Implicit Selection

Tony W. Lai

Derick Wood

Data Structuring Group
Department of Computer Science
University of Waterloo
Waterloo, Ontario N2L 3G1
CANADA

Abstract

We consider the implicit selection problem, in which we want to select the kth smallest element of a multiset of n elements using only a constant amount of additional space. We show that this problem can be solved in $O(n)$ time in the worst case. In particular, we show that $6.7756n + o(n)$ comparisons are sufficient if all elements are distinct and $6.8280n + o(n)$ comparisons are sufficient in the general case.

1 Introduction

The problem of selecting the kth largest of a multiset of elements from some totally-ordered universe has been the subject of vigorous investigation. For many years it was assumed to be as difficult as sorting, but the linear upper bound of Blum et al. [2] demonstrated this not to be the case. Since it is straightforward to obtain a linear lower bound, one might expect the story to end here. However, this is not the case, since the multiple of n resulting from the first algorithm of [2] is large, indeed it is $19.\dot{3}n$. The hunt was for a faster algorithm. The state-of-the-art is the lower bound of $2n$ comparisons [1] and an upper bound of $3n$ comparisons [9].

In this extended abstract, we study the selection problem, for a multiset of n elements, under the assumption that apart from the space for the elements themselves we only allow a constant amount of extra space. The extra space is restricted to $O(\log n)$ bits, thus preventing the possibility of encoding a copy of the n elements. We call this the *implicit selection problem.* Before explaining why we are interested in this problem, observe that it is indeed a new problem. Blum et al. [2] devised an $O(n)$ worst-case selection algorithm; however, it uses $\Theta(\log n)$ extra space [5]. Schönhage et al. [9] also devised an $O(n)$ worst-case algorithm; a straightforward implementation of it requires $\Omega(n^{1/2})$ extra space. This means that the various selection algorithms cannot be used to solve the implicit selection problem without major modifications. We provide a solution by giving an implicit emulation of the basic BFPRT algorithm with preconditioning. This algorithm is first described in Section 3, and the emulation is described in Section 4. As will be proved, our emulation requires fewer than $7n$ comparisons and fewer than $19n$ data movements. Two open problems remain, namely, can the lower bound of $2n$ comparisons for selection be improved for implicit selection, and can the upper bounds of $7n$ comparisons and $19n$ data movements be reduced?

The implicit selection problem arises from various implicit data structures [8]. First, in Lai [6], the maintenance of an implicit minimal height k-d tree under insertions and deletions was explored. This needs an implicit selection algorithm, hence, the implicit selection problem. Second, in van Leeuwen and Wood [10], the notion of a "median" heap is explored. Since a heap is an implicit data structure, it seems reasonable that its construction also be implicit, hence, an implicit median algorithm is needed.

Finally, before describing our implicit selection algorithm in detail, we should point out that in practice one would use the probabilistic algorithm of Floyd and Rivest [3]. This algorithm is, essentially, implicit and is expected to require $1.5n + o(n)$ comparisons. In other words, the implicit selection problem is of limited practical interest and is pursued for its theoretical interest.

2 The implicit selection problem

The *selection problem* is: determine the kth smallest element of a multiset of n elements, given the values of k and the n elements. We define a new problem, the *implicit selection problem*, in which we want to find the kth smallest of n elements using only a constant amount of additional space.

For our model of computation we assume a comparison-based arithmetic RAM. We assume that comparisons have three outcomes ($<$, $=$, or $>$), and that arithmetic operations are allowed only for manipulating indices. We have space to store the n elements and a constant amount of additional space in which we can store data elements and indices in the range $[0, n]$. Note that an index allows us to store $\log n$ bits, so $O(\log n)$ bits can be stored using a constant number of indices. In particular, we can maintain a stack of size $O(\log n)$ bits.

Our main result is:

Theorem 2.1 *The implicit selection problem can be solved in $O(n)$ time in the worst case; furthermore, $6.7756n+o(n)$ comparisons are sufficient if all elements are distinct, and $6.8280n+o(n)$ comparisons are sufficient in the general case.*

In the remainder of this abstract, we describe two algorithms that solve the implicit selection problem in linear time in the worst case by emulating other linear time worst-case selection algorithms. We consider two cases: the case when all elements are distinct and the case when repetitions are permitted. Complications occur in our emulation techniques when repetitions are allowed.

3 The Blum-Floyd-Pratt-Rivest-Tarjan algorithm

Blum et al. devised two selection algorithms that require $\Theta(n)$ time in the worst case. They devised a simple, "slow" algorithm that requires $19.\dot{3}n$ comparisons and a complicated, "fast" algorithm that requires $5.430\dot{5}n$ comparisons. Our algorithms are based on a variant of the slow BFPRT algorithm that incorporates some optimizations of the fast BFPRT algorithm; we refer to this variant as the BFPRT algorithm with presorting. Let c be some odd constant, where $c \geq 5$. Let $\#S$ denote the size of a multiset S. Then the BFPRT algorithm with presorting computes the kth smallest element of a multiset S as follows.

```
function BFPRT-SELECT(S,k)
    1. Arrange $S$ into $\lfloor \#S/c \rfloor$ lists of $c$ elements and sort each list.
    2. Return RSELECT(S,k).
end BFPRT-SELECT

function RSELECT(S,k)
    1. We maintain the invariant that $S$ consists of $\lfloor \#S/c \rfloor$ sorted lists of $c$ elements on entry to
       RSELECT. Let $T$ be the set of medians from each of the lists of size $c$. Arrange $T$ into
       $\lfloor \#T/c \rfloor$ lists of $c$ elements, and sort each list. Compute $m = RSELECT(T, \lceil \frac{\#T}{2} \rceil)$.
    2. Find the rank $r$ of $m$ in $S$, and let $S_<$, $S_=$, $S_>$ be the lists of elements whose middle elements
       are less than, equal to, and greater than $m$, respectively.
    3. If $r = k$, then return $m$. Otherwise, if $r < k$, then set $k'$ to $k - (\#S_< + \#S_=)\lceil \frac{c}{2} \rceil$, discard
       the left $\lceil \frac{c}{2} \rceil$ elements of $S_< \cup S_=$, and merge the lists of $S_< \cup S_=$ to form sorted lists of size
       $c$. Otherwise, set $k'$ to $k$, discard the right $\lceil \frac{c}{2} \rceil$ elements of $S_= \cup S_>$, and merge the lists of
       $S_= \cup S_>$ to form sorted lists of size $c$.
    4. Return $RSELECT(S_< \cup S_= \cup S_>, k')$.
end RSELECT
```

We refer to the problem instance associated with the recursive call in step 2 as the first subproblem and the problem instance associated with the recursive call in step 4 as the second subproblem.

This algorithm can easily be shown to require $O(n)$ time; actually, it requires approximately $6.166n$ comparisons for an appropriate choice of c. However, this algorithm requires $\Theta(n)$ additional space to compute S'.

4 Achieving constant space

Before we proceed further, it is worth noting that we ensure that selections are always performed on the leftmost elements of the input array, and that we simulate the recursion of *RSELECT*. Also, if the number of input elements of a subproblem is less than some constant, then we perform the selection using some other algorithm.

There are four factors that contribute to the storage requirement of the BFPRT algorithm with presorting, the first three being due to the recursion:

1. Saving arguments—endpoints
2. Saving arguments—k
3. Implementing recursive calls
4. Recopying space

We show in turn how each of these costs can be reduced to a constant.

Saving function values may also appear to contribute to the storage requirements of the BFPRT algorithm, but function values never have to be saved. This is because in the *RSELECT*, the result of the recursive call of step 2 is discarded before the recursive call of step 4, which implies that two function values never have to be stored simultaneously.

4.1 Saving arguments—endpoints

Only one endpoint need be saved, since we ensure that selections are performed only on the leftmost elements of the input array. Given an input array of n elements, we guarantee that the number of elements of the first subproblem is $\lfloor n/c \rfloor$, and the number of elements of the second subproblem is $\lceil (1 - \frac{\lceil c/2 \rceil}{2c})n \rceil$. This way, given the number of elements of one of the subproblems, we can multiply a factor and add a term to obtain n. Observe that the factor depends solely on whether the subproblem is the first or second, and the correctional term is a constant. Also, the maximum depth of recursion of *RSELECT* is $O(\log n)$. This suggests that we can maintain a stack of $O(\log n)$ bits to record which subproblems the recursive calls correspond to and another stack of $O(\log n)$ bits to record the correctional terms.

4.2 Saving arguments—k

To encode k, we use a binary encoding scheme. If we have a list of size n, then since $1 \leq k \leq n$, we can encode k in the relative order of $2\lceil \log n \rceil$ elements. We use a pair of unequal elements to encode each bit of k: we place the elements in ascending order to indicate a 0 and in descending order to indicate a 1. Note that we always use $2\lceil \log n \rceil$ elements to encode k; leading zeros are kept. This technique was used by Munro [7] to encode pointers in an implicit data structure based on AVL trees.

Observe that k has to be encoded only during step 2 of *RSELECT*. To encode k, we use the $2\lceil \log n \rceil$ elements immediately following the elements of the first subproblem. This is straightforward when all elements are distinct. In the general case, we may not have enough distinct elements for the encoding to work, so we have to search for elements; this will be discussed in detail below.

4.3 Implementing recursive calls

We use an iterative routine that conceptually performs a postorder traversal of the recursion tree of *RSELECT*. This is straightforward since we maintain a stack and recover previous endpoints and values of k.

4.4 Recopying space

To avoid recopying, we ensure that selections are only performed on the leftmost elements of the input array, and we maintain the invariant that the selections are performed on collections of sorted, contiguous lists of size c. We discuss in detail how we ensure these requirements during the first and second subproblems.

4.4.1 The first subproblem

In the first subproblem, we want to find the median of the medians of the n/c lists of size c. To do this, we want to place these medians in the leftmost n/c positions of the input array. These n/c positions contain n/c^2 medians and $\frac{n}{c} - \frac{n}{c^2}$ non-medians. The remaining positions contain $\frac{n}{c} - \frac{n}{c^2}$ medians and $(\frac{c-1}{c})^2 n$ non-medians. Thus, to move the medians of all lists to the leftmost n/c positions, we swap the non-medians of the leftmost n/c elements with the medians of the remaining elements. To maintain our invariant, we also have to sort the sublists of size c of the list of medians.

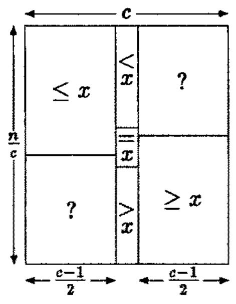

Figure 1: Rearrangement in the second subproblem

4.4.2 The second subproblem

In the second subproblem, we want to discard elements and place the retained elements in the leftmost positions. However, first we must undo the swapping of the medians that we performed during the first subproblem. We swap the center elements of the rightmost $\frac{n}{c} - \frac{n}{c^2}$ lists of size c with the non-center elements of the leftmost n/c^2 lists of size c. It is unlikely that we will obtain the original list because the n/c medians are usually rearranged during the first recursive call. Nevertheless, if the center of each list of size c is removed, then we know we have sorted lists of size $c - 1$.

Actually, this is not quite true. $O(\log n)$ lists of size c may not be restored properly because of the binary encoding of k. The encoding of k interacts with the rearrangement performed during the first subproblem in such a way that these $O(\log n)$ lists must be considered separately. We discuss this in detail below; for now we ignore this problem.

We now have n/c lists of size c; each list of size c contains a sorted list of $c-1$ elements and an element in the center that is the median of some list. We want to determine the rank of the median m of medians computed during the first recursive call, and we want to discard elements. When computing the rank of m, we simultaneously rearrange the elements so as to reduce the time needed to discard elements.

To find the rank of m, we have to consider only the non-medians, since m is the median of medians. However, when discarding elements, it is beneficial to partition the medians such that the medians less than m occupy the uppermost positions, and the medians greater than m occupy the lowermost positions, upper and lower being with respect to Figure 1.

We now want to find the rank of m among the non-medians while rearranging the lists in such a way as to facilitate discards. We consider three types of lists of non-medians:

1. lists in which the left $\frac{c-1}{2} + 1$ elements are less than m
2. lists in which the left $\frac{c-1}{2}$ elements are less than or equal to m, and the right $\frac{c-1}{2}$ elements are greater than or equal to m
3. lists in which the right $\frac{c-1}{2} + 1$ elements are greater than m

Observe that the lists of non-medians are sorted lists of size $c - 1$. When processing a list of non-medians, we identify the type of the list, and we perform a binary search to find the rank of m in the list. While processing these lists, we rearrange them such that the lists of type (1) are the uppermost lists and the lists of type (3) are the lowermost lists.

We now discard some elements. We discard exactly $\frac{\lceil c/2 \rceil}{2c} n + O(1)$ elements, unlike the BFPRT algorithm with presorting, which can potentially discard more elements. Note that this does not affect the worst case. To discard elements, observe that at least half of the lists must be of types

(1) or (2) and at least half of the lists must be of types (2) or (3). Therefore, we discard the left $\lceil\frac{c}{2}\rceil$ elements of the uppermost $\lceil\frac{n}{2c}\rceil$ lists if the rank of m is less than k, and we discard the right $\lceil\frac{c}{2}\rceil$ elements of the lowermost $\lceil\frac{n}{2c}\rceil$ lists if the rank of m is greater than k.

To satisfy the invariant, we must form sorted lists of size c. For $\lfloor\frac{n}{2c}\rfloor$ lists of size $c-1$, we simply perform a binary insertion of a retained median to form a list of size c. For the remaining $\lceil\frac{n}{2c}\rceil$ lists of size $\frac{c-1}{2}$, we merge two lists, steal an element from another list, and perform a binary insertion of this element to form a list of size c.

Once this is done, all of the retained elements will be in one large contiguous block in the left end or the right end of the currently processed part of the array. If the retained elements fall in the right end, then we rotate the elements so that the retained elements fall into the leftmost positions of the array.

There is one complication with this scheme, however. If $O(\log n)$ lists are scrambled, then we cannot ensure that at least half the lists are of types (1) and (2) and at least are of types (2) and (3); furthermore, we cannot ensure that the $\frac{n}{2c}$ uppermost medians are no greater than m and the $\frac{n}{2c}$ lowermost medians are no less than n. Simply sorting each of the $O(\log n)$ lists is not sufficient. Fortunately, we know that there is some ordering of the $O(\log n)$ list elements that ensures these conditions; a greedy algorithm suffices to enforce this condition.

To handle this complication, we swap the $O(\log n)$ scrambled lists into the right end of the currently processed part of the array once we identify them. We then search and rearrange the remaining elements as above. Before we discard any elements, we determine the number of lists whose left halves are no greater than m, the number of lists whose right halves are no less than m, the number of medians that are no greater than m, and the number of medians that are no less than m that we are short of. We then sort the $O(\log n)$ scrambled lists of size c in such a way as to place the smaller elements in the left positions and the larger elements in the right positions. We may then need to swap some elements with some middle elements to obtain the necessary number of medians less than or greater than m. We now rotate the middle elements into their correct positions; that is, we ensure that the $\frac{n}{2c}$ uppermost middle elements are no greater than m and the $\frac{n}{2c}$ lowermost middle elements are no less than m. We then sort the $O(\log n)$ lists of size $c-1$, and rotate these lists into their proper positions.

5 The general case

The preceding techniques for emulating the BFPRT algorithm with presorting are sufficient if all elements are distinct, but are inadequate in the general case. The problem in the general case is that we use $2\lceil\log n\rceil$ elements to encode k, and we must ensure that no element appears more than $\lceil\log n\rceil$ times. To deal with this problem, we first sort the elements in $O(\log n \log\log n)$ time. Second, we observe that if some element e has more than $\lceil\log n\rceil$ occurrences, then the two middle elements are occurrences of e; thus, we take one of the two middle elements, and we perform two binary searches to find the leftmost and rightmost occurrences of this element in $O(\log\log n)$ time. If we find that no element has more than $\lceil\log n\rceil$ occurrences, then we encode a bit using the relative order of the ith and $(i+\lceil\log n\rceil)$th elements, for $1 \leq i \leq \lceil\log n\rceil$.

If some element e occurs more than $2\lceil\log n\rceil$ times, then we must search for elements unequal to e. We inspect the next $\frac{\lceil c/2\rceil}{2c}n$ elements. If they are all equal to e, then we avoid solving the first subproblem as follows. We discard the $\frac{\lceil c/2\rceil}{2c}n$ elements equal to e and move them to the right end of the currently processed block. We sort the $O(\log n)$ scrambled lists of size c, and we perform binary searches to determine the rank of e. We then adjust k depending on the rank of e, and solve the second subproblem.

All that remains is to show how to efficiently search for elements unequal to e and what to do with these elements. In general, we need only $2c$ comparisons to determine if all of the elements of a block of $c-1$ lists of size c are equal to e; the block consists of $c-1$ lists that are sorted if the middle element of each list is removed, and this set of middle elements forms a sorted list of size $c-1$, for a total of c lists, and we compare the first and last element of each list to e. If we find an element unequal to e, we swap it with one of the excess occurrences of e. We then swap the list of size c from which the element came to some positions next to the $2\lceil\log n\rceil$ locations used to encode k, because this list is effectively scrambled, and we want to quickly identify scrambled lists. Note that we need some index computations to determine the starting location of the block of lists whose middle elements are sorted lists of size $c-1$.

6 Analysis

6.1 A sketch of the algorithm

For the purposes of analysis, it is useful to sketch the basic algorithm. Let n_0 be some sufficiently large constant, and let *SELECT* be some selection algorithm.

```
function ISELECT(array $A$, endpoint $u$, rank $k$)
  1. Initialize:
     Arrange $A[1..u]$ into $\lfloor u/c \rfloor$ contiguous lists of size $c$, and sort each list.
  2. Check simple cases:
     If $u \le n_0$, then $m \leftarrow SELECT(A,u,k)$, and go to recovery.
  3. Solve first subproblem:
     Rearrange elements for first subproblem. Set $u' \leftarrow \lfloor \frac{u}{c} \rfloor$. Encode $k$ in $A[u'+1..u'+2\lceil\log n\rceil]$.
     (In the general case, if $\frac{\lceil c/2 \rceil}{2c} n$ occurrences of the same element are inspected, then rearrange
     elements and go to solve second subproblem.) Push $1, u - cu'$. Set $u \leftarrow u'$, $k \leftarrow \lceil \frac{u'}{2} \rceil$. Go to
     check simple cases.
  4. Solve second subproblem:
     Rearrange elements for the second subproblem, using $m$. Set $u', k'$ to the new values of $u, k$.
     Push $2, u - \frac{2c}{2c - \lceil c/2 \rceil} u'$. Set $u \leftarrow u'$, $k \leftarrow k'$. Go to check simple cases.
  5. Recovery:
     If stacks are empty, then return $m$. Pop sub, $d$. If sub $= 1$, set $u' \leftarrow u$, $u \leftarrow cu + d$;
     recover $k$ from $A[u'+1 \ldots u'+2\lceil\log u\rceil]$, and go to solve second subproblem. If sub $= 2$, set
     $u \leftarrow \frac{2c}{2c - \lceil c/2 \rceil} u + d$, and go to recovery.
end ISELECT
```

6.2 The distinct element case

We derive a recurrence relation to measure the cost of *ISELECT* independently of the cost measure. The cost of the algorithm is

$$T(n) = T_0(n) + T'(n)$$

where $T_0(n)$ is the cost of initialization and $T'(n)$ is the cost of the emulation of *RSELECT*. *ISELECT* emulates *RSELECT* by maintaining a stack, so $T'(n)$ is described by the recurrence relation

$$T'(n) = T'(s_1(n)) + T'(s_2(n)) + f_{1,p}(n) + f_{1,r}(n) + f_{2,p}(n) + f_{2,r}(n)$$

where $s_i(n)$ is the number of elements of the ith subproblem, $f_{i,p}(n)$ is the cost of preparing for subproblem i, and $f_{i,r}(n)$ is the cost of recovering from subproblem i, for $i = 1, 2$. If we are measuring the total cost of all operations of *ISELECT*, then $T_0(n)$ is $O(n)$, $s_1(n) = \frac{n}{c}$, $s_2(n) = (1 - \frac{\lceil c/2 \rceil}{2c})n$, and for all i, $f_{i,p}$ is $O(n)$ and $f_{i,r}$ is $O(n)$. Clearly for $c \geq 5$, there exists a constant $d < 1$ such that $s_1(n) + s_2(n) < dn$, which implies that $T(n)$ is $O(n)$.

We now count the number of comparisons $C(n)$ more carefully, using the above formula for $T(n)$. We let $c = 43$ since this is the optimal value of c. The cost of initialization is simply the cost of sorting lists of size 43; since 177 comparisons are required by the Ford-Johnson algorithm [4] to sort 43 elements, $C_0(n) = \frac{177}{43}n$. Consider $C'(n)$. Note that $s_1(n) = \frac{n}{43}$ and $s_2(n) = (1 - \frac{\lceil c/2 \rceil}{2c})n = \frac{32}{43}n$.

To set up the first subproblem, we have to find the medians of the sorted lists, requiring no comparisons; sort lists of size c of the medians, requiring $\frac{177}{43} \cdot \frac{n}{43}$ comparisons; encode u, requiring no comparisons; and encode k, requiring $O(\log n)$ comparisons. Thus $\frac{177n}{43^2} + O(\log n)$ comparisons are required to set up the first subproblem. To recover from the first subproblem, we have to recover u and k, requiring $O(\log n)$ comparisons.

To set up the second subproblem, we have to identify and reposition $O(\log n)$ scrambled lists, requiring no comparisons, and we partition the list of medians, requiring n/c comparisons. We then search and reposition lists of size $c - 1$, requiring $\frac{n}{c} \cdot (2 + \lfloor \log c - 2) \rfloor)$ comparisons: for each list of size $c - 1$, we inspect the two middle elements, and then perform a binary search on $\frac{c-1}{2} - 1$ elements. We then perform binary insertions of medians into $\frac{n}{2c}$ lists, requiring $\frac{n}{2c} \cdot \lfloor \log(2c - 1) \rfloor$ comparisons. We also form lists of size c by merging two lists of size $\frac{c-1}{2}$ and performing a binary insertion of some element into the list of size $c - 1$, requiring $(\frac{1}{2} - \frac{\lceil c/2 \rceil}{2c})\frac{n}{c} \cdot (c - 2 + \lfloor \log(2c - 1) \rfloor)$ comparisons. We finally process the $O(\log n)$ scrambled lists, requiring $O(\log n \log \log n)$ comparisons. Thus the number of comparisons required to set up the second subproblem is $\frac{11}{43}n + \frac{21}{86} \cdot \frac{47}{43}n + O(\log n \log \log n)$. To recover from the second problem, we recover u, requiring no comparisons.

Thus,

$$C'(n) = C'(\frac{n}{43}) + C'(\frac{32n}{43}) + \frac{177}{43^2}n + \frac{11}{43}n + \frac{21}{86} \cdot \frac{47n}{43} + O(\log n \log \log n)$$

By induction we can show that $C'(n) \leq \frac{2287}{860}n + o(n)$. Thus

$$C(n) = C_0(n) + C'(n) \leq \frac{177}{43}n + \frac{2287}{860}n + o(n) = \frac{5827}{860}n + o(n) < 6.7756n + o(n)$$

We now count the number of data movements $M(n)$ more carefully, using the above formula for $T(n)$. The cost of initialization is $M_0(n) = n$ if we sort using an auxiliary array of pointers. Consider $M'(n)$.

To set up the first subproblem, we have to find the medians of the sorted lists, requiring no movements; sort lists of size c of the medians, requiring $\frac{n}{43}$ movements; encode u, requiring no movements; and encode k, requiring $O(\log n)$ movements. Thus $\frac{n}{43} + O(\log n)$ movements are required to set up the first subproblem. To recover from the first subproblem, we have to recover u and k, requiring no movements.

To set up the second subproblem, we have to identify and reposition $O(\log n)$ scrambled lists, requiring $O(\log n)$ movements, and we partition the list of medians, requiring $\frac{3n}{2c}$ movements. We then search and reposition lists of size $c - 1$, requiring $3\frac{c-1}{c}n$ movements. We then perform binary insertions of medians into $\frac{n}{2c}$ lists, requiring $\frac{\lceil c/2 \rceil}{c}n$ data movements. We also form lists of size c by merging two lists of size $\frac{c-1}{2}$ and performing a binary insertion of some element into the list of size $c - 1$, requiring $\frac{c-1}{4c}(2 + \frac{\lceil c/2 \rceil}{c})n$ movements. We finally process the $O(\log n)$ scrambled lists; if we modify the algorithm slightly so that we swap the lists into their proper positions instead of rotating them, then we need only $O(\log n \log \log n)$ movements for sorting and swapping; only $O(\log n)$ additional comparisons are needed because of this change. Thus the number of movements required to set up the second subproblem is $\frac{3n}{86}+3\cdot\frac{42}{43}n+\frac{22}{43}n+\frac{21}{86}(2+\frac{22}{43})n+O(\log n\log\log n)$. To recover from the second problem, we recover u, requiring no movements.

Thus,

$$M'(n)=M'(\frac{n}{43})+M'(\frac{32n}{43})+\frac{23n}{43}+\frac{3}{86}n+3\cdot\frac{42n}{43}+\frac{21}{86}(2+\frac{22}{43})n+O(\log n\log\log n)$$

By induction we can show that $M'(n)\leq\frac{15211}{860}n+o(n)$. Thus

$$M(n)=M_0(n)+M'(n)\leq n+\frac{15211}{860}n+o(n)=\frac{16071}{860}n+o(n)<18.6873n+o(n)$$

6.3 The general case

The analysis of the general case algorithm is similar to the distinct element selection algorithm; the only difference is some additional actions are performed when encoding k in the first subproblem. We can again derive a recurrence relation to measure the cost of *ISELECT* independently of the cost measure, and we can show that the total cost of all operations of our algorithm is $O(n)$.

We count the number of comparisons required in the general case more carefully. If we choose $c=43$, then

$$C'(n)=C'(\frac{n}{43})+C'(\frac{32n}{43})+\frac{177n}{43^2}+\frac{11n}{21\cdot 43}+\frac{11n}{43}+\frac{21}{86}\cdot\frac{47n}{43}+O(\log n\log\log n)$$

By induction we can show that $C'(n)\leq\frac{48973}{18060}n+o(n)$. Thus the number of comparisons required is

$$C(n)=C_0(n)+C'(n)\leq\frac{177}{43}n+\frac{48973}{18060}n+o(n)=\frac{123313}{18060}n+o(n)<6.8280n+o(n)$$

The number of data movements required is the same as before and is $18.6873n+o(n)$.

Acknowledgements

The work of the first author was supported under an NSERC Postgraduate Scholarship and that of the second author under a Natural Sciences and Engineering Research Council of Canada Grant No. A-5692

References

[1] S. W. Bent and J. W. John. Finding the median requires $2n$ comparisons. In *Proceedings of the 17th Annual ACM Symposium on Theory of Computing*, pages 213–216, 1985.

[2] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan. Time bounds for selection. *Journal of Computer and System Sciences*, 7:448–461, 1973.

[3] R. W. Floyd and R. L. Rivest. Expected time bounds for selection. *Communications of the ACM*, 18:165–172, 1975.

[4] L. R. Ford and S. M. Johnson. A tournament problem. *American Mathematical Monthly*, 66:387–389, 1959.

[5] E. Horowitz and S. Sahni. *Fundamentals of Computer Algorithms*, pages 131–135. Computer Science Press, 1978.

[6] T. W. Lai. *Space Bounds for Selection.* Master's thesis, University of Waterloo, 1987.

[7] J. I. Munro. An implicit data structure supporting insertion, deletion, and search in $O(\log^2 n)$ time. *Journal of Computer and System Sciences*, 33:66–74, 1986.

[8] J. I. Munro and H. Suwanda. Implicit data structures for fast search and update. *Journal of Computer and System Sciences*, 21:236–250, 1980.

[9] A. Schönhage, M. Paterson, and N. Pippenger. Finding the median. *Journal of Computer and System Sciences*, 13:184–199, 1976.

[10] J. van Leeuwen and D. Wood. *Interval heaps.* Technical Report, Univ. of Waterloo, 1988.