

# Asymptotically Optimal Encodings for Range Selection\*

Gonzalo Navarro<sup>1</sup>, Rajeev Raman<sup>2</sup>, and Srinivasa Rao Satti<sup>3</sup>

1 Department of Computer Science, University of Chile, Chile  
gnavarro@dcc.uchile.cl

2 Department of Computer Science, University of Leicester, UK  
r.raman@leicester.ac.uk

3 School of Computer Science & Engg, Seoul National University, S. Korea  
ssrao@cse.snu.ac.kr

---

## Abstract

We consider the problem of preprocessing an array  $A[1..n]$  to answer *range selection* and *range top- $k$*  queries. Given a query interval  $[i..j]$  and a value  $k$ , the former query asks for the position of the  $k$ th largest value in  $A[i..j]$ , whereas the latter asks for the positions of all the  $k$  largest values in  $A[i..j]$ . We consider the *encoding* version of the problem, where  $A$  is not available at query time, and an upper bound  $\kappa$  on  $k$ , the rank that is to be selected, is given at construction time. We obtain data structures with asymptotically optimal size and query time on a RAM model with word size  $\Theta(\lg n)$ : our structures use  $O(n \lg \kappa)$  bits and answer range selection queries in time  $O(1 + \lg k / \lg \lg n)$  and range top- $k$  queries in time  $O(k)$ , for any  $k \leq \kappa$ .

**1998 ACM Subject Classification** F.2.2. Nonnumerical Algorithms and Problems, E.2 Data Storage Representations, E.4 Coding and Information Theory

**Keywords and phrases** Data Structures, Order Statistics, Succinct Data Structures, Space-efficient Data Structures

**Digital Object Identifier** 10.4230/LIPIcs.FSTTCS.2014.291

## 1 Introduction

We consider the problem of preprocessing an array  $A[1..n]$  over a totally ordered universe, so that the following queries can be efficiently answered:

- Range selection:  $\text{select}(i, j, k)$  returns the position of the  $k$ th largest element in  $A[i..j]$ .
- Range top- $k$ :  $\text{top}(i, j, k)$  returns the positions of the  $k$  largest elements in  $A[i..j]$ .

We can assume that  $A$  is a permutation of  $[n]$ , since replacing each element  $A[i]$  by its rank in  $A$  yields correct answers to those queries. The range selection problem has received a lot of interest in recent years [4, 3, 13, 5]. Following a series of earlier papers, Brodal and Jørgensen [4] presented a structure using linear space and  $O(\lg n / \lg \lg n)$  time, for any  $k$  given at query time. The model used for this result, as well as the other results in this paper, is the *word RAM* model with word size  $w = \Theta(\log n)$  bits. Jørgensen and Larsen [13] improved the time to  $O(\lg k / \lg \lg n + \lg \lg n)$ , still within linear space, and proved that  $\Omega(\lg k / \lg \lg n)$  time is needed when using  $n \lg^{O(1)} n$  space. Finally, Chan and Wilkinson [5]

---

\* Navarro funded in part by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F; Satti 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).



matched this lower bound, obtaining  $O(1 + \lg k / \lg \lg n)$  time using linear space<sup>1</sup>. This result implies, via a reduction first observed in [4], an optimal  $O(k)$ -time solution to the range top- $k$  problem as well.

In this paper, we are interested in the *encoding model*, where the array  $A$  is not available at query time, and therefore the data structure must contain enough information to answer queries by itself. One can always use a non-encoding data structure such as that of Chan and Wilkinson [5], on a copy  $A'$  of  $A$ , 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, and has time equal to that of the best non-encoding data structure. 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 *select* query).

Existing non-trivial solutions for this problem in the encoding model are as follows. In the case  $k = 1$ , both queries boil down to the well-known *range maximum query (RMQ)*, which can be answered in constant time and  $2n + o(n)$  bits, matching the lower bound of  $2n - O(\lg n)$  bits to within lower-order terms [9]. Note that the space usage is  $O(n / \lg n)$  words, or sublinear. The case  $k = 2$  was recently considered by Davoodi et al. [7]. Grossi et al. [11] considered encodings for general  $k$ , showing that  $\Omega(n \lg k)$  bits are needed to encode answers to either selection or top- $k$  queries. Therefore, interesting encodings can only exist if an upper bound  $\kappa$  on  $k$  is given at construction time—the so-called  *$\kappa$ -bounded rank* variant of this problem [13]. For general  $k$ , Grossi et al. [11] gave an asymptotically optimal-space and  $O(1)$  time solution for the (much simpler) case where  $k$  is fixed at construction time and furthermore, only *one-sided* queries (i.e. query intervals of the form  $A[1, j]$ ) are supported. Optimal-space encodings for the two-sided range selection problem can be obtained via encodings of the range top- $k$  problem given by Grossi et al. [11] described below; these however have poor running times. Chan and Wilkinson gave a (bounded-rank) range selection encoding for general  $k$  that answers *select* queries in  $O(1 + \lg k / \lg \lg n)$  time. Its space usage, however, is  $O(n(\lg \kappa + \lg \lg n + (\lg n)/\kappa))$  bits, which is non-optimal.

In this paper we show that the same optimal time can be obtained in the encoding model, using asymptotically optimal space.

► **Theorem 1.** *Given an array  $A[1..n]$  and a value  $\kappa$ , there is an encoding of  $A$  that uses  $O(n \lg \kappa)$  bits and supports the query  $\text{select}(i, j, k)$  in  $O(1 + \lg k / \lg \lg n)$  time for any  $k \leq \kappa$ .*

Furthermore, our development allows us to obtain asymptotically optimal time and space for the encoding range top- $k$  problem.

► **Theorem 2.** *Given an array  $A[1..n]$  and a value  $\kappa$ , there is an encoding of  $A$  that uses  $O(n \lg \kappa)$  bits and supports the query  $\text{top}(i, j, k)$  in time  $O(k)$ , for any  $k \leq \kappa$ .*

Grossi et al. [11] gave a range top- $k$  encoding using  $O(n \lg \kappa)$  bits that answers top- $k$  queries in  $O(\kappa)$  time, for any  $k \leq \kappa$ . To achieve the optimal  $O(k)$  time, they require  $O(n \lg^2 \kappa)$  bits. Note that Grossi et al.'s result implies an optimal-space (bounded-rank) range selection encoding with running time  $O(\kappa)$ .

In general, the low space usage of encoding data structures is useful when the values in  $A$  themselves are uninteresting, and one just wants to query about their relative magnitudes. An example of range top- $k$  queries used for autocompletion search is given by Grossi et

---

<sup>1</sup> Chan and Wilkinson claim a bound of  $O(1 + \log_w k)$  for the “trans-dichotomous” model where the word size  $w = \Omega(\log n)$ ; this is, however, based on an incorrect application [17] of a result of Grossi et al. [12], and the proof presented in [5] only yields a time bound of  $O(1 + \log k / \log \log n)$ .

al. [11]; the problem arises frequently in data and log mining applications as well. In addition, our result for range selection allows, for example, delivering the top- $k$  results in sorted order. It is also useful for interfaces where, say, the top- $k$  results are displayed and then, upon user request, the  $(k + 1)$ th to  $2k$ th results are displayed, and so on. Even when  $A$  is needed, the sub-linear space usage of encoding data structures means that multiple copies of range selection data structures can be built over one copy of  $A$ , and still take less space than  $A$  (this trick is used already in the non-encoding result of [5]).

The next section gives some basic concepts and the roadmap of the paper.

## 2 Preliminaries

Grossi et al. [11] build their results on top of the *shallow cutting* technique [13, 5]. We revisit (a slight variant of) this construction, as we also build on it.

Let  $A[1..n]$  be a permutation on  $[n]$ . Furthermore, consider each entry  $A[i]$  as a point  $(x, y) = (i, A[i])$ , and set a parameter  $\kappa$ . A horizontal line sweeps the space  $[1, n] \times [1, n]$  from  $y = n$  to  $y = 1$ . The points hit are included in a single *root cell*, which spans a three-sided area called a *slab*, of the form  $[1, n] \times [y, n]$ , including all the points of the cell. Once we reach a point  $(x^*, y^*)$  that makes the root cell contain  $2\kappa$  points, we *close* the cell and leave its final slab as  $[1, n] \times [y^*, n]$ . Then we create two *children cells* of  $\kappa$  points as follows. Let  $x_{\text{split}}$  be the  $\kappa$ th  $x$ -coordinate in the root cell. This is called the *split point*. Then the new cells contain the points whose  $x$ -coordinates are  $\leq x_{\text{split}}$  and  $> x_{\text{split}}$ , respectively, and their initial slabs are thus  $[1, x_{\text{split}}] \times [y^*, n]$  and  $[x_{\text{split}} + 1, n] \times [y^*, n]$  (these will grow downwards as we continue with the sweeping process, independently on each cell). When those cells reach size  $2\kappa$ , they are split again, and so on. A binary tree  $T_C$  is created to reflect the cell refinement process. The root cell is associated with the root node of  $T_C$ , the first two children cells to the left and right children of the root, and so on. The leaves of  $T_C$  are associated with the final cells, which have not been split and contain  $\kappa$  to  $2\kappa - 1$  points (unless  $n < \kappa$ ).

At any moment of the sweeping process, there is a sequence of split points  $x_1, x_2, \dots$ , which grows as further cells are split. The current leaves of  $T_C$  cover an interval of  $x$ -coordinates  $[x_i + 1, x_{i+1}]$  (we implicitly assume split points 0 and  $n$  at the extremes). When the next split occurs, within the cell covering interval  $[x_i + 1, x_{i+1}]$ , we split the cell into two new cells covering the  $x$ -coordinate intervals  $[x_i + 1, x_{\text{split}}]$  and  $[x_{\text{split}} + 1, x_{i+1}]$ . We associate the *keys*  $[x_i + 1, x_{\text{split}}]$  and  $[x_{\text{split}} + 1, x_{i+1}]$  and the *extents*  $[x_{i-1} + 1, x_{i+1}]$  and  $[x_i + 1, x_{i+2}]$ , respectively, with the two new cells. After the sweep finishes, the sequence of split points is of the form  $0 = x_0 < x_1 < x_2 < \dots < x_{n'} = n$ . In the following, we will use  $x_i$  to refer to this final sequence of split points. Then we add  $n'$  further *keyless* cells with extents  $[x_{i-1} + 1, x_{i+1}]$  for all  $1 \leq i \leq n'$ . Note that  $\kappa \leq x_{i+1} - x_i \leq 2\kappa$  for all  $i$  (if  $n \geq \kappa$ ).

This construction has useful properties [13]: (i) it creates  $O(n') = O(n/\kappa)$  cells, each containing  $\kappa$  to  $2\kappa$  points (if  $n \geq \kappa$ ); (ii) if  $c$  is the cell of the highest (closest to the root) node  $v \in T_C$  whose key is contained in a query range  $[i..j]$ , then  $[i..j]$  is contained in the extent of  $c$ ; and (iii) the top- $\kappa$  values in  $[i..j]$  belong to the union of the points in the 3 cells comprising the extent of  $c$ .

With these properties, Chan and Wilkinson [5] reduce the  $O(\lg n / \lg \lg n)$  time of Brodal and Jørgensen [4] as follows. At each node  $v \in T_C$ , they store the structure of Brodal and Jørgensen for the array  $A_v[1..O(\kappa)]$  of the  $y$ -coordinates of the points in the extent of  $v$ . Actually, they store in  $A_v$  the local permutation in  $[O(\kappa)]$  induced by the relative ordering in  $A$ , thus  $A_v$  requires  $O(\kappa \lg \kappa)$  bits in each  $v$  and  $O(n \lg \kappa)$  bits in total. The structure for range selection also uses  $O(\kappa \lg \kappa)$  bits and answers queries in time  $O(1 + \lg_w \kappa)$ . They also

store an array  $P_v[1..O(\kappa)]$ , so that  $P_v[i]$  is the position in  $A[1..n]$  of the value stored in  $A_v[i]$ .

Property (iii) above implies that the  $k$ th largest element of  $A[i..j]$ , for any  $k \leq \kappa$ , is also the  $k$ th largest value in  $A_v[l, r]$ , where  $v$  is the node that corresponds to interval  $[i..j]$  by property (ii) and  $P_v[l-1] < i \leq j < P_v[r+1]$  are the elements in the extent of node  $v$  enclosing  $[i..j]$  most tightly. Thus query  $\text{select}(i, j, k)$  on  $A$  is mapped to query  $p = \text{select}(l, r, k)$  on  $A_v$ . Once the local answer is found in  $A_v[o]$ , the global answer is  $P_v[o]$ . Chan and Wilkinson [5] manage to store all the  $P_v$  arrays in  $O(n \lg(\kappa \lg n) + (n/\kappa) \lg n)$  bits, which gives  $O(n \lg n)$  bits when added over a set of suitable  $\kappa$  values. This is linear space, but too large for an encoding.

Grossi et al. [11] use an  $O(n')$ -bit representation of the topology of  $T_C$  [16] that carries out a number of operations in constant time, plus a bit-vector of length  $n$  to mark the  $x_i$  values. With these and some additional structures of total size  $O(n)$  bits, they show how to find the appropriate node  $v \in T_C$ , as well as the cell and extent limits, corresponding to a range  $A[i..j]$ , in constant time. They can also map between  $i$  and  $x_i$ , and compute the interval  $[x_l, x_r]$  of splitting points contained in any node  $v$ , all in constant time.

In the sequel we build a space- and time-optimal encoding for range selection:

1. In Section 3 we provide constant-time access to any  $P_v$  using only  $O(n \lg \kappa)$  bits in the encoding model. This yields an  $O(\lg \kappa)$  time algorithm for range selection, as we can first find the node  $v$  in constant time, then binary search for  $l$  and  $r$  in  $P_v$ , then run the range selection query on  $A_v$  in time  $O(1 + \lg \kappa / \lg \lg n)$ , to finally return  $P_v[o]$  in  $O(1)$  time. This is obtained by a hierarchical marking of nodes plus a color-based encoding of the inheritance of points along cells in paths of unmarked nodes in  $T_C$ .
2. In Section 4 we address the bottleneck of the previous solution: we replace the binary search by fast predecessor queries on  $P_v$ , so as to obtain  $O(1 + \lg \kappa / \lg \lg n)$  time. This is obtained by storing *succinct string B-trees* (succinct SB-trees) [12] on some nodes, which enable a denser marking, and searches on the color information along (now shorter) paths of unmarked nodes, using global precomputed tables.
3. In Section 5 we wrap up the results in order to prove Theorem 1. Then we show how to answer top- $k$  queries by first finding the  $k$ th element in  $A_v$  and then using existing techniques [15] to collect all the values larger than the  $k$ th. This proves Theorem 2.

### 3 Constant-time Access to $P_v$

We describe a data structure that gives constant-time access to the values  $P_v[1..O(\kappa)]$  in any node  $v$ .

#### 3.1 Marking Nodes

Let  $s(v)$  be the number of descendants of  $v$  in  $T_C$ . We define a decreasing sequence of sizes as follows:  $t_0 = n'$  and  $t_{\ell+1} = \lceil \lg t_\ell \rceil$ , until reaching a  $z$  such that  $t_z = 1$ . Node  $v$  will be of level  $\ell$  if  $t_\ell^2 \leq s(v) < t_{\ell-1}^2$ . For any  $\ell \geq 1$ , we mark a node  $v \in T_C$  if it is of level  $\ell$  and:

- C1. it is a leaf or both its children are of level  $> \ell$ ; or
- C2. both its children are of level  $\ell$ ; or
- C3. it is the root or its parent is of level  $< \ell$ .

► **Lemma 3.** *The number of marked nodes of level  $\ell$  is  $O(n'/t_\ell^2)$ .*

**Proof.** The key property is that the descendants of  $v$  are of the same level of  $v$  or less. So nodes marked by C1 above cannot descend from each other, thus each such marked node has at least  $t_\ell^2$  descendants not shared with another. As  $T_C$  has at most  $2n'$  nodes, there

cannot be more than  $2n'/t_\ell^2$  nodes marked by this condition. By the same key property, nodes marked by C2 form a binary tree whose leaves are those marked by C1, thus there are at most other  $2n'/t_\ell^2$  nodes marked by C2. For C3, note that all unmarked nodes of level  $\ell$  are in disjoint paths (otherwise the parent of two nodes of level  $\ell$  would be marked by C2), and the path terminates in a node already marked by C1 or C2 (contrarily, a node of level  $\ell$  marked by C3 must be a child of a node of level  $< \ell$ , and thus cannot descend from nodes of level  $\ell$ , by the key property). Therefore, C3 marks the highest node of each such isolated path leading to a node marked by C1 or C2, and thus the number of nodes marked this way is limited by those marked by C1 or C2. ◀

### 3.2 Handling Marked Nodes

Marked nodes, across all the levels, are few enough to admit an essentially naive storage of the array  $P_v$ . If a marked node  $v$  represents a slab with left boundary  $x_l + 1$ , we store all its  $P_v[o]$  values as the integers  $P_v[o] - x_l$ . As explained, from  $v$  we can determine  $x_l$ , and thus obtain  $P_v[o]$  in constant time. Since a node of level  $\ell$  contains less than  $t_{\ell-1}^2$  descendants (leaves, in particular), its slab spans  $O(t_{\ell-1}^2)$  consecutive split points  $x_i$ , and thus  $O(\kappa t_{\ell-1}^2)$  positions in  $A$ . Thus, each such integer  $P_v[o] - x_l$  can be represented using  $\lg O(\kappa t_{\ell-1}^2) = O(t_\ell + \lg \kappa)$  bits. The second term adds up to  $O(\kappa \lg \kappa)$  bits per node and  $O(n \lg \kappa)$  overall. Since, by Lemma 3, there are  $O(n'/t_\ell^2)$  marked nodes of level  $\ell$ , the first term,  $O(t_\ell)$ , adds up to  $O((n'/t_\ell^2) \cdot (\kappa t_\ell)) = O(n/t_\ell)$  bits over all marked nodes of level  $\ell$ . Adding over all the levels  $\ell$  we have  $O(n) \sum_{\ell=0}^z 1/t_\ell$ . Since  $t_z = 1$  and  $t_{\ell-1} > 2^{t_\ell-1}$ , it holds  $t_{z-s} > 2^s$  for  $s \geq 4$ , and thus  $O(n) \sum_{\ell=0}^z 1/t_\ell \leq O(n)(O(1) + \sum_{s \geq 0} 1/2^s) = O(n)$  bits overall.

### 3.3 Handling Unmarked Nodes

While the problem of supporting constant-time access to  $P_v$  is solved for marked nodes,  $T_C$  may have  $\Theta(n')$  unmarked nodes. To deal with unmarked nodes, we first observe that an unmarked node  $v$  at level  $\ell$  has exactly one level  $\ell$  child and one child  $x$  at level  $> \ell$  (otherwise  $v$  would be marked by C2). Furthermore,  $x$  is marked by C3. Finally, the marked parent of an unmarked level  $\ell$  node must be the root or at level  $\ell$  itself. Thus, as already observed, level  $\ell$  unmarked nodes form disjoint paths in  $T_C$ , and all nodes adjacent to such a path are marked.

Now consider the points in slabs corresponding to unmarked nodes. When a cell is closed and split into two, the leftmost (rightmost)  $\kappa$  points in its slab become part of its left (right) child slab.

Thus, each child slab starts out with  $\kappa$  *inherited* points which are in common with its parent slab and  $\kappa$  further *original* points will be added to it before it is itself closed and split. For each point of node  $v$ , in  $x$ -coordinate order, we use a bit to specify if the point is inherited or original. Let  $o_v[1..2\kappa]$  be this bit-vector.

Let  $\pi$  be a path of unmarked nodes of level  $\ell$ , let  $u$  be the marked parent of the topmost unmarked node, and let  $v$  be an unmarked node in  $\pi$ . Each original point  $p$  of  $v$  must be an inherited point of some marked descendant  $v'$  that is adjacent to  $\pi$  (recall that  $v'$  represents all its points explicitly). Thus the coordinate of each such original point  $p$  can be specified by recording which marked descendant  $v'$  contains it, and the rank of  $p$  among the points of  $v'$ . Suppose that the  $j$ -th original point in  $v$  is in  $v'$ 's marked descendant at distance  $d_j$  along  $\pi$ . Then we write down the bit-string  $b_v = \mathbf{1}^{d_1-1} \mathbf{0} \mathbf{1}^{d_2-1} \mathbf{0} \dots \mathbf{1}^{d_\kappa-1} \mathbf{0}$ . We claim that, summed across all nodes  $v$  in the path  $\pi$ , this adds  $2|\pi|\kappa$  bits: there are  $|\pi|\kappa$   $\mathbf{0}$  bits, each  $\mathbf{1}$  bit represents an inherited point in a slab on the path  $\pi$ , and there are  $|\pi|\kappa$  inherited points

in  $\pi$ . Thus,  $\sum_{v \in T_C} |b_v| = O(n'\kappa) = O(n)$  bits. As explained, we also store  $O(\lg \kappa)$  bits for each original point in  $v$  telling which rank to pick in the marked node, in an array  $r_v$ . This adds  $O(n'\kappa \lg \kappa) = O(n \lg \kappa)$  bits, which completes the information necessary to identify any original point. Section 3.4 has the details of how to obtain the point value in  $O(1)$  time.

Unfortunately, we cannot apply the same approach to the inherited points in  $v$ , as we cannot bound the size of the bit-strings as we did for  $b_v$ . For any inherited point  $p$  in  $v$ , we instead specify which ancestor of  $v$  on  $\pi$  has  $p$  as an original point (we specify  $u$  if this ancestor is outside  $\pi$ ), and then retrieve the point as an original point in the ancestor. This is done by coding points using  $4\kappa$  colors. Of these colors,  $2\kappa$  are *original* colors and  $2\kappa$  are *inherited* colors. For each original color  $g$  there is a corresponding inherited color  $g'$ . All the points in  $u$  are given arbitrary distinct original colors. Then we traverse the nodes  $v$  in  $\pi$  top to bottom. If point  $p$  in  $v$  is inherited (from its parent  $v'$ ), we look at the color of  $p$  in  $v'$ . If  $p$  has an original color  $g$  in  $v'$ , we give  $p$  color  $g'$  in  $v$ . Otherwise, if  $p$  is also inherited in  $v'$ , having color  $g'$ , it will also have color  $g'$  in  $v$ . On the other hand, if point  $p$  is original in  $v$ , we give it one of the currently unused original colors. Note that no colors  $g$  and  $g'$  can be present simultaneously in any  $v'$ , thus writing  $g'$  in  $v$  unambiguously determines which color is inherited from  $v'$ . Then any other color  $g$  such that  $g'$  is not among the  $\kappa$  inherited colors of  $v$  can be used as an original color for  $v$ .

This scheme gives sufficient information to track the inheritance of points across  $\pi$ : when a new, original, point  $p$  appears in  $v$ , it is given an original color  $g$ . Then the point is inherited along the descendants of  $v$  as long as color  $g'$  exists below  $v$ . Thus, to find the appropriate ancestor of  $v$  that contains a given inherited point  $p$  of color  $g'$ , as an original point, we concatenate all the colors on  $\pi$  into a string, and ask for the nearest preceding occurrence of color  $g$ . The path can be encoded in  $O(|\pi|\kappa \lg \kappa)$  bits, which adds up to  $O(n \lg \kappa)$  bits overall. The position of  $g$  in the nearest ancestor also tells which of the original points does  $p$  correspond to.

### 3.4 Technicalities

Let us fix a representation for  $T_C$  using  $O(n')$  bits and supporting a large number of operations in constant time [16], in particular the preorder rank  $r(v)$  of any node  $v$ . We also use structures that support two operations on bit-vectors and sequences  $X$ :  $rank_a(X, i)$  is the number of occurrences of symbol  $a$  in  $X[1..i]$ , and  $select_a(X, j)$  is the position of the  $j$ th occurrence of letter  $a$  in  $X$ .

We store a bit-vector  $M[1..O(n')]$  in the same preorder of the nodes, where  $M[r(v)] = \mathbf{1}$  iff node  $v$  is marked. Further, we store a string  $S[1..O(n')]$  where we write down the level of each marked node, that is,  $S[rank_1(M, r(v))] = \ell$  iff  $v$  is marked and of level  $\ell$ . Operations  $rank$  and  $select$  on  $M$  can be supported in constant time and  $o(|M|)$  further bits [6, 14]. Since there are  $\lg^* n'$  distinct values of  $\ell$ , the alphabet of  $S$  is small and  $S$  can be represented within  $|S|H_0(S) + o(n')$  bits so that operations  $rank$  and  $select$  on  $S$  can be carried out in constant time [8]. Here  $H_0(S)$  is the *zeroth-order empirical entropy* of  $S$ , defined as  $|S|H_0(S) = \sum_{\ell} n_{\ell} \lg(|S|/n_{\ell})$ , where  $n_{\ell}$  is the number of occurrences of symbol  $\ell$  in  $S$ . Since  $n_{\ell} \lg(|S|/n_{\ell})$  is increasing<sup>2</sup> with  $n_{\ell}$  and  $n_{\ell} = O(n'/t_{\ell}^2)$  by Lemma 3, we have  $|S|H_0(S) = O(n') \sum_{\ell} \lg(t_{\ell}^2)/t_{\ell}^2 = O(n') \sum_{\ell} \lg(t_{\ell})/t_{\ell}^2 \leq O(n') \sum_{\ell} 1/t_{\ell} = O(n')$ .

With  $M$  and  $S$  we can create separate storage areas per level for the explicit  $P_v$  arrays of

<sup>2</sup> At least for  $n_{\ell} \leq |S|/e$ . When  $n_{\ell}$  is larger we can simply bound  $n_{\ell} \lg(|S|/n_{\ell}) = O(n_{\ell})$ , thus we can remove all those large  $n_{\ell}$  terms from the sum and add an extra  $O(n')$  term to absorb them all.



marked nodes, each of which uses the same space for nodes of the same level: if a node  $v$  is marked (i.e.,  $M[r(v)] = \mathbf{1}$ ) and is of level  $\ell = S[\text{rank}_1(M, r(v))]$ , then we store its array  $P_v$  as the  $r$ th one in a separate sequence for level  $\ell$ , where  $r = \text{rank}_\ell(S, \ell)$ .

Now consider unmarked nodes. The vectors  $o_v$ ,  $r_v$  and  $b_v$  are concatenated in the same preorder of the nodes. While vectors  $o_v$  and  $r_v$  are of fixed size, vectors  $b_v$  are not. Their starting positions are thus indicated with  $\mathbf{1}$ s in a second bit-vector  $B[1..O(n)]$ . Given any original point  $o_v[i] = \mathbf{1}$ , it is the  $j$ th original point for  $j = \text{rank}_1(o_v, i)$ ; recall that  $j$  is used to find  $d_j$  in  $b_v$ . Now  $b_v$  starts at position  $\text{select}_1(B, r(v))$  in the concatenation of all the  $b_v$ 's. Finally, we recover  $d_j$  as  $\text{select}_0(b_v, j) - \text{select}_0(b_v, j - 1)$ .

Now we have to find the marked node  $v'$  leaving  $\pi$  at distance  $d_j$  from  $v$ . The strategy is to find the node  $u'$  that is “at the end” of  $\pi$ . More precisely,  $u'$  is a child of the lowest node of  $\pi$  and is the only node leaving  $\pi$  that is of the same level  $\ell$  of  $v$ . Indeed,  $u'$  is the highest marked node of level  $\ell$  in the subtree of  $v$ . Since we can compute node depth and level ancestors in constant time [16], we can compute the ancestor  $a$  of  $u'$  that is at depth  $\text{depth}(v) + d_j - 1$ , and find  $v'$  as the child of  $a$  that is not in  $\pi$ , that is, is not an ancestor of  $u'$ .

Now, to find  $u'$ , we calculate the subtree size of  $v$  (in constant time [16]) and hence its level  $\ell$ .<sup>3</sup> If the nodes are arranged in preorder,  $u'$  is the first node appearing after  $r(v)$ ,  $r(u') > r(v)$ , which is marked  $M[r(u')] = \mathbf{1}$  and whose level is  $S[\text{rank}_1(M, r(u'))] = \ell$ . This corresponds to the first occurrence of  $\ell$  in  $S$  after position  $\text{rank}_1(M, r(v))$ . This is found in constant time with  $\text{rank}$  and  $\text{select}$  operations on  $S$ , and then  $r(u')$  is found with  $\text{select}$  on  $M$ . Finally, the tree representation gives us  $u'$  from its rank  $r(u')$  in constant time as well.

The sequence of colors  $c_\pi$  of path  $\pi$  is also associated with the last node  $u'$  of  $\pi$ , and all are concatenated in preorder of those nodes  $u'$ . As before, a bitmap is used to mark the starting position of each sequence  $c_\pi$ , and another bitmap is used to mark the preorders of the involved nodes  $u'$ .

Now let  $c_\pi$  be the sequence of  $2|\pi|\kappa$  colors for path  $\pi$ , writing from highest to lowest node the  $2\kappa$  colors of each node. The subarray corresponding to each  $v$  is easily found in  $c_\pi$  by knowing the depth of  $v$  and of  $u'$ . In order to find, given a position  $c_\pi[i] = g'$ , the largest  $i' < i$  such that  $c_\pi[i'] = g$ , we build a monotone minimum perfect hash function (MMPHF) [1] for each original color  $g$ , recording the set of positions where either  $g$  or  $g'$  occur in  $c_\pi$ . A MMPHF can be regarded as a support for the limited operation  $\text{rank}_{g,g'}(c_\pi, i)$  that counts the number of occurrences of  $g$  or  $g'$  in  $c_\pi[1..i]$ , provided  $c_\pi[i] \in \{g, g'\}$ . This is answered in constant time and using  $O(|\pi|\kappa \lg \lg \kappa)$  bits. In addition, for each  $g$  we store a bit-vector  $c_\pi^g$  so that  $c_\pi^g[\text{rank}_{g,g'}(c_\pi, i)] = \mathbf{1}$  iff  $c_\pi[i] = g$ . Then, after computing  $r = \text{rank}_{g,g'}(c_\pi, i)$ , we use  $\text{rank}$  and  $\text{select}$  on  $c_\pi^g$  to find the latest  $\mathbf{1}$  in  $c_\pi^g[1..r]$ . This corresponds to the last occurrence of  $g$  preceding  $c_\pi[i] = g'$ . The position is mapped back from  $c_\pi^g[o]$  to  $c_\pi$  using a sequence  $c'_\pi$  that identifies  $g'$  with  $g$ , so that the answer is  $\text{select}_g(c'_\pi, o)$ . We use a representation for  $c'_\pi$  that requires  $O(|\pi|\kappa \lg \kappa)$  bits and gives constant  $\text{select}$  time [10]. Thus the structures representing paths  $\pi$  use space  $O(|\pi|\kappa \lg \kappa)$ , which is independent of the path level  $\ell$ .

### Extending access from cells to extents

We have shown how to provide constant-time access to the points in a cell. In order to extend this to the extent of a node  $v$ , we use the technique of [11] to find in constant time the 3 cells that form the extent of  $v$ , and simulate the concatenation of the 3 arrays  $P$ .

<sup>3</sup> To find the level in constant time from the subtree size, we can check directly for the case  $\ell = 0$ , and store the other answers in a small table of  $\lg n'$  cells.

#### 4 Predecessor Queries on $P_v$

Having constant-time access to  $P_v$  enables binary searching for the desired limits of the array  $A_v$  where the selection query is to be run. However the binary search time becomes the bottleneck. In this section we obtain fast predecessor searches that replace the binary search.

A classical predecessor structure uses  $O(\kappa \lg n)$  bits, as the universe is the set of positions in  $A$ , and this adds up to  $O(n \lg n)$  bits (note that this structure is needed in all the  $O(n')$  nodes of  $T_C$ , not only the marked ones). A low-space predecessor structure when one has independent access to the sequence is the succinct SB-tree [12, Lem. 3.3]. For  $\kappa$  elements over a universe of size  $m$ , this structure supports predecessor queries in time  $O(1 + \lg \kappa / \lg \lg m)$  using  $O(\kappa \lg \lg m)$  bits, and a precomputed table of size  $o(m)$  that depends only on  $m$ .

On a node  $v$  of level  $\ell$ , the universe of positions is of size  $O(\kappa s(v)) = O(\kappa t_{\ell-1}^2)$ , thus the succinct SB-tree would use  $O(\kappa \lg \lg(\kappa t_{\ell-1})) = O(\kappa \lg t_{\ell} + \kappa \lg \lg \kappa)$  bits. The first term is still too large, as just considering the nodes with  $\ell = 1$  we add up to  $O(n \lg \lg n)$  bits.

To improve on this, we will use a marking that is denser than that used in Section 3 (this marking is only used for the predecessor structures). We will further mark every  $(t_{\ell} / \lg^2 t_{\ell})$ th node in the paths  $\pi$  of unmarked nodes of level  $\ell$ . All marked nodes will store a succinct SB-tree. The number of marked nodes of level  $\ell$  is now  $O(n' \lg^2 t_{\ell} / t_{\ell})$ , so storing a succinct SB-tree in a each marked node of level  $\ell$  adds up to  $O(n \lg^3 t_{\ell} / t_{\ell})$  bits. Adding up over all the levels  $\ell$  we have  $O(n) \sum_{\ell} \lg^3 t_{\ell} / t_{\ell} \leq O(n)(O(1) + \sum_{s \geq 0} s^3 / 2^s) = O(n)$  bits. The second term of the succinct SB-tree space,  $O(\kappa \lg \lg \kappa)$ , adds up to  $O(n \lg \lg \kappa)$  bits.

As a result, the paths of unmarked nodes of level  $\ell$  have length  $O(t_{\ell} / \lg^2 t_{\ell}) = O(t_{\ell})$ . Consider one such path. The nodes leaving the path are of level  $> \ell$ , except the node  $u'$  leaving  $\pi$  at the bottom, which is of level  $\ell$ . Therefore, we can divide the range of  $s(v)$  split points covered by  $v$  into three areas: (1) the area covered by the subtrees that leave  $\pi$  to the left, (2) the area covered by the subtrees that leave  $\pi$  to the right, and (3) the area covered by  $u'$ . Each of those areas is contiguous, (1) preceding (3) preceding (2). Since there are  $O(t_{\ell})$  nodes of type (1) and each is of level at least  $\ell + 1$ , the total area covered by those is of size  $O(t_{\ell} \cdot \kappa t_{\ell}^2) = O(\kappa t_{\ell}^3)$ . The case of (2) is analogous. Therefore, for the (unmarked) nodes on  $\pi$  we store a succinct SB-tree for the values in area (1) and another for the values in area (2), both using  $O(\kappa \lg \lg(\kappa t_{\ell}^3)) = O(\kappa \lg \lg(\kappa t_{\ell}))$  bits. Given a predecessor request, we first find the node  $u'$  below  $\pi$  as in Section 3, and determine in constant time whether the query falls in the area (1), (2), or (3) (by obtaining the limits  $[x_l + 1, x_r]$  of  $u'$ , as explained). If it falls in areas (1) or (2) we use the corresponding succinct SB-tree of  $v$ , otherwise we use the succinct SB-tree of  $u'$  (which is marked and hence stores a regular succinct SB-tree). We use the same techniques as in Section 3 to store and access the (variable-sized) representations of the succinct SB-trees.

With this twist, the space over a node of level  $\ell$  is  $O(\kappa \lg \lg(\kappa t_{\ell}))$  bits, adding up to at most  $O(n \lg \lg \lg n + n \lg \lg \kappa)$  bits, again dominated by the nodes of level  $\ell = 1$ . This gives a total space of  $O(n(\lg \kappa + \lg \lg \lg n))$  and a time of  $O(\lg \kappa / \lg \lg n)$ . Note that the time is improved from  $O(\lg \kappa / \lg \lg t_{\ell})$  to  $O(\lg \kappa / \lg \lg n)$  by using the same precomputed table over a universe of size  $n$  for all the nodes, and this table requires  $o(n)$  further bits. This result is already as desired if  $\lg \kappa = \Omega(\lg \lg \lg n)$ . In the sequel we address the case  $\kappa = O(\lg \lg n)$ .

##### 4.1 Handling Small $\kappa$ Values

When  $\kappa = O(\lg \lg n)$  we will not use the mechanism of storing succinct SB-trees for areas (1) and (2) of unmarked nodes as before, but a different mechanism. Let  $\pi$  be a path of unmarked nodes of level  $\ell$ . Let  $u_1, u_2, \dots$  be the nodes that leave  $\pi$  from the left, reading their areas in



left-to-right order (i.e., top-down in  $\pi$ ), and  $v_1, v_2, \dots$  be the nodes that leave  $\pi$  from the right, also reading them in left-to-right order (i.e., bottom-up in  $\pi$ ). Then the area of  $A$  covered by  $\pi$  can be partitioned into the  $|\pi|$  consecutive areas covered by  $u_1, u_2, \dots, u', v_1, v_2, \dots$ . All those nodes are marked and thus store their own succinct SB-tree.

Our problem is to determine, given a node  $v$  in  $\pi$ , which is the predecessor in  $P_v$  of a given position  $p$ . A first predecessor structure, associated with  $\pi$ , determines in which of those  $|\pi|$  areas  $p$  belongs (the node containing that area will descend from  $v$ ). Let  $\ell_i$  be the level of node  $u_i$ . Then the area covered by  $u_i$  is of length  $O(\kappa t_{\ell_i-1}^2)$ . Thus we can encode those lengths with, say,  $\gamma$ -codes [2], within  $O(\sum_i \lg(\kappa t_{\ell_i-1}^2)) = O(|\pi| \lg \kappa + \sum_i t_{\ell_i})$  bits.

From a space accounting point of view, this space can be afforded because we can charge  $O(\lg \kappa + t_{\ell_i})$  bits to the storage of  $u_i$ . As  $u_i$ 's level is larger than  $p$ , it is a marked node (see Section 3). Thus there are  $O(n'/t_{\ell_i}^2)$  such nodes overall, each of which will be charged  $O(t_{\ell_i})$  bits only once, from the path  $\pi$  it leaves, for a total of  $O(n'/t_{\ell_i})$  bits, adding up to  $O(n')$  bits overall. For the other term, note that we can always afford  $\lg \kappa$  bits of space per node.

On the other hand, we note that, since  $\ell_i > \ell$ , it holds  $O(|\pi| \lg \kappa + \sum_i t_{\ell_i}) = O(|\pi| \lg \kappa + |\pi| \lg t_\ell)$ . Since  $|\pi| = O(t_\ell / \lg^2 t_\ell)$ ,  $t_\ell = O(\lg n)$  even for  $\ell = 1$ , and  $\kappa = O(\lg \lg n)$ , the space is  $O(\lg n / \lg \lg n) = o(\lg n)$ , and thus the whole description of the  $u_i$  areas fits in a single computer word, and a global precomputed table of  $o(n)$  bits can be used to answer any predecessor query in constant time.

We proceed analogously with the areas of  $v_1, v_2, \dots$ . Now, a predecessor query for the areas  $u_1, u_2, \dots, u', v_1, v_2, \dots$  can be answered as before: We first determine whether the answer is  $u'$  with a constant number of comparisons, and if not, we use the global precomputed table with the description of the lengths of the areas of the  $u_i$  or the  $v_i$  nodes. This takes  $O(1)$  time. Once we know the area where the answer lies, we use the succinct SB-tree of the corresponding node  $v'$  (which we remind it is marked) to find the position of the predecessor in its  $P_{v'}$  array. Node  $v'$  is found by first computing its parent  $v''$  with level ancestor queries from  $u'$  (found as in Section 3) and then  $v'$  is the child of  $v''$  not in  $\pi$ .

Once we have that the predecessor of  $p$  in  $v'$  is  $P_{v'}[o']$ , the final challenge is to map that position in  $v'$  to the corresponding position in  $v$ . We will reuse the encoding of  $4\kappa$  colors described in Section 3. Note that, in the string of  $2|\pi|\kappa$  colors associated with the path  $\pi$ , we have sufficient information to determine which of the points in  $v$  are inherited in  $v'$ : if the color of the point is  $g$  or  $g'$ , we track  $g'$  downwards in  $\pi$  until it does not appear in some node  $v''$ , then the point is inherited in the sibling  $v'$  of  $v''$  not in  $\pi$ . Note that all the points of  $v$  that are inherited in  $v'$  are contiguous in  $P_v$ .

In addition to the color information  $c_v$ , we store associated with  $v$  a sequence of numbers  $n_v[1..2\kappa]$ , so that  $n_v[i]$  is the rank of the  $i$ th point of  $v$  among the points stored in  $v'$ , where  $v'$  is the first node leaving  $\pi$  that inherits the  $i$ th point of  $v$ . With the information of  $c_v$  and  $n_v$ , and given the predecessor of a point in  $P_{v'}$ , we have sufficient information to determine the predecessor of the point in  $P_v$ : only some of the points of  $P_{v'}$  are inherited from  $P_v$ .

The set of all  $c_v$  and  $n_v$  arrays in  $\pi$  add up to  $O(|\pi|\kappa \lg \kappa)$  bits, and since  $|\pi| = O(t_\ell / \lg^2 t_\ell)$ ,  $t_\ell = O(\lg n)$ , and  $\kappa = O(\lg \lg n)$ , this is  $O(\lg n \lg \lg \lg n / \lg \lg n) = o(\lg n)$ . Thus a global precomputed table of  $o(n)$  bits can precompute all the process of determining the predecessor in any  $v$  given that the answer is at any position in any descendant  $v'$ .

### Predecessors on extents

Once again,  $P_v$  refers to the extent of  $v$ , not only to its cell, whereas we support predecessors only on the points of the cell. With a couple of comparisons we determine whether the predecessor query must be run on the cell of  $v$  or on the cell of a neighboring node.

## 5 Wrapping Up

We can now describe a structure that, given a value  $\kappa$ , uses  $O(n \lg \kappa)$  bits and answers a query  $\text{select}(i, j, k)$  for any  $k \leq \kappa$  in time  $O(1 + \lg \kappa / \lg \lg n)$ , as follows:

1. We find the maximal interval  $[l, r]$  such that  $i \leq x_l + 1 \leq x_r \leq j$ , using *rank/select* on a bit-vector that marks the split points  $x_s$  [11].
2. If the interval is empty, then  $A[i..j]$  is contained in a leaf of  $T_C$ , which covers  $O(\kappa)$  consecutive values of  $A$ . Then the query can be directly run on plain range selection structures [4] associated with each leaf (these structures add up to  $O(n \lg \kappa)$  bits).
3. Otherwise, we find the highest node  $v \in T_C$  containing  $[x_l + 1, x_r]$ , as well as the other two neighbor nodes that span the extent of  $v$ , all in constant time [11].
4. Using the structures of Section 4, we find the predecessor  $P_v[r]$  of  $j$ , and the successor  $P_v[l]$  of  $i$  (the successor needs structures analogous to the predecessor), in time  $O(1 + \lg \kappa / \lg \lg n)$ .
5. We use the range selection structure [4] associated with  $P_v$  to run the query  $o = \text{select}(l, r, k)$ . The time is  $O(1 + \lg_w \kappa)$ .
6. We use the structures of Section 3 to compute the final answer  $P_v[o]$ , in  $O(1)$  time, adding to it the starting offset of node  $v$ .

In order to reduce the time to  $O(1 + \lg k / \lg \lg n)$ , we build our data structures for values  $\kappa_t = 2^{2^t}$ , for  $t = 0, 1, \dots, \tau$ , where  $\tau$  is such that  $2^{2^{\tau-1}} < \kappa \leq 2^{2^\tau}$ . The space for those structures is  $O(n) \sum_{t=0}^{\tau} \lg \kappa_t = O(n) \sum_{t=0}^{\tau} 2^t = O(n 2^\tau) = O(n \lg \kappa)$ . A query  $\text{select}(i, j, k)$  is run on the structure for  $\kappa_t$  such that  $\kappa_{t-1} < k \leq \kappa_t$ , that is,  $2^{t-1} < \lg k \leq 2^t$ ,<sup>4</sup> and thus its query time is  $O(1 + \lg \kappa_t / \lg \lg n) = O(1 + 2^t / \lg \lg n) = O(1 + \lg k / \lg \lg n)$ . This proves Theorem 1.

### Answering the query $\text{top}(i, j, k)$

We proceed as for query  $\text{select}(i, j, k)$  until we find the  $k$ th largest element in  $A_v[l..r]$ , let it be  $A_v[o]$ . Now we must find all the elements  $A_v[s]$  in  $A_v[l..r]$  where  $A_v[s] \geq A_v[o]$ . With an RMQ structure over  $A_v$  we can do this using Muthukrishnan's algorithm [15]: find the maximum in  $A_v[l..r]$ , let it be  $A_v[m_1]$ , then continue recursively with  $A_v[l..m_1 - 1]$  and  $A_v[m_1 + 1..r]$  stopping the recursion when the maximum found at  $A_v[m]$  satisfies  $A_v[m] < A_v[o]$ . Recall that  $A_v$  is a permutation on  $O(\kappa)$  symbols and thus we can afford storing it directly. Finally, when we have the positions  $m_1, \dots, m_k$  of the top- $k$  elements, we return  $P_v[m_1], \dots, P_v[m_k]$ . The overall time is  $O(\lg k / \lg \lg n + k) = O(k)$ . This proves Theorem 2.

Note that we deliver the top- $k$  elements in unsorted order. On the other hand, after  $O(1 + \lg k / \lg \lg n)$  time, each new result is delivered in  $O(1)$  time.

## 6 Conclusions

We have shown how to build an encoding data structure that uses asymptotically optimal space of  $O(n \lg \kappa)$  bits that answers  $\kappa$ -bounded rank range selection queries in time  $O(1 + \lg k / \lg \lg n)$ , and range top- $k$  queries in  $O(k)$  time for any  $k \leq \kappa$ . It would be interesting to obtain exactly optimal space (to within lower-order terms), but the precise lower bound is unknown even for  $k = 2$  [7]. It would also be interesting to obtain optimal time bounds for the general case  $w = \Omega(\lg n)$ .

<sup>4</sup> The search for the right  $t$  can be done in constant time by computing  $\lg \lg k$  and consulting a small precomputed table of  $\lg \lg K \leq \lg \lg n$  entries.

---

**References**

---

- 1 D. Belazzougui, P. Boldi, R. Pagh, and S. Vigna. Monotone minimal perfect hashing: searching a sorted table with  $o(1)$  accesses. In *Proc. 20th SODA*, pages 785–794, 2009.
- 2 T. Bell, J. Cleary, and I. Witten. *Text compression*. Prentice Hall, 1990.
- 3 G. S. Brodal, B. Gfeller, A. G. Jørgensen, and P. Sanders. Towards optimal range medians. *Theor. Comp. Sci.*, 412(24):2588–2601, 2011.
- 4 G. S. Brodal and A. G. Jørgensen. Data structures for range median queries. In *Proc. 20th ISAAC*, LNCS 5878, pages 822–831, 2009.
- 5 T. Chan and B. T. Wilkinson. Adaptive and approximate orthogonal range counting. In *Proc. 24th SODA*, pages 241–251, 2013.
- 6 D. Clark. *Compact Pat Trees*. PhD thesis, University of Waterloo, Canada, 1996.
- 7 P. Davoodi, G. Navarro, R. Raman, and S. Srinivasa Rao. Encoding range minima and range top-2 queries. *Philosophical Transactions of the Royal Society A*, 372:20130131, 2014.
- 8 P. Ferragina, G. Manzini, V. Mäkinen, and G. Navarro. Compressed representations of sequences and full-text indexes. *ACM Trans. Alg.*, 3(2):article 20, 2007.
- 9 J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comp.*, 40(2):465–492, 2011.
- 10 A. Golynski, I. Munro, and S. Rao. Rank/select operations on large alphabets: a tool for text indexing. In *Proc. 17th SODA*, pages 368–373, 2006.
- 11 R. Grossi, J. Iacono, G. Navarro, R. Raman, and S. Srinivasa Rao. Encodings for range selection and top- $k$  queries. In *Proc. 21st ESA*, LNCS 8125, pages 553–564, 2013.
- 12 R. Grossi, A. Orlandi, R. Raman, and S. Srinivasa Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *Proc. 26th STACS*, pages 517–528, 2009.
- 13 A. G. Jørgensen and K. G. Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proc. 22nd SODA*, pages 805–813, 2011.
- 14 I. Munro. Tables. In *Proc. 16th FSTTCS*, LNCS 1180, pages 37–42, 1996.
- 15 S. Muthukrishnan. Efficient algorithms for document retrieval problems. In *Proc 13th SODA*, pages 657–666, 2002.
- 16 K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. 21st SODA*, pages 134–149, 2010.
- 17 B. T. Wilkinson. Personal Communication, 2014.