CS-88-33 - ON EFFICIENT ENTREEINGS

ABSTRACT:

A *data encoding* is a formal model of how a logical data structure is mapped into or represented in a physical storage structure. Both structures are complete trees in this paper, and we encode the logical or guest tree in the leaves of the physical or host tree giving a restricted class of encodings called *entreeings*. The *cost* of an entreeing is the total amount that the edges of the guest tree are stretched or dilated when they are replaced by shortest paths in the host tree. We are particularly interested in the *asymptotic average cost* of families of similar entreeings.

Our investigation is a continuation of the study initiated in [6].

AUTHORS: Paul S. Amerins, Ricardo A. Baeza-Yates, Derick Wood

PRICE: $2.00

CS-88-34 - THE SUBSEQUENCE GRAPH OF A TEXT

ABSTRACT:

We define the directed acyclic subsequence graph of a text as the smallest deterministic partial finite automaton that recognizes all possible subsequences of that text. We define the size of the automaton as the size of the transition function and not the number of states. We show that it is possible to build this automaton using $O(n \log n)$ time and space for a text of size n. We extend this construction to the case of multiple strings obtaining a $O(n^2 \log n)$ time and $O(n^2)$ space algorithm, where n is the size of the set of strings. For the later case, we discuss its application to the longest common subsequence problem improving previous solutions.

AUTHOR: Ricardo A. Baeza-Yates

PRICE: $2.00

If you would like to order any reports please forward your order, along with a cheque or international bank draft payable to the Department of Computer Science, University of Waterloo, Waterloo, Ontario, N2L 3G1, to the Research Report Secretary.

Please indicate your current mailing address and if you wish to remain on our mailing list.

MAILING ADDRESS:

University of Tennessee
Computer Science Librarian
107 Ayres Hall
Knoxville TN 37996

YES, REMAIN ON MAILING LIST

NO, DELETE FROM MAILING LIST

sent Feb. 10/89

Please Send Technical Report CS-88-327 Implicit Selection to: University of Tennessee
Computer Science Dept.
107 Ayres Hall
Knoxville, TN 37996

c/o Lisa D. Lay

* Please let us know of any changes

Printing Requisition / Graphic Services

15067

1. Please complete unshaded areas on form as applicable.
2. Distribute copies as follows: White and Yellow to Graphic Services. Retain Pink Copies for your records.
3. On completion of order the Yellow copy will be returned with the printed material
4. Please direct enquiries, quoting requisition number and account number, to extension 3451.

TITLE OR DESCRIPTION: CS-88-32 Implicit Selection

DATE REQUISITIONED: August 15/88

DATE REQUIRED: ASAP

ACCOUNT NO.: 1 2 6 6 1 7 6 4 1

REQUISITIONER – PRINT: D. Wood

PHONE: 4456

SIGNING AUTHORITY: S. DeAngelis / D. Wood

MAILING INFO – NAME: Sue DeAngelis

DEPT.: C.S.

BLDG. & ROOM NO.: DC 2314

[X] DELIVER [ ] PICK-UP

Copyright: I hereby agree to assume all responsibility and liability for any infringement of copyrights and/or patent rights which may arise from the processing of, and reproduction of, any of the materials herein requested. I further agree to indemnify and hold blameless the University of Waterloo from any liability which may arise from said processing or reproducing. I also acknowledge that materials processed as a result of this requisition are for educational use only.

NUMBER OF PAGES: 19

NUMBER OF COPIES: 150

TYPE OF PAPER STOCK: [ ] BOND [ ] NCR ____ PT. [ ] COVER [ ] BRISTOL [ ] SUPPLIED [ ] Alpac Ivory 140M

PAPER SIZE: [ ] 8½ x 11 [ ] 8½ x 14 [ ] 11 x 17 [ ] 10x14 Glosscoat

PAPER COLOUR: [ ] WHITE [X] 10 pt Rolland Tint

INK: [X] BLACK [ ]

PRINTING: [ ] 1 SIDE ____ PGS. [X] 2 SIDES ____ PGS.

NUMBERING: FROM ____ TO ____

BINDING/FINISHING: [X] COLLATING [ ] STAPLING [ ] ____ HOLE PUNCHED [ ] PLASTIC RING

FOLDING/PADDING: 7x10 saddle stitched

CUTTING: ~~7x10~~

Special Instructions

Beaver Cover

Both cover and inside in black ink please

COPY CENTRE — OPER. NO. | BLDG. | MACH. NO.

DESIGN & PASTE-UP — OPER. NO. | TIME | LABOUR CODE

D 0 1

D 0 1

D 0 1

TYPESETTING — QUANTITY

P A P 0 0 0 0 0 0 — T 0 1

P A P 0 0 0 0 0 0 — T 0 1

P A P 0 0 0 0 0 0 — T 0 1

PROOF

P R F

P R F

P R F

NEGATIVES — QUANTITY | OPER. NO. | TIME | LABOUR CODE

F L M — C 0 1

F L M — C 0 1

F L M — C 0 1

F L M — C 0 1

F L M — C 0 1

PMT

P M T — C 0 1

P M T — C 0 1

P M T — C 0 1

PLATES

P L T — P 0 1

P L T — P 0 1

P L T — P 0 1

STOCK

0 0 1

0 0 1

0 0 1

0 0 1

BINDERY

R N G — B 0 1

R N G — B 0 1

R N G — B 0 1

M I S 0 0 0 0 0 0 — B 0 1

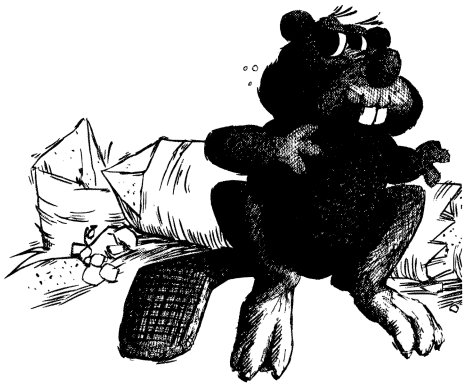
OUTSIDE SERVICES

$ ____ COST

TAXES – PROVINCIAL [ ] FEDERAL [ ]

GRAPHIC SERV. OCT. 85 482-2

UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT
UNIVERSITY OF WATERLOO COMPUTER SCIENCE DEPARTMENT



Implicit Selection

Tony W. Lai
and
Derick Wood

Data Structuring Group
Research Report CS-88-32

August, 1988

Implicit Selection*

Tony W. Lai† Derick Wood†

August 8, 1988

Abstract

We consider the problem of selecting the kth smallest element of a multiset of n elements using only a constant amount of additional space, the *implicit selection problem*. We demonstrate that this problem can be solved in $O(n)$ time in the worst case. Moreover, we prove that $6.4217n + o(n)$ comparisons are sufficient if all elements are distinct, $6.4514n + o(n)$ comparisons are sufficient in the general case, and $24.8388n + o(n)$ data movements are sufficient in either case.

1 Introduction

The problem of selecting the kth smallest 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 on for a faster algorithm; the state-of-the-art is a lower bound of $2n$ comparisons [1] and an upper bound of $3n$ comparisons [10].

In this paper, we study the selection problem, for a multiset of n elements, under the assumption that apart from the space for the elements themselves we allow only 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

*The work of the first author was supported under an NSERC Postgraduate Scholarship and that of the second under a Natural Sciences and Engineering Research Council of Canada Grant No. A-5692 and under an Information Technology Research Centre Grant. A preliminary version of this paper was presented at SWAT 1988; see [7]; however, we have improved the bounds given in [7].

†Data Structuring Group, Department of Computer Science, University of Waterloo, Waterloo, Ontario, Canada.

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. [10] also devised an $O(n)$ worst-case algorithm; a straightforward implementation of it requires $\Omega(n^{1/2})$ extra space. This means that these 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; see [2]. 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 $6.5n$ comparisons and fewer than $25n$ data movements. Two open problems remain, namely, is the lower bound for implicit selection greater than $2n$ comparisons, and can the upper bounds of $6.5n$ comparisons and $25n$ data movements be reduced?

The implicit selection problem arises from various implicit data structures [9]. 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 [11], 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 solely 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, $6.4217n + o(n)$ comparisons are sufficient if all elements are distinct, and $6.4514n + o(n)$ comparisons are sufficient in the general case; $24.8388n + o(n)$ movements are sufficient in either case.*

In the remainder of this paper, 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. [2] 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 $5.4305\dot{5}n$ comparisons. Our algorithms are based on a well known 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 preconditioning. Let c be some odd integer constant, where $c \geq 5$. Let $\#S$ denote the size of a multiset S. Then the BFPRT algorithm with preconditioning 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 of the lists of size c. Arrange T into $\lfloor \#T/c \rfloor$ lists of c elements, and sort each list. Compute $m = RSELECT(T, \lceil \#T/2 \rceil)$.
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 leftmost $\lceil \frac{c}{2} \rceil$ elements of $S_< \cup S_=$, and merge

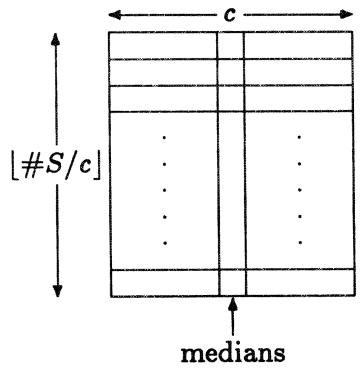


Figure 1. The conceptual organization of elements in *RSELECT*.

the lists of $S_< \cup S_=$ to form sorted lists of size c. Otherwise, set k' to k, discard the rightmost $\lceil \frac{c}{2} \rceil$ 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 1 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 $c = 31$. However, this algorithm requires $\Theta(n)$ additional space to compute $S_< \cup S_= \cup S_>$.

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 a specified 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 *RSELECT*, the result of the recursive call of step 1 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 during each recursive call, 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 by 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 added term is bounded by 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 added 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. Use a pair of unequal elements to encode each bit of k, placing the elements in ascending order to indicate a 0 and in descending order to indicate a 1. Figure Figure 2 displays such an encoding of 5 in a list of size 8. Note that we always use $2\lceil \log n \rceil$ elements to encode k; leading zeros are kept. This technique was used by Munro [8] to encode pointers in an implicit data structure based on AVL trees.

Observe that k has to be encoded only during step 1 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, there may not be enough distinct elements for the encoding to work, so we have to search for elements; this will be discussed in detail in Section 5.

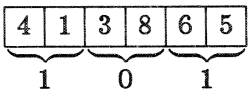


Figure 2. Binary encoding of 5 in a list of size 8.

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

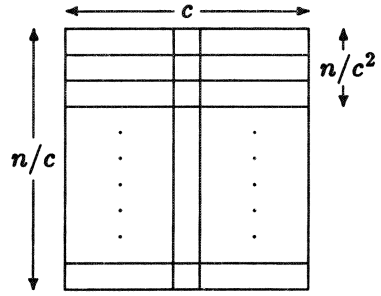
To avoid recopying, we ensure that selections are only performed on the leftmost elements of the input array, and maintain the invariant that the selections are performed on collections of sorted, contiguous lists of size c. We discuss in detail how we ensure these requirements during the first and second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

4.4.1 The first subproblem

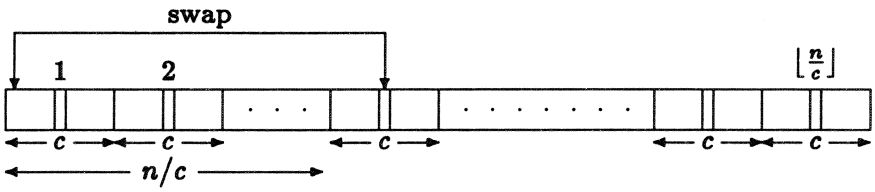
In the first subproblem, we want to find the median of the medians of the n/c lists of size c. To do this, we want to place these medians in the leftmost n/c positions of the input array. These n/c positions contain n/c^2 medians and $\frac{n}{c} - \frac{n}{c^2}$ non-medians. The remaining positions 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, swap the non-medians of the leftmost n/c elements with the medians of the remaining elements. To maintain our invariant, sort the sublists of size c of the list of medians, and reorder the corresponding lists of non-medians such that we obtain blocks of c lists of size $c-1$ in which the lists of each block are sorted by their original median elements.

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 when solving the first subproblem. 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



The conceptual arrangement of the elements.



The actual arrangement of the elements.

Figure 3. Rearrangement in the first subproblem.

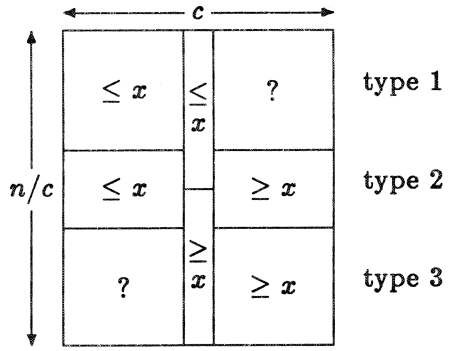


Figure 4. Rearrangement in the second subproblem.

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 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 4.

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. There are three types of lists of non-medians.

1. Lists in which the leftmost $\frac{c-1}{2} + 1$ elements are less than m.
2. Lists in which the leftmost $\frac{c-1}{2}$ elements are less than or equal to m, and the rightmost $\frac{c-1}{2}$ elements are greater than or equal to m.
3. Lists in which the rightmost $\frac{c-1}{2} + 1$ 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, identify the type of the list, and perform a binary search to find the rank of m in the list. While processing these lists, rearrange them such that the lists of type (1) are the uppermost lists and the lists of type (3) are the lowermost lists.

Care must be taken when determining the rank of m if repetitions are allowed. Occurrences of m in the leftmost $\frac{c-1}{2}$ positions of any list must be considered equivalent to elements less than m, and occurrences of m in the rightmost $\frac{c-1}{2}$ positions of any list must be considered equivalent to elements greater than m; otherwise, we may erroneously discard too many occurrences of m if the kth smallest element is in fact m.

Now 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, discard the leftmost $\lceil \frac{c}{2} \rceil$ elements of the uppermost $\lceil \frac{n}{2c} \rceil$ lists if the rank of m is less than k, and discard the rightmost $\lceil \frac{c}{2} \rceil$ 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 $c-1$, 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 $\frac{c-1}{2}$, 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 contiguous block in the left end or the right end of the part of the array currently being processed. If the retained elements fall in the right end, then rotate the elements so that the retained elements fall into the leftmost positions of the array.

There is one complication with this scheme, however. The encoding of k scrambles $O(\log n)$ lists, so we cannot ensure that at least half the lists are of types (1) and (2) and at least half 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, swap the $O(\log n)$ scrambled lists into the right end of the part of the array currently being processed once we identify them. Then search for and rearrange the remaining elements as above. Before discarding any elements, determine the number of lists whose left halves are no greater than m that are needed, the number of lists whose

right halves are no less than m that are needed, the number of medians that are no greater than m that are needed, and the number of medians that are no less than m that are needed. Then sort the $O(\log n)$ scrambled lists of size c so as to place the smaller elements in the leftmost positions and the larger elements in the rightmost 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. Now swap the middle elements into their correct positions; that is, 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. Finally, sort the $O(\log n)$ lists of size $c-1$, and swap 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, first sort the elements in $O(\log n \log\log n)$ time. Second, observe that if some element e has more than $\lceil \log n \rceil$ occurrences, then the two middle elements are occurrences of e; thus, take one of the two middle elements, and 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 $1 \le i \le \lceil \log n \rceil$.

If some element e occurs more than $\lceil \log n \rceil$ times, then we must search for elements distinct from e. We inspect the next $R(c)n + \lceil \log n \rceil$ elements, for some $0 < R(c) < 1$. If at least $R(c)n$ elements are equal to e, we avoid solving the first subproblem as follows. Discard the $R(c)n$ elements equal to e and move them to the rightmost end of the currently processed block. Sort the $O(\log n)$ scrambled lists of size c, and perform binary searches to determine the rank of e. 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 distinct from 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 distinct from 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 $u \leq n_0$, 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

$$T(n) = T_0(n) + T_1(n) \tag{1}$$

where $T_0(n)$ is the cost of initialization and $T_1(n)$ is the cost of the emulation of *RSELECT*. *ISELECT* emulates *RSELECT* by maintaining a stack, so $T_1(n)$ is described by the recurrence relation

$$T_1(n) = T_1(s_1(n)) + T_1(s_2(n)) + f_{1,p}(n) + f_{1,r}(n) + f_{2,p}(n) + f_{2,r}(n) \tag{2}$$

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 $O(n)$.

6.2.1 Counting comparisons

We count the number $C(n)$ of comparisons more carefully, using equations 1 and 2. Let $c = 31$ since this is the optimal value of c. The cost of initialization is simply the cost of sorting lists of size 31; since 116 comparisons are required by the Ford-Johnson algorithm [4] to sort 31 elements, $C_0(n) = \frac{116}{31}n$. Consider $C_1(n)$, noting that $s_1(n) = \frac{n}{31}$ and $s_2(n) = (1 - \frac{\lceil c/2 \rceil}{2c})n = \frac{23}{31}n$.

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 $\frac{1}{c} \cdot C_0(n)$ comparisons; encode u, requiring no comparisons; and encode k, requiring $O(\log n)$ comparisons. Thus $\frac{116n}{31^2} + O(\log n)$ 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.

Before discussing the second subproblem, we show how to efficiently identify the types of lists and search for the median m of medians. The most straightforward method is to inspect one of the two middle elements, and perform a binary search on $\frac{c-1}{2} - 1$ elements, requiring $1 + \lfloor \log c \rfloor$ comparisons per list. Observe that if we know that a list is of type (1) or (2), we need only perform a binary search on its right half, saving one comparison; if we know that a list is of type (2) or (3), we need only search its left half. Since we have sorted blocks of c lists by the former contents of their middle elements, if we find a type (1) list, all lists above it in the same block must be of types (1) or (2), and if we find a type (3) list, all lists below must be of types (2) or (3).

This suggests a more efficient search method. For a given block of b lists, we search in the $\lceil b/2 \rceil$th list in the following manner: inspect the two middle elements, and then perform a binary search on one half. If the list is of type (2), we do not perform the binary search and, therefore, save $\lfloor \log(c-2) \rfloor - 1$ comparisons. Then continue searching the $(\lceil b/2 \rceil - i)$th and $(\lceil b/2 \rceil + i)$th lists for $i = 1, 2, \ldots$ in this manner until either a list of type (1) or (3) is found, or all lists are exhausted. If the list is not type (2), we save comparisons from the new knowledge of the types of some of the lists; we apply our technique recursively on the portion of the block that has yielded no information. We use the straightforward search technique for blocks of less than 5 lists. By induction, we can show that we save at least $s(b)$ comparisons, where

$$s(b) = \begin{cases} 0, & \text{if } b < 5 \\ \lceil \frac{b}{2} \rceil - 2 + s(\lfloor \frac{b}{2} \rfloor), & \text{otherwise} \end{cases}$$

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 $\frac{n}{c}[1 + \lfloor \log c \rfloor - \frac{s(c)}{c}]$ comparisons using the above technique, and 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 $(\frac{1}{2} - \frac{\lceil c/2 \rceil}{2c})\frac{n}{c}[c - 2 + \lfloor \log(2c-1) \rfloor]$ 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 $\frac{7}{62}n + \frac{388}{31^2}n + O(\log n \log\log n)$. To recover from the second problem, we recover u, requiring no comparisons.

Thus,

$$C_1(n) = C_1(\frac{n}{31}) + C_1(\frac{23n}{31}) + \frac{116}{31^2}n + \frac{7}{62}n + \frac{388}{31^2}n + O(\log n \log\log n)$$

By induction we can show that $C_1(n) \leq \frac{175}{62}n + o(n)$. Thus

$$C(n) = C_0(n) + C_1(n) \leq \frac{116}{31}n + \frac{175}{62}n + o(n) = \frac{407}{62}n + o(n) < 6.5646n + o(n)$$

6.2.2 Counting data movements

We compute the number $M(n)$ of data movements using equations 1 and 2. Note that we count only the movements from or to the array of n elements; we do not charge for movements entirely within the $O(1)$ extra storage. The cost of initialization is $M_0(n) = \frac{3}{2}n$ if we sort using an auxiliary array of pointers. Consider $M_1(n)$.

To set up the first subproblem, we have to swap the non-medians in the leftmost n/c positions with medians, requiring $3 \cdot \frac{c-1}{c^2} n$ movements; sort lists of size c of the medians and reorder the non-medians, requiring $\frac{3}{2}n$ data movements; encode u, requiring no movements; and encode k, requiring $O(\log n)$ movements. Thus $\frac{3}{2}n + \frac{90n}{31^2} + O(\log n)$ 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 undo the movement in the first subproblem, requiring $3 \cdot \frac{c-1}{c^2} n$ movements. We 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 $3 \cdot \frac{c-1}{c} n$ movements, and perform binary insertions of medians into $\frac{n}{2c}$ lists, requiring $\frac{\lceil c/2 \rceil}{2c} n$ data movements. We also form lists of size c by merging two lists of size $\frac{c-1}{2}$ and inserting some element into the list of size $c-1$, requiring $(\frac{1}{2} - \frac{\lceil c/2 \rceil}{2c}) \frac{n}{c} \cdot (2c + \lceil \frac{c}{2} \rceil)$ movements. We then may need to perform $3 \cdot \frac{\lceil c/2 \rceil}{2c} n$ movements to ensure that the discarded elements are the rightmost elements. Finally, we finally process the $O(\log n)$ scrambled lists, requiring $O(\log n \log\log n)$ movements. Thus the number of movements required to set up the second subproblem is $\frac{122n}{31} + \frac{3n}{62} + \frac{39 \cdot 15 + 90}{31^2} n + O(\log n \log\log n)$. To recover from the second problem, we recover u, requiring no movements.

Thus,

$$M_1(n) = M_1(\frac{n}{31}) + M_1(\frac{23n}{31}) + \frac{170n}{31} + \frac{765}{31^2} n + O(\log n \log\log n)$$

By induction we can show that $M_1(n) \leq \frac{6035}{217} n + o(n)$. Thus

$$M(n) = M_0(n) + M_1(n) \leq \frac{3}{2} n + \frac{6035}{217} n + o(n) = \frac{12721}{434} n + o(n) < 29.3111n + o(n)$$

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 again apply equations 1 and 2.

To count the number of comparisons, again choose $c = 31$. The main difference from the distinct element case is the encoding of k. We need $O(\log n \log\log n)$ comparisons to sort $2\lceil \log n \rceil$ elements, and we may have to inspect $R(c)n$ elements, requiring $\frac{2}{c-1} R(c)n$ comparisons. Thus the number of comparisons required to encode k is $\frac{2}{c-1} R(c)n + O(\log n \log\log n)$ comparisons, as opposed to $O(\log n)$ comparisons in the distinct element case.

If we find that $R(c)n$ elements are equal, we avoid one subproblem of *ISELECT*. We discard the elements, requiring no comparisons; undo the rearrangement of elements in the first subproblem, requiring no comparisons; correct the order of scrambled lists, requiring $O(\log n)$ comparisons; and perform a binary search on each list of size c, requiring $(1-R(c))\frac{n}{c}\lfloor\log(2c+1)\rfloor$ comparisons.

If we do not find $R(c)n$ elements that are equal, the worst case occurs when we inspect $R(c)n$ elements but are unable to discard elements and avoid the first subproblem. Thus,

$$\begin{aligned} C_1(n) = {} & \frac{2}{c-1}R(c)n \\ & + \max\left\{C_1(\frac{n}{31}) + C_1(\frac{23n}{31}) + \frac{116}{31^2}n + \frac{7}{62}n + \frac{388}{31^2}n,\right. \\ & \qquad \left. C_1((1-R(c))n) + \frac{C_0(n)}{c} + \frac{\lfloor\log(2c+1)\rfloor}{c}(1-R(c))n\right\} \\ & + o(n) \end{aligned}$$

To minimize $C_1(n)$, we set $R(c)$ such that

$$\begin{aligned} & C_1(\frac{n}{31}) + C_1(\frac{23n}{31}) + \frac{504n}{31^2} + \frac{7n}{62} + \frac{2R(c)n}{c-1} = \\ & \quad C_1((1-R(c))n) + \frac{C_0(n)}{c} + \frac{\lfloor\log(2c+1)\rfloor}{c}(1-R(c))n + \frac{2R(c)n}{c-1} \end{aligned}$$

We obtain

$$R(c) = \frac{-18991+\sqrt{374771761}}{3844} \approx 0.095740$$

Thus,

$$C_1(n) = C_1(\frac{n}{31}) + C_1(\frac{23n}{31}) + \frac{504n}{31^2} + \frac{7n}{62} + \frac{2}{c-1}R(c)n + O(\log n \log\log n)$$

By induction we can show that $C_1(n) \le (\frac{2537}{1860} + \frac{\sqrt{374771761}}{13020})n + o(n)$. Thus

$$\begin{aligned} C(n) = C_0(n) + C_1(n) &\le \frac{116}{31}n + \left(\frac{2537}{1860} + \frac{\sqrt{374771761}}{13020}\right)n + o(n) \\ &= \left(\frac{9497}{1860} + \frac{\sqrt{374771761}}{13020}\right)n + o(n) < 6.5928n + o(n) \end{aligned}$$

If we avoid the first subproblem, we discard elements, requiring $3R(c)n + O(\log n)$ movements; undo the rearrangement performed in the first subproblem, requiring $3 \cdot \frac{c-1}{c^2}n$ movements; correct the order of scrambled lists, requiring $O(\log n)$ movements; and then perform a binary search on each

list of size c, requiring no movements. If we must solve the first subproblem, we have the situation of the distinct element case. Thus the number of data movements required in the general case is

$$M_1(n) = \max\left\{M_1(\frac{n}{31}) + M_1(\frac{23n}{31}) + \frac{170n}{31} + \frac{765}{31^2}n + O(\log n \log\log n),\right.$$
$$\left. M_1((1-R(c))n) + 3R(c)n + 3\frac{c-1}{c^2}n + O(\log n)\right\}$$
$$= M_1(\frac{n}{31}) + M_1(\frac{23n}{31}) + \frac{170n}{31} + \frac{765}{31^2}n + O(\log n \log\log n)$$

As in the distinct element case, $M_1(n) \leq \frac{6035}{217}n + o(n)$. Thus

$$M(n) = M_0(n)+M_1(n) \leq n+\frac{6035}{217}n+o(n) = \frac{12721}{434}n+o(n) < 29.3111n+o(n)$$

6.4 Optimizations

We are able to save movements and comparisons by relaxing some of our invariants. By allowing discarded elements to be either leftmost or rightmost, we save $3 \cdot \frac{\lceil c/2 \rceil}{2c}n$ movements when setting up the second subproblem. We can efficiently determine the left and right endpoints of the elements of previous subproblems by using an extra bit in each recursive call to indicate whether discarded elements are leftmost or rightmost.

We save $6 \cdot \frac{c-1}{c^2}n$ movements when setting up the first subproblem by relaxing the invariant that elements that we are selecting from must be contiguous; we instead ensure that the elements that we are selecting from must be evenly spaced. Observe that the distance between elements increases by a factor of c whenever we enter a first subproblem and decreases by a factor of c whenever we exit a first subproblem. Thus we do not need to explicitly store the element distances on the stack of $O(\log n)$ bits.

Observe that by relaxing the invariant that retained elements must be leftmost, elements of rank less than k are moved to the leftmost positions and elements of rank greater than k are moved to the right positions of our array. In conjunction with our other optimization, we no longer need to partition the medians when setting up the second subproblem, saving n/c comparisons and $\frac{3n}{2c}$ data movements.

To count the number of comparisons required in the distinct element case, using equations 1 and 2, choose $c = 31$. We obtain

$$C_1(n) = C_1(\frac{n}{31}) + C_1(\frac{23n}{31}) + \frac{504}{31^2}n + \frac{5}{62}n + O(\log n \log\log n)$$

By induction this yields $C_1(n) \leq \frac{1163}{434}n + o(n)$. Thus

$$C(n) = C_0(n)+C_1(n) \leq \frac{116}{31}n+\frac{1163}{434}n+o(n) = \frac{2787}{434}+o(n) < 6.4217n+o(n)$$

To count the number of comparisons required in the general case, choose $c = 31$ and $R(c) = \frac{-18061+\sqrt{340313401}}{3844}$. We obtain

$$C_1(n) = C_1(\frac{n}{31}) + C_1(\frac{23n}{31}) + \frac{504}{31^2}n + \frac{5}{62}n + \frac{2R(c)n}{c-1} + O(\log n \log\log n)$$

which, by induction yields $C_1(n) \leq \frac{16829+\sqrt{340313401}}{13020}n + o(n)$. Thus

$$\begin{aligned} C(n) = C_0(n) + C_1(n) &\leq \frac{116}{31}n + \frac{16829+\sqrt{340313401}}{13020}n + o(n) \\ &= \frac{65549+\sqrt{340313401}}{13020}n + o(n) < 6.4514n + o(n) \end{aligned}$$

Counting the number of movements with $c = 31$, we obtain

$$M_1(n) = M_1(\frac{n}{31}) + M_1(\frac{23n}{31}) + \frac{289}{62} + \frac{585}{31^2} + O(\log n \log\log n)$$

Therefore, $M_1(n) \leq \frac{1447}{62}n + o(n)$, and

$$M(n) = M_0(n)+M_1(n) \leq \frac{3}{2}n+\frac{1447}{62}n+o(n) = \frac{770}{31}n+o(n) < 24.8388n+o(n)$$

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, Rockeville, Maryland, 1978.

W. Lai. *Space Bounds for Selection.* Technical Report CS-88-26, iversity of Waterloo, 1988.

W. Lai and D. Wood. Implicit selection. In *Proceedings of the 1st ındinavian Workshop on Algorithm Theory, Lecture Notes in Com-ːer Science 318*, pages 14–23, Springer-Verlag, Heidelberg, 1988.

[. Munro. An implicit data structure supporting insertion, deletion, l search in $O(\log^2 n)$ time. *Journal of Computer and System Sci-ːes*, 33:66–74, 1986.

[. Munro and H. Suwanda. Implicit data structures for fast search l update. *Journal of Computer and System Sciences*, 21:236–250, ;0.

Schönhage, M. Paterson, and N. Pippenger. Finding the median. ırnal of Computer and System Sciences, 13:184–199, 1976.

van Leeuwen and D. Wood. *Interval heaps.* Technical Report CS-13, Univ. of Waterloo, 1988.