A randomized in-place algorithm for positioning the kth element in a multiset*

Jyrki Katajainen
Department of Computing, University of Copenhagen,
Universitetsparken 1, DK-2100 Copenhagen East, Denmark
jyrki@diku.dk

Tomi A. Pasanen
Turku Centre for Computer Science, University of Turku
Lemminkäisenkatu 14 A, FIN-20520 Turku, Finland
tomi.pasanen@cs.utu.fi

Abstract. A variant of the classical selection problem, called the *positioning problem*, is considered. In this problem we are given a sequence $A[1{:}n]$ of size n, an integer k, $1 \leq k \leq n$, and a strict weak ordering $\otimes$, and the task is to rearrange the elements of the sequence in such way that $A[k] \otimes A[j]$ is false for all j, $1 \leq j < k$, and $A[\ell] \otimes A[k]$ is false for all ℓ, $k < \ell \leq n$. We present a Las-Vegas algorithm which carries out this rearrangement efficiently using only a constant amount of additional space even if the input contains equal elements and if only pairwise element comparisons are permitted. To be more precise, the algorithm solves the positioning problem in-place in linear time using at most $n + k + o(n)$ element comparisons, $k + o(n)$ element exchanges, and the probability for succeeding within stated time bounds is at least $1 - e^{-n^{\Omega(1)}}$.

1. Introduction

In the *selection problem* the task is to find, given a multiset and an integer k, the kth smallest element of the multiset. In this paper we consider a variant of this problem—and call it the *positioning problem*—examined by Hoare [9]: given a sequence $A[1{:}n]$ of n elements, an integer k, $1 \leq k \leq n$, and a strict weak ordering $\otimes$, rearrange the sequence in such a way that $A[k] \otimes A[j]$ is false for all j, $1 \leq j < k$, and $A[\ell] \otimes A[k]$ is false for all ℓ, $k < \ell \leq n$. The routine accomplishing this task is called `nth_element` in the C++ standard library [10, Clause 25].

A strict weak ordering $\otimes$ on the set of elements can be extended to the *lexicographical ordering* on the set of element-index pairs in the usual way: $(A[i], i) \otimes (A[j], j)$ if and only if $A[i] \otimes A[j]$ or if $A[i] \ominus A[j]$ and $i < j$. Here the relation $\ominus$ is that induced by $\otimes$, i.e., $A[i] \ominus A[j]$ if and only

*Partially supported by Danish Natural Science Research Council under contract 9701414 (project Experimental Algorithmics) and contract 9801749 (project Performance Engineering).

if both $A[i] \not\otimes A[j]$ and $A[j] \not\otimes A[i]$ are false, or it can be given as part of input. If we had extra space available, the positioning problem for multisets could be solved by tagging each element with its index, comparing these pairs lexicographically, and applying any of the existing algorithms for sets. However, if the rearrangement of the elements is to be done *in-place*, i.e., using only constant amount of extra space, the problem becomes non-trivial.

Without loss of generality, we assume that $k \leq \lceil n/2 \rceil$. If this is not the case, the problem can be solved symmetrically by considering the end of the input sequence $A[1{:}n]$ as the beginning, by positioning the $(n-k)$th smallest element in this reverse sequence, and by using the converse of the given ordering relation in element comparisons.

The efficiency of sorting algorithms is traditionally measured by calculating the *number* of *element comparisons* performed. In particular, observe that we allow only pairwise element comparisons. When the movement of elements is done in a strict in-place manner, i.e., by swapping the elements wordwise, the *number* of *element exchanges* is another natural performance measure.

Of the classical selection algorithms [1, 5, 6, 9, 21] only the algorithm by Hoare [9] operates in-place. If in his algorithm the partitioning is carried out by using the indices to make the elements distinct as proposed, for example, in [2], and if the median of three random elements is used as the partitioning element in each partitioning, the algorithm performs at most $2.75n + o(n)$ element comparisons on an average when positioning the $\lceil n/2 \rceil$th element in a sequence of n elements; for the exact bounds for general k, see [15]. It is well-known that for a permutation of n distinct elements the average number of element exchanges performed during each partitioning is $1/6$ times that of element comparisons (see, for example, [22, pp. 333–334]). That is, when positioning the $\lceil n/2 \rceil$th element the average number of element exchanges performed is bounded by $0.46n + o(n)$.

The algorithm of Floyd and Rivest [5, 6] (see also [18, Section 3.3] and [20]), on which our algorithm is based, selects the kth smallest of n elements using at most $n + k + o(n)$ element comparisons. This algorithm can also be used for positioning, not only for selection, but it was designed for sets. Furthermore, it is of the Las Vegas type, i.e., it may fail to finish in time, but it will always produce a correct result. From the lower bound of [4] and Yao's minimax principle (see, e.g., [18, Proposition 2.5]) it follows that $n+k-O(1)$ is a lower bound on the expected number of element comparisons performed by any Las-Vegas algorithm for selecting the kth smallest of n elements.

In this paper we describe an in-place adaptation of the algorithm of Floyd and Rivest [5, 6] which can handle multiset data efficiently. It carries out the positioning of the kth element in a multiset of size n in $O(n)$ time using at most $n + k + o(n)$ pairwise element comparisons and at most $k + o(n)$ element exchanges; the probability that these resource bounds are exceeded is at most $e^{-n^{\Omega(1)}}$. To achieve these bounds two ordering relations $\not\otimes$ and

$\ominus$ must be provided. If only $\oslash$ is given as part of input, the algorithm may require $2n + o(n)$ element comparisons. Due to the above-mentioned lower bound, if we ignore the lower order terms, the number of element comparisons performed is best possible. Also, the number of element exchanges performed is optimal if we do not make any assumptions about the input.

For the selection problem several deterministic in-place algorithms have been proposed [3, 7, 16], but none of these solve the positioning problem as such. First, the algorithm of Lai and Wood [16] can be used for finding the kth smallest element, but it will not carry out the partitioning required. Of course, a simple solution is to carry out a three-way partitioning after the selection, but this may require $2n$ additional element comparisons and k additional element exchanges. It is possible to avoid these, but careful modifications to the algorithm and its analysis will be necessary. Second, both the algorithms of Carlsson and Sundström [3], and Lai and Wood [16], as well as the algorithm of Geffert [7] which uses one of the above-mentioned algorithms as a subroutine, rely on three-way element comparisons. A naive solution is to replace each three-way comparison with two binary comparisons, but this will double the comparison count. In the full version of this paper we describe a deterministic in-place algorithm for positioning which requires $3.64n + 0.72k + o(n)$ pairwise element comparisons, but not even this will get close to the bound achieved by the randomized method.

2. Bit encoding

Our in-place positioning algorithm relies on the bit encoding technique introduced by Munro [19]. This technique has turned out to be crucial in many other in-place algorithms (see, for example, [11, 12, 14, 16, 17]). In this section we describe how the technique is used in the present context.

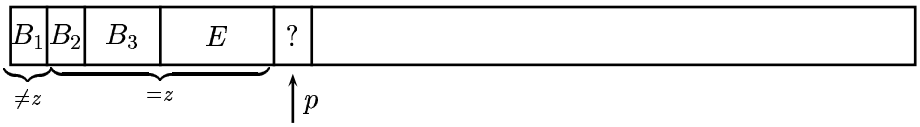
Two distinct elements x and y, $x \oslash y$, can be used to represent a 0-bit by storing them in two consecutive locations in order xy, and a 1-bit by storing them in order yx. By using $\lceil \log_2(n+1) \rceil$ such pairs an integer value up to n can be represented. To read the value or to update the value of such an integer, $O(\log_2 n)$ element comparisons and $O(\log_2 n)$ element exchanges might be necessary.

Assume that n is the input size of the positioning problem. For fixed integer $c \geq 1$ and real number $0 < \beta < 1$, our algorithm will need cn^β integers whose value is between 0 and n. In order to represent these integers using the bit encoding technique we find $cn^\beta 2\lceil \log_2(n+1) \rceil$ distinct elements and transfer these into the beginning of the input sequence. Next we describe how this preprocessing is done.

For the sake of brevity, let $b = cn^\beta 2\lceil \log_2(n+1) \rceil$. First, we sort the b first elements of the input; let B be the resulting section. Here any in-place sorting algorithm can be used, e.g., heapsort [23] or in-place mergesort [13]. This requires $O(b \log_2 b)$ time. If each element in B has less than $b/2$ duplicates—this check requires $O(b)$ time—the elements $B[i]$ and $B[b/2+i]$ must be dis-

tinct for all $i = 1, 2, \ldots, b/2$ and each of these pairs can be used to represent a bit. The interleaving of the first and the second half of B, i.e., moving the pairs $B[i]$ and $B[b/2 + i]$ into consecutive locations for $i = 1, 2, \ldots, b/2$, is easily carried out in-place in $O(b \log_2 b)$ time. At the same time the bits can be initialized to zero. In this case the construction is complete.

Second, if some element, say z, appears in B at least $b/2$ times, we scan the remaining sequence to find $b/2$ elements that are different from z. Before starting the scan we move all elements of B different from z to the beginning of B forming a section B_1. This requires a single block interchange, i.e., $O(b)$ time. The rest elements, elements equal to z, are divided to two consecutive sections, B_2 and B_3, so that B_3 includes $b/2$ elements and B_2 what is left. The section B_2 between B_1 and B_3 is to be filled with elements different from z. After B_3 there is a section, call it E, that also contains elements equal to z. That is, during the scan the input sequence has the form:



If the element pointed to by cursor p is equal to z, the cursor is incremented by one, E is made one larger, and the element at the new cursor position is examined next. Otherwise, the element pointed to is swapped with the first element of B_2, B_1 becomes one larger, B_2 one smaller, E one larger, the cursor is incremented by one, and the new cursor position is considered next. The scan is stopped when B_2 gets full or when the end of the input sequence is reached.

If the end of the sequence is reached before B_2 gets full, we know that most of the elements are equal to z and the underlying positioning problem is easy to solve. The elements in B_1 are sorted; let $B_<$ denote the section of elements less than z in the sorting results and $B_>$ the section of elements larger than z. Then the contents of $B_>$ is swapped with a block of equal size at the end of $B_2 \cup B_3 \cup E$. Since after this the whole sequence is in sorted order, the kth smallest element is correctly positioned. The sorting of B_1 requires at most $O(b \log_2 b)$ time, and the block swap involving $B_>$ at most $O(b)$ time. During the scan one element comparison is made for each element kept in E if the ordering relation $\ominus$ is provided. Since $b \log_2 b = o(n)$, this special case is solved in $O(n)$ time using at most $n + o(n)$ element comparisons and at most $o(n)$ element exchanges. If only the ordering relation $\oslash$ is provided, two comparisons per element might be necessary.

Let us hereafter assume that there are enough elements different from z. Assume that the configuration after the scan is $B_1 B_3 E U$, where U denotes the rest of the sequence not yet touched. As earlier, the interleaving of B_1 and B_3 requires $O(b \log_2 b)$ time. This completes the preprocessing and the final configuration is BEU, B containing the consecutive pairs that can be used for representing $b/2$ bits. To get to this configuration $O(n)$ time has been used and it is necessary to carry out one element comparison for

each element in E (or two if only the ordering relation $\lessdot$ is provided). Potentially, E can be large, but later on we can avoid the comparisons involving the elements in E since these are known to be equal to z.

3. Randomized positioning

As our starting point we use a non-recursive variant of the selection algorithm of Floyd and Rivest [5, 6] described, for example, in [18, Section 3.3]. The basic idea is to draw a small random sample from the input sequence with replacement, choose two sample elements x and y, $x \lessdot y$, and then partition the sequence into three sections: the elements lexicographically less than x, the elements between x and y, including themselves, and the elements lexicographically greater than y. The crux is how to choose elements x and y such that the number of elements between them is $o(n)$, x is lexicographically less than the kth element, and y is lexicographically greater than the kth element. If these conditions are satisfied, it is easy to move the kth element to its correct position by sorting the elements falling between x and y, and this sorting takes only $o(n)$ time.

Let us now turn our attention to implementation details. Assume that we have preprocessed the input sequence as described in Section 2 for $b = 4n^{\beta}\lceil \log_2(n+1) \rceil$, where the constant β, $0 < \beta < 1$, will be determined later, and that the sequence has the form BEU. Let T be a shorthand notation for the union of sections E and U, and let $m = |T| = n - b$. The positioning of the kth element is accomplished as follows.

Take a random sample. Let $s = m^{\beta}$. Draw s random integers, independently and uniformly, from the range $\{1, 2, \ldots, m\}$ and store these numbers in B in encoded form. Clearly, this requires $O(s \log_2 n)$ time.

Sort the s integers in B using any in-place sorting algorithm. Each time two numbers are swapped their encodings in B are swapped as well. Therefore, this sorting requires $O(s \log_2 s \log_2 n)$ time.

Scan the sorted number sequence, starting from the back, determine the multiplicity of each number, and save the original number together with its multiplicity in encoded form in B. Because the B section is is so large, the old sequence and the new sequence cannot overlap during this process. Due to the manipulation of the encoded number representations the scan requires $O(s \log_2 n)$ time.

The integers in the number sequence indicate the indices of the elements to be chosen to the random sample, and the multiplicities tell how many times each element appears in the sample. That is, the sampling is done with replacement. Now scan the number sequence and gather the elements chosen to the sample together to the beginning of T, call the resulting section S. Again, since encoded representations are manipulated, this scan requires $O(s \log_2 n)$ time.

Choose elements x and y. Sort the elements in S using any in-place sorting algorithm. When in this process two elements are swapped, also the

encoded indices and multiplicities in B are swapped. Thus, this requires $O(s \log_2 s \log_2 n)$ time.

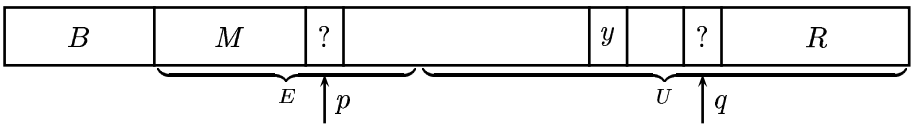
Let α and γ be some fixed constants, $0 < \alpha < \beta < \gamma < 1$. If $k < m^\gamma$, let $\lambda = \nu = 2m^\gamma s/m$. On the other hand, if $k \geq m^\gamma$, let $\mu_\ell = (k-b)s/m$, $\mu_r = (k+b)s/m$, $\Delta_\ell = m^\alpha \mu_\ell^{1/2}$, $\Delta_r = m^\alpha \mu_r^{1/2}$, $\lambda = \max\{1, \lfloor \mu_\ell - \Delta_\ell \rfloor\}$, and $\nu = \min\{\lceil \mu_r + \Delta_r \rceil, s\}$. Now scan the S section to find the λth element of the sample, call it x, and the νth element, call it y. Since in this scan we have to determinate the value of multiplicities one by one, it takes $O(s \log_2 n)$ time. After this the indices of x and y are saved.

Undo element moves in T. Once more sort the elements in S, but now use their indices as keys. Again the encoded indices and multiplicities in B are moved accordingly. After this, move the elements in S back to their original positions. This is done backwards starting from the rear. In total, this requires $O(s \log_2 s \log_2 n)$ time.

Carry out two-way partitioning. If $k < m^\gamma$, perform a two-way partitioning of T using y as the partitioning element. Let M (R) denote the section of elements lexicographically smaller than or equal to (larger than) y. Initially, both M and R are empty. We use a modification of the meeting cursors strategy by Hoare [9]. Initially, the two cursors p and q are at their respective ends of T, and they are gradually moved toward each other until they meet. The partitioning should be done carefully so that each element of U is compared to y exactly once, and that the comparison between the elements of E and y are avoided as far as possible. To achieve this, the relationship between z and y is determined before partitioning, and during the partitioning, when cursors p or q are in the E section, the elements pointed to by them are not compared to y, but the outcome of the initial comparison is recalled.

The partitioning procedure consists of three loops which all are similar in structure. In the first loop the cursors p and q are moved toward the centre until one of them meets the index of y. Depending on whether p or q meets the index first, there are two other loops. In both of these the movement of the cursors is continued until the cursors meet each other. Let us now consider the first loop in detail; the other two loops are similar and hence their consideration is omitted.

We maintain the following invariant in the first loop:



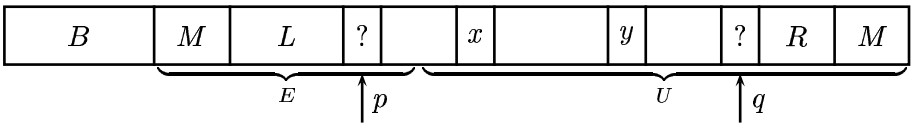
If the element pointed to by p is larger than y, the cursor is stopped; otherwise M is made one larger and the cursor p is incremented by one. If the element pointed to by q is smaller than y, also this cursor is stopped; otherwise R is made one larger and the cursor q is decremented by one. After both cursors are stopped, the elements pointed to by them are swapped. This makes both M and R one larger. Cursor p is incremented by one and

cursor q is decremented by one, and the process is repeated until one of the cursors meets the index of y.

Clearly, this two-way partitioning is carried out in $O(m)$ time and at most $|U| + O(1)$ element comparisons are performed. For each element moved to M an element exchange might be necessary. With high probability the final size of the M section will be $o(m)$ so only $o(m)$ element exchanges will be necessary in this case.

Perform three-way partitioning. If $m^\gamma \leq k \leq \lceil m/2 \rceil$, perform three-way partitioning using x and y as the partitioning elements and collect the elements falling between them (including x and y) into M. Let L denote the section containing elements lexicographically smaller than x and R the section containing those larger than y. There are two symmetric cases depending on whether the index of x is smaller than or larger than the index of y. We consider only the first case here; the other case is similar.

Again the partitioning is based on the meeting cursors strategy. The partitioning routine consists of several symmetric loops depending on the relative positions of the cursors p and q and the index of x and the index of y. We consider here only the first loop when the cursors have not yet met the index of x or the index of y. During the process we maintain the following invariant:



We maintain an M section both in the beginning of the sequence and in the end of the sequence, and then at the end swap the M sections into the middle. The elements in the E section are handled as in two-way partitioning in order to avoid unnecessary element comparisons.

Consider the invariant maintained in the first partitioning loop. Each element under consideration is always compared first to y, and thereafter to x if necessary. If the element pointed to by cursor p is larger than y, the cursor is stopped; if not and if it is larger than x, it belongs to M and it is exchanged with the first element of L, which makes the first M section one larger, moves the L section one position to the right, and thereafter cursor p is incremented by one. Otherwise, the element under consideration belongs to L, and that section is made one larger and cursor p is incremented by one. The movement of q backwards is done equally carefully using the M section at the end. After both cursors are stopped, the underlying elements are exchanged, and the process is repeated until one of the cursors meets the index of x or the index of y.

From this implementation it is clearly seen that three-way partitioning runs in $O(m)$ time. Each element of U is compared to y and possibly to x. Hence, the total number of element comparisons is bounded by $|U| + \min\{|U|, k + o(m)\} + O(1)$. With high probability, the final size of the M section will be $o(m)$. Therefore, only the element exchanges involving an

element of L are significant. With high probability, the L section will get at most $k + o(m)$ elements, so the number of element exchanges performed is bounded by $k + o(m)$.

Reordering. Change the configuration of the sequence from $BVXZ$ (where L might be empty if $k < m^\gamma$) to $VBXZ$ by swapping the elements in B with a block of the same size at the end of L; if $|L| < |B|$, simply interchange the order of the blocks in these sections. Since B contains $O(s \log_2 n)$ elements, this interchange requires $O(s \log_2 n)$ time.

Finishing up. If $|L| \geq k - b$ or $|R| \geq n - k - b$, the correct positioning of the kth element is guaranteed by sorting the whole sequence using any in-place sorting algorithm. This takes $O(n \log_2 n)$ time, but the sorting will be necessary with negligible probability.

Otherwise, the correct positioning of the kth element is guaranteed by sorting the section BX in-place. This is the normal mode. With high probability the final size of the M section will be $o(m)$, and since the size of the B section is bounded by $O(s \log_2 s)$, which is also $o(m)$, this sorting requires $o(m)$ time.

4. Analysis

Now we are ready to prove our main theorem.

Theorem 1. *Our Las-Vegas algorithm carries out the positioning of the kth element in a multiset of size n in-place in $O(n)$ time using at most $n + k + o(n)$ pairwise element comparisons, and at most $k + o(n)$ element exchanges; these resource bounds are exceeded with probability $e^{-n^{\Omega(1)}}$.*

Proof. The space bound is obvious since all computations are performed in-place. The final sorting guarantees that the algorithm is of the Las-Vegas type. Since $s = m^\beta$, $0 < \beta < 1$, and $m \leq n$, all the phases where we operate with the sample take $O(s(\log_2 n)^2) = o(n)$ time. Hence, the computational costs are dominated by the preprocessing phase, one of the partitioning phases, and the final sorting phase. Consider first the normal mode, i.e., the case that the kth element falls in M and $|M| < n^\varepsilon$ for some ε, $0 < \varepsilon < 1$. Clearly, in this case the running time of the algorithm is $O(n)$, the number of pairwise element comparisons sums to $n + k + o(n)$, and that of element exchanges to $k + o(n)$.

The algorithm may fail to meet these resource bounds in six ways:

1. If $k < m^\gamma$ and M gets too small, so that the kth element is not in the union of B and M. We will be pessimistic and say that a failure occurs if y is lexicographically larger than the m^γth element of T.
2. If $k < m^\gamma$ and M gets too large, so that the sorting phase is too costly. Here we say that a failure occurs if $|M| > 4m^\gamma$.
3. If $k \geq m^\gamma$ and U gets too large, so that U can contain the kth element. This failure occurs if $|U| \geq k - b$.
4. If $k \geq m^\gamma$ and R gets too large, so R can contain the kth element. This failure occurs if $|R| \geq n - k - b$.

5. If $k \geq m^\gamma$ and the left boundary of M gets too far away to the left, so that the sorting phase becomes costly. We say that this failure occurs if x is lexicographically smaller than the $(k - b - \Delta_\ell m/s)$th element in T.
6. Finally, it is possible that $k \geq m^\gamma$ and the right boundary of M gets too far away to the right. We say that this failure occurs if y is lexicographically larger than the $(k + b + \Delta_r m/s)$th element in T.

The probabilities of these failures can be bounded above by using Chernoff bounds (see [18, Theorem 4.2 and Theorem 4.3]). We consider here the failure modes 2 and 3; the other four modes are handled in a similar way.

Failure mode 2. If this failure occurs, more than $4m^\gamma s/m$ of the sample elements are lexicographically larger than the $(2m^\gamma)$th element of T. Let $X_i = 1$, if the ith sample element is smaller than or equal to the $(2m^\gamma)$th element of T, and $X_i = 0$ otherwise. Thus, $\Pr[X_i = 1] = 2m^\gamma/m$, and $\Pr[X_i = 0] = 1 - 2m^\gamma/m$. For $X = \sum_{i=1}^{s} X_i$, $\mathbf{E}[X] = 2m^\gamma s/m$. Since X is binomially distributed and $X_1, X_2, \ldots, X_s$ are all independent, we can use [18, Theorem 4.2] to bound its upper tail probability:

$$\begin{aligned} \Pr[X > 4m^\gamma s/m] &= \Pr[X > 2\mathbf{E}[X]] \\ &\overset{\text{Theorem 4.2}}{\leq} e^{-m^{\gamma+\beta-1}/2} \\ &\overset{\beta=2/3 \text{ and } \gamma=5/6}{=} e^{-m^{1/2}/2}. \end{aligned}$$

Failure mode 3. Recall that $\mu_\ell = (k - b)s/m$ and $\Delta_\ell = m^\alpha \mu_\ell^{1/2}$. If this failure occurs, less than $(k - b)s/m - \Delta_\ell$ of the sample elements are lexicographically smaller than the $(k - b)$th element of T. Define $X_i = 1$, if the ith sample element is lexicographically smaller than the $(k-b)$th element of T, and $X_i = 0$ otherwise. For $X = \sum_{i=1}^{s} X_i$, $\mathbf{E}[X] = \mu_\ell = (k - b)s/m$. Now we can bound the lower tail probability of X using [18, Theorem 4.3]:

$$\begin{aligned} \Pr[X < \mu_\ell - \Delta_\ell] &\overset{\delta=\Delta_\ell/\mu_\ell}{=} \Pr[X < (1 - \delta)\mu_\ell] \\ &\overset{\text{Theorem 4.3}}{\leq} e^{\mu_\ell \delta^2/2} \\ &= e^{-m^{2\alpha}/2}. \end{aligned}$$

With parameters $\alpha = 1/6$, $\beta = 2/3$, and $\gamma = 5/6$. we have that $\delta \leq 2e - 1$, so we can use the simplified Chernoff bound [18, Theorem 4.3].

Summing up. Since the probability of the union of events is at most the sum of their probabilities, the probability that some of the mentioned failures occurs is still negligible. Up to now we have expressed the failure probabilities as a function of m, but $m = n - s \geq n/2$ when n is large enough, so the failure probability is of the form $e^{-n^{\Omega(1)}}$. □□

5. Concluding remarks

Since the selection problem has been extensively studied in the literature, it was a surprise for us when we observed that Hoare's randomized algorithm

was the only one that solves the positioning problem, operates in-place, is able to handle multiset data, and relies only on pairwise element comparisons. In this paper we described a more efficient randomized algorithm for positioning having all these desirable properties.

Due to the in-place requirement our algorithm is quite complicated but, if $o(n)$ extra space is available, the bit emulation can be avoided. In the experiments carried out when writing the paper [2] such a randomized method using $o(n)$ extra space turned out to be the fastest alternative if there were many duplicates among the input. However, if the duplicates were rare, this method specialized for multisets was slower than the unspecialized methods due to more complex index manipulations.

Assuming that assignments of the form $x = A[i]$ and $A[i] = A[j]$ are considered as element moves, an element exchange may require three moves. Using the hole technique (see, e.g., [8]), it is possible to implement our randomized positioning algorithm so that with high probability it will carry out at most $2k + o(n)$ element moves. If we are only interested in selection, it would be easy to modify the algorithm so that it will perform $o(n)$ element moves with high probability.

References

[1] 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** (1973), 448–461.

[2] J. Bojesen, J. Katajainen, and M. Spork, Performance engineering case study: heap construction, *The ACM Journal of Experimental Algorithmics* (to appear).

[3] S. Carlsson and M. Sundström, Linear-time in-place selection in less than $3n$ comparisons, *Proceedings of the 6th International Symposium on Algorithms and Computation*, *Lecture Notes in Computer Science* **1004**, Springer-Verlag, Berlin/Heidelberg, Germany (1995), 244–253. A longer version is available at `http://www.sm.luth.se/~msm/publications.html`.

[4] W. Cunto and J. I. Munro, Average case selection, *Journal of the ACM* **36** (1989), 270–279.

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

[6] R. W. Floyd and R. L. Rivest, Algorithm 489: The algorithm Select—for finding the ith smallest of n elements, *Communications of the ACM* **18** (1975), 173.

[7] V. Geffert, Linear-time in-place selection in $\varepsilon \cdot n$ element moves, Unpublished typescript (2000).

[8] V. Geffert, J. Katajainen, and T. Pasanen, Asymptotically efficient in-place merging, *Theoretical Computer Science* **237** (2000), 159–181.

[9] C. A. R. Hoare, Algorithm 65: Find, *Communications of the ACM* **4** (1961), 321–322.

[10] ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission), *International Standard ISO/IEC 14882: Programming Languages — C++*, Genève, Switzerland (1998).

[11] J. Katajainen and T. Pasanen, Stable minimum space partitioning in linear time, *BIT* **32** (1992), 580–585.

[12] J. Katajainen and T. Pasanen, Sorting multiset stably in minimum space, *Acta Informatica* **31** (1994), 301–313.

[13] J. Katajainen, T. Pasanen, and J. Teuhola, Practical in-place mergesort, *Nordic Journal of Computing* **3** (1996), 27–40.

[14] J. Katajainen and T. A. Pasanen, In-place sorting with fewer moves, *Information*

Processing Letters **70** (1999), 31–37.

[15] P. Kirschenhofer, H. Prodinger, and C. Martínez, Analysis of Hoare's Find algorithm with median-of-three partition, *Random Structures and Algorithms* **10** (1997), 143–156.

[16] T. W. Lai and D. Wood, Implicit selection, *Proceedings of the 1st Scandinavian Workshop on Algorithm Theory*, *Lecture Notes in Computer Science* **318**, Springer-Verlag, Berlin/Heidelberg, Germany (1988), 14–23.

[17] C. Levcopoulos and O. Petersson, Exploiting few inversions when sorting: Sequential and parallel algorithms, *Theoretical Computer Science* **163** (1996), 211–238.

[18] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, Cambridge, England (1995).

[19] 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** (1986), 66–74.

[20] J. T. Postamus, A. H. G. Rinnooy Kan, and G. T. Timmer, An efficient dynamic selection method, *Communications of the ACM* **26** (1983), 878–881.

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

[22] R. Sedgewick, The analysis of Quicksort programs, *Acta Informatica* **7** (1977), 327–355.

[23] J. W. J. Williams, Algorithm 232: Heapsort, *Communications of the ACM* **7** (1964), 347–348.