Encodings for Range Selection and Top-k Queries

Roberto Grossi[1], John Iacono[2], Gonzalo Navarro[3⋆], Rajeev Raman[4], and S. Srinivasa Rao[5⋆⋆]

[1] Dip. di Informatica, Univ. of Pisa, Italy
[2] Dept. of Comp. Sci. and Eng., Polytechnic Institute of New York Univ., USA
[3] Dept. of Comp. Sci., Univ. of Chile, Chile
[4] Dept. of Comp. Sci., Univ. of Leicester, UK
[5] School of Comp. Sci. and Eng., Seoul National Univ.

Abstract. We study the problem of *encoding* the positions the top-k elements of an array $A[1..n]$ for a given parameter $1 \leq k \leq n$. Specifically, for any i and j, we wish create a data structure that reports the positions of the largest k elements in $A[i..j]$ in decreasing order, *without* accessing A at query time. This is a natural extension of the well-known encoding range-maxima query problem, where only the position of the maximum in $A[i..j]$ is sought, and finds applications in document retrieval and ranking. We give (sometimes tight) upper and lower bounds for this problem and some variants thereof.

1 Introduction

We consider the problem of *encoding* range top-k queries over an array of distinct values $A[1..n]$. Given an integer $1 \leq k \leq n$, we wish to preprocess A and create a data structure that can answer top-k-pos queries: given two indices i and j, return the *positions* where the largest k values in $A[i..j]$ occur.

The encoding version of the problem requires this query to be answered *without* accessing A: this is useful when the values in A are intrinsically uninteresting and only the indices where the top-k values occur are of interest. An example is auto-completion search in databases and search engines [13,15]. Here, as the user types in a query, the system presents the user with the k most popular completions, chosen from a lexicon of phrases, based on the text entered so far. Viewing the lexicon as a sorted sequence of strings with popularity scores stored in A, the indices i and j can specify the range of phrases prefixed by the text typed in so far. Similarly, in document search engines, A could contain the (virtual) sequence of PageRank values of the pages in an inverted list sorted by URL. Then we could efficiently retrieve the k most highly ranked documents that contain a query term, restricted to a range of page identifiers (which can model a domain

⋆ Partially funded by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Mideplan, Chile.
⋆⋆ Research partly supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education, Science and Technology (Grant number 2012-0008241).

of any granularity). An encoding data structure for top-k queries will allow us to reduce the amount of space needed to perform these searches in main memory.

One can always use a non-encoding data structure (i.e., one that accesses A during the execution of queries) for top-k-pos, such as that of Brodal et al. [4], on an array A' that contains the sorted permutation of the elements in A, and thus trivially avoid access to A at query time. This yields an encoding that uses $O(n)$ words, or $O(n \lg n)$ bits, of memory and answers top-k-pos queries in optimal $O(k)$ time. We aim to find non-trivial encodings of size $o(n \lg n)$ bits (from which, of course, it is not possible to recover the sorted permutation, but one can still answer any top-k-pos query). As we prove a lower bound of $\Omega(n \lg k)$ bits on the space of any top-k encoding, non-trivial encodings can exist only if $\lg k = o(\lg n)$. This is a reasonable assumption for the aforementioned applications.

Related work. The encoding top-k problem is closely related to the problem of encoding *range maximum query (RMQ)*, which is the particular case with $k = 1$: $\mathsf{RMQ}_A(i, j) = \operatorname{argmax}_{i \le p \le j} A[p]$. The RMQ problem has a long history and many applications [2, 3, 12], and the problem of encoding RMQs has been studied in [8, 19]. In particular, Fischer and Heun [8] gave an encoding of A that uses $2n + o(n)$ bits and answers RMQ in $O(1)$ time; their space bound is asymptotically optimal to within lower-order terms. We are not aware of any work on top-k encoding for $k > 1$.

Our work is also related to *range selection*, which is to preprocess A to find the kth largest element in a range $A[i..j]$, with i, j, k given at query time. This problem has recently been studied intensively in its non-encoding version [5,6,9,10,14]. Jørgensen and Larsen [14] obtained a query time of $O(\lg k / \lg \lg n + \lg \lg n)$, very recently improved to $O(\lg k / \lg \lg n)$ by Chan and Wilkinson [6], both using $\Theta(n)$ words, i.e., $\Omega(n \lg n)$ bits. Jørgensen and Larsen [14] introduced the *κ-capped* range selection problem, where a parameter κ is given at preprocessing time, and the data structure only supports selection for ranks $k \le \kappa$. They showed that even the one-sided κ-capped range selection problem requires query time $\Omega(\lg k / \lg \lg n)$ for structures using $O(n \operatorname{polylog} n)$ words, and the result of Chan and Wilkinson is therefore the best possible. Although the problems we consider are different in essential ways, we borrow some techniques, most notably that of *shallow cuttings* [6, 14], in some of our results.

Contributions. We present new lower and upper bounds shown in Table 1, where we assume that the word RAM model has word size of $w = \Omega(\lg n)$ bits, for the following operations on encoding data structures for one- and two-sided range selection and range top-k queries and for any $1 \le i \le j \le n$.

1. kth-pos(i) returns the position of the k-th largest value in range $A[1..i]$ for any array A, and
2. top-k-pos(i) returns the top-k largest positions in $A[1..i]$ (one-sided variant) or top-k-pos(i, j) for the top-k largest positions in $A[i..j]$ (two-sided variant).

We make heavy use of rank and select operations on bitmaps. Given a bitmap $B[1..n]$, operation $\mathsf{rank}_b(B, i)$ is the number of occurrences of bit b in the prefix

Problem	Lower bound space (bits)	Upper bound space (bits)	Upper bound time
kth-pos(i)	$n \lg k - O(n)$	$n \lg k + O(\epsilon(k) n \lg k)$	$O(1/\epsilon(k))$
top-k-pos(i)	$n \lg k - O(n)$	$n \lg k + O(n)$	$O(k)$
top-k-pos(i, j)	$n \lg k - O(n)$	$O(n \lg k)$	$O(k)$

Table 1. Our lower and upper bounds for encodings for one- and two-sided range selection and range top-k queries, simplified for the interesting case $\lg k = o(\lg n)$, and valid for any function $\lg \lg k / \lg k \preceq \epsilon(k) \preceq 1$.

$B[1..i]$, whereas $\mathsf{select}_b(B, j)$ is the position in B of the j-th occurrence of bit b. These operations generalize in the obvious way to sequences over an alphabet $[\sigma]$. We also make use of the Cartesian tree [21] of an array $A[1..n]$, which is fundamental for RMQ solutions. The root of the Cartesian tree represents the position m of a maximum of $A[1..n]$, thus $m = \textsc{rmq}_A(1, n)$. Its left and right subtrees are the Cartesian trees of $A[1..m-1]$ and $A[m+1..n]$, respectively.

2 One-sided queries

We start by considering one-sided queries. We are given an array A of n integers and a fixed value k. We can assume w.l.o.g. that A is a permutation in $[n]$, otherwise we can replace each $A[i]$ by its rank in $\{A[1], \ldots, A[n]\}$, breaking ties as desired, and obtain the same results from queries. We want to preprocess and encode A to support the one-sided operations of Table 1 efficiently.

2.1 Lower bounds

For queries kth-pos(i) and top-k-pos(i), we give a lower bound of $\Omega(n \lg k)$ bits on the size of the encoding. Assume for simplicity that $n = \ell k$, for some integer ℓ. Consider an array A of length n, with $A[i]$ initialized to i, for $1 \le i \le n$, and re-order its elements as follows: take $\ell - 1$ permutations π_j of size k, $0 \le j < \ell - 1$, and permute the elements in the subarray $A[jk+1..(j+1)k]$ according to permutation π_j, $A[jk+i] = jk + \pi_j(i)$ for $0 \le j < \ell - 1$ and $1 \le i \le k$. Observe that, for each $1 \le j < \ell$, $A[x] < A[y]$ for any $x \le jk$ and $y > jk$. We now show how to reconstruct the $\ell - 1$ permutations by performing several kth-pos queries on the array A. By the above property of A, the position of the k-th value in the prefix $A[1..jk+i-1]$ is the position of value $(j-1)k+i$, for any $1 \le j < \ell$ and $1 \le i \le k$. This position is $(j-1)k + \pi_{j-1}^{-1}(i)$. Then, any π_{j-1} can be easily computed with the k queries, kth-pos$(jk+i-1)$ for $1 \le i \le k$. Since the $\ell - 1$ permutations require $(\ell - 1) \lg k! = (n-k) \lg k - O(n)$ bits, the claim follows.

Similar arguments apply to top-k-pos(i) as well, even if it gives the results not in order: using the array A above, kth-pos(i) is precisely the element that disappears from the answer when we move from top-k-pos(i) and top-k-pos$(i+1)$.

Theorem 1. *Any encoding of an array $A[1..n]$ answering* kth-pos *or* top-k-pos *queries requires at least $(n-k) \lg k - O(n)$ bits of space.*

Fig. 1. Encoding of an array A of length n to support kth-pos and top-k-pos queries, for $k = 3$. The encoding consists of a bitmap P of length $n = 18$ with $n' = 13$ ones, and a string X of length n' over the alphabet $[1..k]$.

2.2 Upper bounds and encodings

We first consider query $\mathsf{kth\text{-}pos}(i)$. We scan the array from left to right, and keep track of the top-k values in the prefix seen so far. At any position $i > k$, if we insert $A[i]$ into the top-k values, then we have to remove the k-th largest value of the prefix $A[1..i-1]$. The idea to solve these queries is to record the position of that leaving k-th largest value, so that to solve $\mathsf{kth\text{-}pos}(i)$ we find the next $i' > i$ where the top-k list changes, and then find the value leaving the list when $A[i']$ enters it. This one was the k-th largest value in $A[1..i]$. We wish to store this data using $O(n \lg k)$ bits.

We store a bit vector P of length n, where $P[i] = 1$ iff a new element is inserted into the top-k values at position i (or equivalently, the k-th largest value of $A[1..i-1]$ is deleted at position i). Let $n' \leq n$ be the number of ones in P. The first k bits of P are set to 1. We encode P using $n + o(n)$ bits, while supporting constant-time rank and select operations on it [7, 16].

Further, we store string X of length n', such that $X[j] = j$ for $1 \leq j \leq k$, and $X[j] = X[\mathsf{rank}_1(P, \mathsf{kth\text{-}pos}(\mathsf{select}_1(P, j) - 1))]$, for $k < j \leq n'$. String X encodes the positions of the top-k values in $A[1..i]$ as follows. Let $j = \mathsf{rank}_1(P, i) > k$. Then the last occurrence of $X[j] = \alpha$ in $X[1..j-1]$ marks the position of the element that was the k-th in the segment $A[1..i-1]$. This is because the last occurrence of each distinct symbol α in $X[1..j]$ is the position of a top-k value in $A[1..i]$. This is obviously true for $i = j = k$, and by induction it stays true when, at a position $P[i] = 1$, we set $X[j+k] = \alpha$, where α marked the position of the k-th maximum in $A[1..i-1]$. Figure 1 shows an example.

Note that X is a string of length n' over the alphabet $[k]$. Hence, X can be encoded using $(1 + \epsilon(k))n' \lg k$ bits, so that select is supported in $O(1)$ time and access in $O(1/\epsilon(k))$ time [11], for any $\epsilon(k) = O(1)$ (including functions in $o(1)$). On top of this we add $O(n' \lg \lg k)$ bits. to support in constant time *partial* rank queries, of the form $\mathsf{rank}_{X[i]}(X, i)$. This is obtained by storing one monotone minimum perfect hash function (mmphf) [1] per distinct symbol c appearing $n_c > 0$ times in X. The space for each c is $O(n_c \lg \lg(n'/n_c))$ bits, which adds up to $O(n' \lg \lg k)$ by Jensen's inequality.

By the discussion above, for $i < n$ we compute $j = \mathsf{rank}_1(P, i) + 1$, and $\alpha = X[j]$. Then it holds $\mathsf{kth\text{-}pos}(i) = \mathsf{select}_1(P, \mathsf{select}_\alpha(X, \mathsf{rank}_\alpha(X, j) - 1))$. This is correct because the top-k list changes when $A[i]$ enters the list, and we find the next time $X[j] = \alpha$ is mentioned, which is where $A[i]$ is finally displaced

from the top-k list. Thus, this operation can be supported in $O(1/\epsilon(k))$ time, where the time to access X dominates. Theorem 2 follows.

Theorem 2. *Given an array $A[1..n]$ and a value k, there is an encoding of A and k on a RAM machine of $w = \Omega(\lg n)$ bits that uses $n \lg k + O(\epsilon(k) n \lg k)$ bits and support* $\mathsf{kth\text{-}pos}(i)$ *queries in $O(1/\epsilon(k))$ time, for any function $\epsilon(k) \in O(1)$ and $\epsilon(k) \in \Omega(\lg \lg k / \lg k)$.*

For supporting $\mathsf{top\text{-}k\text{-}pos}(i)$ queries we need a different query on X: Given a position j, find the rightmost occurrence preceding j of every symbol in $[k]$. This can be done in $O(k)$ time using our representation of X [11]: The string is cut into chunks of size k. We can traverse the chunk of j in time $O(k)$ to find all the last occurrences, preceding j, of distinct symbols[6]. For each symbol not found in the chunk, we use constant-time rank and select on bitmaps already present in the representation to find the previous chunk where it appears, and finally find in constant time its last occurrence in that previous chunk (as we have already chosen, for our purposes, constant-time select inside the chunks).

By the discussion above on the meaning of X, it is clear that the rightmost occurrences, up to position $j = \mathsf{rank}_1(P, i) + 1$, of the distinct symbols in $[k]$, form precisely the answer to $\mathsf{top\text{-}k\text{-}pos}(i)$. Thus we find all those positions p in time $O(k)$ and remap them to the original array using $\mathsf{select}_1(P, p)$. Since we need only select queries on X, we need only $n \lg k + O(n)$ bits for it [11].

Note the top-k positions do not come sorted by largest value. By the same properties of X, if the first occurrence of α after $X[j]$ precedes the first occurrence of β after $X[j]$, then the value associated to α in our answer is smaller than that associated to β, as it is replaced earlier. Thus we find the first occurrence, after j, of all the symbols in $[k]$, analogously as before, and sort the results according to the positions found to the right, in $O(k \lg k)$ time. Thus Theorem 3 follows.

Theorem 3. *Given an array $A[1..n]$ and a value k, there is an encoding of A and k that uses $n \lg k + O(n)$ bits and supports* $\mathsf{top\text{-}k\text{-}pos}(i)$ *queries in $O(k)$ time on a RAM machine of $w = \Omega(\lg n)$ bits. The result can be sorted from largest to lowest value in $O(k \lg k)$ time. The encoding is a subset of that of Theorem 2.*

3 Two-sided range top-k queries

We now consider the problem of encoding the array $A[1..n]$ so as to answer the query $\mathsf{top\text{-}k\text{-}pos}(i, j)$. We will also consider solving top-k queries for any $k \leq \kappa$, where κ is set at construction time. Clearly, a lower bound on the encoding size of $\Omega(n \lg \kappa)$ bits follows from Section 2.1.

Corollary 1. *Any encoding of an array $A[1..n]$ answering* $\mathsf{top\text{-}k\text{-}pos}(i, j)$ *queries requires at least $(n - k) \lg k - O(n)$ bits of space.*

We give now two upper bounds for query $\mathsf{top\text{-}k\text{-}pos}(i, j)$. The first is weaker, but it is used to obtain the second.

[6] Although we do not have constant-time access to the symbols, we can select all the (overall) k positions of all the k distinct symbols in the chunk, in time $O(k)$.

3.1 Using $O(kn)$ bits and $O(k^2)$ time

Let $A[1..n] = a_1 \dots a_n$. We define, for each element a_j, κ *pointers* $j > P_1[j] > \dots > P_\kappa[j]$, to the last κ elements to the left of j that are larger than a_j.

Definition 1. *Given a sequence $a_1, \dots, a_n$, we define arrays of pointers $P_0[1..n]$ to $P_\kappa[1..n]$ as $P_0[j] = j$, and $P_{k+1}[j] = \max(\{i,\ i < P_k[j] \wedge a_i > a_j\} \cup \{0\})$.*

These pointers allow us to answer top-k queries without accessing A. We now prove a result that is essential for their space-efficient representation.

Lemma 1. *Let $1 \le j_1, j_2 \le n$ and $0 < k \le \kappa$, and let us call $i_1 = P_{k-1}[j_1]$ and $i_2 = P_{k-1}[j_2]$. Then, if $i_1 < i_2$ and $P_k[j_2] < i_1$, it holds $P_k[j_1] \ge P_k[j_2]$.*

Proof. It must hold $a_{i_1} < a_{i_2}$, since otherwise $P_k[j_2] \ge i_1$ by Definition 1 (as it would hold $a_{j_2} < a_{i_2} \le a_{i_1}$ and $0 < i_1 < i_2$), contradicting the hypothesis.

Now let us call $r_1 = P_k[j_1]$ and $r_2 = P_k[j_2] < i_1$. If it were $r_1 < r_2$ (and thus $r_2 > 0$), then we would have the following contradiction: (1) $a_{j_1} \ge a_{r_2}$ (because otherwise it would be $r_1 = P_k[j_1] \ge r_2$, as implied by Definition 1 since $r_2 = P_k[j_2] < i_1 = P_{k-1}[j_1]$ and $a_{r_2} > a_{j_1}$); (2) $a_{r_2} > a_{j_2}$ (because $r_2 = P_k[j_2]$); (3) $a_{j_2} \ge a_{i_1}$ (because otherwise it would be $r_2 = P_k[j_2] \ge i_1$, as implied by Definition 1 since $i_1 < i_2 = P_{k-1}[j_2]$ and $a_{i_1} > a_{j_2}$, and $r_2 \ge i_1$ contradicts the hypothesis); (4) $a_{i_1} > a_{j_1}$ (because $i_1 = P_{k-1}[j_1]$). □

This lemma shows that if we draw, for a given k, all the arcs starting at $P_{k-1}[j]$ and ending at $P_k[j]$ for all j, then no arc "crosses" another. This property enables a space-efficient implementation of the pointers.

Pointer representation. We represent each "level" $k > 0$ of pointers separately, as a set of arcs leading from $P_{k-1}[j]$ to $P_k[j]$. For a level $k > 0$ and for any $0 \le i \le n$, let $p_k[i] = |\{j,\ P_k[j] = i\}|$ be the number of pointers of level k that point to position i. We store a bitmap

$$T_k[1..2n+1] \quad = \quad \mathtt{1}\mathtt{0}^{p_k[0]}\,\mathtt{1}\mathtt{0}^{p_k[1]}\,\mathtt{1}\mathtt{0}^{p_k[2]} \dots \mathtt{1}\mathtt{0}^{p_k[n]},$$

where we mark the number of times each position is the target of pointers from level k. Each `1` corresponds to a new position and each `0` to the target of an arc. Note that the sources of those arcs correspond to the `0`s in bitmap T_{k-1}, that is, to arcs that go from $P_{k-2}[j]$ to $P_{k-1}[j]$. Arcs that enter the same position i are sorted according to their source position, so that we associate the leftmost `0`s of $\mathtt{0}^{p_k[i]}$ to the arcs with the rightmost sources. Conversely, we associate the rightmost `0`s of $\mathtt{0}^{p_{k-1}[i]}$ to the arcs with the leftmost targets. This rule ensures that those arcs entering, or leaving from, a same position do not cross in T_k. Then we define a balanced sequence of parentheses

$$B_k[1..2n] \quad = \quad (^{p_{k-1}[0]})^{p_{k-1}[0]}(^{p_k[0]-p_{k-1}[0]})^{p_{k-1}[1]}(^{p_k[1]})^{p_{k-1}[2]}(^{p_k[2]} \dots)^{p_{k-1}[n]}.$$

This sequence matches arc targets (opening parentheses) and sources (their corresponding closing parentheses). The arcs that leave from and enter at the special position 0 receive special treatment to make the sequence balanced.

Calling *findopen*(B_k, i) the position of the opening parenthesis matching the closing parenthesis at $B_k[i]$, the following algorithm computes the position z_k of the `0` of T_k corresponding to $P_k[j]$, given the position z_{k-1} of the `0` of T_{k-1} corresponding to $P_{k-1}[j]$.

1. $p \leftarrow rank_{\mathtt{0}}(T_{k-1}, z_{k-1})$
2. $z \leftarrow select_{\mathtt{1}}(T_{k-1}, z_{k-1} - p)$
3. $z' \leftarrow select_{\mathtt{1}}(T_{k-1}, z_{k-1} - p + 1)$
4. $p \leftarrow z' - (z_{k-1} - z)$
5. $c \leftarrow select_{)}(B_k, p)$
6. $o \leftarrow findopen(B_k, c)$
7. $r \leftarrow rank_{(}(B_k, o)$
8. $z_k \leftarrow select_{\mathtt{0}}(T_k, r)$

The code works as follows. Given the position $T_{k-1}[z_{k-1}] = \mathtt{0}$ corresponding to the target of pointer $P_{k-1}[j]$, we first compute in p the number of `0`s up to z_{k-1} in T_{k-1}. This position is corrected so as to (virtually) reverse the `0`s that form the run where z_{k-1} lies (between the `1`s at positions z and z'), in order to convert entering into leaving arcs. Then we find c, the p-th closing parenthesis in B_k, which is the target of this arc, and find its source o, the matching opening parenthesis. Finally we compute the rank r of o among the opening parentheses to the left, and match it with the corresponding `0` in T_k, z_k.

We use the code as follows. Starting with $P_0[j] = j$ and $z_0 = 2j + 1$, we use the code up to κ times in order to find, consecutively, $z_1, z_2, \ldots, z_\kappa$. At any point we have that $P_k[j] = rank_1(T_k, z_k) - 1$. Finally, since we know $T_0 = 1(10)^n$, we can avoid storing it, and replace lines 1–4 by $p \leftarrow j + 1$, when computing z_1.

By using a representation of T_k that supports constant-time *rank* and *select* operations in $2n + O(n/\lg^2 n)$ bits [17], and a representation of B_k that in addition supports operation *findopen* in constant time and the same space [20], we have that the overall space is $4\kappa n + o(n)$ bits for any $\kappa = O(\lg n)$. With such a representation, we can compute any $P_k[j]$, for any j and any $1 \leq k \leq \kappa$, in time $O(k)$ and, more precisely, in time $O(1)$ after having computed $P_{k-1}[j]$ using the same procedure.

Top-k algorithm. To find the k largest elements of $A[i..j]$ we use the structure of Fischer and Heun [8] that takes $2n + o(n)$ bits and answers RMQs in constant time. Our algorithm reconstructs the top part of the Cartesian tree [21] of $A[i..j]$ that contains the top-k elements, and also their children. The invariant of the algorithm is that, at any time, the internal nodes of the reconstructed tree are top-k elements already reported, whereas the next largest element is one of the current leaves. The tree that is reconstructed is of size at most $3k$.

The nodes p of the tree will be associated with an interval $[i_p..j_p]$ of $[1..n]$, and with a position m_p where the maximum of $A[i_p..j_p]$ occurs. In internal nodes it will hold $i_p = j_p = m_p$. Those intervals will form a cover of $[i..j]$ (i.e., will be disjoint and their union will be $[i, j]$), and values i_p (and j_p) will increase as we traverse the tree in inorder form.

We start taking $\text{RMQ}(i, j)$, which gives the position m of the maximum (top-1); this would be enough for $k = 1$. In general, we initialize a tree of just one leaf and no internal nodes. The leaf is associated to position m and interval $[i..j]$. This establishes the invariants.

To report the next largest element, we take the position m_l of the maximum of the rightmost leaf l, and traverse the interval $[i..m_l]$ backwards using $P_1[m_l]$, $P_2[m_l]$, and so on. Each position (larger than 0) we arrive at contains an element larger than $A[m_l]$. However, if those are elements we have already reported, our candidate m_l is still good to be the next one to report. To determine this in constant time, we traverse the tree in reverse inorder at the same time we do the backward interval traversal. When we are at an internal node, we know that the backward traversal will stop there, as the element is larger than $A[m_l]$. Leaves, instead, are not yet reported and their interval may be skipped by the traversal.

If, however, the backward traversal stops at a position $P_r[m_l]$ that falls within the interval of another leaf p, then m_l is not the next largest element, since $P_r[m_l]$ is not yet reported. Instead of continuing with the new candidate at position $P_r[m_l]$, we take the leaf position m_p, which is indeed the largest of the interval. We restart the backward traversal from $l \leftarrow p$, using again $P_1[m_l]$, $P_2[m_l]$, and so on. When the backward traversal surpasses the left limit i, the current candidate is the next largest element to report. We split its area into two, compute RMQs to define the two new leaves of l, and restart the process.

For example, the first thing that happens when we start this algorithm for $k > 1$ is that $P_1[m] < i$, thus we report m and create a left child with interval $[i..m-1]$ and position $\text{RMQ}(i, m-1)$ and a right child with interval $[m+1..j]$ and position $\text{RMQ}(m+1, j)$. Then we go on to report the second element.

Since each step can be carried out in constant time, and our backward traversal performs k to $3k$ steps to determine the k-th answer, it follows that the time complexity of the algorithm is $O(k^2)$. We are able to run this algorithm for any $k \leq \kappa + 1$. By renaming κ we have our final result, that with $\kappa = 1$ matches the RMQ lower bounds.

Theorem 4. *Given an array $A[1..n]$ and a value κ, there is an encoding of A and κ that uses $(4\kappa - 2)n + o(n)$ bits and supports* top-k-pos(i, j) *queries for any $k \leq \kappa$, in $O(k^2)$ time on a RAM machine of $w = \Omega(\lg n)$ bits. The positions are given sorted by value.*

3.2 Using $O(n \lg k)$ bits and $O(k)$ time

Our final solution achieves asymptotically optimal time and space, building on the results of Section 3.1. It uses Jørgensen and Larsen's "shallow cuttings" idea [14], which we now outline. Imagine the values of $A[1..n]$, already mapped to the interval $[n]$, as a grid of points $(i, A[i])$. Now sweep a horizontal line from $y = n$ to $y = 1$. Include all the points found along the sweep in a *cell*, that is, a rectangle $[1, n] \times [y, n]$. Once we reach a point y_0 such that the cell reaches the threshold of containing $2\kappa+1$ points, create two new cells by splitting the current cell. Let (x, y) be the point whose x-coordinate is the median in the current cell.

This will be called a *split point*; note it is not necessarily the point (x_0, y_0) that caused the split. Then the two new cells are initialized as $[1, x] \times [y_0, n]$ and $[x, n] \times [y_0, n]$ (note the vertical limit is y_0, that of the point causing the split, which now belongs to one of the two cells). Both cells now contain κ points, and the sweep continues, further splitting the new cells as we include more points. We create a *binary tree of cells* T_C, where the new cells are the left and right children, respectively, of the current cell.

In general, at any point in time, we will have a sequence of split points already determined, $x_1, x_2, \ldots$, and the cells that are leaves in the current T_C cover an x-coordinate interval of the form $[x_i, x_{i+1}]$ (we implicitly add split points 1 and n at the extremes). When the next split occurs at a point (x_0, y_0) within the cell covering the interval $[x_i, x_{i+1}]$, we will split it into two new cells covering $[x_i, x] \times [y_0, n]$ and $[x, x_{i+1}] \times [y_0, n]$, for some x. We will associate to those cells the *keys* $[x_i, x]$ and $[x, x_{i+1}]$, respectively, and the *extents* $[x_{i-1}, x_{i+1}]$ and $[x_i, x_{i+2}]$, respectively. Finally, once we have finished the sweep on the plane, we are left with a final set of split points $x_1, x_2, \ldots, x_t$ (from now on x_i will refer to this final sequence of split points). We add t further *keyless* cells with extents $[x_{i-1}, x_{i+1}]$ for all $1 \leq i \leq t$.

Jørgensen and Larsen prove various interesting properties of this process: (i) it creates $O(t) = O(n/\kappa)$ cells, each containing κ to 2κ points (if $n \geq \kappa$); (ii) if $c = [x_i, x_j] \times [y_0, n]$ is the cell with maximum y_0 value whose key is contained in a query range $[l, r]$, then $[l, r]$ is contained in the extent of c and (iii) the top-κ values in $[l, r]$ belong to the union of the 3 cells that comprise the extent of c.

We give now an encoding of this data structure that contains two parts. The first part uses $O((n/\kappa) \lg \kappa) + o(n) = o(n)$ bits[7] and identifies in constant time the desired cell whose extent contains $[l, r]$. The second part uses $O(n \lg \kappa)$ bits and gives us the first κ elements in the extent, in $O(\kappa)$ time.

Finding the cell. We mark the final split points x_i in a bitmap $S[1..n]$ with constant-time rank and select support. As there are t bits set, S can be implemented in $O((n/\kappa) \lg \kappa) + o(n)$ bits [18]. It allows us finding in constant time the range $[m, M]$ of split points $l \leq x_m < \ldots < x_M \leq r$ contained in $[l, r]$. If this range contains zero or one split point (i.e., $m \geq M$), then $[l, r]$ is contained in the extent of the keyless cell number m and we are done for the first part.

Otherwise, the following procedure finds the desired key [14]. Find the split point x_i with maximum associated y_0-coordinate (this is the y_0 coordinate given to the two cells created by split point x_i). Find the split point x_j with second maximum. If $j < i$ (i.e., x_j is to the left of x_i), then the desired key is $[x_j, x_i]$, else it is $[x_i, x_j]$.

We map, using rank_1 on S, the range $[l, r]$ to the range $[m, M]$. Consider the array $Y[1..t]$ of y_0 values associated to the t split points. We store a range top-2 encoding T on the array Y, using the result of Section 3.1. This requires $O(n/\kappa)$ bits and returns the positions of the first and second maxima in $Y[m, M]$, x_i and

[7] It is not $o(n)$ if $\kappa = O(1)$, yet in this case the results of Section 3.1 are asymptotically equivalent.

x_j, in $O(1)$ time. Assume w.l.o.g. that $i < j$ and thus the desired key is $[x_i, x_j]$; the case $[x_j, x_i]$ is symmetric.

Now the final problem is to find the extent associated to the key $[x_i, x_j]$. For this we need to find the split points that, at the moment when the key $[x_i, x_j]$ was created, preceded x_i and followed x_j. Since, at the time we created split point x_j, the split points that existed were precisely those with y_0 larger than that associated to x_j, it follows that the split point that preceded x_i is $x_{i'}$, where $i' = P_2[j]$, as defined in Section 3.1 (Def. 1). Similarly, the split point that followed x_j was $x_{j'}$, where $j' = N_1[j]$ (N_k is defined symmetrically to P_k but pointing to the right, and it can be represented analogously). Those structures still require $O(n/\kappa)$ bits and answer in $O(1)$ time.

Thus, using $O((n/\kappa)\lg\kappa) + o(n)$ bits and $O(1)$ time, we find the extent that contains query $[l, r]$. The actual x-coordinates of the extent, $[x_{i'}, x_{j'}]$, are found using select_1 on S.

Traversing the maxima. For the second structure, we represent the tree of cells T_C using $O(n/\kappa)$ bits, so that a number of operations are supported in constant time [20]. Since the key $[x_i, x_j]$ was created with the split point x_j, the corresponding node of T_C is the left child of the j-th node of T_C in inorder. This node with inorder j is computed in constant time, and then we can compute the preorder of its left child also in constant time (note that leaves do not have inorder number, but all nodes have a preorder position) [20].

Associated to the preorder index of each node of T_C we store an array $M[1..O(\kappa)]$ over $[O(\kappa)]$, using $O(\kappa\lg\kappa)$ bits (and $O(n\lg\kappa)$ overall). This array stores the information necessary to find the successive maxima of the extent of the node. We use RMQ queries on $A[x_{i'}, x_{j'}]$. Clearly the maximum in the extent is $m_1 = \textsc{rmq}_A(x_{i'}, x_{j'})$. The second maximum is either $m_2 = \textsc{rmq}_A(x_{i'}, m_1 - 1)$ or $m_2 = \textsc{rmq}_A(m_1 + 1, x_{j'})$. Which of the two is greater is stored (in some way to be specified soon) in $M[1]$. Assume $M[1]$ indicates that $m_2 = \textsc{rmq}_A(x_{i'}, m_1 - 1)$ (the other case is symmetric). Then the third maximum is either $m_3 = \textsc{rmq}_A(x_{i'}, m_2 - 1)$, $m_3 = \textsc{rmq}_A(m_2 + 1, m_1 - 1)$, or $m_3 = \textsc{rmq}_A(m_1 + 1, x_{j'})$. Which of the three is the third maximum is indicated by $M[2]$, and so on. Note that we cannot store directly the maxima positions in M because we would need $O(\kappa\lg n)$ bits. Rather, we use M to guide the search across the Cartesian tree slice that covers the extent.

To achieve $O(\kappa)$ time we will encode the values in M in the following way. At query time, will initialize an array $I[1..k]$ and start with the interval $I[1] = [x_{i'}, x_{j'}]$. Now $M[1] = 1$ will tell us that we must now split the interval at $I[1]$ using an RMQ query, $m_1 = \textsc{rmq}_A(I[M[1]]) = \textsc{rmq}_A(I[1]) = \textsc{rmq}_A(x_{i'}, x_{j'})$. We write the two resulting subintervals in $I[2] = [x_{i'}, m_1 - 1]$ and $I[3] = [m_1 + 1, x_{j'}]$. Now $M[2] \in \{2, 3\}$ will tell us in which of those intervals is the second maximum. Assume again it is in $M[2] = 2$. Then we compute $m_2 = \textsc{rmq}_A(I[2]) = \textsc{rmq}(x_{i'}, m_1 - 1)$, and write the two resulting subintervals in $I[4] = [x_{i'}, m_2 - 1]$ and $I[5] = [m_2 + 1, m_1 - 1]$. Now $M[3] \in \{3, 4, 5\}$ tells which interval contains

the third maximum, and so on. Note the process is deterministic, so we can precompute the M values.

Therefore, we obtain in $O(\kappa)$ time the $O(\kappa)$ elements that belong to the extent, that is, the union of the three cells. By the properties of the shallow cutting, those contain the κ maxima of the query interval $[l, r]$. Therefore, in $O(\kappa)$ time we obtain the successive maxima of the extent, and filter out those not belonging to $[l, r]$. We are guaranteed to have seen the κ maxima of $[l, r]$, in order, after examining $O(\kappa)$ maxima of the extent. Note that if we want only the top-k, for $k \leq \kappa$, we also need $O(\kappa)$ time.

Theorem 5. *Given an array $A[1..n]$ and a value κ, there is an encoding of A and κ that uses $O(n \lg \kappa)$ bits and supports* top-k-pos(i, j) *queries for any $k \leq \kappa$, in $O(\kappa)$ time on a RAM machine of $w = \Omega(\lg n)$ bits. The positions are given sorted by value.*

By building this structure for $\lceil \lg \kappa \rceil$ successive powers of 2, we can use one where the search cost is $O(k)$, for any $k \leq \kappa$.

Corollary 2. *Given an array $A[1..n]$ and a value κ, there is an encoding of A and κ that uses $O(n \lg^2 \kappa)$ bits and supports* top-k-pos(i, j) *queries for any $k \leq \kappa$, in $O(k)$ time on a RAM machine of $w = \Omega(\lg n)$ bits. The positions are given sorted by value.*

4 Conclusions

We have given lower and upper bounds to several extensions of the RMQ problem, considering the encoding scenario. Some variants of range selection and range top-k queries were considered in the simpler one-sided version, where the interval starts at the beginning of the array. For those, we have obtained optimal or nearly-optimal time, and matched the space lower bound up to lower-order terms. For the general two-sided version of the problem we have largely focused on the range top-k query, where we have obtained optimal time and asymptotically optimal space (up to constant factors). Several problems remain open, especially handling range selection queries in the two-sided case, which we have not addressed. Tightening the constant space factors is also possible. Finally, most of our results fix k at construction time (although for two-sided queries we can fix a maximum k at construction time, at the price of an $O(\lg k)$ extra space factor). Removing these restrictions is also of interest.

References

1. Belazzougui, D., Boldi, P., Pagh, R., Vigna, S.: Monotone minimal perfect hashing: searching a sorted table with $o(1)$ accesses. In: Proc. SODA. pp. 785–794 (2009)
2. Bender, M., Farach-Colton, M.: The level ancestor problem simplified. Theor. Comp. Sci. 321(1), 5–12 (2004)

3. Berkman, O., Vishkin, U.: Recursive star-tree parallel data structure. SIAM J. Comp. 22(2), 221–242 (1993)
4. Brodal, G., Fagerberg, R., Greve, M., Lopez-Ortiz, A.: Online sorted range reporting. In: Proc. ISAAC. pp. 173–182. LNCS 5878 (2009)
5. Brodal, G., Gfeller, B., Jørgensen, A., Sanders, P.: Towards optimal range medians. Theor. Comp. Sci. 412(24), 2588–2601 (2011)
6. Chan, T., Wilkinson, B.: Adaptive and approximate orthogonal range counting. In: Proc. SODA. pp. 241–251 (2013)
7. Clark, D.: Compact Pat Trees. Ph.D. thesis, Univ. of Waterloo, Canada (1996)
8. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comp. 40(2), 465–492 (2011)
9. Gagie, T., Navarro, G., Puglisi, S.: New algorithms on wavelet trees and applications to information retrieval. Theor. Comp. Sci. 426-427, 25–41 (2012)
10. Gagie, T., Puglisi, S., Turpin, A.: Range quantile queries: another virtue of wavelet trees. In: Proc. SPIRE. pp. 1–6. LNCS 5721 (2009)
11. Golynski, A., Munro, I., Rao, S.: Rank/select operations on large alphabets: a tool for text indexing. In: Proc. SODA. pp. 368–373 (2006)
12. Harel, D., Tarjan, R.: Fast algorithms for finding nearest common ancestors. SIAM J. Comp. 13(2), 338–355 (1984)
13. Hsu, P., Ottaviano, G.: Space-efficient data structures for top-k completion. In: Proc. WWW. pp. 583–594 (2013)
14. Jørgensen, A., Larsen, K.: Range selection and median: Tight cell probe lower bounds and adaptive data structures. In: Proc. SODA. pp. 805–813 (2011)
15. Li, G., Ji, S., Li, C., Feng, J.: Efficient type-ahead search on relational data: a tastier approach. In: Proc. SIGMOD. pp. 695–706. ACM (2009)
16. Munro, I.: Tables. In: Proc. FSTTCS. pp. 37–42. LNCS 1180 (1996)
17. Pătraşcu, M.: Succincter. In: Proc. FOCS. pp. 305–313 (2008)
18. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Trans. Alg. 2(4), 43:1–43:25 (2007)
19. Sadakane, K.: Succinct representations of lcp information and improvements in the compressed suffix arrays. In: Proc. SODA. pp. 225–232 (2002)
20. Sadakane, K., Navarro, G.: Fully-functional succinct trees. In: Proc. SODA. pp. 134–149 (2010)
21. Vuillemin, J.: A unifying look at data structures. Comm. ACM 23(4), 229–239 (1980)