Selection from Heaps, Row-Sorted Matrices, and $X + Y$ Using Soft Heaps

Haim Kaplan[1]
Blavatnik School of Computer Science, Tel Aviv University, Israel
haimk@post.tau.ac.il

László Kozma[2]
Department of Mathematics and Computer Science, TU Eindhoven, The Netherlands
lkozma@gmail.com

Or Zamir[3]
Blavatnik School of Computer Science, Tel Aviv University, Israel
orzamir@mail.tau.ac.il

Uri Zwick[4]
Blavatnik School of Computer Science, Tel Aviv University, Israel
zwick@tau.ac.il

Abstract

We use *soft heaps* to obtain simpler optimal algorithms for selecting the k-th smallest item, and the set of k smallest items, from a heap-ordered tree, from a collection of sorted lists, and from $X + Y$, where X and Y are two unsorted sets. Our results match, and in some ways extend and improve, classical results of Frederickson (1993) and Frederickson and Johnson (1982). In particular, for selecting the k-th smallest item, or the set of k smallest items, from a collection of m sorted lists we obtain a new optimal *"output-sensitive"* algorithm that performs only $O(m + \sum_{i=1}^m \log(k_i + 1))$ comparisons, where k_i is the number of items of the i-th list that belong to the overall set of k smallest items.

1 Introduction

The input to the standard *selection problem* is a set of n items, drawn from a totally ordered domain, and an integer $1 \le k \le n$. The goal is to return the k-th smallest item in the set. A classical result of Blum et al. [1] says that the selection problem can be solved deterministically in $O(n)$ time, i.e., faster than *sorting* the set. The number of comparisons required for selection was reduced by Schönhage et al. [26] to $3n$, and by Dor and Zwick [8, 9] to about $2.95n$.

In the *generalized selection* problem, we are also given a *partial order P* known to hold for the set of n input items. The goal is again to return the k-th smallest item. The corresponding *generalized sorting* problem was extensively studied. It was shown by Kahn and Saks [20] that the problem can be solved using only $O(\log e(P))$ comparisons, where $e(P)$ is the number of *linear extensions* of P. Thus, the information-theoretic lower bound is tight for generalized sorting. The algorithm of Kahn and Saks [20] performs only $O(\log e(P))$ comparisons, but may spend much more time on deciding which comparisons to perform. Kahn and Kim [19] and Cardinal et al. [4] gave algorithms that perform only $O(\log e(P))$ comparisons and run in polynomial time.

A moment's reflection shows that an algorithm that finds the k-th smallest item of a set, must also identify the set of k smallest items of the set.[5] Given a partial order P, let $s_k(P)$ be the number of subsets of size k that may possibly be the set of k smallest items in P. Then, $\log_2 s_k(P)$ is clearly a lower bound on the number of comparisons required to select the k-th smallest item, or the set of k smallest items. Unlike sorting, this information-theoretic lower bound for selection may be extremely weak. For example, the information-theoretic lower bound for selecting the *minimum* is only $\log_2 n$, while $n - 1$ comparisons are clearly needed (and are sufficient). To date, there is no characterization of pairs (P, k) for which the information-theoretic lower bound for selection is tight, nor an alternative general technique to obtain a tight lower bound.

Frederickson and Johnson [12, 13, 14] and Frederickson [11] studied the generalized selection problem for some interesting specific partial orders. Frederickson [11] considered the case in which the input items are items of a binary *min-heap*, i.e., they are arranged in a binary tree, with each item smaller than its two children. Frederickson [11] gave a complicated algorithm that finds the k-th smallest item using only $O(k)$ comparisons, matching the information-theoretic lower bound for this case. (Each subtree of size k of the heap, containing the root, can correspond to the set of k smallest items, and there are $\frac{1}{k+1}\binom{2k}{k}$, the k-th *Catalan* number, such subtrees.)

Frederickson and Johnson [12, 13, 14] considered three other interesting special cases. (i) The input items are in a collection of sorted lists, or equivalently they reside in a row-sorted matrix; (ii) The input items reside in a collection of matrices, where each matrix is both row- and column-sorted; (iii) The input items are $X + Y$, where X and Y are unsorted sets of items.[6] For each of these cases, they present a selection algorithm that matches the information-theoretic lower bound.

We note in passing that *sorting $X + Y$* is a well studied problem. Fredman [15] showed that $X + Y$, where $|X| = |Y| = n$, can be sorted using only $O(n^2)$ comparisons, but it is not known how to do it in $O(n^2)$ time. (An intriguing situation!) Fredman [15] also showed that $\Omega(n^2)$ comparisons are required, if only comparisons between items in $X + Y$, i.e., comparisons of the form $x_i + y_j \leq x_k + y_\ell$, are allowed. Lambert [24] and Steiger and Streinu [27] gave algorithms that sort $X + Y$ in $O(n^2 \log n)$ time using only $O(n^2)$ comparisons. Kane et al. [21], in a breakthrough result, have shown recently that $X + Y$ can be sorted using only $O(n \log^2 n)$ comparisons of the form $(x_i + y_j) - (x_{i'} + y_{j'}) \leq (x_k + y_\ell) - (x_{k'} + y_{\ell'})$, but it is again not known how to implement their algorithm efficiently.

---

[5]   The information gathered by a comparison-based algorithm corresponds to a partial order which can be represented by a DAG (Directed Acyclic Graph). Every *topological sort* of the DAG corresponds to a total order of the items consistent with the partial order. Suppose that e is claimed to be the k-th smallest item and suppose, for the sake of contradiction, that the set I of the items that are *incomparable* with e is non-empty. Then, there is a topological sort in which e is before all the items of I, and another topological sort in which e is after all the items of I, contradicting the fact that e is the k-th smallest item in all total orders consistent with the partial order.

[6]   By $X + Y$ we mean the set of pairwise sums $\{x + y \mid x \in X, y \in Y\}$.

The *median* of $X + Y$, on the other hand, can be found in $O(n \log n)$ time, and $O(n \log n)$ comparisons of items in $X + Y$, as was already shown by Johnson and Mizoguchi [18] and Johnson and Kashdan [17]. The selection problem from $X + Y$ becomes more challenging when $k = o(n^2)$.

Frederickson [11] gives two applications for selection from a binary min-heap. The first is in an algorithm for listing the k smallest spanning trees of an input graph. The second is a certain resource allocation problem. Eppstein [10] uses the heap selection algorithm in his $O(m + n \log n + k)$ algorithm for generating the k shortest paths between a pair of vertices in a digraph. As pointed out by Frederickson and Johnson [13], selection from $X + Y$ can be used to compute the Hodges-Lehmann [16] estimator in statistics. Selection from a matrix with sorted rows solves the problem of "optimum discrete distribution of effort" with concave functions, i.e., the problem of maximizing $\sum_{i=1}^{m} f_i(k_i)$ subject to $\sum_{i=1}^{m} k_i = k$, where the f_i's are concave and the k_i's are non-negative integers. (See Koopman [23] and other references in [13].) Selection from a matrix with sorted rows is also used by Brodal et al. [3] and Bremner et al. [2].

The $O(k)$ heap selection algorithm of Frederickson [11] is fairly complicated. The naïve algorithm for the problem runs in $O(k \log k)$ time. Frederickson first improves this to $O(k \log \log k)$, then to $O(k 3^{\log^* k})$, then to $O(k 2^{\log^* k})$, and finally to $O(k)$.

Our first result is a very simple $O(k)$ heap selection algorithm obtained by running the naïve $O(k \log k)$ algorithm using an auxiliary *soft heap* instead of a standard heap. Soft heaps, discussed below, are fairly simple data structures whose implementation is not much more complicated than the implementation of standard heaps. Our overall $O(k)$ algorithm is thus simple and easy to implement and comprehend.

Relying on our simple $O(k)$ heap selection algorithm, we obtain simplified algorithms for selection from row-sorted matrices and from $X + Y$. Selecting the k-th item from a row-sorted matrix with m rows using our algorithms requires $O(m + k)$ time, if $k \leq 2m$, and $O(m \log \frac{k}{m})$ time, if $k \geq 2m$, matching the optimal results of Frederickson and Johnson [12]. Furthermore, we obtain a new optimal "output-sensitive" algorithm whose running time is $O(m + \sum_{i=1}^{m} \log(k_i + 1))$, where k_i is the number of items of the i-th row that belong to the set of k smallest items in the matrix. We also use our simple $O(k)$ heap selection algorithm to obtain simple optimal algorithms for selection from $X + Y$.

Soft heaps are "approximate" priority queues introduced by Chazelle [6]. They form a major building block in his deterministic $O(m\alpha(m, n))$-time algorithm for finding minimum spanning trees [5], which is currently the fastest known deterministic algorithm for the problem. Chazelle [6] also shows that soft heaps can be used to obtain a simple linear time (standard) selection algorithm. (See the next section.) Pettie and Ramachandran [25] use soft heaps to obtain an *optimal* deterministic minimum spanning tree algorithm with a yet unknown running time. A simplified implementation of soft heaps is given in Kaplan et al. [22].

All algorithms considered in the paper are *comparison-based*, i.e., the only operations they perform on the input items are pairwise comparisons. In the selection from $X + Y$ problem, the algorithms make pairwise comparisons in X, in Y and in $X + Y$. The number of comparisons performed by the algorithms presented in this paper dominates the total running time of the algorithms.

The rest of the paper is organized as follows. In Section 2 we review the definition of soft heaps. In Section 3 we describe our heap selection algorithms. In Section 3.1 we describe our basic algorithm for selection from *binary* min-heaps. In Sections 3.2 and 3.3 we extend the algorithm to d-ary heaps and then to general heap-ordered trees and forests. In Section 4 we

describe our algorithms for selection from row-sorted matrices. In Section 4.1 we describe a simple $O(m + k)$ algorithm which is optimal if $k = O(m)$. In Section 4.2 we build on the $O(m + k)$ algorithm to obtain an optimal $O(m \log \frac{k}{m})$ algorithm, for $k \geq 2m$. In Sections 4.3 and 4.4 we obtain new results that were not obtained by Frederickson and Johnson [12]. In Section 4.3 we obtain an $O(m + \sum_{i=1}^{m} \log n_i)$ algorithm, where $n_i \geq 1$ is the length of the i-th row of the matrix. In Section 4.4 we obtain the new $O(m + \sum_{i=1}^{m} \log(k_i + 1))$ optimal output-sensitive algorithm. In Section 5 we present our algorithms for selection from $X + Y$. In Section 5.1 we give a simple $O(m + n + k)$ algorithm, where $|X| = m$, $|Y| = n$. In Section 5.2 we give a simple $O(m \log \frac{k}{m})$ algorithm, for $m \geq n$ and $k \geq 6m$. We conclude in Section 6 with some remarks and open problems.

2 Soft heaps

Soft heaps, invented by Chazelle [6], support the following operations:

    `soft-heap`(ε): Create and return a new, empty soft heap with *error parameter ε*.

    `insert`(Q, e): Insert item e into soft heap Q.

    `meld`(Q_1, Q_2): Return a soft heap containing all items in heaps Q_1 and Q_2, destroying Q_1 and Q_2.

    `extract-min`(Q): Delete from the soft heap and return an item of minimum key in heap Q.

In Chazelle [6], `extract-min` operations are broken into `find-min` and `delete` operations. We only need combined `extract-min` operations. We also do not need `meld` operations in this paper.

The main difference between soft heaps and regular heaps is that soft heaps are allowed to *increase* the keys of some of the items in the heap by an arbitrary amount. Such items are said to become *corrupt*. The soft heap implementation chooses which items to corrupt, and by how much to increase their keys. The only constraint is that for a certain *error parameter* $0 \leq \varepsilon < 1$, the number of corrupt items *in* the heap is at most εI, where I is the number of *insertions* performed so far. The ability to corrupt items allows the implementation of soft heaps to use what Chazelle [6] calls the "data structures equivalent of car pooling" to reduce the amortized time per operation to $O(\log \frac{1}{\varepsilon})$, which is the best possible dependency on ε. In the implementation of Kaplan et al. [22], `extract-min` operations take $O(\log \frac{1}{\varepsilon})$ amortized time, while all other operations take $O(1)$ amortized time. (In the implementation of Chazelle [6], `insert` operations take $O(\log \frac{1}{\varepsilon})$ amortized time while the other operations take $O(1)$ time.)

An `extract-min` operation returns an item whose current, possibly corrupt, key is the smallest in the heap. Ties are broken arbitrarily. (Soft heaps usually give many items the same corrupt key, even if initially all keys are distinct.) Each item e thus has two keys associated with it: its original key $e.key$, and its current key in the soft heap $e.key'$, where $e.key \leq e.key'$. If $e.key < e.key'$, then e is corrupt. The current key of an item may increase several times.

At first sight, it may seem that the guarantees provided by soft heaps are extremely weak. The only bound available is on the number of corrupt items currently *in* the heap. In particular, *all* items extracted from the heap may be corrupt. Nonetheless, soft heaps prove to be an extremely useful data structure. In particular, they play a key role in the fastest known deterministic algorithm of Chazelle [5] for finding minimum spanning trees.

Soft heaps can also be used, as shown by Chazelle [6], to select an *approximate median* of n items. Initialize a soft heap with some error parameter $\varepsilon < \frac{1}{2}$. Insert the n items into the soft heap and then perform $(1-\varepsilon)\frac{n}{2}$ `extract-min` operations. Find the maximum item e, with respect to the original keys, among the extracted items. The rank of e is between $(1-\varepsilon)\frac{n}{2}$ and $(1+\varepsilon)\frac{n}{2}$. The rank is at least $(1-\varepsilon)\frac{n}{2}$ as e is the largest among $(1-\varepsilon)\frac{n}{2}$ items. The rank is at most $(1+\varepsilon)\frac{n}{2}$ as the items remaining in the soft heap that are smaller than e must be corrupt, so there are at most εn such items. For, say, $\varepsilon = \frac{1}{4}$, the running time of the algorithm is $O(n)$.

Using a linear time approximate median algorithm, we can easily obtain a linear time algorithm for selecting the k-th smallest item. We first compute the true rank r of the approximate median e. If $r = k$ we are done. If $r > k$, we throw away all items larger than e. Otherwise, we throw away all items smaller than e and replace k by $k - r$. We then continue recursively. In $O(n)$ time, we reduced n to at most $(\frac{1}{2} + \varepsilon)n$, so the total running time is $O(n)$.

In this paper, we show the usefulness of soft heaps in solving generalized selection problems. We obtain simpler algorithms than those known before, and some results that were not known before.

In Chazelle [6] and Kaplan et al. [22], soft heaps may corrupt items while performing any type of operation. It is easy, however, to slightly change the implementation of [22] such that corruptions only occur *following* `extract-min` operations. In particular, `insert` operations do not cause corruption, and an `extract-min` operation returns an item with a smallest current key at the *beginning* of the operation. These assumptions simplify algorithms that use soft heaps, and further simplify their analysis. The changes needed in the implementation of soft heaps to meet these assumptions are minimal. The operations `insert` (and `meld`) are simply implemented in a *lazy* way. The implementation of [22] already has the property that `extract-min` operations cause corruptions only *after* extracting an item with minimum current key.

We assume that an `extract-min` operation returns a pair (e, C), where e is the extracted item, and C is a list of items that became corrupt after the extraction of e, i.e., items that were not corrupt before the operation, but are corrupt after it. We also assume that *e.corrupt* is a bit that says whether e is corrupt. (Note that *e.corrupt* is simply a shorthand for *e.key* < *e.key'*.) It is again easy to change the implementation of [22] so that `extract-min` operations return a list C of newly corrupt items, without affecting the amortized running times of the various operations. (In particular, the amortized running time of an `extract-min` operation is still $O(\log \frac{1}{\varepsilon})$, independent of the length of C. As each item becomes corrupt only once, it is easy to charge the cost of adding an item to C to its insertion into the heap.)

We stress that the assumptions we make on soft heaps in this paper can be met by minor and straightforward modifications of the implementation of Kaplan et al. [22], as sketched above. No complexities are hidden here. We further believe that due to their usefulness, these assumptions will become the standard assumptions regarding soft heaps.

3   Selection from heap-ordered trees

In Section 3.1 we present our simple, soft heap-based, $O(k)$ algorithm for selecting the k-th smallest item, and the set of k smallest items from a binary min-heap. This algorithm is the cornerstone of this paper. For simplicity, we assume throughout this section that the input heap is infinite. In particular, each item e in the input heap has two children *e.left* and *e.right*. (A non-existent child is represented by a dummy item with key $+\infty$.) In Section 3.2

Heap-Select(r):

> $S \leftarrow \emptyset$
> $Q \leftarrow \text{heap}()$
> $\text{insert}(Q, r)$
> **for** $i \leftarrow 1$ **to** k **do**
> > $e \leftarrow \text{extract-min}(Q)$
> > $\text{append}(S, e)$
> > $\text{insert}(Q, e.left)$
> > $\text{insert}(Q, e.right)$
>
> **return** S

Figure 1 Extracting the k smallest items from a binary min-heap with root r using a standard heap.

Soft-Select(r):

> $S \leftarrow \emptyset$
> $Q \leftarrow \text{soft-heap}(1/4)$
> $\text{insert}(Q, r)$
> $\text{append}(S, r)$
> **for** $i \leftarrow 1$ **to** $k - 1$ **do**
> > $(e, C) \leftarrow \text{extract-min}(Q)$
> > **if** ***not*** $e.corrupt$ **then**
> > > $C \leftarrow C \cup \{e\}$
> >
> > **for** $e \in C$ **do**
> > > $\text{insert}(Q, e.left)$
> > > $\text{insert}(Q, e.right)$
> > > $\text{append}(S, e.left)$
> > > $\text{append}(S, e.right)$
>
> **return** $\text{select}(S, k)$

Figure 2 Extracting the k smallest items from a binary min-heap with root r using a soft heap.

we adapt the algorithm to work for d-ary heaps, for $d \geq 3$, using "on-the-fly ternarization via heapification". In Section 3.3 we extend the algorithm to work on any heap-ordered tree or forest. The results of Section 3.3 are new.

3.1 Selection from binary heaps

The naïve algorithm for selection from a binary min-heap is given in Figure 1. The root r of the input heap is inserted into an auxiliary heap (priority queue), denoted Q. The minimal item e is extracted from Q and appended to a list S. The two children of e (in the input heap), if they exist, are inserted into Q. This operation is repeated k times. After k iterations, the items in S are the k smallest items in the input heap, in sorted order. Overall, $2k + 1$ items are inserted into Q and k items are extracted, so the total running time is $O(k \log k)$, which is optimal if the k smallest items are to be reported in sorted order.

Frederickson [11] devised a very complicated algorithm that outputs the k smallest items, not necessarily in sorted order, in only $O(k)$ time, matching the information-theoretic lower bound. In Figure 2 we give our very simple algorithm Soft-Select(r) for the same task, which also performs only $O(k)$ comparisons and runs in optimal $O(k)$ time. Our algorithm is a simple modification of the naïve algorithm of Figure 1 with the auxiliary heap replaced by a soft heap. The resulting algorithm is much simpler than the algorithm of Frederickson [11].

Algorithm Soft-Select(r) begins by initializing a soft heap Q with error parameter $\varepsilon = 1/4$ and by inserting the root r of the input heap into it. Items inserted into the soft heap Q are also inserted into a list S. The algorithm then performs $k - 1$ iterations. In each iteration, the operation $(e, C) \leftarrow \text{extract-min}$ extracts an item e with the smallest (possibly corrupt) key currently in Q, and also returns the set of items C that become corrupt as a result of the removal of e from Q. If e is *not* corrupt, then it is added to C. Now, for each item $e \in C$, we insert its two children $e.left$ and $e.right$ into the soft heap Q and the list S.

■ **Figure 3** Types of items in the input heap. White nodes belong to A, i.e., were not inserted yet into the soft heap Q; black nodes belong to the barrier B; gray nodes belong to C, i.e., are corrupt; striped nodes belong to D, i.e., were already deleted.

Lemma 1 below claims that `Soft-Select`(r) inserts the k smallest items of the input heap into the soft heap Q. Lemma 2 claims that, overall, only $O(k)$ items are inserted into Q, and hence into S. Thus, the k smallest items in the input heap can be found by selecting the k smallest items in the list S using a standard selection algorithm.

▶ **Lemma 1.** *Algorithm* `Soft-Select`(r) *inserts the k smallest items from the input binary min-heap into the soft heap Q. (Some of them may subsequently be extracted from the heap.)*

Proof. At the beginning of an iteration of algorithm `Soft-Select`, let A be the set of items of the input binary heap that were not yet inserted into the soft heap Q; let B be the set of items that were inserted, not yet removed and are not corrupt; let C be the set of items that were inserted, not yet removed, and are *corrupt*; let D be the set of items that were inserted and already *deleted* from Q. We prove below, by easy induction, the following two invariants:
(a) All strict ancestors of items in B are in $C \cup D$.
(b) Each item in A has an ancestor in B.

Thus, the items in B form a *barrier* that separates the items of A, i.e., items that were not inserted yet into the heap, from the items of $C \cup D$, i.e., items that were inserted and are either corrupt or were already removed from the soft heap Q. For an example, see Figure 3.

Invariants (a) and (b) clearly hold at the beginning of the first iteration, when $B = \{r\}$ and $C \cup D$ is empty. Assume that (a) and (b) hold at the beginning of some iteration. Each iteration removes an item from the soft heap. The item removed is either a corrupt item from C, or an item (in fact the smallest item) on the barrier B. Following the extraction, some items on the barrier B become corrupt and move to C. The barrier is 'mended' by inserting to Q the children of items on B that were extracted or became corrupt. By our simplifying assumption, insertions do not corrupt items, so the newly inserted items belong to B and are thus part of the new barrier, reestablishing (a) and (b).

We now make the following two additional claims:
(c) The item extracted at each iteration is smaller than or equal to the smallest item on the barrier. (With respect to the original keys.)
(d) The smallest item on the barrier cannot decrease.

Claim (c) follows immediately from the definition of an `extract-min` operation and our assumption that corruption occurs only after an extraction. All the items on the barrier, and in particular the smallest item e on the barrier, are in the soft heap and are not corrupt.

Thus, the extracted item is either e, or a corrupt item f whose corrupt key is still smaller than e. As corruption can only increase keys, we have $f < e$.

Claim (d) clearly holds as items on the barrier at the end of an iteration were either on the barrier at the beginning of the iteration, or are children of items that were on the barrier at the beginning of the iteration.

Consider now the smallest item e on the barrier after $k - 1$ iterations. As all extracted items are smaller than it, the rank of e is at least k. Furthermore, all items smaller than e must be in $C \cup D$, i.e., inserted at some stage into the heap. Indeed, let f be an item of A, i.e., an item not inserted into Q. By invariant (b), f has an ancestor f' on the barrier. By heap order and the assumption that e is the smallest item on the barrier we indeed get $e \leq f' < f$. Thus, the smallest k items were indeed inserted into the soft heap as claimed. ◄

The proof of Lemma 1 relies on our assumption that corruptions in the soft heap occur only after `extract-min` operations. A slight change in the algorithm is needed if `insert` operations may cause corruptions; we need to repeatedly add children of newly corrupt items until no new items become corrupt. (Lemma 2 below shows that this process must end if $\varepsilon < \frac{1}{2}$. The process may *not* end if $\varepsilon \geq \frac{1}{2}$.) The algorithm, without any change, remains correct, and in particular Lemma 1 holds, if `extract-min` operations are allowed to corrupt items before extracting an item of minimum (corrupt) key. The proof, however, becomes more complicated. (Claim (c), for example, does not hold in that case.)

▶ **Lemma 2.** *Algorithm* `Soft-Select`(r) *inserts only* $O(k)$ *items into the soft heap* Q.

Proof. Let I be the number of insertions made by `Soft-Select`(r), and let C be the number of items that become corrupt during the running of the algorithm. (Note that `Soft-Select`(r) clearly terminates.) Let $\varepsilon(= \frac{1}{4})$ be the error parameter of the soft heap. We have $I < 2k + 2C$, as each inserted item is either the root r, or a child of an item extracted during one of the $k - 1$ iterations of the algorithm, and there are at most $2k - 1$ such insertions, or a child of a corrupt item, and there are exactly $2C$ such insertions. We also have $C < k + \varepsilon I$, as by the definition of soft heaps, at the end of the process at most εI items in the soft heap may be corrupt, and as only $k - 1$ (possibly corrupt) items were removed from the soft heap. Combining these two inequalities we get $C < k + \varepsilon(2k + 2C)$, and hence $(1 - 2\varepsilon)C < (1 + 2\varepsilon)k$. Thus, if $\varepsilon < \frac{1}{2}$ we get

$$C \; < \; \frac{1 + 2\varepsilon}{1 - 2\varepsilon} k \quad , \quad I \; < \; 2\left(1 + \frac{1 + 2\varepsilon}{1 - 2\varepsilon}\right) k \; .$$

The number of insertions I is therefore $O(k)$, as claimed. (For $\varepsilon = \frac{1}{4}$, $I < 8k$.) ◄

Combining the two lemmas we easily get:

▶ **Theorem 3.** *Algorithm* `Soft-Select`(r) *selects the* k *smallest items of a binary min-heap in* $O(k)$ *time.*

Proof. The correctness of the algorithm follows from Lemmas 1 and 2. Lemma 2 also implies that only $O(k)$ operations are performed on the soft heap. As $\varepsilon = 1/4$, each operation takes $O(1)$ amortized time. The total running time, and the number of comparisons, performed by the loop of `Soft-Select`(r) is thus $O(k)$. As the size of S is $O(k)$, the selection of the smallest k items from S also takes only $O(k)$ time. ◄

3.2 Selection from d-ary heaps

Frederickson [11] claims, in the last sentence of his paper, that his algorithm for binary min-heaps can be modified to yield an optimal $O(dk)$ algorithm for d-ary min-heaps, for any $d \geq 2$, but no details are given. (In a d-ary heap, each node has (at most) d children.)

We present two simple $O(dk)$ algorithms for selecting the k smallest items from a d-ary min-heap. The first is a simple modification of the algorithm for the binary case. The second in a simple reduction from the d-ary case to the binary case.

Algorithm $\texttt{Soft-Select}(r)$ of Figure 2 can be easily adapted to work on d-ary heaps. We simply insert the d children of an extracted item, or an item that becomes corrupt, into the soft heap. If we again let I be the number of items inserted into the sort heap, and C be the number of items that become corrupt, we get $I < d(k + C)$ and $C < k + \varepsilon I$, and hence

$$ C \; < \; \frac{1 + d\varepsilon}{1 - d\varepsilon}\, k \quad , \quad I \; < \; d\left(1 + \frac{1 + d\varepsilon}{1 - d\varepsilon}\right) k \; , $$

provided that $\varepsilon < \frac{1}{d}$, e.g., $\varepsilon = \frac{1}{2d}$. The algorithm then performs $O(dk)$ $\texttt{insert}$ operations, each with an amortized cost of $O(1)$, and $k - 1$ $\texttt{extract-min}$ operations, each with an amortized cost of $O(\log \frac{1}{\varepsilon}) = O(\log d)$. The total running time is therefore $O(dk)$. (Note that it is important here to use the soft heap implementation of [22], with an $O(1)$ amortized cost of $\texttt{insert}$.)

An alternative $O(dk)$ algorithm for d-ary heaps, for any $d \geq 2$, can be obtained by a simple reduction from d-ary heaps to 3-ary (or binary) heaps using a process that we call "on-the-fly ternarization via heapification". We use the well-known fact that an array of d items can be *heapified*, i.e., converted into a binary heap, in $O(d)$ time. (See Williams [28] or Cormen et al. [7].) We describe this alternative approach because we think it is interesting, and because we use it in the next section to obtain an algorithm for general heap-ordered trees, i.e., trees in which different nodes may have different degrees, and the degrees of the nodes are not necessarily bounded by a constant.

In a d-ary heap, each item e has (up to) d children $e.child[1], \ldots, e.child[d]$. We construct a *ternary* heap on the same set of items in the following way. We *heapify* the d children of e, i.e., construct a *binary* heap whose items are these d children. This gives each child f of e two new children $f.left$ and $f.right$. (Some of these new children are null.) We let $e.middle$ be the root of the heap composed of the children of e. Overall, this gives each item e in the original heap three new children $e.left$, $e.middle$ and $e.right$, some of which may be null. Note that e gets its new children $e.left$ and $e.right$ when it and its siblings are heapified. (The names *left*, *middle* and *right* are, of course, arbitrary.) For an example, see Figure 4.

This heapification process can be carried out on-the-fly while running $\texttt{Soft-Select}(r)$ on the resulting ternary heap. The algorithm starts by inserting the root of the d-ary heap, which is also the root of its ternarized version, into the soft heap. When an item e is extracted from the soft heap, or becomes corrupt, we do not immediately insert its d original children into the soft heap. Instead, we *heapify* its d children, in $O(d)$ time. This assigns e its middle child $e.middle$. Item e already has its left and right children $e.left$ and $e.right$ defined. The three new children $e.left$, $e.middle$ and $e.right$ are now inserted into the soft heap. We call the resulting algorithm $\texttt{Soft-Select-Heapify}(r)$.

▶ **Theorem 4.** *Algorithm* $\texttt{Soft-Select-Heapify}(r)$ *selects the k smallest items from a d-ary heap with root r in $O(dk)$ time.*
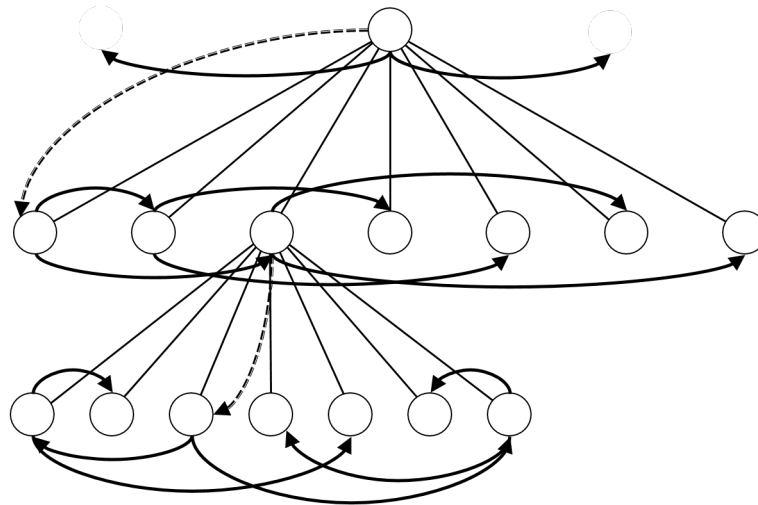
Figure 4 On-the-fly ternarization of a 7-ary heap. Thin lines represent the original 7-ary heap. Bold arrows represent new *left* and *right* children. Dashed arrows represent new *middle* children.

Proof. Algorithm `Soft-Select-Heapify`(r) essentially works on a ternary version of the input d-ary heap constructed on the fly. Simple adaptations of Lemmas 1 and 2 show that the total running time, excluding the heapifications' cost, is $O(k)$. As only $O(k)$ heapifications are performed, the cost of all heapifications is $O(dk)$, giving the total running time of the algorithm. ◀

It is also possible to *binarize* the input heap on the fly. We first ternarize the heap as above. We now convert the resulting ternary tree into a binary tree using the standard first child, next sibling representation. This converts the ternary heap into a binary heap, *if* the three children of each item are sorted. During the ternarization process, we can easily make sure that the three children of each item appear in sorted order, swapping children if necessary, so we can apply this final binarization step.

3.3 Selection from general heap-ordered trees

Algorithm `Soft-Select-Heapify`(r) works, of course, on arbitrary heap-ordered trees in which different nodes have different degrees. Algorithm `Soft-Select`(r), on the other hand, is not easily adapted to work on general heap-ordered trees, as it is unclear how to set the error parameter ε to obtain an optimal running time. To bound the running time of `Soft-Select-Heapify`(r) on an arbitrary heap-ordered tree, we introduce the following definition.

▶ **Definition 5** ($D(T, k)$)**.** Let T be a (possibly infinite) rooted tree and let $k \geq 1$. Let $D(T, k)$ be the maximum *sum of degrees* over all subtrees of T of size k rooted at the root of T. (The degrees summed are in T, not in the subtree.)

For example, if T_d is an infinite d-ary tree, then $D(T_d, k) = dk$, as the sum of degrees in each subtree of T_d containing k vertices is dk. For a more complicated example, let T be the infinite tree in which each node at level i has degree $i + 2$, i.e., the root has two children, each of which has three children, etc. Then, $D(T, k) = \sum_{i=2}^{k+1} i = k(k + 3)/2$, where the subtrees achieving the maximum are paths from the root. A simple adaptation of Theorem 4 gives:

▶ **Theorem 6.** *Let T be a heap-ordered tree with root r. Algorithm* `Soft-Select-Heapify`(r) *selects the k-th smallest item in T, and the set of k smallest items in T, in $O(D(T, 3k))$ time.*

Proof. We use the on-the-fly binarization and a soft heap with $\varepsilon = \frac{1}{6}$. The number of corrupt items is less than $2k$. The number of extracted items is less than k. Thus, the algorithm needs to heapify the children of less than $3k$ items that form a subtree T' of the original tree T. The sum of the degrees of these items is at most $D(T, 3k)$, thus the total time spent on the heapifications, which dominates the running time of the algorithm, is $O(D(T, 3k))$. We note that $D(T, 3k)$ can be replaced by $D(T, (2 + \delta)k)$, for any $\delta > 0$, by choosing ε small enough. ◀

▶ **Theorem 7.** *Let T be a heap-ordered tree and let $k \geq 1$. Any comparison-based algorithm for selecting the k-th smallest item in T must perform at least $D(T, k-1) - (k-1)$ comparisons on some inputs.*

Proof. Let T' be the subtree of T of size $k - 1$ that achieves the value $D(T, k - 1)$, i.e., the sum of the degrees of the nodes of T' is $D(T, k - 1)$. Suppose the $k - 1$ items of T' are the $k - 1$ smallest items in T. The nodes of T' have at least $D(T, k - 1) - (k - 2)$ children that are not in T'. The k-th smallest item is the minimum item among these items, and no information on the order of these items is implied by the heap order of the tree. Thus, finding the k-th smallest item in this case requires at least $D(T, k - 1) - (k - 1)$ comparisons. ◀

4 Selection from row-sorted matrices

In this section we present algorithms for selecting the k smallest items from a row-sorted matrix, or equivalently from a collection of sorted lists. Our results simplify and extend results of Frederickson and Johnson [12]. The algorithms presented in this section use our `Soft-Select` algorithm for selection from a binary min-heap presented in Section 3.1. (Frederickson's [11] algorithm could also be used, but the resulting algorithms would become much more complicated, in particular more complicated than the algorithms of Frederickson and Johnson [12].) In Section 4.1 we give an $O(m + k)$ algorithm, where m is the number of rows, which is optimal for $k = O(m)$. In Section 4.2 we give an $O(m \log \frac{k+m}{m})$ algorithm which is optimal for $k = \Omega(m)$. These results match results given by Frederickson and Johnson [12]. In Sections 4.3 and 4.4 we give two new algorithms that improve in some cases over the previous algorithms.

4.1 An $O(m + k)$ algorithm

A sorted list may be viewed as a heap-ordered path, i.e., a 1-ary heap. We can convert a collection of m sorted lists into a (degenerate) binary heap by building a binary tree whose leaves are the first items in the lists. The values of the $m - 1$ internal nodes in this tree are set to $-\infty$. Each item in a list will have one real child, its successor in the list, and a dummy child with value $+\infty$. To find the k smallest items in the lists, we simply find the $m + k - 1$ smallest items in the binary heap. This can be done in $O(m + k)$ time using algorithm `Soft-Select` of Section 3.1. More directly, we can use the following straightforward modification of algorithm `Soft-Select`. Insert the m first items in the lists into a soft heap. Perform $k - 1$ iterations in which an item with minimum (corrupt) key is extracted. Insert into the soft heap the child of the item extracted as well as the children of all the items that became corrupt following the `extract-min` operation.
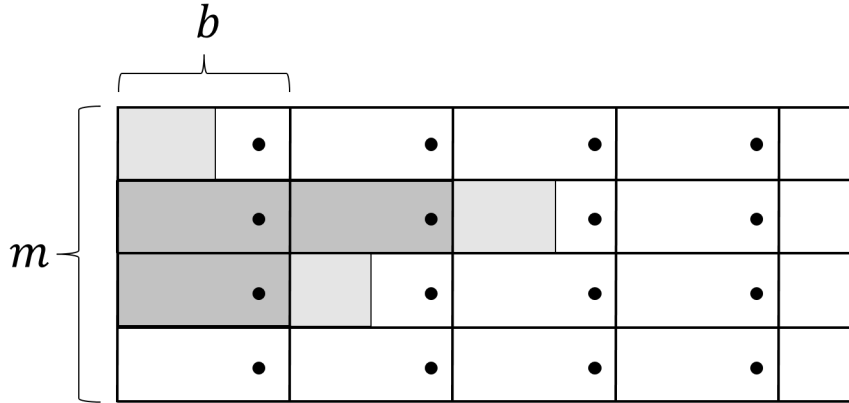
■ **Figure 5** Partitioning the items in each row to blocks of size b. Block representatives are shown as small filled circles. The shaded regions contains the k smallest items. The darkly shaded region depicts blocks all of whose items are among the k smallest.

Alternatively, we can convert the m sorted lists into a heap-ordered tree $T_{m,1}$ by adding a root with value $-\infty$ that will have the m first items as its children. All other nodes in the tree will have degree 1. It is easy to see that $D(T_{m,1}, k) = m + k - 1$. By Theorem 6 we again get an $O(m + k)$ algorithm. We have thus presented three different proofs of the following theorem.

▶ **Theorem 8.** *Let A be a row-sorted matrix containing m rows. Then, the k-th smallest item in A, and the set of k smallest items in A, can be found in $O(m + k)$ time.*

We refer to the algorithm of Theorem 8 as $\mathtt{Mat\text{-}Select}_1(A, k)$. The $O(m + k)$ running time of $\mathtt{Mat\text{-}Select}_1(A, k)$ is asymptotically optimal if $k = O(m)$, as $\Omega(m)$ is clearly a lower bound; each selection algorithm must examine at least one item in each row of the input matrix.

4.2   An $O(m \log \frac{k}{m})$ algorithm, for $k \geq 2m$

We begin with a verbal description of the algorithm. Let A be the input matrix and let $k \geq 2m$. Partition each row of the matrix A into *blocks* of size $b = \lfloor \frac{k}{2m} \rfloor$. The *last* item in each block is the *representative* of the block. Consider the (yet unknown) distribution of the k smallest items among the m rows of the matrix. Let k_i be the number of items in the i-th row that are among the k smallest items in the whole matrix. These k_i items are clearly the first k_i items of the i-th row. They are partitioned into a number of full blocks, followed possibly by one partially filled block. (For an example, see Figure 5.) The number of items in partially filled blocks is at most $m \lfloor \frac{k}{2m} \rfloor$. Thus, the number of filled blocks is at least

$$\frac{k - m \lfloor \frac{k}{2m} \rfloor}{\lfloor \frac{k}{2m} \rfloor} \geq m .$$

Apply algorithm $\mathtt{Mat\text{-}Select}_1$ to select the smallest m block representatives. This clearly takes only $O(m)$ time. (Algorithm $\mathtt{Mat\text{-}Select}_1$ is applied on the implicitly represented matrix A' of block representatives.) All items in the m blocks whose representatives were selected are among the k smallest items of the matrix. The number of such items is $mb = m \lfloor \frac{k}{2m} \rfloor \geq \frac{k}{4}$, as $k \geq 2m$. These items can be removed from the matrix. All that remains is to select the $k - mb$ smallest remaining items using a recursive call to the algorithm. In each recursive

---

Mat-Select$_2(A, k)$:

$m \leftarrow$ *num-rows*(A)
if $k \leq 2m$ **then**
$\quad$ **return** Mat-Select$_1(A, k)$
else
$\quad b \leftarrow \lfloor k/(2m) \rfloor$
$\quad A' \leftarrow jump(A, b)$
$\quad K \leftarrow$ Mat-Select$_1(A', m)$
$\quad A'' \leftarrow shift(A, bK)$
$\quad$ **return** $bK +$ Mat-Select$_2(A'', k - bm)$

■ **Figure 6** Selecting the k smallest items from a row-sorted matrix A. (Implicit handling of submatrices passed to recursive calls.)

Mat-Select$_2(\langle A, c, D \rangle, k)$:

$m \leftarrow$ *num-rows*(A)
if $k \leq 2m$ **then**
$\quad$ **return** Mat-Select$_1(\langle A, c, D \rangle, k)$
else
$\quad b \leftarrow \lfloor k/(2m) \rfloor$
$\quad K \leftarrow$ Mat-Select$_1(\langle A, bc, D \rangle, m)$
$\quad$ **return** $bK +$ Mat-Select$_2(\langle A, c, D + bcK \rangle, k - bm)$

■ **Figure 7** Selecting the k smallest items from a row-sorted matrix A. (Explicit handling of submatrices passed to recursive calls.)

call (or iteration), the total work is $O(m)$. The number of items to be selected drops by a factor of at least $3/4$. Thus after at most $\log_{4/3} \frac{k}{2m} = O(\log \frac{k}{m})$ iterations, k drops below $2m$ and then Mat-Select$_1$ is called to finish the job in $O(m)$ time.

Pseudo-code of the algorithm described above, which we call Mat-Select$_2(A, k)$ is given in Figure 6. The algorithm returns an array $K = (k_1, k_2, \ldots, k_m)$, where k_i is the number of items in the i-th row that are among the k smallest items of the matrix. The algorithm uses a function *num-rows*(A) that returns the number of rows of a given matrix, a function *jump*(A, b) that returns an (implicit) representation of a matrix A' such that $A'_{i,j} = A_{i,bj}$, for $i, j \geq 1$, and a function *shift*(A, K) that returns an (implicit) representation of a matrix A'' such that $A''_{i,j} = A_{i,j+k_i}$, for $i, j \geq 1$.

In Figure 7 we eliminate the use of *jump* and *shift* and make everything explicit. The input matrix is now represented by a triplet $\langle A, c, D \rangle$, where A is a matrix, $c \geq 1$ is an integer, and $D = (d_1, d_2, \ldots, d_m)$ is an array of non-negative integral *displacements*. Mat-Select$_2(\langle A, c, D \rangle, k)$ selects the k smallest items in the matrix A' such that $A'_{i,j} = A_{i,cj+d_i}$, for $i, j \geq 1$. To select the k smallest items in A itself, we simply call Mat-Select$_2(\langle A, 1, \mathbf{0} \rangle, k)$, where $\mathbf{0}$ represents an array of m zeros. The implementation of Mat-Select$_2(\langle A, c, D \rangle, k)$ in Figure 7 is recursive. It is easy to convert it into an equivalent iterative implementation.

▶ **Theorem 9.** *Let A be a row-sorted matrix containing m rows and let $k \geq 2m$. Algorithm* Mat-Select$_2(A, k)$ *selects the k smallest items in A in $O(m \log \frac{k}{m})$ time.*

Frederickson and Johnson [12] showed that the $O(m \log \frac{k}{m})$ running time of algorithm $\mathtt{Mat\text{-}Select}_2(A, k)$ is optimal, when $k \geq 2m$. A simple proof of this claim can also be found in Section 4.5.

4.3 An $O(m + \sum_{i=1}^{m} \log n_i)$ algorithm

Assume now that the i-th row of A contains only n_i items. We assume that $n_i \geq 1$, as otherwise, we can simply remove the i-th row. We can run algorithms $\mathtt{Mat\text{-}Select}_1$ and $\mathtt{Mat\text{-}Select}_2$ of the previous sections by adding dummy $+\infty$ items at the end of each row, but this may be wasteful. We now show that a simple modification of $\mathtt{Mat\text{-}Select}_2$, which we call $\mathtt{Mat\text{-}Select}_3$, can solve the selection problem in $O(m + \sum_{i=1}^{m} \log n_i)$ time. We focus first on the number of comparisons performed by the new algorithm.

At the beginning of each iteration, $\mathtt{Mat\text{-}Select}_2$ sets the block size to $b = \lfloor \frac{k}{2m} \rfloor$. If $n_i < b$, then the last item in the first block of the i-th row is $+\infty$. Assuming that $k \leq \sum_{i=1}^{m} n_i$, no representatives from the i-th row will be selected in the current iteration. There is therefore no point in considering the i-th row in the current iteration. Let m' be the number of *long* rows, i.e., rows for which $n_i \geq \lfloor \frac{k}{2m} \rfloor$. We want to reduce the running time of the iteration to $O(m')$ and still reduce k by some constant factor.

The total number of items in the short rows is less than $m \lfloor \frac{k}{2m} \rfloor \leq \frac{k}{2}$. The long rows thus contain at least $\frac{k}{2}$ of the k smallest items of the matrix. We can thus run an iteration of $\mathtt{Mat\text{-}Select}_2$ on the long rows with $k' = \frac{k}{2}$. In other words, we adjust the block size to $b' = \lfloor \frac{k'}{2m'} \rfloor = \lfloor \frac{k}{4m'} \rfloor$ and use $\mathtt{Mat\text{-}Select}_1$ to select the m' smallest representatives. This identifies $b'm' \geq \frac{k'}{4} \geq \frac{k}{8}$ items as belonging to the k smallest items in A. Thus, each iteration takes $O(m')$ time and reduces k by a factor of at least $\frac{7}{8}$.

In how many iterations did each row of the matrix participate? Let k_j be the number of items still to be selected at the beginning of iteration j. Let $b_j = \lfloor \frac{k_j}{2m} \rfloor$ be the threshold for long rows used in iteration j. As k_j drops exponentially, so does b_j. Thus, row i participates in at most $O(\log n_i)$ of the *last* iterations of the algorithm. The total number of comparisons performed is thus at most $O(m + \sum_{i=1}^{m} \log n_i)$, as claimed.

To show that the algorithm can also be implemented to run in $O(m + \sum_{i=1}^{m} \log n_i)$ time, we need to show that we can quickly identify the rows that are long enough to participate in each iteration. To do that, we sort $\lceil \log n_i \rceil$ using bucket sort. This takes only $O(m + \max_i \lceil \log n_i \rceil)$ time. When a row loses some of its items, it is easy to move it to the appropriate bucket in $O(1)$ time. In each iteration we may need to examine rows in one bucket that turn out not to be long enough, but this does not affect the total $O(m + \sum_{i=1}^{m} \log n_i)$ running time of the algorithm.

▶ **Theorem 10.** *Let A be a row-sorted matrix containing m rows, and let $N = (n_1, n_2, \ldots, n_m)$, where $n_i \geq 1$ be the number of items in the i-th row of the matrix, for $1 \leq i \leq m$. Let $k \leq \sum_{i=1}^{m} n_i$. Algorithm $\mathtt{Mat\text{-}Select}_3(A, N, k)$ selects the k smallest items in A in $O(m + \sum_{i=1}^{m} \log n_i)$ time.*

In Section 4.5 below we show that the running time of $\mathtt{Mat\text{-}Select}_3(A, N, k)$ is optimal for some values of $N = (n_1, n_2, \ldots, n_m)$ and k, e.g., if $k = \frac{1}{2} \sum_{i=1}^{m} n_i$, i.e., for median selection. The $O(m \log \frac{k}{m})$ running time of $\mathtt{Mat\text{-}Select}_2(A, k)$ is sometimes better than the $O(m + \sum_{i=1}^{m} \log n_i)$ running time of $\mathtt{Mat\text{-}Select}_3(A, N, k)$. We next describe an algorithm, $\mathtt{Mat\text{-}Select}_4(A, k)$, which is always at least as fast as the three algorithms already presented, and sometimes faster.

4.4 An $O(m + \sum_{i=1}^{m} \log(k_i + 1))$ algorithm

As before, let k_i be the (yet unknown) number of items in the i-th row that belong to the smallest k items of the matrix. In this section we describe an algorithm for finding these k_i's that runs in $O(m + \sum_{i=1}^{m} \log(k_i + 1))$ time.

We partition each row this time into blocks of size $1, 2, 4, \ldots$. The representative of a block is again the last item in the block. Note that the first k_i items in row i reside in $\lfloor \log(k_i + 1) \rfloor$ complete blocks, plus one incomplete block, if $\log(k_i + 1)$ is not an integer. Thus $L = \sum_{i=1}^{m} \lfloor \log(k_i + 1) \rfloor$ is exactly the number of block representatives that belong to the k smallest items of the matrix.

Suppose that $\ell \geq L$ is an upper bound on the true value of L. We can run Mat-Select$_1$ to select the ℓ smallest block representatives in $O(m + \ell)$ time. If ℓ_i representatives were selected from row i, we let $n_i = 2^{\ell_i+1} - 1$. We now run Mat-Select$_3$ which runs in $O(m + \sum_{i=1}^{m} \log n_i) = O(m + \sum_{i=1}^{m} (\ell_i + 1)) = O(m + \ell)$. Thus, if $\ell = O(L)$, the total running time is $O(m + \sum_{i=1}^{m} \log(k_i + 1))$, as promised.

How do we find a tight upper bound on $L = \sum_{i=1}^{m} \lfloor \log(k_i + 1) \rfloor$? We simply try $\ell = m, 2m, 4m, \ldots$, until we obtain a value of ℓ that is high enough. If $\ell < L$, i.e., ℓ is not large enough, we can discover it in one of two ways. Either $\sum_{i=1}^{m} n_i < k$, in which case ℓ is clearly too small. Otherwise, the algorithm returns an array of k_i values. We can check whether these values are the correct ones in $O(m)$ time. First compute $M = \max_{i=1}^{m} A_{i,k_i}$. Next check that $A_{i,k_i+1} > M$, for $1 \leq i \leq m$. As ℓ is doubled in each iteration, the cost of the last iteration dominates the total running time which is thus $O(m + 2L) = O(m + \sum_{i=1}^{m} \log(k_i + 1))$. We call the resulting algorithm Mat-Select$_4$.

▶ **Theorem 11.** *Let A be a row-sorted matrix containing m rows and let $k \geq 2m$. Algorithm Mat-Select$_4(A, k)$ selects the k smallest items in A in $O\left(m + \sum_{i=1}^{m} \log(k_i + 1)\right)$ time, where k_i is the number of items selected from row i.*

4.5 Lower bounds for selection from row-sorted matrices

We begin with a simple proof that the $O(m \log \frac{k}{m})$ algorithm is optimal for $k \geq 2m$.

▶ **Theorem 12.** *Any algorithm for selecting the k smallest items from a matrix with m sorted rows must perform at least $(m - 1) \log \frac{m+k}{m}$ comparisons on some inputs.*

Proof. We use the information-theoretic lower bound. We need to lower bound $s_k(m)$, which is the number of m-tuples $(k_1, k_2, \ldots, k_m)$, where $0 \leq k_i$, for $1 \leq i \leq m$, and $\sum_{i=1}^{m} k_i = k$. It is easily seen that $s_k(m) = \binom{m+k-1}{m-1}$, as this is the number of ways to arrange k identical balls and $m - 1$ identical dividers in a row. We thus get a lower bound of

$$\log \binom{m+k-1}{m-1} \geq \log \left(\frac{m+k-1}{m-1}\right)^{m-1} = (m-1) \log \frac{m+k-1}{m-1} \geq (m-1) \log \frac{m+k}{m},$$

where we used the well-known relation $\binom{n}{k} > \left(\frac{n}{k}\right)^k$. ◄

We next show that our new $O(m + \sum_{i=1}^{m} \log n_i)$ algorithm is optimal, at least in some cases, e.g., when $k = \frac{1}{2} \sum_{i=1}^{m} n_i$ which corresponds to median selection.

▶ **Theorem 13.** *Any algorithm for selecting the $k = \frac{1}{2} \sum_{i=1}^{m} n_i$ smallest items from a row-sorted matrix with m rows of lengths $n_1, n_2, \ldots, n_m \geq 1$ must perform at least $\sum_{i=1}^{m} \log(n_i + 1) - \log\left(1 + \sum_{i=1}^{m} n_i\right)$ comparisons on some inputs.*

Proof. The number of possible solutions to the selection problem for all values of $0 \leq k \leq \sum_{i=1}^{m} n_i$ is $\prod_{i=1}^{m}(n_i + 1)$. (Each solution corresponds to a choice $0 \leq k_i \leq n_i$, for $i = 1, 2, \ldots, m$.) We prove below that the number of solutions is maximized for $k = \lfloor \frac{1}{2} \sum_{i=1}^{m} n_i \rfloor$ (and $k = \lceil \frac{1}{2} \sum_{i=1}^{m} n_i \rceil$). The number of possible solutions for this value of k is thus at least $(\prod_{i=1}^{m}(n_i + 1))/(1 + \sum_{i=1}^{m} n_i)$. Taking logarithm, we get the promised lower bound.

We next prove that the number of solutions is maximized when $k = \lfloor \frac{1}{2} \sum_{i=1}^{m} n_i \rfloor$. Let X_i be a uniform random variable on $\{0, 1, \ldots, n_i\}$, and let $Y = \sum_{i=1}^{m} X_i$. The number of solutions for a given value k is proportional to the probability that Y attains the value k. Let $Y_j = \sum_{i=1}^{j} X_i$. We prove by induction on j that the distribution of Y_j is maximized at $\mu_j = \frac{1}{2} \sum_{i=1}^{j} n_i$, is symmetric around μ_j, and is increasing up to μ_j and decreasing after μ_j. The base case is obvious as $Y_1 = X_1$ is a uniform distribution. The induction step follows from an easy calculation. Indeed, $Y_j = Y_{j-1} + X_j$, where X_j is uniform and Y_{j-1} has the required properties. The distribution of Y_j is the *convolution* of the distributions of Y_{j-1} and X_j, which corresponds to taking the *average* of $n_j + 1$ values of the distribution of Y_{j-1}. It follows easily that Y_j also has the required properties. ◀

We next compare the lower bound obtained, $\sum_{i=1}^{m} \log(n_i + 1) - \log(1 + \sum_{i=1}^{m} n_i)$, with the upper bound $O(m + \sum_{i=1}^{m} \log n_i)$. The subtracted term in the lower bound is dominated by the first term, i.e., $\log(1 + \sum_{i=1}^{m} n_i) \leq \frac{\log(m+1)}{m} \sum_{i=1}^{m} \log(n_i + 1)$, where equality holds only if $n_i = 1$, for every i. When the n_i's are large, the subtracted term becomes negligible. Also, as $n_i \geq 1$, we have $\sum_{i=1}^{m} \log(n_i + 1) \geq m$. Thus, the lower and upper bound are always within a constant multiplicative factor of each other.

The optimality of the $O(m + \sum_{i=1}^{m} \log n_i)$ algorithm also implies the optimality of our new "output-sensitive" $O(m + \sum_{i=1}^{m} \log(k_i + 1))$ algorithm. As $k_i \leq n_i$, an algorithm that performs less than $c(m + \sum_{i=1}^{m} \log(k_i + 1))$ comparisons on all inputs, for some small enough c, would contradict the lower bounds for the $O(m + \sum_{i=1}^{m} \log n_i)$ algorithm. We also note that an argument very similar to the one used in the proof of Theorem 13 shows that $\Omega(\sum_{i=1}^{m} \log(k_i + 1))$ comparisons are required even if the algorithm is given a multiplicative approximation of the k_i's.

5   Selection from $X + Y$

We are given two *unsorted* sets X and Y and would like to find the k-th smallest item, and the set of k smallest items, in the set $X + Y$. We assume that $|X| = m$, $|Y| = n$, where $m \geq n$.

5.1   An $O(m + n + k)$ algorithm

Heapify X and heapify Y, which takes $O(m + n)$ time. Let $x_1, \ldots, x_m$ be the heapified order of X, i.e., $x_i \leq x_{2i}, x_{2i+1}$, whenever the respective items exists. Similarly, let $y_1, \ldots, y_n$ be the heapified order of Y. Construct a heap of maximum degree 4 representing $X + Y$ as follows. The root is $x_1 + y_1$. Item $x_i + y_1$, for $i \geq 1$ has four children $x_{2i} + y_1, x_{2i+1} + y_1, x_i + y_2, x_i + y_3$. Item $x_i + y_j$, for $i \geq 1$, $j > 1$, has two children $x_i + y_{2j}, x_i + y_{2j+1}$, again when the respective items exist. (Basically, this is a heapified version of $X + y_1$, where each $x_i + y_1$ is the root of a heapified version of $x_i + Y$.) We can now apply algorithm `Soft-Select` on this heap, which takes only $O(k)$ time. We call the resulting algorithm `X+Y-Select`$_1(X, Y)$.

▶ **Theorem 14.** *Let X and Y be unordered sets of m and n items respectively. Then, algorithm `X+Y-Select`$_1(X, Y)$ finds the k-th smallest item, and the set of k smallest items in $X + Y$, in $O(m + n + k)$ time.*

We note that if X and Y are given to us as heaps, and in particular, if they are sorted, then the running time of algorithm $\texttt{X+Y-Select}_1(X, Y)$ becomes $O(k)$, as there is no need to heapify X and Y.

5.2 An $O(m \log \frac{k}{m})$ algorithm, for $k \geq 6m$, $m \geq n$

If Y is sorted, then $X + Y$ is a row-sorted matrix, and we can use algorithm $\texttt{Mat-Select}_2$ of Theorem 9. We can sort Y in $O(n \log n)$ time and get an $O(m \log \frac{k}{m} + n \log n)$ algorithm. The running time of this algorithm is $O(m \log \frac{k}{m})$ when $k \geq mn^\varepsilon$, for any fixed $\varepsilon > 0$. But, for certain values of m, n and k, e.g., $m = n$ and $k = mn^{o(1)}$, the cost of sorting is dominant. We show below that the sorting can always be avoided.

We first regress and describe an alternative $O(m \log \frac{k}{m})$ algorithm for selection from row-sorted matrices. The algorithm is somewhat more complicated than algorithm $\texttt{Mat-Select}_2$ given in Section 4.2. The advantage of the new algorithm is that much less assumptions are made about the order of the items in each row. A similar approach was used by Frederickson and Johnson [12] but we believe that our approach is simpler. In particular we rely on a simple partitioning lemma (Lemma 16 below) which is not used, explicitly or implicitly, in [12].

Instead of partitioning each row into blocks of equal size, as done by algorithm $\texttt{Mat-Select}_2$ of Section 4.2, we partition each row into exponentially increasing blocks, similar, but not identical, to the partition made by algorithm $\texttt{Mat-Select}_3$ of Section 4.3.
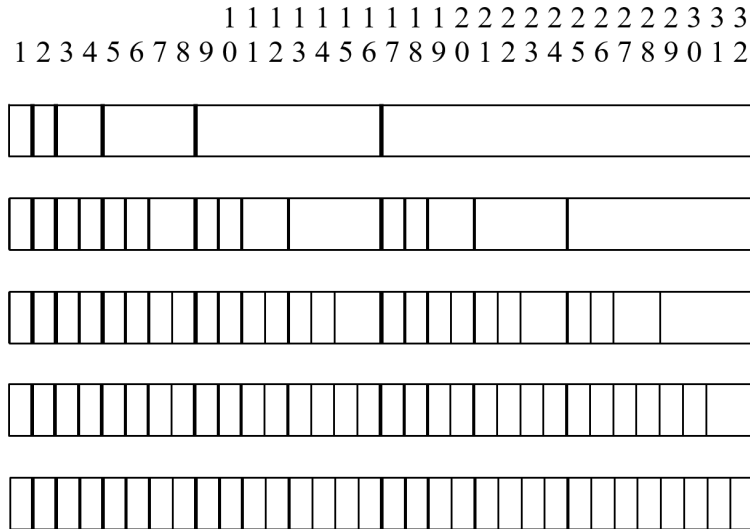
Let $b = \lfloor \frac{k}{3m} \rfloor$. Partition each row into blocks of size $b, b, 2b, 4b, \ldots, 2^j b, \ldots$. The representative of a block is again the last item in the block. We use algorithm $\texttt{Mat-Select}_1$ to select the m smallest representatives. This takes $O(m)$ time. Let $e_1 < e_2 < \ldots < e_m$ denote the m selected representatives in (the unknown) sorted order, and let $s_1, s_2, \ldots, s_m$ be the sizes of their blocks. We next use an $O(m)$ *weighted selection* algorithm (see, e.g., Cormen et al. [7], Problem 9.2, p. 225) to find the smallest ℓ such that $\frac{k}{6} \leq \sum_{j=1}^{\ell} s_j$ and the items $e_1, e_2, \ldots, e_\ell$, in some order. Such an $\ell \leq m$ must exist, as $m \lfloor \frac{k}{3m} \rfloor \geq m(\frac{k}{3m} - 1) = \frac{1}{3}(k - 3m) \geq \frac{k}{6}$, as $k \geq 6m$. Also note that $\frac{k}{6} \leq \sum_{j=1}^{\ell} s_j < \frac{k}{3}$, as the addition of each block at most doubles the total size, i.e., $\sum_{j=1}^{\ell} s_j < 2 \sum_{j=1}^{\ell-1} s_j$, for $\ell > 1$.

▶ **Lemma 15.** *All items of the blocks whose representatives are $e_1, e_2, \ldots, e_\ell$ are among the k smallest items in the matrix.*

Proof. Let S_k be the set of k smallest items of the matrix. Consider again the partition of S_k among the m rows of the matrix. Less than $mb = m \lfloor \frac{k}{3m} \rfloor \leq \frac{k}{3}$ of the items of S_k belong to rows that do not contain a full block of S_k items. Thus, at least $\frac{2k}{3}$ of the items of S_k are contained in rows that contain at least one full block of S_k items. The exponential increase in the size of the blocks ensures that in each such row, at least half of the items of S_k are contained in full blocks. Thus, at least $\frac{k}{3}$ of the items of S_k are contained in full blocks. In particular, if $e_1, e_2, \ldots, e_\ell$ are the smallest block representatives, and $\sum_{j=1}^{\ell} s_j \leq \frac{k}{3}$, then all the items in the blocks of $e_1, e_2, \ldots, e_\ell$ belong to S_k. ◀

We can thus remove all the items in the blocks of $e_1, e_2, \ldots, e_\ell$ from the matrix and proceed to find the $k - \sum_{j=1}^{\ell} s_j \leq \frac{5k}{6}$ smallest items of the remaining matrix. When k drops below $6m$, we use the algorithm of $\texttt{Mat-Select}_1$ of Section 4.1. The resulting algorithm performs $O(\log \frac{k}{m})$ iterations, each taking $O(m)$ time, so the total running time is $O(m \log \frac{k}{m})$. (This matches the running time of $\texttt{Mat-Select}_2$, using a somewhat more complicated algorithm.)

We make another small adaptation to the new $O(m \log \frac{k}{m})$ algorithm before returning to the selection from $X + Y$ problem. Instead of letting $b = \lfloor \frac{k}{3m} \rfloor$ and using blocks of size

$$1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,2\,2\,2\,2\,2\,2\,2\,2\,2\,2\,3\,3\,3$$
$$1\,2\,3\,4\,5\,6\,7\,8\,9\,0\,1\,2\,3\,4\,5\,6\,7\,8\,9\,0\,1\,2\,3\,4\,5\,6\,7\,8\,9\,0\,1\,2$$

Figure 8 Partitions of Y for $n = 32$.

$b, b, 2b, 4b, \ldots$, we let $b' = 2^{\lfloor \log_2 b \rfloor}$, i.e., b' is the largest power of 2 which is at most b, and use blocks of size $b', b', 2b', 4b', \ldots$. All block sizes are now powers of 2. As the sizes of the blocks may be halved, we select the $2m$ smallest block representatives. The number of items removed from each row in each iteration is now also a power of 2.

Back to the $X + Y$ problem. The main advantage of the new algorithm is that we do not really need the items in each row to be sorted. All we need are the items of ranks $b, b, 2b, 4b, \ldots$, where $b = 2^\ell$ for some $\ell > 0$, in each row. In the $X + Y$ problem the rows, or what remains of them after a certain number of iterations, are related, so we can easily achieve this task.

At the beginning of the first iteration, we use repeated median selection to find the items of Y whose ranks are $1, 2, 4, \ldots$. This also partitions Y into blocks of size $1, 2, 4, \ldots$ such that items of each block are smaller than the items of the succeeding block. We also place the items of ranks $1, 2, 4, \ldots$ in their corresponding places in Y. This gives us enough information to run the first iteration of the matrix selection algorithm.

In each iteration, we refine the partition of Y. We apply repeated median selection on each block of size 2^ℓ in Y, breaking it into blocks of size $1, 1, 2, 4, \ldots, 2^{\ell-1}$. The total time needed is $O(n)$ per iteration, which we can easily afford. We assume for simplicity that $n = |Y|$ is a power of 2 and that all items in Y are distinct. We now have the following fun lemma:

▶ **Lemma 16.** *After i iterations of the above process, if $1 \leq r \leq n$ has at most i 1's in its binary representation, then $Y[r]$ is the item of rank r in Y, i.e., $Y[1 : r-1] < Y[r] < Y[r+1 : n]$. Additionally, if $r_1 < r_2$ both have at most i 1's in their binary representation, and r_2 is the smallest number larger than r_1 with this property, then $Y[1 : r_1] < Y[r_1+1 : r_2] < Y[r_2+1 : n]$, i.e., the items in $Y[r_1 + 1 : r_2]$ are all larger than the items in $Y[1 : r_1]$ and smaller than the items in $Y[r_2 + 1 : n]$.*

For example, if $n = 32$, then after the first iteration we have the partition

$$Y[1], Y[2], Y[3 : 4], Y[5 : 8], Y[9 : 16], Y[17 : 32] \; .$$

After the second iteration, we have the partition

$$Y[1], Y[2], \quad Y[3], Y[4], \ Y[5], Y[6], Y[7:8], \quad Y[9], Y[10], Y[11\!:\!12], Y[13:16],$$
$$Y[17], Y[18], Y[19:20], Y[21:24], Y[25:32] \ .$$

(Actually, blocks of size 2 are also sorted.) The partitions obtained for $n = 32$ in the first five iterations are also shown in Figure 8.

Proof. The claim clearly holds after the first iteration, as numbers with a single 1 in their binary representation are exactly powers of 2. Let $Y[r_1 + 1 : r_2]$ be a block of Y generated after i iterations. If r_1 has less than i 1's, then $r_2 = r_1 + 1$, so the block is trivial. Suppose, therefore, that r_1 has exactly i 1's in its representation and that $r_2 = r_1 + 2^\ell$. (ℓ is actually the index of the rightmost 1 in the representation of r_1, counting from 0.) In the $(i+1)$-st iteration, this block is broken into the blocks $Y[r_1 + 1], Y[r_1 + 2], Y[r_1 + 3 : r_1 + 4], \ldots, Y[r_1 + 2^{\ell-1} + 1 : r_1 + 2^\ell]$. As the numbers $r_1 + 2^j$, for $1 \le j < \ell$ are exactly the number between r_1 and r_2 with at most $i + 1$ 1's in their binary representation, this establishes the induction step.   ◄

After i iterations of the modified matrix selection algorithm applied to an $X + Y$ instance, we have removed a certain number of items d_i from each row. The number of items removed from each row in each iteration is a power of 2. By induction, d_i has at most i 1's in its representation. In the $(i + 1)$-st iteration we set $b = 2^\ell$, for some $\ell \ge 1$ and need the items of rank $b, 2b, 4b, \ldots$ from what remains of each row. The items needed from the i-th row are exactly $X[i] + Y[d_i + 2^j b]$, for $j = 0, 1, \ldots$. The required items from Y are available, as $d_i + 2^j b$ has at most $i + 1$ 1s in its binary representation! We call the resulting algorithm X+Y-Select$_2(X, Y)$.

▶ **Theorem 17.** *Let X and Y be unordered sets of m and n items respectively, where $m \ge n$, and let $k \ge 6m$. Algorithm* X+Y-Select$_2(X, Y)$ *finds the k-th smallest item, and the set of k smallest items in $X + Y$, in $O(m \log \frac{k}{m})$ time.*

6   Concluding remarks

We used soft heaps to obtain a very simple $O(k)$ algorithm for selecting the k-th smallest item from a binary min-heap, greatly simplifying the previous $O(k)$ algorithm of Frederickson [11]. We used this simple heap selection algorithm to obtain simpler algorithms for selection from row-sorted matrices and from $X + Y$, simplifying results of Frederickson and Johnson [12]. The simplicity of our algorithms allowed us to go one step further and obtain some improved algorithms for these problems, in particular an $O(m + \sum_{i=1}^{m} \log(k_i + 1))$ "output-sensitive" algorithm for selection from row-sorted matrices.

Our results also demonstrate the usefulness of soft heaps outside the realm of minimum spanning tree algorithms. It would be nice to find further applications of soft heaps.

── **References** ──

1   Manuel Blum, Robert W. Floyd, Vaughan R. Pratt, Ronald L. Rivest, and Robert Endre Tarjan. Time Bounds for Selection. *J. Comput. Syst. Sci.*, 7(4):448–461, 1973. `doi:10.1016/S0022-0000(73)80033-9`.

2   David Bremner, Timothy M. Chan, Erik D. Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, Mihai Patrascu, and Perouz Taslakian. Necklaces, Convolutions, and $X + Y$. *Algorithmica*, 69(2):294–314, 2014. `doi:10.1007/s00453-012-9734-3`.

3   Gerth Stølting Brodal, Rolf Fagerberg, Mark Greve, and Alejandro López-Ortiz. Online Sorted Range Reporting. In *Proc. of 20th ISAAC*, pages 173–182, 2009. `doi:10.1007/978-3-642-10631-6_19`.

4   Jean Cardinal, Samuel Fiorini, Gwenaël Joret, Raphaël M. Jungers, and J. Ian Munro. Sorting under partial information (without the ellipsoid algorithm). *Combinatorica*, 33(6):655–697, 2013. `doi:10.1007/s00493-013-2821-5`.

5   Bernard Chazelle. A minimum spanning tree algorithm with Inverse-Ackermann type complexity. *J. ACM*, 47(6):1028–1047, 2000. `doi:10.1145/355541.355562`.

6   Bernard Chazelle. The soft heap: an approximate priority queue with optimal error rate. *J. ACM*, 47(6):1012–1027, 2000. `doi:10.1145/355541.355554`.

7   Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, 3rd Edition*. MIT Press, 2009. URL: `http://mitpress.mit.edu/books/introduction-algorithms`.

8   Dorit Dor and Uri Zwick. Selecting the Median. *SIAM Journal on Computing*, 28(5):1722–1758, 1999. `doi:10.1137/S0097539795288611`.

9   Dorit Dor and Uri Zwick. Median Selection Requires $(2+\epsilon)n$ Comparisons. *SIAM Journal on Discrete Mathematics*, 14(3):312–325, 2001. `doi:10.1137/S0895480199353895`.

10   David Eppstein. Finding the k Shortest Paths. *SIAM J. Comput.*, 28(2):652–673, 1998. `doi:10.1137/S0097539795290477`.

11   Greg N. Frederickson. An Optimal Algorithm for Selection in a Min-Heap. *Information and Computation*, 104(2):197–214, 1993. `doi:10.1006/inco.1993.1030`.

12   Greg N. Frederickson and Donald B. Johnson. The Complexity of Selection and Ranking in $X + Y$ and Matrices with Sorted Columns. *J. Comput. Syst. Sci.*, 24(2):197–208, 1982. `doi:10.1016/0022-0000(82)90048-4`.

13   Greg N. Frederickson and Donald B. Johnson. Generalized Selection and Ranking: Sorted Matrices. *SIAM Journal on Computing*, 13(1):14–30, 1984. `doi:10.1137/0213002`.

14   Greg N. Frederickson and Donald B. Johnson. Erratum: Generalized Selection and Ranking: Sorted Matrices. *SIAM Journal on Computing*, 19(1):205–206, 1990. `doi:10.1137/0219013`.

15   Michael L. Fredman. How Good is the Information Theory Bound in Sorting? *Theor. Comput. Sci.*, 1(4):355–361, 1976. `doi:10.1016/0304-3975(76)90078-5`.

16   Joseph L. Hodges Jr and Erich L. Lehmann. Estimates of location based on rank tests. *The Annals of Mathematical Statistics*, pages 598–611, 1963.

17   Donald B. Johnson and Samuel D. Kashdan. Lower Bounds for Selection in $X + Y$ and Other Multisets. *J. ACM*, 25(4):556–570, 1978. `doi:10.1145/322092.322097`.

18   Donald B. Johnson and Tetsuo Mizoguchi. Selecting the K-th element in $X + Y$ and $X_1 + X_2 + \ldots + X_m$. *SIAM J. Comput.*, 7(2):147–153, 1978. `doi:10.1137/0207013`.

19   Jeff Kahn and Jeong Han Kim. Entropy and Sorting. *J. Comput. Syst. Sci.*, 51(3):390–399, 1995. `doi:10.1006/jcss.1995.1077`.

20   Jeff Kahn and Michael Saks. Balancing poset extensions. *Order*, 1(2):113–126, 1984. `doi:10.1007/BF00565647`.

21   Daniel M. Kane, Shachar Lovett, and Shay Moran. Near-optimal linear decision trees for k-SUM and related problems. In *Proc. of 50th STOC*, pages 554–563, 2018. `doi:10.1145/3188745.3188770`.

22   Haim Kaplan, Robert Endre Tarjan, and Uri Zwick. Soft Heaps Simplified. *SIAM Journal on Computing*, 42(4):1660–1673, 2013. `doi:10.1137/120880185`.

23   Bernard O. Koopman. The optimum distribution of effort. *Journal of the Operations Research Society of America*, 1(2):52–63, 1953. `doi:10.1287/opre.1.2.52`.

24   Jean-Luc Lambert. Sorting the sums $(x_i + y_j)$ in $O(n^2)$ comparisons. *Theoretical Computer Science*, 103(1):137–141, 1992.

25   Seth Pettie and Vijaya Ramachandran. An optimal minimum spanning tree algorithm. *J. ACM*, 49(1):16–34, 2002. `doi:10.1145/505241.505243`.

26   Arnold Schönhage, Mike Paterson, and Nicholas Pippenger. Finding the Median. *J. Comput. Syst. Sci.*, 13(2):184–199, 1976. `doi:10.1016/S0022-0000(76)80029-3`.

27   William L. Steiger and Ileana Streinu. A Pseudo-Algorithmic Separation of Lines from Pseudo-Lines. *Inf. Process. Lett.*, 53(5):295–299, 1995. `doi:10.1016/0020-0190(94)00201-9`.

28   J.W.J. Williams. Algorithm 232: Heapsort. *cacm*, 7:347–348, 1964.