Alternate Approach of Comparison for Selection Problem

Nikhil Shaw

nikhilshaw_ra@srmuniv.edu.in

SRM University, Chennai

Abstract

This paper proposes an alternate approach to solve the selection problem and is comparable to best-known algorithm of Quickselect. In computer science, a selection algorithm is an algorithm for finding the Kth smallest number in an unordered list or array. Selection is a subproblem of more complex problems like the nearest neighbor and shortest path problems. Previous known approaches work on the same principle to optimize the sorting algorithm and return the Kth element. This algorithm uses window method to prune and compare numbers to find the Kth smallest element. The average time complexity of the algorithm is linear and has the worst case of O(n square).

Keywords: Order statistics, Selection

1. Introduction

The selection problem is defined as follows: given a set X of n elements in an unsorted order, find Kth smallest/largest element in X where k lies between 1 and n. One may generalize the selection problem to apply to ranges within a list, yielding the problem of range queries. In data structures, a range query consists of preprocessing some input data into a data structure to efficiently answer any number of queries on any subset of the input. The question of range median queries (computing the medians of multiple ranges) has been analyzed.

Various approaches are used to solve the problem, selection by sorting, partition based selection and using data structures to select in linear time. The new proposed algorithm works on the principle that given an unordered set X of n elements, Kth smallest element has k-1 elements smaller and n-k elements greater than it in X. Reading set from left and by comparing each element with all other elements to its right, kth element is determined. The algorithm uses an upper and lower limit which are encountered elements just smaller and greater than the kth element. The limits are used to skip over elements for comparison which do not fall inside it. Below we discuss some algorithms to solve the problem.

1.1. Heapselect

Beneficial when the aim is to find the smallest/largest element. A min/max heap can be formed with an insertion operation of O(nlogn) and O(1) for returning the element. However to retrieve Kth smallest/largest element, k return/delete operation(s) has to be performed costing O(klogn).

```
heapselect(A,k)
{
heap H = heapify(A)
for (i = 1; i < k; i++)
    remove min(H)
return min(H)
}
```

1.2. quickselect

Linear performance can be achieved by a partition-based selection algorithm, most basically quickselect. Quickselect is a variant of quicksort in both one chooses a pivot and then partitions the data by it, but while quicksort recurses on both sides of the partition, quickselect only recurses on one side, namely the side on which the desired Kth element is. As with quicksort, this has optimal average performance, in this case linear, but poor worst-case performance, in this case quadratic.

```
quickselect(A,k)
{
pick x in A
partition A into A1<x, A2=x, A3>x
if (k <= length(A1))
    return quickselect(A1,k)
else if (k > length(A1)+length(A2))
    return quickselect(A3,k−length(A1)-length(A2))
else return x
}
```

1.3. Floyd-Rivest Algorithm

```
sampleselect(A,n,k)
{
given n,k choose parameters m,j "appropriately"
pick a random subset A' having m elements of A
x = sampleselect(A',m,j)
partition A into A1<x, A2=x, A3>x
if (k <= length(A1))
    return sampleselect(A1,k)
else if (k > length(A1)+length(A2))
    return sampleselect(A3,k−length(A1)−length(A2))
else return x
}
```

The basic idea is that the closer x is to the Kth position, the more items we'll eliminate in the final recursive call. By taking a median of a sample, instead of just choosing randomly, we're more likely to get something closer to the Kth position.

2. Algorithm

```
sampleselect(A, n, k)
{
   Assign L and U(lower limit and upper limit) to k-1 and n-k,
   El and Eu(Encountered element just lower and just larger than A[k])
   to -inf and +inf respectively
   for(i=0; i<n; i++)
   {
      if(i equals n−1)
      { A[i] is the Kth element)}
      if (A[i] doesn't lie between El and Eu)
      {
         if(A[j]<E1)
         {L= L−1}
         else
         {U= U−1}
         continue the loop
      }
      Assign both Cs and Cl (counters) to 0
```

```
            for(j=i+1; j<n; j++)
            {
                if(arr[j] < arr[i])
                { Cs = Cs+1}
                else
                {Cl = Cl+1}
                if (Cs > L)
                {
                    U = U−1
                    Eu= A[i]
                    break
                }
                else if (Cl > U)
                {
                    L = L−1
                    El = A[i]
                    break
                }
            }
            if(Cl equals El) and (Cu equals Eu)
            { A[i] is the Kth element}
        }
    }
```

The basic idea is that for Kth element, there can only be k-1 smaller and n-k larger elements. The algorithm uses a window El and Es to skip comparing all of the element of the array. Let L(i) be length of the window (El − Es), then

$$L(i) <= L(i+1) \tag{1}$$

3. Analysis

The algorithm works faster in cases when K is either close to 1 or to the size of the array. In worst case, it will check all of the elements of the integer array to reach the solution. Best case is when the Kth element is in the first position of an unsorted array. Time Complexity in such a case is O(n). On the other hand, when Kth element is present at the end of the unsorted array and elements are arranged in alternate order smaller and greater than the

Kth element (from greatest to smallest) time complexity will be O(n square). Below are examples to illustrate the arrangement.

Arrangement: **6** 4 7 1 9 0 11 (unsorted)

n=7, k=4

0 1 4 **6** 7 9 11 (sorted)

Figure 1: Best case

Arrangement: 11 0 9 1 7 4 **6** (unsorted)

n=7, k=4

0 1 4 **6** 7 9 11 (sorted)

Figure 2: Worst case

The algorithm performs better when K is closer to 1 or the size of the array. Below diagram illustrates the performance with the variation of K.

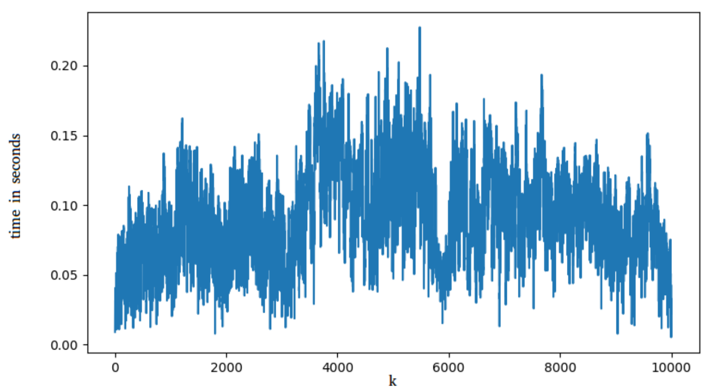

Figure 3: Variation of completion time for different values of k processed on 4x Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz.

Below is comparison of the proposed algorithm with insertion sort and quick select algorithm.

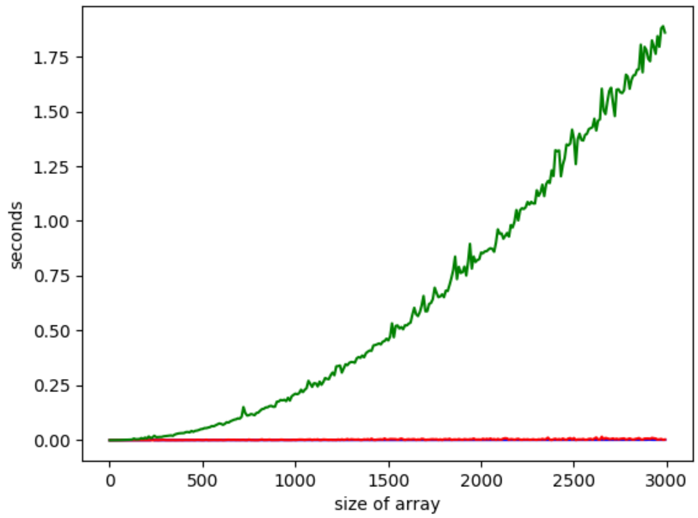

Figure 4: Variation of completion time for k=1 for proposed algorithm (blue), insertion sort (green) and quick select (red) processed on 4x Intel(R) Core(TM) i5-4210U CPU @ 1.70GHz.

4. Conclusion

The algorithm works in linear time when Kth element is present in the starting position or when elements just smaller or greater than Kth element is present in the initial positions of the unsorted array. Although the proposed algorithm is comparable to quick select algorithm, quick select works better in certain cases.

5. Reference

[1] http://www.sgi.com/tech/stl/nth_element.html

[2] http://www.ics.uci.edu/ eppstein/161/960125.html

[3] https://en.wikipedia.org/wiki/Selection_algorithm