2)) e1))
          0 quicksort_output))

(defn lazy_select
  "Returns the kth smallest element in S, together with the number of
  comparisons performed during sorting and calculating the rank of a
  and b"
  [S k & {:keys [comps] :or {comps 0}}]
  (let [n (count S)
        R (R S)
        q_sort_R (quick_sort R)
        R_sorted (sorted q_sort_R)
        R_comp (+ comps (count_comp q_sort_R))
        x (* k (math/expt n (/ -1 4)))
        l (max (math/floor (- x (math/sqrt n))) 0)
        h (min (math/ceil (+ x (math/sqrt n))) (math/floor (math/expt n (/ 3 4))))
        a (nth R_sorted l)
        b (nth R_sorted h)
        ra (rank_of a S)
        rb (rank_of b S)
        rank_comp (- (* 2 n) 2)
        first_case (atom false)
        P (cond
            (< k (math/expt n (/ 1 4)))
            (do
```

```
                (swap! first_case (fn [a] true))
                (filter #(<= % b) S))
              (> k (- n (math/expt n (/ 1 4)))) (filter #(>= % a) S)
              :else (filter #(and (>= % a) (<= % b)) S))
          q_sort_P (quick_sort P)
          P_sorted (sorted q_sort_P)
          P_comp (count_comp q_sort_P)]
      (if (and (>= k ra)
               (<= k rb)
               (<= (count P) (+ (* (math/expt n (/ 3 4)) 4) 2)))
        (let [elem (if @first_case
                     (nth P_sorted (- k 1))
                     (nth P_sorted (- k ra)))]
          {:elem elem
           :comps (+ R_comp P_comp rank_comp)})
        (lazy_select S k :comps (+ R_comp P_comp rank_comp)))))


(defn -main []
  (let [n 1000
        k (/ n 2)
        ;;k (math/floor (math/expt n (/ 1 5)))
        calls (repeatedly #(lazy_select (random_array n) k))]
    (println
     (let [res (map #(:comps %) (take 10 calls))]
       (int (/ (apply + res) (count res)))))))
```