Radix Selection Algorithm for the *k*th Order Statistic

Cayman Mitchell, Nelson Schoenbrot, Joshua Shor, Keith Thomas, Sung-Hyuk Cha

Computer Science Department, Pace University
1 Pace Plaza, New York, NY 10038 USA
{kt87309n,js01380p,ns75228n,cm36262n,scha}@pace.edu

Abstract. Finding the *k*th order statistic is an important sub-problem in many complex problems such as nearest neighbor, shortest path, etc. While linear time algorithms, which utilize the divide and conquer paradigm, are known, they require an extra pivot-selection step. Here is a simple linear time algorithm, which utilizes the *Radix sort* algorithm model. First, elements are separated into a constant number of subgroups, which in this case are individual bits, and the algorithm partitions the data according to those subgroups. Searching for the element continues recursively on the corresponding subgroup only. The expected and worst case time complexities are $O(n)$ and $O(bn)$, respectively, where b is the number of bits in the number.

1 Introduction

Finding the *k*th smallest item in a set, A of n numbers is known as the *k*th *order statistic* or *k*th *selection* problem [1]. It is formally defined as:

Input: A set A of n numbers and an integer k, with $1 \leq k \leq n$.
Output: The element $x \in A$ such that $k - 1$ other elements of A are $\leq x$
and $n - k + 1$ other remaining elements of A are $\geq x$.

Special case problems include the minimum ($k = 1$) and maximum ($k = n$) problems, which can be solved in linear time relatively simply. A straightforward algorithm for the *k*th order statistic problem is first sorting the input array A, followed by selecting the *k*th element in the sorted array, as illustrated in Table 1 where $k = median$.

Table 1. Illustration of the straightforward algorithm for the $k = 5$ (*median*) order statistic.

A =	2	11	12	4	5	13	1	7	3
Sorted A =	1	2	3	4	(5)	7	11	12	13

This naïve algorithm requires the hard sorting step whose lower bound is $O(n\log n)$ and thus the computational running time complexity of this simple algorithm is also $O(n\log n)$.

Designing a linear time selection algorithm to find any k, as opposed to the special case maximum or minimum, is more difficult. Hoare, the inventor of the famous sorting algorithm *Quick sort*, also introduced a comparable selection algorithm, which utilizes a random pivot point [2]. Hoare's selection algorithm is also known as *Quick select.* The algorithm recursively partitions the array into two sub-arrays using a random pivot. The running time of *Quick select* in the average case is $O(n)$ [1]. Its worst case running time, though rare, is $O(n^2)$, and occurs when every chosen pivot point is the lowest or highest number in the array. Blum, Floyd, Pratt, Rivest, and Tarjan devised an algorithm (BFPRT) where the time complexity is linear even in the worst case [3]. This BFPRT algorithm is identical to the *Quick select* algorithm, except that a "good pivot" is used instead of a random pivot. For each partitioning step, a linear time subroutine to select a good pivot is executed.

Here, a simple linear time *Radix select* algorithm is presented. This algorithm can be considered very similar to partition-based algorithms, such as *Quick select* and BFPRT, but it does not require an actual pivot. The next section describes the *Radix select* algorithm from a partition-based perspective for the sake of comparison. The *Radix select* algorithm also resembles *Radix sort* [1] whose time complexity is $\Theta(bn)$. Both *Radix sort* and *Radix select* utilize counting elements. Section 3 describes the *Radix select* algorithm in terms of counting.

2 Radix Select vs. Partition-based Algorithms

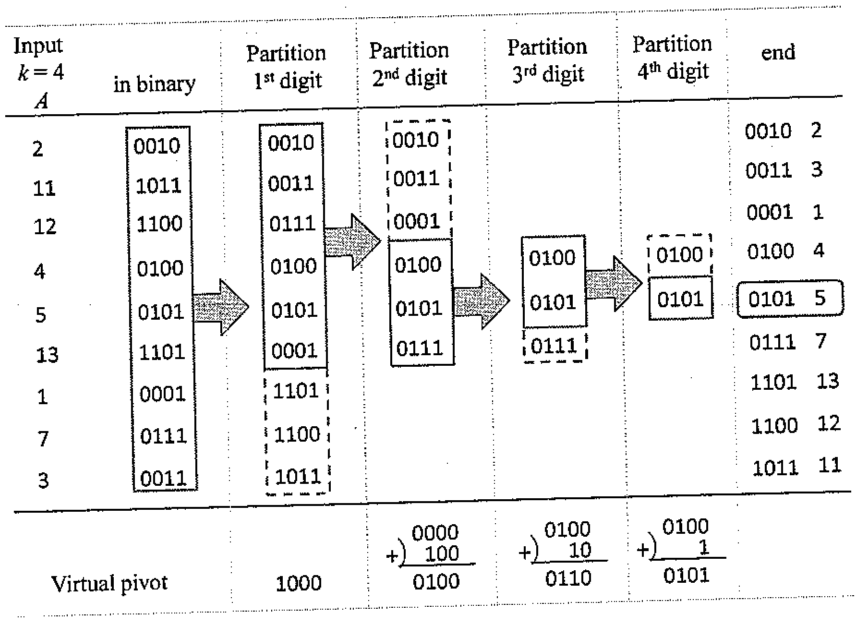


Figure 1. The bitwise selection algorithm illustration.

Most computers use a binary (*radix* = 2) representation of numbers to store and use internally. The *k*th order statistic problem can easily be solved in an array that uses binary number representation. Suppose all of the numbers in the set A can be represented by b numbers of bits, *e.g.* $b = 4$ in Figure 1. Starting from the most significant bit, one can partition the set into two groups, *i.e.* zero and one groups. If the size of the zero group is larger than or equal to k, the kth smallest element must be in the zero group. Similarly, if the size of the zero group is smaller than k, then the kth smallest element must be in the one group. Ergo, only one of the groups is solved recursively. Figure 1 illustrates the *Radix select* algorithm when $k = 5$.

The following pseudo code for the radix selection algorithm resembles the *Quickselect* in [1, 2].

Algorithm 1 radixselect(A, p, r, k, j)

```
1  if $j = b$
2      return $A[p]$
3  $q$ = bit-partition($A, p, r, j$)
4  if $k < q$
5      return radixselect($A, p, q-1, k, j+1$)
6  else
7      return radixselect($A, q, r, k, j+1$)
```

The radixselect is initially called with radixselect(A, 1, n, k, 1) and terminates when it reaches the least significant digit $j = b$ and the answer is $A[p]$. At the end of the radixselect algorithm, the array A is not a sorted array but all elements in $A[1, .., k-1]$ are guaranteed to be less than or equal to $A[k]$ and all elements in $A[k+1, .., n]$ are guaranteed to be greater than or equal to $A[k]$.

Let $A_j[i]$ denote the ith bit of the ith element in the array A with $1 \leq j \leq b$ and $1 \leq i \leq n$. Let the most significant digit is 1 and the least significant digit is b. Although there are many ways to partition elements according to zeros and ones, here is one version that does not require any extra space. Unlike the partition based selection algorithms, the radix selection does not require an actual pivot but uses virtual pivots, e.g., 8, 4, 6, and 5 for each step in Figure 1. Figure 2 illustrates this sub-routine.

Sub-routine 1 bit-partition(A, i, j, p)

```
1  while j > i
2      while A_p[i] == 0, i++
3      while A_p[j] == 1, j--
4      if j > i, swap(A[i], A[j])
5  end
6  return i
```

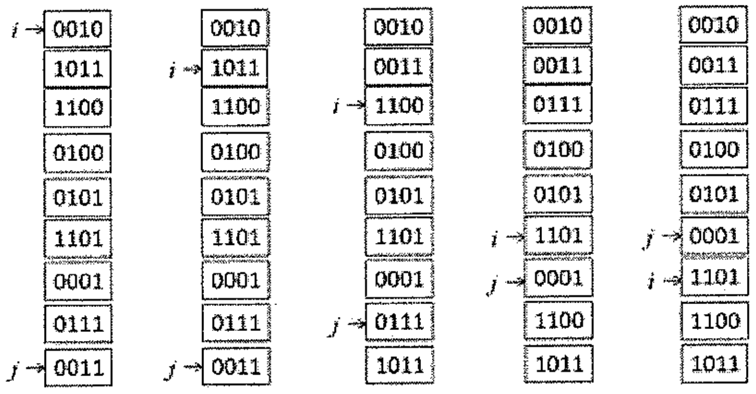


Figure 2. bit-partition(A, 1, n, 1) sub-routine illustration.

The index i increases until it finds 1 in the p bit and index j decreases until it finds 0 in the p bit. Then it swaps the element in ith and jth positions. The sub-routine repeats it until the index i becomes greater than or equal to j. The space required for the radix selection algorithm is $\Theta(bn)$ bits or simply linear, $\Theta(n)$ if b is considered to be a small constant number.

Theorem 1: The worst case running time complexity for the algorithm 1 radix selection is $\Theta(bn)$.

Proof: Suppose that the size of the partitioned array is the same as the previous array. For each bit, exactly n amount of comparisons need to be made and there are b number of bits. Hence, the worst case running time complexity is $\Theta(bn)$. It occurs when there are few unique elements in the input Array. ■

Theorem 2: The expected running time complexity for the algorithm 1 radix selection is $\Theta(n)$.

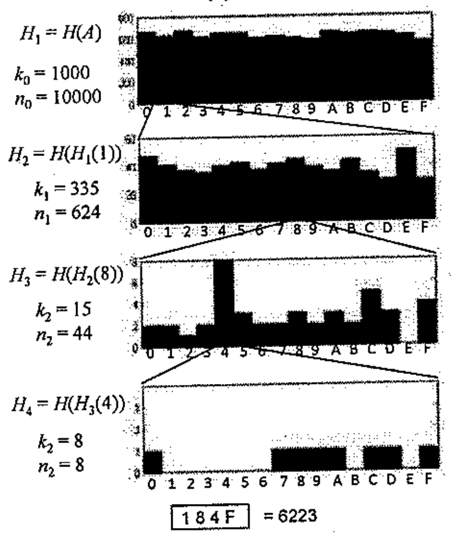
Proof: The expected size of the partitioned array is the half of the previous array. Then the running time function follows the geometric series:

$$n+\frac{n}{2}+\frac{n}{2^2}+\cdots+\frac{n}{2^{b-1}}=n\sum_{i=0}^{b-1}\frac{1}{2^i}=\left(2-(1/2)^{b-1}\right)n<2n.$$

3 Radix Selection Algorithm by Counting

Input	Hexa-decimal	$H_1(1)$	$H_2(8)$	$H_3(4)$	$H_4(F)$	
$k = 1000$	$k_0 = 1000$	$k_1 = 335$	$k_2 = 15$	$k_2 = 8$		
$n = 10000$	$n_0 = 10000$	$n_1 = 624$	$n_2 = 44$	$n_2 = 8$		
2679	0A77	1B15	180F	184B	1840	
6933	1B15	1F2B	184B	184A	1847	
2932	0B74	13FA	18B2	184C	1848	
48777	BE89	184B	⋮	1847	1849	
59	003B	⋮	⋮	1848	184A	
⋮	⋮	⋮	⋮	1840	184B	
⋮	⋮	⋮	⋮	184F	184C	
⋮	⋮	⋮	⋮	1849	$\boxed{184F}$	= 6223
⋮	⋮	⋮	189A			
⋮	⋮	1B02	183B			
		109F	1849			
		1230	18A2			
21679	54AF	12A2				
26779	689B					
30178	75E2					
13011	32D3					

(a)



(b)

Figure 3. The radix selection by counting algorithm illustration.

Suppose that elements in the array A are represented in *hexadecimal* which is a positional numeric system with a radix of 16 (r = 16), i.e., R = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}. The *radix selection by counting* or simply *RSC* algorithm resembles the *radixsort* in [1] but the important difference is that the *RSC* starts from the most significant digit whereas the *radixsort* starts from the least significant digit.

In Figure 3, n = 10,000 and k = 1,000. First, the histogram of radix = 16 bins is constructed from the most significant bit. Next, the prefix sum of the histogram is computed. Notice that in $C_1(1)$ bin contains 1,289 in Table 2 and this value exceeds k = 1,000. Hence, only elements that start with '1' are the candidates and there are 624 elements. Now the problem is a new n_2 = 624 and k = 1,000 – $C_1(0)$ = 335 problem. This process is repeated until the end of the digits. Figure 3 illustrates the algorithm and Table 2 shows the respective histograms and their prefix sums.

Table 2. Histograms and their prefix sums in the RSC algorithm.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
H_1	665	624	670	610	643	643	600	619	598	586	655	629	647	640	618	553
C_1	665	1289														
H_2	47	41	37	35	40	42	37	41	44	39	36	43	34	29	50	29
C_2	47	88	125	160	200	242	279	320	364							
H_3	2	2	1	2	8	3	2	2	3	2	3	2	5	3	0	4
C_3	2	4	5	7	15											
H_4	1	0	0	0	0	0	0	1	1	1	1	0	1	1	0	1
C_4	1	1	1	1	1	1	1	2	3	4	5	5	6	7	7	8

The space complexity is $\Theta(n+b)$ and the worst case running time complexity is $\Theta(bn)$. The expected running time is $\Theta(n)$.

4 Algorithm Analysis

As previously stated, several other algorithms address the kth selection problem. *Heap select*, for instance, is commonly used in cases where a more reliable time complexity is required. Heaps are structures in which the smallest or largest (min-heap, or max-heap, respectively) values can be found in *O(nlog(n))* time. Heaps may be implemented to find the kth smallest number by constructing a max-heap of size k, inserting numbers from the array until the heap is full, and then iterating through the rest of the array, inserting any values that are smaller than the maximum of this heap. When the end of the array is reached, the kth smallest number is the root of the heap.

The most commonly used algorithm for this problem, however, is *Quick select*, a partition-based algorithm that is generally faster than most other algorithms. For the experiments of this paper, a variant of *Quick select* was implemented. This variant chooses the pivot using the "median-of-three" approach. In this implementation, instead of choosing an arbitrary number to be the pivot point, the algorithm takes three numbers from the array, and selects median of these three values to be the pivot point. This helps to combat *Quick select's* worst case scenario $O(n^2)$ by ensuring that the chosen pivot point is not the very first or last element of the array.

The worst of the three, *Heap select*, takes *O(nlog(k))* time to find the kth smallest number. This is because insertion of every element into the max-heap takes *O(log(k))*, and the entire array must be looped through once. Thus, since there are n elements, the entire algorithm takes *O(nlog(k))* time [6].

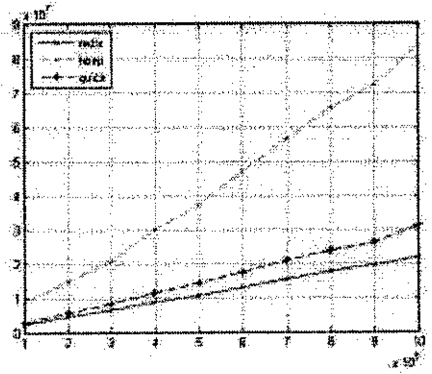
Quick select is an interesting case because it uses a random pivot point. Its best and average case time complexities are *O(n)*; however, its worst case time complexity is $O(n^2)$. Fortunately, the probability of this algorithm reaching that worst case scenario is extremely low, thus creating its relatively consistent results. The probability of the worst case scenario occurring is further reduced by the implementation of the "median-of-three" variant of the algorithm, which helps increase the likelihood of selecting a "good pivot", *i.e.* one that is roughly in the middle of the set of values.

Radix select, as previously discussed, has a best case time complexity of *O(n)*, and a worst case time complexity of $\Theta(bn)$.

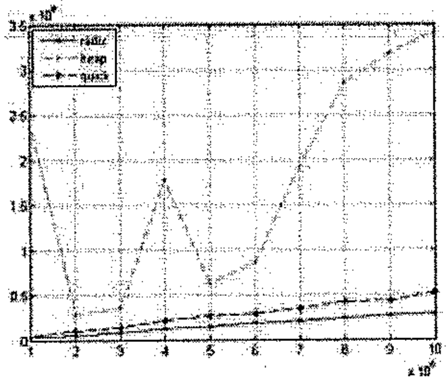
It is evident that *Radix select* has a time complexity equal to *Quick select* in the best case, and a much better time complexity in the worst case. In theory, this should imply that *Radix select* will be faster than *Quick select*.

5 Experimentation

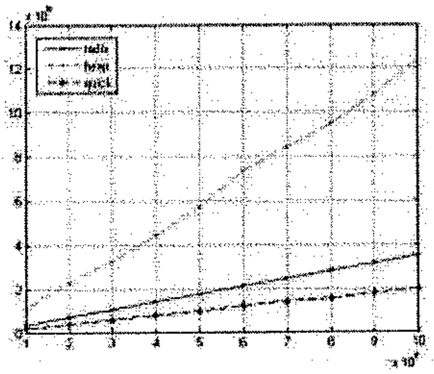
Three separate algorithms were implemented to compare time complexity, iterations, and memory. *Heap select* (using the previously discussed max-heap implementation), *Quick select* with "median-of-three" pivot selection, and *Radix select* using a *binary* implementation. Each algorithm was run with ten different array sizes, ranging from 10,000 to 1,000,000. Each number in that array was set to a pseudorandom value between 0 and $2^{\wedge 31} - 1$. Next, *k* was randomly selected as a number between 0 and the size of the array. The code was implemented so that two values could be recorded: the number of iterations of the algorithm and the amount of time in nanoseconds it took the system to complete. Then each algorithm was executed 100 times for each array size. Graphs of the test results are shown below:



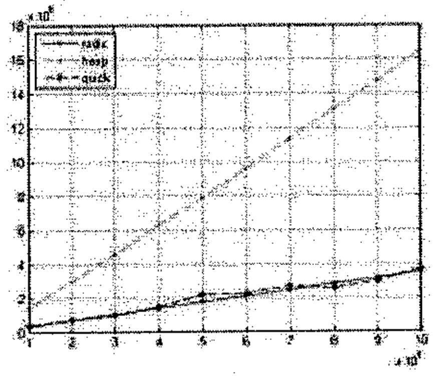
(a) Average Time (in nanoseconds) vs. Array Size.



(b) Worst Case Time (in nanoseconds) vs. Array Size.



(c) Average Iterations vs. Array Size.



(d) Worst Case Iterations vs. Array Size.

Figure 2. bit-partition(*A*, 1, *n*, 1) sub-routine illustration.

6 Conclusion

It is quite obvious that of the three algorithms tested, *Heap select* is the worst, as the amount of time taken and number of iterations were both consistently greater than either *Quick select* or *Radix* select. In addition, *Heap select* had the largest range of values for both time and iterations, meaning that it cannot be considered reliable. Furthermore, *Heap select* will crash on a MacBook Pro running Eclipse with array sizes greater than 800,000, throwing an OutOfMemory Exception.

While *Quick select* and *Radix select* are both *O(n)*, it is evident that *Radix select* has several advantages over *Quick select*. First, *Radix select* is faster on average than *Quick select*; in most circumstances, *Radix select* took less time to find *k* than *Quick select* did. Second, the variability in the amount of time and, particularly, the number of iterations is quite small, meaning that *Radix* is not only faster than *Quick* but it is also significantly more reliable. Both *Quick* and *Radix* experienced problems with memory at around 800,000 as well, however this problem is inconsistent.

Overall, it can be clearly seen that *Radix* can outdo *Quick select* in reliability and speed, while requiring a comparable amount of memory. Thus, *Radix select* ought to become the new standard in *k*th selection algorithms.

Additionally, further research should be done on David Musser's *Introselect* algorithm, which combines *Quick select* with *Heap select* [7], resulting in a worst case time complexity of *O(nlog(n))* [4]. In this situation, if *Quick select* begins to approach its worst case scenario, the program switches to *Heap select*. Perhaps *Radix select* could be used to improve this dynamic algorithm, or perhaps it is already superior.

References

1 Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. Introduction to Algorithms, 2nd ed. MIT Press and McGraw-Hill, 2001.

2 C.A.R. Hoare, "Quicksort", Computer Journal. 5(1):10-15, 1962.

3 M. Blum, R.W. Floyd, V. Pratt, R. Rivest and R. Tarjan, "Time bounds for selection," J. Comput. System Sci. 7(1973) 448-461.

4 "Introspective Sorting and Selection Algorithms," Software—Practice and Experience, Vol. 27(8), 983–993 (August 1997).

5 M. Blum, R.W. Floyd, V. Pratt, R. Rivest and R. Tarjan, "Time bounds for selection," *J. Comput. System Sci.* 7 (1973) 448-461.

6 D. Eppstein. "Selection and Order Statistics." *Donald Bren School of Information and Computer Sciences at the University of California, Irvine*. 09 July 2003. Web. 09 Jan. 2012. <http://www.ics.uci.edu/~eppstein/161/960125.html>.

7 David R. Musser. Introspective sorting and selection algorithms. *Software Practice and Experience*, 27(8):983–993, 1997.

8 "Geometric Series -- from Wolfram MathWorld." *Wolfram MathWorld: The Web's Most Extensive Mathematics Resource*. Web. 24 Dec. 2011. <http://mathworld.wolfram.com/GeometricSeries.html>.