Optimal Encodings for Range Top-k, Selection, and Min-Max

Paweł Gawrychowski[1]* and Patrick K. Nicholson[2]

[1] Institute of Informatics, University of Warsaw, Poland
[2] Max-Planck-Institut für Informatik, Saarbrücken, Germany

Abstract. We consider encoding problems for range queries on arrays. In these problems the goal is to store a structure capable of recovering the answer to all queries that occupies the information theoretic minimum space possible, to within lower order terms. As input, we are given an array $A[1..n]$, and a fixed parameter $k \in [1, n]$. A *range top-k* query on an arbitrary range $[i, j] \subseteq [1, n]$ asks us to return the ordered set of indices $\{\ell_1, ..., \ell_k\}$ such that $A[\ell_m]$ is the m-th largest element in $A[i..j]$, for $1 \leq m \leq k$. A *range selection* query for an arbitrary range $[i, j] \subseteq [1, n]$ and query parameter $k' \in [1, k]$ asks us to return the index of the k'-th largest element in $A[i..j]$. We completely resolve the space complexity of both of these heavily studied problems—to within lower order terms—for all $k = o(n)$. Previously, the constant factor in the space complexity was known only for $k = 1$. We also resolve the space complexity of another problem, that we call *range min-max*, in which the goal is to return the indices of both the minimum and maximum elements in a range.

1 Introduction

Many important algorithms make use of range queries over arrays of values as subroutines [14,18]. As a prime example, text indexes that support pattern matching queries often maintain an array storing the lengths of the longest common prefixes between consecutive suffixes of the text. During a search for a pattern this array is queried in order to find the position of the minimum value in a given range. That is, a subroutine is needed that can preprocess an array A in order to answer *range minimum queries*. Formally, as input to such a query we are given a range $[i, j] \subseteq [1, n]$, and wish to return the index $k = \arg\min_{i \leq \ell \leq j} A[\ell]$. In text indexing applications memory is often the constraining factor, so the question of how many bits are needed to answer range minimum queries has been heavily studied. After a long line of research (see [2,17]), it has been determined that such queries can be answered in constant time, by storing a data structure of size $2n + o(n)$ bits [7]. Furthermore, this space bound is optimal to within lower order terms (see [7, Sec. 1.1.2]). The interesting thing is that the space does not depend on the number of bits required to store individual

* Currently holding a post-doctoral position at Warsaw Center of Mathematics and Computer Science.

elements of the array A. After constructing the data structure we can discard the array A, while still retaining the ability to answer range minimum queries.

Results of this kind, where it is shown that the solutions to all queries can be stored using less space than is required to store the original array, fall into the category of *encodings*, and, more generally, *succinct* data structures [11]. Specifically, given a set of combinatorial objects χ we wish to represent an arbitrary member of χ using $\lg|\chi| + o(\lg|\chi|)$ bits[3], while still supporting queries, if possible. If queries can be supported by the representation then we refer to it as a data structure, but if not, then we refer to it as an encoding. For the case of range minimum queries or range maximum queries, the set χ turns out to be *Cartesian trees*, which were introduced by Vuillemin [19]. For a given array A, the Cartesian tree encodes the solution to all range minimum queries, and similarly, if two arrays have the same solutions to all range minimum queries, then their Cartesian trees are identical [7].

Recently, there has been a lot of interest the following two problems, that generalize range maximum queries in two different ways. The input to each of the following problems is an array $A[1..n]$, that we wish to preprocess into an encoding occupying as few bits as possible, such that the answers to all queries are still recoverable. We assume a value $k \geq 1$ is fixed at preprocessing time.

- **Range top-k:** Given an arbitrary query range $[i,j] \subseteq [1,n]$ and $k' \in [1,k]$, return the indices of the k' largest values in $[i,j]$. This problem is the natural generalization of range maximum queries and has been the focus of a several papers, leading to asymptotically optimal lower and upper space bounds of $\Omega(n \lg k)$ and $\mathcal{O}(n \lg k)$ bits, proved by Grossi et al. [10] and Navarro, Raman, and Rao [15], respectively. The latter upper bound is a data structure that can answer range top-k' queries in optimal $\mathcal{O}(k')$ time.
- **Range k-selection:** Given an arbitrary query range $[i,j] \subseteq [1,n]$ and $k' \leq k$, return the index of the k'-th largest value in $[i,j]$. This problem was studied in a series of recent papers (see [8] and [3] for further references), culminating in data structures that occupy a linear number of words, and can answer queries in $\mathcal{O}(\lg k'/\lg\lg n + 1)$ time [4]. This query time matches a cell-probe lower bound for near-linear space data structures [12]. It is straightforward to see that any encoding of range top-k queries is also an encoding for range k-selection queries, though the question of how much time is required during a query remains unclear [15]. Very recently, Navarro, Raman, and Rao [15] described a data structure that can be used to answer range k-selection queries in optimal $\mathcal{O}(\lg k'/\lg\lg n + 1)$ time [15], and, like the range top-k data structure, occupies $\mathcal{O}(n \lg k)$ bits of space.

Our Results We present the first space-optimal encodings to range top-k—and therefore range selection also—as well as a new problem that we call *range min-max*, in which the goal is to return the indices of both the minimum and maximum element in the array. We emphasize that, on their own, the encodings

[3] We use $\lg x$ to denote $\log_2 x$.

Table 1. Old and new results. Both upper and lower bounds are expressed in bits. Our bounds make use of the binary entropy function $H(x) = x \lg(\frac{1}{x}) + (1-x)\lg(\frac{1}{1-x})$. For the entry marked with a † the claimed bound holds when $k = o(n)$.

Ref.	Query	Lower Bound	Upper Bound	Query Time
[7]	max	$2n - \Theta(\lg n)$	$2n + o(n)$	$\mathcal{O}(1)$
[10,15]	top-k	$\Omega(n \lg k)$	$\mathcal{O}(n \lg k)$	$\mathcal{O}(k')$
[5]	top-2	$2.656n - \Theta(\lg n)$	$3.272n + o(n)$	$\mathcal{O}(1)$
Thm. 1, 2	min-max	$3n - \Theta(\lg(n))$	$3n + o(n)$	$\mathcal{O}(1)$
Thm. 3, 4	top-2	$3nH(\frac{1}{3}) - \Theta(\text{polylog}(n))$	$3nH(\frac{1}{3}) + o(n)$	—
Thm. 3, 4	top-k	$(k+1)nH(\frac{1}{k+1})(1-o(1))$†	$(k+1)nH(\frac{1}{k+1}) + o(n)$	—

for range top-k and selection do not support queries efficiently: they merely store the solutions to all queries in a compressed form. However, our encoding for range min-max can be augmented with $o(n)$ additional bits of data to create a data structure that supports queries in $\mathcal{O}(1)$ time. Furthermore, even without query support, our encodings for range top-k and selection address a problem posed in the papers of Grossi et al. [10] and Navarro et al. [15].

In Table 1 we present a summary of previous and new results. Prior to this work, the only value for which the exact coefficient of n was known was the case in which $k = 1$ (i.e., range maximum queries). For even $k = 2$ the best previous estimate was that the coefficient of n is between 2.656 and 3.272 [5]. The lower bound of 2.656 was derived using generating functions and an extensive computational search [5]. In contrast, our method is purely combinatorial and gives the exact coefficient for all $k = o(n)$. For $k = 2, 3, 4$ the coefficients are (rounding up) 2.755, 3.245, and 3.610, respectively.

As mentioned above, a negative aspect of our encodings is that they appear to be somewhat difficult to use as the basis for a data structure. However, in Section 4, we present a data structure based on our encoding that *nearly* matches the optimal space bound. Explicitly, we can achieve a space bound of $(k + 1.5)nH(\frac{1.5}{k+1.5}) + o(n \lg k)$ bits with query time $\mathcal{O}(\text{poly}(k \lg n))$. Thus, our data structure achieves space much closer to the optimal bound than the previous best result [15], but the query time is worse. We leave the following data structure problem open: how can range top-k and selection queries be supported with optimal query time using space matching our encodings (to within lower order terms)?

Finally, we wish to point out that although our formulation of the range top-k problem returns the indices in sorted order, the constant factor in our lower bound also holds for the *unsorted* version, in which we return the indices in an arbitrary order, provided $k = o(n)$. This follows since any encoding strategy for unsorted range top-k can be used to construct a sorted top-k encoding, by padding the end of the input array with $k - 1$ values larger than any other. The unsorted encoding of this padded array can be used to infer the solution to an arbitrary sorted top-k query $[i, j]$ by examining the solutions to queries $[i, j], [i, j+1], ..., [i, n+k-1]$: see Appendix A for details.

Discussion of Techniques and Road Map Prior work for top-k, for $k \geq 2$, focused on encoding a decomposition of the array, called a shallow cutting [10,15]. Since shallow cuttings are a general technique used to solve many other range searching problems [13,12], these previous works [10,15] required additional information beyond storing the shallow cutting in order to recover the answers to top-k queries. Furthermore, in these works the exact constant factor is not disclosed, though we estimate it to be at least twice as large as the bounds we present. For the specific case of range top-2 queries a different encoding has been proposed based on *extended Cartesian trees* [5]. In contrast to both of the previous approaches, our encoding is based the approach of Fischer and Heun [7], who describe what is called a 2D min-heap (resp. max-heap) in order to encode range minimum queries (resp. range maximum queries). We begin in Section 2 by showing how to generalize their technique to simultaneously answer both range minimum and range maximum queries. Our encoding provides the answer to both using $3n + o(n)$ bits in total, compared to $4n + o(n)$ bits using the trivial approach of constructing both encodings separately. We then show this bound is optimal by proving that any encoding for range min-max queries can be used to distinguish a certain class of permutations. We move on in Section 3 to generalize Fischer and Heun's technique in a clean and natural way to larger values of k. Indeed, the encoding we present—like that of Fischer and Heun—is simple enough to implement. The main difficulty is proving that the bound achieved by our technique is optimal. For this we enumerate a particular class of walks, via an application of the so-called cycle lemma of Dvoretzky and Motzkin [6].

Finally, in Section 4 we show that our encoding can be used as the basis for a range top-k data structure. Though the resultant space bound and query time are suboptimal, we note that interesting challenges had to be overcome to design a data structure based on our encoding. Concisely, we required the ability to decompose the encoding into smaller blocks in order to support queries efficiently. To do this we, in some sense, generalized the pioneers approach of Jacobson [11] via a non-trivial decomposition theorem. Since balanced parentheses representations appear in many succinct data structures, we believe this will likely be of independent interest.

2 Optimal Encodings of Range Min-Max Queries

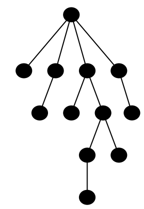
In this section we describe our encoding for range min-max queries. We use $\textsc{RMinMax}(A[i..j])$ to denote a range min-max query on a subarray $A[i..j]$. The solution to the query is the ordered set of indices $\{\ell_1, \ell_2\}$ such that $\ell_1 = \arg\max_{\ell \in [i,j]} A[\ell]$ and $\ell_2 = \arg\min_{\ell \in [i,j]} A[\ell]$.

2.1 Review of Fischer and Heun's Technique

We review the algorithm of Fischer and Heun [7] for constructing the encoding of range minimum (resp. maximum) queries.

	Min		Max	
i	Stack	Emit	Stack	Emit
1	1	1	1	1
2	2	01	1,2	1
3	2,3	1	1,3	01
4	2,3,4	1	1,4	01
5	2,3,5	01	1,4,5	1
6	2,6	001	1,4,5,6	1
7	2,6,7	1	1,4,5,7	01
8	2,8	001	1,4,5,7,8	1
9	2,8,9	1	1,4,5,9	001
10	2,8,10	01	1,4,5,9,10	1
11	2,8,10,11	1	1,4,5,9,11	01
		00001		000001

$T_{\min}$: 0000110110011001011101l



$T_{\max}$: 0000010110011011101011l

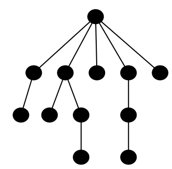


Fig. 1. A trace of Fischer and Huen's algorithm for constructing the encoding for range minimum and maximum queries on an array $A[1..11] = (11, 1, 7, 10, 9, 3, 4, 2, 8, 5, 6)$.

Consider an array $A[1..n]$ storing n numbers. Without loss of generality we can alter the values of the numbers so that they are a permutation, breaking ties in favour of the leftmost element. To construct the encoding for range minimum queries we sweep the array from left to right[4], while maintaining a stack. A string of bits $T_{\min}$ (resp. $T_{\max}$) will be emitted in reverse order as we scan the array. Whenever we push an element onto the stack, we emit a one bit, and whenever we pop we emit a zero bit. Initially the stack is empty, so we push the position of the first element we encounter on the stack, in this case, 1. Each time we increment the current position, i, we compare the value of $A[i]$ to that of the element in the position t, that is stored on the top of the stack. While $A[t]$ is not less than (resp. not greater than) $A[i]$, we pop the stack. Once $A[t]$ is less than (resp. greater than) the current element or the stack becomes empty, we push i onto the stack. When we reach the end of the array, we pop all the elements on the stack, emitting a zero bit for each element popped, followed by a one bit. An example illustrating a trace of the algorithm described here can be found in Figure 1.

Fischer and Heun showed that the string of bits output by this process can be used to encode a rooted ordinal tree in terms of its *depth first unary degree sequence* or DFUDS [7]. To extract the tree from a sequence, suppose we read d zero bits until we hit the first one bit. Based on this, we create a node v of degree d, and continue building first child of v recursively. Since there are at most $2n$ stack operations, the tree is therefore represented using $2n$ bits. We omit the technical details of how a query is answered, but the basic idea

[4] In the original paper the sweeping process moves from right to left, but either direction yields a correct algorithm by symmetry.

is to augment this tree representation with succinct data structures supporting navigation operations. The following corollary summarizes part of their result:

Lemma 1 (Corollary 5.6 [7]). *Given the DFUDS representation of* $T_{\min}$ *(resp.* $T_{\max}$*) any query* $\text{RMin}(A[i..j])$ *(resp.* $\text{RMax}(A[i..j])$*) can be answered in constant time using an index occupying* $\mathcal{O}(\frac{n \log\log n}{\log n}) = o(n)$ *additional bits of space.*

2.2 Upper Bound for Range Min-Max Queries

We propose the following encoding for a simultaneous representation of $T_{\min}$ and $T_{\max}$. Scan the array from left to right and maintain two stacks: a min-stack for range minimum queries, and a max-stack for range maximum queries. Notice that in each step except for the first and last, we are popping an element from exactly one of the two stacks. This crucial observation allows us to save space. We describe our encoding in terms of the min-stack and the max-stack maintained as above. Unlike before however, we maintain two separate bit strings, T and U. If the new element causes $\delta \geq 1$ elements on the min-stack to be popped, then we prepend $0^{\delta-1}1$ to the string T, and prepend 0 to the string U. Otherwise, if the new element causes δ elements on the max-stack to be popped, we prepend $0^{\delta-1}1$ to the string T, and 1 to the string U. Since exactly $2n$ elements are popped during n push operations, the bit string T has length $2n$, and the bit string U has length n, for a total of $3n$ bits.

Before stating our theorem, we require the following result by Raman, Raman, and Rao [16]:

Lemma 2 ([16]). *Let* $\mathcal{V}$ *be a bit vector of length* n *bits, containing* m *one bits. In the word-RAM model with word size* $\Theta(\lg n)$ *bits, there is a data structure of size* $\lg \binom{n}{m} + \mathcal{O}(\frac{n \lg\lg n}{\lg n}) \leq nH(\frac{m}{n}) + \mathcal{O}(\frac{n \lg\lg n}{\lg n})$ *bits that supports the following operations in* $\mathcal{O}(1)$ *time, for any* $i \in [1, n]$*:*

1. $\mathsf{access}(\mathcal{V}, i)$*: return the bit at index* i *in* $\mathcal{V}$.
2. $\mathsf{rank}_\alpha(\mathcal{V}, i)$*: return the number of bits with value* $\alpha \in \{0, 1\}$ *in* $\mathcal{V}[1..i]$.
3. $\mathsf{select}_\alpha(\mathcal{V}, i)$*: return the index of the* i*-th bit with value* $\alpha \in \{0, 1\}$.

Next, we show that by using our encoding and Lemma 2 it is possible to also support queries on this encoding in $\mathcal{O}(1)$ time.

Theorem 1. *There is a data structure that occupies* $3n + o(n)$ *bits of space, such that any query* $\text{RMinMax}(A[i..j])$ *can be answered in* $\mathcal{O}(1)$ *time.*

Proof. By Corollary 1, to prove the theorem, it is sufficient to show that there is a data structure that occupies $3n + o(n)$ bits of space, and can recover any block of $\lg n$ consecutive bits from both $T_{\min}$ and $T_{\max}$ in $\mathcal{O}(1)$ time.

If we have such a structure that can extract any block from either DFUDS representation, then we can use it as an oracle to access the DFUDS representation of either tree. Thus, we need only apply Lemma 1 to complete the theorem. The data structure makes use of the bit strings T and U, as well as the following auxiliary data structures:

1. We precompute a lookup table $\mathcal{L}$ of size $\Theta(\sqrt{n}\lg n)$ bits. The lookup table takes two bit strings as input, s_1 and s_2, both with length $\frac{\lg n}{4}$, as well as a single bit b. We conceptually think of the bit string s_1 as having the format $0^{\gamma_1}10^{\gamma_2}1...0^{\gamma_{t-1}}10^{\gamma_t}1$, where each $\gamma_i \geq 0$. The table returns a new bit string s_3, of length no greater than $\frac{\lg n}{4}$, that we will define next. Let $\cdot$ be the concatenation operator, and define the function:

$$f(x,y,y') = \begin{cases} 0 \cdot x & \text{if } y = y' \\ 1 & \text{otherwise.} \end{cases}$$

If $u_i = 0^{\gamma_i}1$ then $s_3 = f(u_1, s_2[1], b) \cdot f(u_2, s_2[2], b) \cdots f(u_k, s_2[k], b)$, and $s_2[i]$ denotes the i-th bit of s_2. Such a table occupies no more than the claimed amount of space, and can return s_3 (as well as k) in $\mathcal{O}(1)$ time.
2. Each bit in T corresponds to at least one bit in $T_{\min}$ or $T_{\max}$. Also recall that at each step during preprocessing we append the value $\delta - 1$ in unary to T rather than δ (as in the representation of Fischer and Heun). Thus, we can treat each push operation (with the exception of the first and last) corresponding to a single one bit in T as representing three bits: two bits in $T_{\min}$ and one bit in $T_{\max}$ or two bits in $T_{\max}$ and one bit in $T_{\min}$, depending on the corresponding value in U. We store a bit vector $B_{\min}$ of length $2n$ which marks the position in T of the bit corresponding to the $(i \lg n + 1)$-th bit of $T_{\min}$, for $0 \leq i \leq \lfloor \frac{2n}{\lg n} \rfloor$. We do the analogous procedure for $T_{\max}$ and call the resulting bit vector $B_{\max}$.

Suppose now that we support the operations rank and select on $B_{\min}$, $B_{\max}$, and T. We use the data structure of Lemma 2 that for $B_{\min}$ and $B_{\max}$ will occupy

$$O\left(\lg \binom{n}{\frac{n}{\lg n}} + \frac{n \lg \lg n}{\lg n}\right) = O\left(\frac{n \lg \lg n}{\lg n}\right)$$

bits, and for T will occupy no more than $2n + \mathcal{O}(\frac{n \lg \lg n}{\lg n})$ bits. Thus, our data structures at this point occupy $3n + o(n)$ bits in total, counting the space for U. We will describe how to recover $\lg n$ consecutive bits of $T_{\min}$; the procedure for $T_{\max}$ is analogous. Consider the distances between two consecutive 1 bits having indices x_i and x_{i+1} in $B_{\min}$. Suppose $x_{i+1} - x_i \leq c \lg n$ in $B_{\min}$, for some constant $c \geq 9$. In this case we call the corresponding block β_i of $\lg n$ consecutive bits of $B_{\min}$ *min-good*, and otherwise we call β_i *min-bad*. We also define similar notions for *max-good* and *max-bad* blocks. The problem now becomes recovering any block (good or bad), since if the $\lg n$ consecutive bits we wish to extract are not on block boundaries we can simply extract two consecutive blocks which overlap the desired range, then recover the bits in the range using bit shifting and bitwise arithmetic.

If β_i is min-good, then we can recover it in $\mathcal{O}(c) = \mathcal{O}(1)$ time, since all we need to do is scan the corresponding segment of T between the two 1s, as well as the segment of U starting at $\texttt{rank}_1(T, x_i)$. We process the bits of T and

U together in blocks of $\frac{\lg n}{4}$ each, using the lookup table $\mathcal{L}$: note that we can advance in U correctly by determining t by counting the number of 1 bits in either in s_1 or s_3. This can be done using either an additional lookup table of size $\Theta(\sqrt{n})$ using constant time, or by storing the answer explicitly in $\mathcal{L}$. When we do this, there is one border case which we must handle, which occurs when the last bit in s_1 is not a 1. However, we can simply append a 1 to end of s_1 in this case, and then delete either 1 or 01 from the end of s_3, depending on the value of $s_2[t]$. This correction can be done in $\mathcal{O}(1)$ time using bit shifting and bitwise arithmetic.

If β_i is min-bad, then we store the answer explicitly. This can be done by storing the answer for each bad β_i in an array of size $z \lg n$ bits, where z is the number of bad blocks. Since $z \leq \lceil \frac{n}{c \lg n} \rceil$ this is $\lceil \frac{n}{c} \rceil$ bits in total. We also must store yet another bit vector, encoded using Lemma 2, marking the start of the min-bad blocks, which occupies another $\mathcal{O}(\frac{n \lg \lg n}{\lg n})$ bits by a similar calculation as before. Thus, we can recover any block in $B_{\min}$ using $3n + \lceil \frac{n}{c} \rceil + o(n)$ bits in $\mathcal{O}(c) = \mathcal{O}(1)$ time.

In fact, by examining the structure of Lemma 2 in more detail we can argue that it compresses T slightly for each bad block, to get a better space bound than $2n + o(n)$ bits. Consider all the min-bad blocks $\beta_1, ..., \beta_z$ in $B_{\min}$ and the max-bad blocks $\beta'_1, ..., \beta'_{z'}$ in $B_{\max}$. For a given min-bad block β_i, any max-bad block β'_j can only overlap its first or last $2 \lg n$ bits in T. This follows since each bit in T corresponds to at least one bit in either $T_{\min}$ or $T_{\max}$, and because less than half of these $2 \lg n$ bits can correspond to bits in $T_{\min}$ (since the block is min-bad). Thus, each bad block has a middle part of at least $(c-4) \lg n$ bits, which are not overlapped by any other bad block. We furthermore observe that these $(c-4) \lg n$ middle bits are highly compressible, since they contain at most $\lg n$ one bits, by the definition of a bad block. Since these $(c-4) \lg n$ middle bits are compressed to their zeroth-order entropy in chunks of $\frac{\lg n}{2}$ consecutive bits by Lemma 2, we get that the space occupied by each of them is at most

$$\left\lceil \lg \binom{(c-4) \lg n}{\lg n} \right\rceil + \Theta(c) \leq (c-4) H\left(\frac{1}{c-4}\right) \lg n + \Theta(c) \ .$$

The cost of explicitly storing the answer for the bad block was $\lg n$ bits. Since $c \geq 9$, and assuming n is sufficiently large, we get that this additional $\lg n$ bits of space can be added to the cost of storing the middle part of the bad block in compressed form, without exceeding the cost of storing the middle part of the bad block in uncompressed form. The value of $c \geq 9$ came from a numeric calculation by finding the first value of c such that $(c-4)H(\frac{1}{c-4}) + 1 < (c-4)$. Thus, the total space bound is $3n + o(n)$ bits. □

2.3 Lower Bound for Range Min-Max Queries

Given a permutation $\pi = (p_1, ..., p_n)$, we say π contains the permutation pattern s_1-s_2-...-s_m if there exists a subsequence of π whose elements have the same relative ordering as the elements in the pattern. That is, there exist some $x_1 <$

$x_2 < ... < x_m \in [1, n]$ such that for all $i, j \in [1, m]$ we have that $\pi(x_i) < \pi(x_j)$ if and only if $s_i < s_j$. For example, if $\pi = (1, 4, 2, 5, 3)$ then π contains the permutation pattern 1-3-4-2: we use this hyphen notation to emphasize that the indices need not be consecutive. In this case, the series of indices in π matching the pattern are $x_1 = 1$, $x_2 = 2$, $x_3 = 4$ and $x_4 = 5$. If no hyphen is present between elements s_i and s_{i+1} in the permutation pattern, then the indices x_i and x_{i+1} must be consecutive: i.e., $x_{i+1} = x_i + 1$. In terms of the example, π does not contain the permutation pattern 1-34-2.

A permutation $\pi = (p_1, ..., p_n)$ is a *Baxter permutation* if there exist no indices $1 \le i < j < k \le n$ such that $\pi(j+1) < \pi(i) < \pi(k) < \pi(j)$ or $\pi(j) < \pi(k) < \pi(i) < \pi(j+1)$. Thus, Baxter permutations are those that do not contain 2-41-3 and 3-14-2. Permutations with less than 4 elements are trivially Baxter permutations, and for permutations on 4 elements the non-Baxter permutations are exactly $(2, 4, 1, 3)$ and $(3, 1, 4, 2)$. Baxter permutations are well studied, and their asymptotic behaviour is known (see, e.g., OEIS A001181 [1]).

We have the following lemma:

Lemma 3. *Suppose π is a Baxter permutation, stored in an array $A[1..n]$ such that $A[i] = \pi(i)$. If an encoding that can recover all range minimum and maximum queries is constructed on A, then π can be recovered from the encoding.*

Proof. In order to recover the permutation, it suffices to show that we can perform pairwise comparisons on any two elements in A using range minimum and range maximum queries. The proof follows by induction on n.

For the base case, for $n = 1$ there is exactly one permutation, so there is nothing to recover. Thus, let us assume that the lemma holds for all permutations on less than $n \ge 2$ elements. For a permutation on n elements, consider the subpermutation induced by the array prefix $A[1..(n-1)]$ and suffix $A[2..n]$. These subpermutations must be Baxter permutations, since deleting elements from the prefix or suffix of a Baxter permutation cannot create a 2-41-3 or a 3-14-2. Thus, it suffices to show that we can compare $A[1]$ and $A[n]$, as all the remaining pairwise comparisons can be performed by the induction hypothesis.

Let $x = \text{RMin}(A[1..n])$ and $y = \text{RMax}(A[1..n])$ be the indices of the minimum and maximum elements in the array, respectively. If $x \in \{1, n\}$ or $y \in \{1, n\}$ we can compare $A[1]$ and $A[n]$, so assume $x, y \in [2, n-1]$. Without loss of generality we consider the case where $x < y$: the opposite case is symmetric (i.e., replacing 3-14-2 with 2-41-3), and $x \ne y$ because $n \ge 2$. Consider an arbitrary index $i \in [x, ..., y]$, and the result of comparing $A[1]$ to $A[i]$ and $A[i]$ to $A[n]$ (that can be done by the induction hypothesis, as $i \in [2, n-1]$). The result is a partial order on three elements, and is either:

1. One of the two chains $A[1] < A[i] < A[n]$ or $A[n] < A[i] < A[1]$, in which case we are done since $A[1]$ and $A[n]$ can be compared; or
2. A partial order in which $A[i]$ is the minimum or maximum element, and $A[1]$ is incomparable with $A[n]$.

If we are in the latter case for all $i \in [x, y]$, then let $f(i) = 0$ if $A[i]$ is the minimum element in this partial order, and $f(i) = 1$ otherwise. Because of how

x and y were chosen, $f(x) = 0$ and $f(y) = 1$. If we consider the values of $f(i)$ for all $i \in [x, y]$, there must exist two indices $i, i+1 \in [x, y]$ such that $f(i) = 0$ and $f(i+1) = 1$. Therefore, the indices $1, i, i+1, n$ form the forbidden pattern 3-14-2, unless $A[1] < A[n]$. □

Theorem 2. *Any data structure encoding range minimum and maximum queries simultaneously must occupy* $3n - \Theta(\log n)$ *bits, for sufficiently large values of* n.

Proof. Let $L(n)$ be the number of Baxter permutations on n elements. It is known (cf. [1]) that $\lim_{n\to\infty} \frac{L(n)\pi\sqrt{3}n^4}{2^{3n+5}} = 1$. Since we can encode and recover each one by the procedure discussed in Lemma 3, our encoding data structure must occupy at least $\lg L(n) = 3n - \Theta(\log n)$ bits, if n is sufficiently large. □

3 Optimal Encodings for Top-k Queries

In this section we use RTopK($A[i..j]$) to denote a range top-k query on the subarray $A[i..j]$. The solution to such a query is an ordered list of indices $\{\ell_1, ..., \ell_k\}$ such that $A[\ell_m]$ is the m-th largest element in $A[i..j]$.

3.1 Upper Bound for Encoding Top-k Queries

Like the encoding for range min-max queries, our encoding for range top-k queries is based on representing the changes to a certain structure as we scan through the array A. Each prefix in the array will correspond to a different structure. We denote the structure, that we will soon describe, for prefix $A[1..j]$ as $S_k(j)$, for all $1 \le j \le n$. The structure $S_k(j)$ will allow us to answer RTopK($A[i..j]$) for any $i \in [1, j]$. Our encoding will store the differences between $S_k(j)$ and $S_k(j+1)$ for all $j \in [1, n-1]$. Let us begin by defining a single instance for an arbitrary j.

We first define the directed graph $G_j = (V, E)$ with vertices labelled $\{1, ..., j\}$, and where an edge $(i', j') \in E$ iff both $i' < j'$ and $A[i'] < A[j']$ for all $1 \le i' < j' \le j$. We call G_j the *dominance graph* of $A[1..j]$, and say j' *dominates* i', or i' is dominated by j', if $(i', j') \in E$. Next consider the out-degree $d_j(\ell)$ of the vertex labelled $\ell \in [1, j]$ in G_j. We define an array $S[1..j]$, where $S[\ell] = d_j(\ell)$ for $1 \le \ell \le j$. The structure $S_k(j)$ is defined as follows: take the array $S[1..j]$, and for each entry $\ell \in [1, j]$ such that $S[\ell] > k$, replace $S[\ell]$ with k. We use the notation $S_k(j, \ell)$ to refer to the ℓ-th array entry in the structure $S_k(j)$. We refer to an index ℓ to be *active* iff $S_k(j, \ell) < k$, and as *inactive* otherwise. We note that $S_k(n)$ is reminiscent of the one-sided top-k structure of Grossi et al. [10].

Lemma 4. *The total ordering of elements* $A[i_1], ..., A[i_{j'}]$, *where* $\{i_1, ..., i_{j'}\}$ *are the active indices in* $S_k(j)$, *can be recovered by examining only* $S_k(j)$.

Proof. We scan the structure $S_k(j)$ from index j down to 1, maintaining a total ordering on the active elements seen so far. Initially, we have an empty total ordering. At each active location ℓ the value $S_k(j, \ell)$ indicates how many active

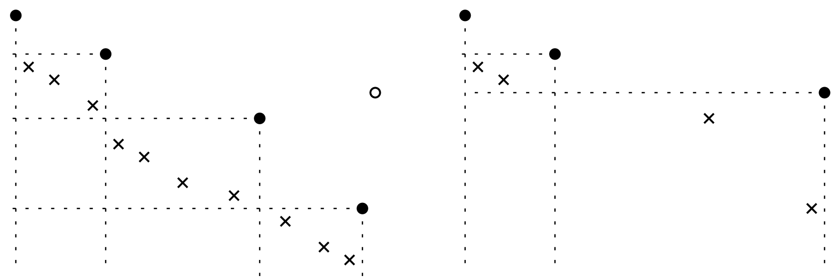


Fig. 2. Geometric interpretation of how the structure $S_k(j)$ is updated to $S_k(j+1)$. In the example $k = 2$, and the value of each active element in the array is represented by its height. Black circles denote 0 values in the array $S_2(j)$, whereas crosses represent 1 values, and 2 values (inactive elements) are not depicted. When the new point (empty circle) is inserted to the structure on the left, it increments the counters of the smallest 10 active elements, resulting in the picture on the right representing $S_2(j + 1)$.

elements in locations $[\ell+1, j]$ are larger than $A[\ell]$. This follows since an inactive element cannot dominate an active element in the graph G_j. Thus, we can insert $A[\ell]$ into the current total ordering of active elements. $\square$

We define the *size* of $S_k(j)$ as follows: $|S_k(j)| = \sum_{\ell=1}^{j}(k - S_k(j, \ell))$. The key observation is that the structure $S_k(j + 1)$ can be constructed from $S_k(j)$ using the following procedure:

1. Compute the value $\delta_j = |S_k(j)| - |S_k(j+1)| + k$. This quantity is always non-negative, as we add one new element to the large staircase, which increases the size by at most k.
2. Find the δ_j indices among the active elements in $S_k(j)$ such that their values in A are the smallest via Lemma 4. Denote this set of indices as $\mathcal{I}$.
3. For each $\ell \in [1, j]$, set $S_k(j+1, \ell) = S_k(j, \ell) + 1$ iff $\ell \in \mathcal{I}$, and $S_k(j+1, \ell) = S_k(j, \ell)$ otherwise.
4. Add the new element at the end of the array, setting $S_k(j + 1, j + 1) = 0$.

Thus, to construct $S_k(j + 1)$ all that is needed is $S_k(j)$ and the value δ_j: see Figure 2. This implies that by storing δ_j for $j \in [1, n-1]$ we can build any $S_k(j)$.

Theorem 3. *Solutions to all queries* RTopK$(A[i..j])$ *can be encoded in at most* $(k + 1)nH(\frac{1}{k+1})$ *bits of space.*

Proof. Suppose we store the bitvector $0^{\delta_1}10^{\delta_2}1\ldots0^{\delta_{n-1}}1$. This bitvector contains no more than kn zero bits. This follows since each active counter can be incremented k times before it becomes inactive. Thus, storing the bitvector requires no more than $\lg\binom{(k+1)n}{n} \leq (k + 1)nH(\frac{1}{k+1})$ bits.

Next we prove that this is all we need to answer a query RTopK$(A[i..j])$. We use the encoding to construct $S_k(j)$. We know that for every element at inactive index ℓ in $S_k(j)$ there are at least k elements with larger value in $A[\ell + 1..j]$.

Consequently, these elements need not be returned in the solution, and it is enough to recover the indices of the top-k values among the elements at active indices at least i. We apply Lemma 4 on $S_k(j)$ to recover these indices and return them as the solution. □

3.2 Lower Bound for Encoding Top-k Queries

The goal of this section is to show that the encoding from Section 3.1 is, in fact, optimal. The first observation is that all structures $S_k(j)$ for $j \in [1, n]$ can be reconstructed with RTopK queries.

Lemma 5. *Any $S_k(j)$ can be reconstructed with* RTopK *queries.*

Proof. To reconstruct $S_k(j)$, we execute the query RTopK$(A[\ell..j])$ for each $\ell \in [1, j]$. If index ℓ is returned as the k'-th largest element in $[\ell, j]$, then by definition there are exactly $k' - 1$ elements in locations $A[\ell + 1..j]$ with value larger than $A[\ell]$. Thus, ℓ is an active location and $S_k(j, \ell) = k' - 1$. If ℓ is not returned by the query, then it is inactive and we set $S_k(j, \ell) = k$. □

Recall that we encode all structures by specifying $\delta_1, \delta_2, \ldots, \delta_{n-1}$. We call an $(n-1)$-tuple of nonnegative integers $(\delta_1, \delta_2, \ldots, \delta_{n-1})$ *valid* if it encodes some $S_k(1), S_k(2), \ldots, S_k(n)$, i.e., if there exists at least one array $A[1..n]$ consisting of distinct integers such that the structure constructed for $A[1..j]$ is exactly the encoded $S_k(j)$, for every $j = 1, 2, \ldots, n$. Then the number of bits required by the encoding is at least the logarithm of the number of valid $(n-1)$-tuples $(\delta_1, \delta_2, \ldots, \delta_{n-1})$. Our encoding from Section 3.1 shows this number is at most $\binom{(k+1)n}{n}$, but we need to argue in the other direction, which is far more involved.

Recall that the size of a particular $S_k(j)$ is $|S_k(j)| = \sum_{i=1}^{j}(k - S_k(j, i))$. We would like to argue that there are many valid $(n-1)$-tuples $(\delta_1, \delta_2, \ldots, \delta_{n-1})$. This will be proven in a series of transformations.

Lemma 6. *If $(\delta_1, \delta_2, \ldots, \delta_{n-1})$ is valid, then for any $\delta_n \in \{0, 1, \ldots, \lceil \frac{M}{k} \rceil\}$ where $M = \sum_{i=1}^{n-1}(k - \delta_i)$, the tuple $(\delta_1, \delta_2, \ldots, \delta_{n-1}, \delta_n)$ is also valid.*

Proof. Let $A[1..n]$ be an array such that the structure constructed for $A[1..j]$ is exactly $S_k(j)$, for every $j = 1, 2, \ldots, n$. By definition of δ_j, we have that $M = \sum_{i=1}^{n-1}(k - \delta_i) < |S_k(n)|$. Denote the number of active elements in $S_k(j)$ with the corresponding entry set to α as m_α for $\alpha \in [0, k-1]$. For any $s \in \{0, 1, \ldots, \sum_{\alpha=0}^{k-1} m_\alpha\}$, we can adjust $A[n+1]$ so that it is larger than exactly the s smallest active elements in $S_k(n)$. Thus, choosing any $\delta_n \in \{0, 1, \ldots, \sum_{\alpha=1}^{k} m_\alpha\}$ results in a valid $(\delta_1, \delta_2, \ldots, \delta_n)$. Since $|S_k(n)| = \sum_{\alpha=0}^{k-1}(k-\alpha)m_\alpha \leq k\sum_{\alpha=0}^{k-1} m_\alpha$, we have $\sum_{\alpha=0}^{k-1} m_\alpha \geq \left\lceil \frac{|S_k(n)|}{k} \right\rceil$, proving the claim. □

Every valid $(n-1)$-tuple $(a_1, a_2, \ldots, a_{n-1})$ corresponds in a natural way to a walk of length $n-1$ in a plane, where we start at $(0, 0)$ and perform steps of the form $(1, a_i)$, for $i = 1, 2, \ldots, n-1$. We consider a subset of all such walks.

Denoting the current position by (x_i, y_i), we require that a_i is an integer from $[k - \lceil \frac{y_i}{k} \rceil, k]$. Under such conditions, any walk corresponds to a valid $(n-1)$-tuple $(\delta_1, \delta_2, \ldots, \delta_{n-1})$, because we can choose $\delta_i = k - a_i$ and apply Lemma 6. Therefore, we can focus on counting such walks.

The condition $[k - \lceil \frac{y_i}{k} \rceil, k]$ is not easy to work with, though. We will count more restricted walks instead. A Y-restricted nonnegative walk of length n starts at $(0,0)$ and consists of n steps of the form $(1, a_i)$, where $a_i \in Y$ for $i = 1, 2, \ldots, n$, such that the current y-coordinate is always nonnegative. Y is an arbitrary set of integers.

Lemma 7. *The number of valid $(n-1)$-tuples is at least as large as the number of $[k - \Delta, k]$-restricted nonnegative walks of length $n - 1 - \Delta$.*

Proof. We have already observed that the number of valid $(n-1)$-tuples is at least as large as the number of walks consisting of $n-1$ steps of the form $(1, a_i)$, where $a_i \in [k - \lceil \frac{y_i}{k} \rceil, k]$ for $i = 1, 2, \ldots, n-1$. We distinguish a subset of such walks, where the first Δ steps are of the form $(1, k)$, and then we always stay above (or on) the line $y = k\Delta$. Under such restrictions, $a_i \in [k - \Delta, k]$ implies $a_i \in [k - \lceil \frac{y_i}{k} \rceil, k]$, so counting $[k - \Delta, k]$-restricted nonnegative walks gives us a lower bound on the number of valid $(n-1)$-tuples. □

We move to counting Y-restricted nonnegative walks of length n. Again, counting them directly is non-trivial, so we introduce a notion of Y-restricted returning walk of length n, where we ignore the condition that the current y-coordinate should be always nonnegative, but require the walk ends at $(n, 0)$.

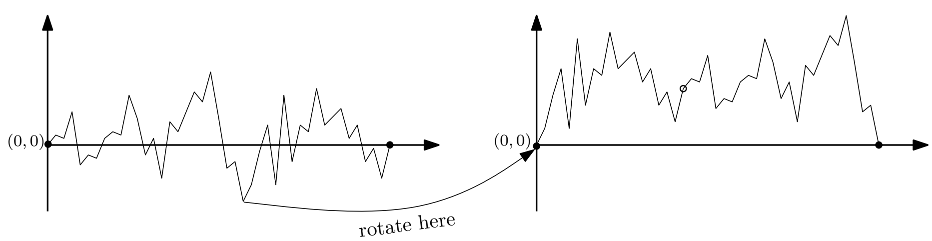


Fig. 3. Left: a Y-restricted walk ending at $(n, 0)$. Right: a cyclic rotation of the walk on the left such that the walk is always nonnegative.

Lemma 8. *The number of Y-restricted nonnegative walks of length n is at least as large as the number of Y-restricted returning walks of length n divided by n.*

Proof. This follows from the so-called cycle lemma [6], but we prefer to provide a simple direct proof. We consider only Y-restricted nonnegative walks of length n ending at $(n, 0)$, and denote their set by W_1. The set of Y-restricted returning walks of length n is denoted by W_2. The crucial observation is that a cyclic rotation of any walk in W_2 is also a walk in W_2. Moreover, there is always at least one such cyclic rotation which results in the walk becoming nonnegative (see Figure 3). Therefore, we can define a total function $f : W_2 \to W_1$, that takes a walk w and rotates it cyclically as to make it nonnegative. Because there are just n cyclic rotations of a walk of length n, any element of W_1 is the image of at most n elements of W_2 through f. Therefore, $|W_1| \geq \frac{|W_2|}{n}$ as claimed. □

The only remaining step is to count $[k-\Delta, k]$-restricted returning walks of length $n-1-\Delta$. This is equivalent to counting ordered partitions of $k(n-1-\Delta)$ into parts $a_1, a_2, \ldots, a_{n-1-\Delta}$, where $a_i \in [0, \Delta]$ for every $i = 1, 2, \ldots, n-1-\Delta$. This follows since a partition of size ℓ corresponds to a step of size $k - \ell$.

Lemma 9. *The number of ordered partitions of N into g parts, where every part is from $[0, B]$, is at least $\binom{N-2g'+g-1}{g-g'-1}$, where $g' = \lfloor \frac{N}{B} \rfloor$.*

Proof. The number of ordered partitions of N into g parts, where there are no restrictions on the sizes of the parts, is simply $\binom{N+g-1}{g-1}$. To take the restrictions into the account, we first split N into blocks of length B (except for the last block, which might be shorter). This creates $g'+1$ blocks. Then, we additionally split the blocks into smaller parts, which ensures that all parts are from $[0, B]$. We restrict the smaller parts, so that the first and the last smaller part in every block is strictly positive. This ensures that given the resulting partition into parts, we can uniquely reconstruct the blocks. Therefore, we only need to count the number of ways we can split the blocks into such smaller parts, and by standard reasoning this is at least $\binom{N-2g'+g-1}{g-g'-1}$. This follows by conceptually merging the last element in block i with the first element in block $i+1$, so that no further partitioning can happen between them, and then partitioning the remaining set into $g - g'$ pieces. Every such partition corresponds to a distinct restricted partition obtained by splitting between the merged elements, which creates g' additional blocks. □

We are ready to combine all the ingredients. Setting $N = k(n-1-\Delta)$, $g = n-1-\Delta$, $g' = \left\lfloor \frac{k(n-1-\Delta)}{\Delta} \right\rfloor = \left\lfloor \frac{k(n-1)}{\Delta} \right\rfloor - k$ and substituting, the number of bits required by the encoding is:

$$\lg \binom{N-2g'+g-1}{g-g'-1} > \lg \binom{(k+1)(n-2-\Delta-g')}{n-2-\Delta-g'} .$$

Using the entropy function as a lower bound, this is at least $(k+1)n'H(\frac{1}{k+1}) - \Theta(\log n')$, where $n' = n-2-\Delta-g' \geq n(1-\frac{k}{\Delta}) + \frac{k}{\Delta} + k - 2 - \Delta$. Thus, we have the following theorem:

Theorem 4. *For sufficiently large values of n, any data structure that encodes range top-k queries must occupy $(k+1)n'H(\frac{1}{k+1}) - \Theta(\log n')$ bits of space, where $n' \geq n(1-\frac{k}{\Delta}) + \frac{k}{\Delta} + k - 2 - \Delta$, and $\Delta \geq 1$ can be selected to be any positive integer. If $k = o(n)$, then Δ can be chosen such that $\Delta = \omega(k)$ and $\Delta = o(n)$, yielding that the lower bound is $(k+1)nH(\frac{1}{k+1})(1-o(1))$ bits.*

4 Data Structure for Top-k Queries

In this section we show how to use the encoding of Section 3.1 to construct a data structure that supports top-k queries efficiently.

The high-level idea is to decompose the array into blocks, and construct a new array by storing the k largest elements in each block. Then, we build a naive structure over the new (short) array, called the macro structure, and additionally store a small separate structure for every block, called the micro structure. This is a standard approach in succinct data structures, but as soon as we try to apply it in the top-k setting, quite a few difficulties appear. The micro structures should be based on the encoding from Section 3.1, which in turn is based on encoding how the $S_k(j)$'s change. But these changes can be, in some cases, very non-local, and hence it is not obvious how the blocks should be defined. This problem also occurs in, for example, encodings for balanced parenthesis, where the so-called pioneers approach is used [9]. Here the situation is even more complex, and we start with developing an appropriate decomposition through a series of technical lemmas. Then, using the decomposition, we construct the macro structure, which allows us to answer any query spanning more than one block, and the micro structure, which allows us to answer any query fully inside a single block.

4.1 Good Decompositions

Consider the array A, and the structure $S_k(j)$ at each array index $j \in [1, n]$. Recall that the structure $S_k(j)$ is an array, where each entry is an integer drawn from the range $[1, k]$. For technical reasons we define $S_k(0)$ to be an empty array. See Table 2 for an example of these definitions for $k = 2$.

Table 2. Suppose $A = \{46, 31, 93, 16, 45, 77, 25, 57, 26\}$. We give the structures $S_k(j)$ for A in the following table. The encoding for A is: 1100110010001100101.

i	1	2	3	4	5	6	7	8	9
$A[i]$	46	31	93	16	45	77	25	57	26
$S_2(0, i)$									
$S_2(1, i)$	**0**								
$S_2(2, i)$	**0**	**0**							
$S_2(3, i)$	**1**	**1**	**0**						
$S_2(4, i)$	**1**	**1**	**0**	**0**					
$S_2(5, i)$	**1**	**2**	**0**	**1**	**0**				
$S_2(6, i)$	**2**	**2**	**0**	**2**	**1**	**0**			
$S_2(7, i)$	**2**	**2**	**0**	**2**	**1**	**0**	**0**		
$S_2(8, i)$	**2**	**2**	**0**	**2**	**2**	**0**	**1**	**0**	
$S_2(9, i)$	**2**	**2**	**0**	**2**	**2**	**0**	**2**	**0**	**0**

Let $\mathcal{C}(i) = \{a_1, ..., a_z\}$ be the set of all indices such that $S_k(i-1, a_\ell) \neq S_k(i, a_\ell)$ for $1 \leq \ell \leq z$; this set will include the index i. Furthermore, define

$\mathcal{C}(i_1, i_2) = \cup_{i=i_1}^{i_2} \mathcal{C}(i)$. In the example, $\mathcal{C}(5) = \{2, 4, 5\}$, and $\mathcal{C}(5, 6) = \{1, 2, 4, 5, 6\}$. Note that the encoding described in Section 3.1 is such that $\delta_i = |\mathcal{C}(i) \setminus \{i\}|$ for $i \in [1, n]$.

Conceptually, we divide the range $[1, n]$ into disjoint *even-blocks* of length B: $[1, B], [B + 1, 2B], ...,$ for some parameter $B \geq 1$ that we will fix later, and without loss of generality, assume that B divides n. We use the notation $\mathcal{B}_i$ to denote the range $[Bi + 1, B(i + 1)]$ for $i \in [1, \frac{n}{B}]$.

Our goal is to decompose the array into a collection of disjoint *blocks*. Each block will have the property that it consists of a range of at most B contiguous array elements, and will be also contained within at most one even-block. We refer to blocks that span a single array element as *singletons*.

Suppose our decomposition $\mathcal{D}$ consists of h blocks, $\mathcal{G}_1, ...\mathcal{G}_h$, and that block $\mathcal{G}_i$ consists of the contiguous range $[g(i), g(i + 1) - 1]$ in A, where $1 \leq i \leq h$, $g(1) = 1$, and $g(h + 1) = n + 1$. We call $\mathcal{D}$ *good* if:

D1 Size Constraint: the total number of blocks is $h = \mathcal{O}(\frac{k^2 n}{B})$.

D2 Weight Constraint: Consider the changes in the structures $S_k(g(i)), S_k(g(i)+1), \ldots$ that occur as we scan the indices of an arbitrary block $\mathcal{G}_i$, from left to right. A good decomposition has that the number of changes (i.e., increment operations) occurring in the structures as a result of the elements in a block is relatively small, if the block is not a singleton. Formally, we have that $\sum_{j=g(i)}^{g(i+1)-1} |\mathcal{C}(j)| \leq B$ for $1 \leq i \leq h$ if $\mathcal{G}_i$ is not a singleton. Note that this implies that the bit string $0^{\delta_{g(i)}} 1 0^{\delta_{g(i)+1}} 1 \ldots 0^{\delta_{g(i+1)-1}} 1$ has length at most B.

D3 Window Constraint: Consider the changes in the structures that occur as we process each individual block. The indices of the structures that change are located in a relatively small range, if the block is not a singleton. Formally, suppose that $\mathcal{G}_i \subseteq \mathcal{B}_t$ for some $t \in [1, \frac{n}{B}]$. Then we have that $(\mathcal{C}(g(i), g(i + 1) - 1) \setminus \mathcal{B}_t) \subseteq \mathcal{B}_w$ for some $w \in [1, t - 1]$, if the block $\mathcal{G}_i$ is not a singleton. We call $\mathcal{B}_w$ the *window* of block $\mathcal{G}_i$.

The remainder of this section proves that we can construct a good decomposition.

Lemma 10. *There is a good decomposition $\mathcal{D}$ of the array A.*

Proof. We describe a procedure for computing a decomposition satisfying these conditions. For each position $i \in [1, n]$, we define the *weight* $w_i = |\mathcal{C}(i)|$. The weight of a range in A is equal to the sum of the weights of the positions it spans. Positions with weight larger than B are called *fat*, and will be singletons in our decomposition. Since each w_i corresponds to w_i zero bits in the encoding plus one, and there are at most kn zero bits, the number of fat elements is at most $\mathcal{O}(\frac{kn}{B})$.

Consider the remaining non-fat elements. We combine these non-fat elements into $\mathcal{O}(\frac{kn}{B})$ blocks such that the weights of the ranges is at most B. This can be done by iteratively merging pairs of blocks (initially blocks are just individual non-fat elements), until the sum of the weight of any two adjacent blocks exceeds

B. When this happens, every other block will have weight at least $\frac{B}{2}$, and by the argument above there can be at most $\mathcal{O}(\frac{kn}{B})$ such blocks. Furthermore, we subdivide these blocks along the boundaries of even-blocks, introducing at most $\mathcal{O}(\frac{n}{B})$ additional blocks.

We refer to the above decomposition as the *initial decomposition*. The initial decomposition satisfies conditions **D1** (in fact it has $\mathcal{O}(\frac{kn}{B})$ blocks rather than $\mathcal{O}(\frac{k^2n}{B})$), and **D2**, but not necessarily **D3**. Thus, we must further refine the blocks in order to ensure to create a good decomposition. We do this by splitting them using an iterative procedure that we now describe.

For each block $\mathcal{G}_i \subseteq \mathcal{B}_t$ in the initial decomposition, we scan it from left to right, calling the current position x_0. We will split it into a (potentially large) number of new blocks. At each step, there are two cases depending on whether the set $\mathcal{C}(g(i), x_0) \setminus \mathcal{B}_t$ is contained within a single even-block.

1. If it is, then we extend the current block which begins at position $g(i)$ by adding position x_0 to it.
2. If not, then we split the current block between positions $x_0 - 1$ and x_0, i.e., set $g(i+1) = x_0$. Furthermore, when this occurs we make position x_0 a singleton block. We then recursively apply the same procedure to the remaining unscanned part of the block adjusting the parameters appropriately. Thus, we have introduced two additional blocks.

Such a refinement clearly has the desired window property. However, the difficulty is arguing that the second case only occurs $\mathcal{O}(k^2 \frac{n}{B})$ times. To show this, we use a charging argument in which each split is charged to the rightmost even-block $\mathcal{B}_w$ containing a position in $\mathcal{C}(x_0) \setminus \mathcal{B}_t$. We will bound the number of times $\mathcal{B}_w$ can be charged for a split by $\mathcal{O}(k^2)$.

We say a position is y-active if it is active in structure $S_k(y)$. Consider the (x_0-1)-active elements immediately before a split occurs. Consider the position $a \in \mathcal{B}_w$ such that $a \in \mathcal{C}(x_0)$ and $A[a]$ is maximum. We have that $S_k(x_0-1, a) < k$ since a is, by definition, $(x_0 - 1)$-active. Moreover, since a split occurred, there must be some block $\mathcal{B}_{w'}$ where $w' < w$ containing a position $a' \in \mathcal{B}_{w'}$ such that $a' \in \mathcal{C}(x_0)$. Since a' is also $(x_0 - 1)$-active this implies that there are at most $k - 1$ $(x_0 - 1)$-active positions contained in $\mathcal{B}_w$, whose corresponding elements have values larger than $A[a]$. Thus, when a split occurs, all but at most $k-1$ of the (x_0-1)-active locations contained in $\mathcal{B}_w$ are incremented in $S_k(x_0)$. Furthermore, any location not incremented must be among the $k - 1$ largest values in $A[Bw + 1, B(w + 1)]$. Thus, after k split operations, all but the $k - 1$ largest active locations become inactive. Since each split increments at least one location in $\mathcal{B}_w$ at most $k(k-1)$ additional splits occur before all elements in $\mathcal{B}_w$ become inactive. Overall, at most $k + k(k - 1) = \mathcal{O}(k^2)$ splits can occur before all elements in $\mathcal{B}_w$ become inactive.

Since there are $\frac{n}{B}$ even-blocks, we have that the total number of blocks created by splits (or otherwise) is $\mathcal{O}(\frac{k^2n}{B})$, completing the proof. □

4.2 Navigating the Encoding

Before discussing the data structures we store, we require an additional result, called an *indexable dictionary*, by Raman, Raman, and Rao [16]:

Lemma 11 ([16]). *Let $\mathcal{V}$ be a bit vector of length n bits, containing m one bits. In the word-RAM model with word size $\Theta(\lg n)$ bits, there is a data structure of size $\lg \binom{n}{m} + \mathcal{O}(m) + \mathcal{O}(\lg \lg n) \leq nH(\frac{m}{n}) + \mathcal{O}(m) + \mathcal{O}(\lg \lg n))$ bits that supports the following operations in $\mathcal{O}(1)$ time, for any $i \in [1, n]$:*

1. $\mathsf{access}(\mathcal{V}, i)$*: return the bit at index i in $\mathcal{V}$.*
2. $\mathsf{rank}_1(\mathcal{V}, i)$*: return the number of bits with value 1 in $\mathcal{V}[1..i]$, iff $\mathsf{access}(\mathcal{V}, i) = 1$. If $\mathsf{access}(\mathcal{V}, i) = 0$, then a flag is returned indicating that the operation cannot be supported.*
3. $\mathsf{select}_1(\mathcal{V}, i)$*: return the index of the i-th bit with value 1.*

We apply Theorem 10 to partition A into $\mathcal{O}(\frac{nk^2}{B})$ blocks $\mathcal{G}_1, \mathcal{G}_2, \ldots$, where B is some parameter that will be fixed later on. We then construct the following indexes:

1. *Block index:* This is the rank/select data structure of Lemma 2 constructed on a bit vector of length n marking the block boundaries. This allows us to find the start of an arbitrary block in constant time. This bit vector occupies:
$$\lg \binom{n}{\frac{nk^2}{B}} + \mathcal{O}\left(\frac{n \lg \lg n}{\lg n}\right) \leq nH\left(\frac{k^2}{B}\right) + \mathcal{O}\left(\frac{n \lg \lg n}{\lg n}\right)$$
bits of space, by Lemma 2.
2. *Encoding index:* Consider the bit vector storing the encoding E (described in Section 3.1) on A. For each zero bit in the encoding E, we say that bit is *associated* with the one bit immediately to its right. That is, the zero bit at position i is associated with the one bit in position $\mathsf{select}_1(E, \mathsf{rank}_1(E, i)+1)$. Since the j-th one bit in the encoding is representing element $A[j]$, each zero bit associated with this one bit can also be said to be associated with $A[j]$. Suppose $A[j]$ is part of a block $\mathcal{G}_i$ which is contained in even-block $\mathcal{B}_t$ and has a window contained in even-block $\mathcal{B}_w$. The 0 bits associated with position $A[j]$ come in exactly two flavors:
   (a) Internal increment: if the 0 bit corresponds an increment operation in $S_k(j)$ that occurs inside even-block B_t
   (b) Window increment: if the 0 bit corresponds an increment operation in $S_k(j)$ that occurs inside the window B_w

   Suppose that for each $j \in [1, n]$ we create two bit vectors $E_{\text{INT}}(j)$ and $E_{\text{WIN}}(j)$. These two bit vectors will be of the form $0^{\alpha}1$ and $0^{\beta}1$, respectively, where α_j is the number of internal increments associated with position j and β_j is the number of window increments associated with position j. Note that $\delta_j = \alpha_j + \beta_j$. Let E_{INT} and E_{WIN} be the concatenation of the $E_{\text{INT}}(j)$ and $E_{\text{WIN}}(j)$ bit vectors, respectively. Both of these bit vectors together have

$2n$ one bits, and kn zero bits. Thus, storing E_{INT} and E_{WIN} in the smaller of the two representations discussed (either Lemma 2 or 11) will occupy $(k+2)nH(\frac{2}{k+2}) + \mathcal{O}(\min\{\frac{nk \lg\lg(nk)}{\lg(nk)}, n\})$ bits in total. Note that we cannot perform rank operations on arbitrary positions in these bit vectors using the bound just stated, though we can perform arbitrary select operations.

Lemma 12. *Using the above data structures we can recover the length j' suffix of the structure $S_k(Bi+j)$ for any $i \in [1, \frac{n}{B}]$, $j \in [1, B-1]$ and $j' \in [1, j]$ in $\mathcal{O}(B^2)$ time.*

Proof. Let $m_\alpha = \mathsf{select}_1(E_{\text{INT}}, \alpha)$, and consider the range of E_{INT} between $[m_{Bi-1}+1, m_{Bi+j}]$. This range contains all the one bits in E_{INT} associated with elements $A[Bi], A[Bi+1], \ldots, A[Bi+j]$, and also the zero bits associated with these one bits that are internal increments. Furthermore the length of this range in E_{INT} is at most $\mathcal{O}(B^2)$, since even in the case where every position in the even-block is a singleton, the number of internal increments for each of these is upper bounded by the length of the even-block, B. Thus, to recover the fragment of the structure $S_k(Bi+j)$, we construct an array of length $j+1$ in which each index stores a $\lceil \lg(k+1) \rceil$ bit number. We process the internal increments associated with the elements $A[Bi]$, $A[Bi+1]$, ..., etc. in order, calling the current position ℓ, where $Bi \leq \ell \leq Bi + j$. We maintain the total ordering over all currently active elements in length $\ell - Bi + 1$ suffix of $S_k(\ell)$ as follows. Suppose position ℓ is associated with x internal increments (we can determine this by comparing m_ℓ and $m_{\ell-1}$ in $\mathcal{O}(1)$ time using the encoding index). We insert position ℓ into the total order as the $(x+1)$-th smallest element, set its counter value to 0, and increment the counters associated with the x smallest elements. If an incremented counter exceeds $k-1$, then we remove it from the total order. Maintaining the total ordering as a linked list is sufficient to process the fragment in $\mathcal{O}(x) = \mathcal{O}(B)$ time per position ℓ. Since there are at most $\mathcal{O}(B)$ positions, the total time is $\mathcal{O}(B^2)$. □

Lemma 13. *Given a subarray $A[x_1..x_2]$ that is contained within a block, we can return a list L such that $L[p]$ stores the position of the p-th largest element in $A[x_1..x_2]$ in $\mathcal{O}(B^2)$ time.*

Proof. Using Lemma 12 we can build the length $\ell = x_2 - x_1 + 1$ suffix of the structure $S_k[x_2]$, which stores ℓ $\lceil \lg(k+1) \rceil$-bit numbers. Once we have this length ℓ array, we scan it from right to left, constructing the total order of elements in $A[x_1..x_2]$ by Lemma 4. As before, using a linked list to store the total order is sufficient to achieve the claimed time bound. □

4.3 Version Control

One issue that arises is that to answer queries we will need to construct fragments of the structure $S_k(j)$ for various values of j, which are not necessarily short suffixes. In particular, given a block $\mathcal{G}_i$, we wish to be able to reconstruct its *window fragment*, which is the fragment of the structure $S_k(g(i)-1)$ corresponding to the window of block $\mathcal{G}_i$. Suppose the window is even-block $\mathcal{B}_w$. Lemma 12 only allows us to construct the length B suffix of structure $S_k(B(w+1))$, rather than the window fragment of $\mathcal{G}_i$. Thus, we are interested in how much space is required to recover a window fragment given what we can recover using Lemma 12.

Lemma 14. *Suppose block $\mathcal{G}_i$ has window $\mathcal{B}_w$. The difference $\mathtt{diff}(i)$ between the window fragment of $\mathcal{G}_i$ and the length B suffix of $S_k(B(w+1))$ can be stored using $\Theta(k \lg(B+1))$ bits. Using $\mathtt{diff}(i)$, in addition to the other data structures described thus far, we can construct the window fragment of $\mathcal{G}_i$ in time $\mathcal{O}(B^2)$.*

Proof. Lemmas 12 and 4 allow us to recover the total order $\mathcal{L}$ of the $(B(w+1))$-active elements in the window fragment in $\mathcal{O}(B^2)$ time. Consider the sequence of positions in the array A, $\{x_1, ..., x_z\}$ that have window increments associated with them occurring within the window fragment, where $x_1 > B(w+1)$ and $x_z < g(i)$. Each element $A[x_\ell]$ can be mapped to a position y_ℓ in the total order $\mathcal{L}$. It is sufficient to record the k largest values in this mapping, as all $B(w+1)$-active positions represented in $\mathcal{L}$ which are smaller than the k-th largest such value will become $(g(i)-1)$-inactive. Storing how these k values interleave with the ordering $\mathcal{L}$ requires at most $k\lceil\lg(B+1)\rceil$ bits of space. Note that we do not need to know the positions where these elements occur in A in order to reconstruct the window fragment, just their positions in the total ordering $\mathcal{L}$, which contains at most B elements. □

We store $\mathtt{diff}(i)$ for each $i \in [1, h]$ (recall h is the number of blocks). This requires $\mathcal{O}(hk \lg B) = \mathcal{O}(\frac{nk^3 \lg(B+1)}{B})$ bits of space in total.

4.4 Decomposing Queries

Any range top-k query is either fully within a single block, or consists of three parts: a suffix of a block $\mathcal{G}_i$ that we call the *left part*, then a number of full blocks $\mathcal{G}_{i+1}, \ldots, \mathcal{G}_{j-1}$ that we call the *middle part*, and finally a prefix of a block $\mathcal{G}_j$ that we call the *right part*. Note that any of these three parts may be an empty range. Using the block index we can determine these parts in $\mathcal{O}(1)$ time.

We construct a new array A' by keeping the k largest elements from every block (if a block is a singleton, this is just one element) and normalizing all the elements by sorting. A' is stored explicitly and augmented with a range maximum query structure, which allows us to locate the k largest element in any query range via a three-sided range reporting query: this can be done in $\mathcal{O}(k)$ time and $\mathcal{O}(\frac{nk^3 \lg n}{B})$ bits of space, using successive queries to a range maximum structure built over A' since we have access to these elements.

Additionally, for every j such that $A[j]$ appears in A', i.e., is one of the k largest elements in its block, we store the positions of the first k larger elements on its left in A. This requires space $\mathcal{O}(\frac{nk^3 \lg n}{B})$ bits.

4.5 Wrap Up

Now that we have described all of the data structures, we can explain how to extract the positions of the top-k elements, given a query range $A[i..j]$. The algorithm will consist of first finding the positions of the top-k elements in the middle part, and the total ordering of elements in the left and right parts. Extracting the solution from the middle part is trivial, since we have a top-k data structure explicitly stored on the top-k elements in each block. Extracting the total ordering of elements from the left (or right) part can be done by applying Lemma 13 to the even-block containing the left or right part.

At this point, we have at most three lists L_1,L_2, and L_3, storing positions of the top elements from the left, middle, and right parts respectively, i.e., $L_p[q]$ is the position of the q-th largest element in list p. We now argue that we can merge these lists.

Lemma 15. *Suppose we are given a query $A[i..j]$. A list L can be constructed such that $L[q]$ is the position of the q-th largest element, for $1 \leq q \leq k$, in $A[i..j]$ in time $\mathcal{O}(k + B^2)$.*

Proof. First, we construct the three lists L_1, L_2 and L_3 as described above in time $\mathcal{O}(B^2)$. Then we merge the lists L_1 (the left part) with the list L_2 (the middle part). This is done by examining the left pointers of each position in L_2. Consider the subset $\{\Upsilon_1, ..., \Upsilon_{k'}\}$ of positions in the left part such that $L_2[p]$ has a left pointer to Υ_r, for $1 \leq r \leq k'$. If $p + k' \leq k$, then implies that $L_2[p]$ is the $(p + k')$-th largest element in the combination of the left and middle parts. Otherwise, it implies that $L_2[p]$ is not in the top-k in the combination of the two parts. Using this procedure we merge the lists L_1 and L_2, calling the result L'.

Next we describe how to merge L' and L_3 (the right part). Recall that the right part is a prefix of some block $\mathcal{G}_r$. We reconstruct the window fragment of $\mathcal{G}_r$ using $\texttt{diff}(r)$. We then scan through $\mathcal{G}_r$ up to position j, performing window increments on the window fragment by reading E_{WIN}. Let $m_\alpha = \mathsf{select}_1(E_{\mathrm{WIN}}, \alpha)$. We read the window increments of E_{WIN} from the range $[m_{g(r)-1} + 1, m_j]$. Since $\mathcal{G}_r$ is not a singleton block (otherwise it would be fully contained in the middle part), we have that the length of this range in E_{WIN} is bounded by B. We process the window increments in order to reconstruct the range $\mathcal{B}_w$ spanned by the window of $\mathcal{G}_r$ in the structure $S_k(j)$. During this process, considering a position $j' \in [g(r), j]$, we observe that if $L'[p] \notin \mathcal{C}(j')$, then $A[L'[p]] > A[j']$, unless position $L'[p]$ had been made inactive by a previous window increment earlier in the process. If $L'[p]$ is not in $\mathcal{B}_w$, then we can infer that $A[L'[p]] > A[j']$ immediately. Thus, it is possible to insert the positions $g(r), \ldots, j$ into the list L' to construct the final list L containing the top-k positions in $A[i..j]$. □

From the above lemmas, we immediately get the following theorem:

Theorem 5. *There is a data structure occupying*

$$(k+2)nH\left(\frac{2}{k+2}\right) + nH\left(\frac{k^2}{B}\right) + \mathcal{O}\left(\frac{k^3 n \lg n}{B} + \min\left\{\frac{nk \lg\lg(nk)}{\lg(nk)}, n\right\}\right)$$

bits of space, and supports range top-k queries in $\mathcal{O}(k + B^2)$ time.

By setting $B = k^3 \lg n \sqrt{f(n)}$, for a strictly increasing function f, we get the following result:

Corollary 1. *For any strictly increasing function f, there is a data structure occupying $(k+2)nH(\frac{2}{k+2}) + o(n \lg k)$ bits of space, and supports range top-k queries in $\mathcal{O}(k^6 \lg^2 n f(n))$ time.*

4.6 Improvement to Space Bound

Our final theorem argues that we can slightly improve the space bound:

Theorem 6. *For any strictly increasing function f, there is a data structure occupying $(k+1.5)nH\left(\frac{1.5}{k+1.5}\right) + o(n \lg k)$ bits of space, and supports range top-k queries in $\mathcal{O}(k^6 \lg^2 n f(n))$ time.*

Proof. We observe that we need not store a 1 in the bit vector E_{WIN} for elements that are not in the top-k of their prefix of their even block, as such elements perform no window increments. Initially, this does not seem to buy us anything, since every position can be in the top-k of the prefix of its even block, but in this case we can take the reversal of the array. We call an element *bad* if it is in the top-k of the prefix or suffix of its even block, and *good* otherwise.

To bound the number of bad positions, consider the top-$2k$ elements in each even-block. No other elements can be bad, since there is a subset of size at least k of these top-$2k$ elements on either its right or left. Next consider a good element. It can only contribute a one bit to E_{WIN} in A or to the reverse of A, but not both. Thus, we have $n - \frac{2kn}{B}$ elements contributing $n - \frac{2kn}{B}$ one bits to the window encodings for either A or its reverse. We therefore need only record $\frac{n}{2} - \frac{2kn}{2B} + \frac{2kn}{B} = \frac{n}{2} - o(n)$ one bits for the window encoding bit vector of A or its reverse. This reduces the leading term of the space cost to $(k+1.5)nH(\frac{1.5}{k+1.5})$. To correct for the fact that we have removed one bits from E_{WIN}, we must adjust select operations on this bit vector by explicitly storing, for each block, how many elements in the block are good. Then, when we process the window increments in a block, we can determine whether an element is good by examining its internal increments. This adds an overhead of $\mathcal{O}(\frac{nk^2 \lg n}{B}) = o(n)$ bits of space and adds an $\mathcal{O}(B^2)$ time cost for determining which elements are good in a block. □

References

1. OEIS Foundation Inc. (2011), The On-Line Encyclopedia of Integer Sequences, Number of Baxter permutations of length n. `http://oeis.org/A001181`, accessed: 2014-09-24
2. Bender, M.A., Farach-Colton, M., Pemmasani, G., Skiena, S., Sumazin, P.: Lowest common ancestors in trees and directed acyclic graphs. Journal of Algorithms 57(2), 75–94 (2005)

3. Brodal, G.S., Gfeller, B., Jørgensen, A.G., Sanders, P.: Towards optimal range medians. Theoretical Computer Science 412(24), 2588–2601 (2011)
4. Chan, T.M., Wilkinson, B.T.: Adaptive and Approximate Orthogonal Range Counting. In: Proc. of the 24th Annual ACM-SIAM Symposium on Discrete Algorithms (SODA). pp. 241–251. SIAM (2013)
5. Davoodi, P., Navarro, G., Raman, R., Rao, S.: Encoding Range Minima and Range Top-2 Queries. Phil. Trans. R. Soc. A 372(2016), 1471–2962 (2014)
6. Dvoretzky, A., Motzkin, T.: A problem of arrangements. Duke Mathematical Journal 14(2), 305–313 (1947)
7. Fischer, J., Heun, V.: Space-efficient preprocessing schemes for range minimum queries on static arrays. SIAM J. Comput. 40(2), 465–492 (2011)
8. Gagie, T., Puglisi, S., Turpin, A.: Range quantile queries: Another virtue of wavelet trees. In: Proc. 16th International Symposium on String Processing and Information Retrieval (SPIRE). LNCS, vol. 5721, pp. 1–6. Springer (2009)
9. Geary, R.F., Rahman, N., Raman, R., Raman, V.: A simple optimal representation for balanced parentheses. Theor. Comput. Sci. 368(3), 231–246 (2006)
10. Grossi, R., Iacono, J., Navarro, G., Raman, R., Satti, S.R.: Encodings for Range Selection and Top-k Queries. In: Proc. 21st Annual European Symposium on Algorithms (ESA). LNCS, vol. 8125, pp. 553–564. Springer (2013)
11. Jacobson, G.: Space-efficient static trees and graphs. pp. 549–554. IEEE (1989)
12. Jørgensen, A.G., Larsen, K.G.: Range selection and median: Tight cell probe lower bounds and adaptive data structures. In: Proc. of the Twenty-Second Annual ACM-SIAM Symposium on Discrete Algorithms. pp. 805–813. SIAM (2011)
13. Matoušek, J.: Reporting points in halfspaces. Computational Geometry 2(3), 169–186 (1992)
14. Navarro, G.: Spaces, trees, and colors: The algorithmic landscape of document retrieval on sequences. ACM Comput. Surv. 46(4), 52 (2013)
15. Navarro, G., Raman, R., Satti, S.R.: Asymptotically Optimal Encodings for Range Selection. In: Proc. 34th International Conference on Foundation of Software Technology and Theoretical Computer Science, (FSTTCS). LIPIcs, vol. 29, pp. 291–301. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2014)
16. Raman, R., Raman, V., Rao, S.S.: Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. ACM Transactions on Algorithms 3(4) (2007)
17. Sadakane, K.: Succinct data structures for flexible text retrieval systems. Journal of Discrete Algorithms 5(1), 12 – 22 (2007)
18. Skala, M.: Array range queries. In: Space-Efficient Data Structures, Streams, and Algorithms - Papers in Honor of J. Ian Munro on the Occasion of His 66th Birthday. LNCS, vol. 8066, pp. 333–350. Springer (2013)
19. Vuillemin, J.: A unifying look at data structures. Communications of the ACM 23(4), 229–239 (1980)

A Lower bound for Unsorted Range Top-k

Let a *sorted* range top-k query denote the problem of returning the indices $i_1, \ldots, i_k$ of the k largest values in a query range $[i, j]$, in ascending order: i.e., $A[i_j]$ is the j-th largest value. Let an *unsorted* range top-k query denote the weaker query in which the indices $i_1, \ldots, i_k$ are returned in an arbitrary order.

Lemma 16. *If $\mathcal{S}(n, k)$ is the number of bits required to store an encoding of sorted range top-k queries on an array $A[1..n]$, then at least $\mathcal{S}(n-k, k)$ bits are required to store an encoding of unsorted range top-k queries.*

Proof. Suppose there exists an encoding for unsorted range top-k queries that requires strictly less than $\mathcal{S}(n-k, k)$ bits. We will show that such an encoding can be used to construct an encoding for sorted range top-k queries that occupies strictly less than $\mathcal{S}(n, k)$ bits. We pad the input array $A[1..n]$ with k additional values $A[n+1], \ldots, A[n+k]$ such that $A[n+i] > A[j]$ for all $i \in [1, k]$ and $j \in [1, n]$. We now claim that the unsorted encoding for the padded array can be used to recover solutions to all sorted range top-k queries on ranges in $[1, n]$. Given a query range $[i, j]$, we examine the solutions to unsorted range top-k queries $[i, j], [i, j+1], \ldots, [i, n+k]$. Let $\kappa(j')$ denote the set of indices in $[i, j']$, $\kappa_0 = \kappa(j)$, $\kappa_\ell = \kappa(\ell')$ where ℓ' is the minimum index such that $\kappa(\ell'-1) \neq \kappa_{\ell_0}$, for $\ell \in [1, k]$. By the method we use to pad A, it implies that $\kappa_k \cap \kappa_0 = \emptyset$, since the solution to query $[i, n+k]$ is the set of indices in $[n+1, n+k]$. Thus, the index of the $k-i$-th largest element in the sorted solution can be extracted by computing $\kappa_i \setminus \kappa_{i+1}$ for $i \in [0, k-1]$. This follows since the smallest element in κ_i is removed, and a new elemented added to create κ_{i+1}. Therefore, we have a contradiction, since any encoding for the sorted variant must occupy $\mathcal{S}(n, k)$ bits, and we have given an encoding that occupies strictly less than $\mathcal{S}(n+k-k, k) = S(n, k)$ bits. □

Thus, for $k = o(n)$ the previous lemma, combined with Theorem 4 (which provides the function $\mathcal{S}(n, k)$), implies that the space required for the unsorted encoding on an array of n elements is within additive lower order terms of the space required for the sorted encoding on n elements.