Encodings for Range Majority Queries *

Gonzalo Navarro[1] and Sharma V. Thankachan[2]

[1] Dept. of Computer Science, Univ. of Chile. gnavarro@dcc.uchile.cl
[2] Cheriton School of Computer Science, Univ. of Waterloo. thanks@uwaterloo.ca

Abstract. We face the problem of designing a data structure that can report the majority within any range of an array $A[1,n]$, without storing A. We show that $\Omega(n)$ bits are necessary for such a data structure, and design a structure using $O(n \log^* n)$ bits that answers majority queries in $O(\log n)$ time. We extend our results to τ-majorities.

1 Introduction

Given an array $A[1,n]$ of n numbers or arbitrary elements, an *array range query* problem asks to build a data structure over A, such that whenever an interval $[i,j]$ with $1 \leq i \leq j \leq n$ comes as an input, we can efficiently answer queries on the elements in $A[i,j]$ [16]. Many array range queries arise naturally as subproblems of combinatorial problems, and are also of direct interest in data mining applications. Well-known examples are range maximum queries (RMQs, which seek the largest element in $A[i,j]$) [7] and top-k queries (which report the k largest elements in $A[i,j]$) [3].

An *encoding* for array range queries is a data structure that answers the queries without accessing A. This is useful when the values of A are not of interest themselves, and thus A may be deleted, potentially saving much space. It is also useful when array A does not fit in main memory, so it can be kept in secondary storage while a much smaller encoding can be maintained in main memory, speeding up queries. In this setting, instead of reporting an element in A, we only report a position in A where it occurs. Otherwise in many cases the encodings would be able to reconstruct A, and thus could not be small. As examples of encodings, RMQs can be solved in constant time using just $2n+o(n)$ bits [7], and top-k queries can be solved in $O(k)$ time using $O(n \log k)$ bits [10].

Frequency based array range queries, in particular variants of heavy-hitter-like problems, are very popular in data mining. Queries such as finding the most frequent element in a range (known as the range mode query) are known to be harder than problems like RMQs. For range mode queries, known data structures with constant query time require nearly quadratic space [14, 13]. The best known linear space solution requires $O(\sqrt{n/\log n})$ query time [4], and conditional lower bounds in that paper show that a significant improvement is highly unlikely.

* Funded in part by Millennium Nucleus Information and Coordination in Networks ICM/FIC P10-024F, Chile.

Still, efficient solutions exist for some useful variations of the range mode problem. An example are approximate range mode queries, where we are required to output an element whose number of occurrences in $A[i, j]$ is at least $1/(1+\epsilon)$ times the number of occurrences of the mode in $A[i, j]$ [9, 2].

In this paper we focus on a popular variation of range mode queries called *range majority queries*, which ask to report the range mode in $A[i, j]$ only if it occurs more than half of the times in $A[i, j]$. We also consider an extension where any element occurring a fraction larger than τ of the times in $A[i, j]$ can be reported. More formally, a majority is defined in the following way.

Definition 1. *A* majority *in an array $B[1, m]$, if it exists, is the element that occurs more than $m/2$ times in B. Given a real number $0 < \tau \leq 1/2$, a* τ-majority *in an array $B[1, m]$, if it exists, is any element that occurs more than τm times in B. Thus a majority is a τ-majority for $\tau = 1/2$.*

The problem we address in this paper can be stated as follows.

Definition 2. *Given an array $A[1, n]$, a* range majority query *gives an interval $[i, j]$ and must return whether $A[i, j]$ has a majority, and if it has, also return any position $i \leq k \leq j$ where the majority of $A[i, j]$ occurs. A* range τ-majority query *is defined analogously, returning any position of any τ-majority in $A[i, j]$.*

Range majority queries can be answered in constant time by maintaining a linear space (i.e., $O(n)$ words or $O(n \log n)$ bits) data structure [6]. Similarly, range τ-majority queries can be solved in time $O(1/\tau)$ and linear space if τ is fixed at construction time, or $O(n \log \log n)$ space (i.e., $O(n \log n \log \log n)$ bits) if τ is given at query time [1].

In this paper, we focus for the first time on *encodings for range majority (and τ-majority) queries*. In this scenario, a valid question is how much space is necessary for an encoding that correctly answers such queries (where we recall that A itself is not available at query time). We easily show in Section 2 that any such encoding needs $\Omega(n)$ bits, which reduces to $\Omega(\tau \log(1/\tau)n)$ bits for τ-majorities. Our main result is that it is possible to solve range majority queries within logarithmic time and almost linear-bit space. We achieve $O(n \log \log n)$ bits in Section 3, and our final result in Section 4:

Theorem 1. *There exists an encoding using $O(n \log^* n)$ bits answering range majority queries in time $O(\log n)$.*

In Section 5 we extend the results to τ-majorities, where the time and space obtained for range majority queries are divided by τ. Finally, in Section 6 we show how to build our structures in $O(n \log n)$ time.

Related work. Range τ-majority queries were introduced by Karpinski and Nekrich [11], who presented an $O(n/\tau)$-words structure with $O((1/\tau)(\log \log n)^2)$ query time. Durocher et al. [6] improved their space and query time to $O(n \log(1/\tau))$ and $O(1/\tau)$, respectively. The currently best result is by Belazzougui et al. [1],

where the space is $O(n)$ words and the query time is $O(1/\tau)$. All these results assume τ is fixed during the construction time. For the case where τ is also a part of the query input, a data structure of $O(n \log n)$ words was proposed by Chan et al. [5]. Very recently, Belazzougui et al. [1] brought down the space occupancy to $O(n \log \log n)$ words. The query time is $O(1/\tau)$ in all cases. All these solutions include a representation of A (sometimes aiming at compressing it [8, 1]), thus they are not encodings. As far as we know, ours is the first encoding for this problem.

2 Lower Bounds

We first derive a couple of simple lower bounds on the minimum size our encodings may have. First, $\Omega(n)$ bits are needed to answer majority queries.

Lemma 1. *Any encoding for range majority queries requires $\lfloor n/2 \rfloor$ bits, even for an array with 2 distinct symbols.*

Proof. We can encode any bitmap $C[1, n]$ using an encoding for range majorities on an array $A[1, 2n]$, hence establishing the result. Set $A[2k+1] = 0$ for all valid k values, and $A[2k] = C[k]$. For example, let $C[1, 3] = \langle \mathbf{0}\ \mathbf{1}\ \mathbf{1} \rangle$, then $A[1, 6] = \langle 0\ \mathbf{0}\ 0\ \mathbf{1}\ 0\ \mathbf{1} \rangle$. Then, if $C[k] = 0$ then $A[2k-1, 2k]$ has a majority, whereas if $C[k] = 1$ it does not. □

Second, we show that τ-majority queries require $\Omega(\tau \log(1/\tau)n)$ bits.

Lemma 2. *Any encoding for range τ-majority queries requires $n \lg\lceil 1/\tau \rceil / (1 + \lceil 1/\tau \rceil) > (\tau \lg(1/\tau)/2)n$ bits.*

Proof. Let $c = \lceil 1/\tau \rceil$. We can encode any array $C[1, n]$ over alphabet $[1, c]$ using an encoding for range majorities on an array $A[1, (c+1)n]$. In each bucket $A[(c+1)k+1, (c+1)(k+1)]$ we write the values $\langle 1, 2, 3, \ldots, c \rangle$, except that the value $C[k+1]$ is written twice. Therefore, $A[(c+1)k+1, (c+1)(k+1)]$ has only one τ-majority, precisely at offset $C[k+1]$ within the bucket. Therefore, the encoding for τ-majorities in $A[1, (c+1)n]$ requires at least $n \lg c$ bits, as any possible array C can be reconstructed from it. □

3 An $O(n \log \log n)$ Bits Encoding for Range Majorities

In this section we obtain an encoding using $O(n \log \log n)$ bits and solving majority queries in $O(\log n)$ time. In the next section we reduce the space.

Consider each distinct symbol x appearing in $A[1, n]$. Now consider the set of segments S_x within $[1, n]$ where x is a majority (this includes, in particular, all the segments $[k, k]$ where $A[k] = x$). Segments in S_x may overlap each other. For example, if $A[1, 7] = \langle 1\ 3\ 2\ 3\ 3\ 1\ 1 \rangle$, then

$$S_1 = \{[1, 1], [6, 6], [7, 7], [6, 7], [5, 7]\},$$
$$S_2 = \{[3, 3]\},$$
$$S_3 = \{[2, 2], [4, 4], [5, 5], [4, 5], [2, 4], [3, 5], [4, 6], [2, 5], [1, 5], [2, 6]\}.$$

Now let $A_x[1,n]$ be a bitmap such that $A_x[k] = 1$ iff position k belongs to some segment in S_x. In our example, $A_1 = \langle 1\ 0\ 0\ 0\ 1\ 1\ 1\rangle$, $A_2 = \langle 0\ 0\ 1\ 0\ 0\ 0\ 0\rangle$, and $A_3 = \langle 1\ 1\ 1\ 1\ 1\ 1\ 0\rangle$.

We recall operation $rank(B,i)$ in bitmaps $B[1,m]$, which returns the number of 1s in $B[1,i]$. Operation $rank$ can be implemented using $o(m)$ bits on top of B and in constant time [12].

We define a second bitmap related to x, M_x, so that if $A_x[k] = 1$, then $M_x[rank(A_x,k)] = 1$ iff $A[k] = x$. In our example, $M_1 = \langle 1\ 0\ 1\ 1\rangle$, $M_2 = \langle 1\rangle$, and $M_3 = \langle 0\ 1\ 0\ 1\ 1\ 0\rangle$. Then the following result is not difficult to prove.

Lemma 3. *An element x is a majority in $A[i,j]$ iff $A_x[k] = 1$ for all $i \le k \le j$, and 1 is a majority in $M_x[rank(A_x,i), rank(A_x,j)]$.*

Proof. If x is a majority in $A[i,j]$, then by definition $[i,j] \in S_x$, and therefore all the positions $k \in [i,j]$ are set to 1 in A_x. Therefore, the whole segment $A_x[i,j]$ is mapped bijectively to $M_x[rank(A_x,i), rank(A_x,j)]$, which is of the same length. Finally, the number of occurrences of x in $A[i,j]$ is the number of occurrences of 1 in $M_x[rank(A_x,i), rank(A_x,j)]$, which establishes the result.

Conversely, if $A_x[k] = 1$ for all $i \le k \le j$, then $A[i,j]$ is bijectively mapped to $M_x[rank(A_x,i), rank(A_x,j)]$, and the 1s in this range correspond one to one with occurrences of x in $A[i,j]$. Thus, if 1 is a majority in $M_x[rank(A_x,i), rank(A_x,j)]$, then x is a majority in $A[i,j]$. □

In our example, 1 is a majority in $A[5,7]$, and it holds $A_1[5,7] = \langle 1\ 1\ 1\rangle$ and $M_1[rank(A_1,5), rank(A_1,7)] = M_1[2,4] = \langle 0\ 1\ 1\rangle$, where 1 is a majority. Thus, with A_x and M_x we can determine whether x is a majority in any range.

Lemma 4. *It is sufficient to have rank-enabled bitmaps A_x and M_x to determine, in constant time, whether x is a majority in any $A[i,j]$.*

Proof. We use Lemma 3. We compute $i' = rank(A_x,i)$ and $j' = rank(A_x,j)$. If $j' - i' \neq j - i$, then $A_x[k] = 0$ for some $i \le k \le j$ and thus x is not a majority in $A[i,j]$. Otherwise, we find out whether 1 is a majority in $M_x[i',j']$, by checking whether $rank(M_x,j') - rank(M_x,i'-1) > (j'-i'+1)/2$. □

To find out any position $i \le k \le j$ where $A[k] = x$, we need operation $select(B,j)$, which gives the position of the jth 1 in a bitmap $B[1,m]$. This operation can also be solved in constant time with $o(m)$ bits on top of B [12]. Then, for example, if x is a majority in $A[i,j]$, its first occurrence in $A[i,j]$ is $i - i' + select(M_x, rank(M_x,i'-1)+1)$. With a similar formula we can retrieve any of the positions of x in $A[i,j]$.

We cannot afford to store all the bitmaps A_x and M_x for all x, however. The next lemma is the first step to reduce the total space to slightly superlinear.

Lemma 5. *Any position $A[k] = x$ induces at most five 1s in A_x.*

Proof. Consider a process where we start with $A[k] = \perp$ for all k, and set the values $A[k] = x$ for increasing positions k (left to right). Setting $A[k] = x$ induces

a segment $[k,k] \in S_x$, which may induce a new 1 in A_x. It might also induce some segments of the form $[i,k] \in S_x$, for $i < k$, depending on previous values. If x is a majority in $[i,k]$ with $A[k] = x$ and it was not a majority in $[i,k]$ with $A[k] = \perp$, then x occurs $\lfloor (k-i+1)/2 \rfloor$ times in $A[i,k-1]$. If $A[k-1] \neq x$, then x also occurs $\lfloor (k-i+1)/2 \rfloor > (k-i-1)/2$ times in $A[i,k-2]$, and thus it is a majority in $A[i,k-2]$. Thus all the range $A_x[i,k-2]$ was already 1s and setting $A[k] = x$ has only induced two new 1s, $A_x[k-1] = A_x[k] = 1$. If, on the other hand, $A[k-1] = x$, let l be the smallest value such that $[l,k-1] \in S_x$. Setting $A[k] = x$ will add new 1s to A_x only if $i < l$. By the definition of l, it must hold that $A[l-1] \neq x$ and $A[l-2] \neq x$, that x occurs $\lfloor (k-l+1)/2 \rfloor$ times in $A[l,k-1]$, and that $\lfloor (k-l+1)/2 \rfloor > (k-l)/2$. That is, $k-l$ must be odd and therefore $\lfloor (k-l+1)/2 \rfloor = (k-l+1)/2$. Now, this implies that x occurs $\lfloor (k-l+1)/2 \rfloor + 1 = (k-l+3)/2$ times in $A[l-1,k]$, so x is a majority in $A[l-1,k]$ and setting $A[k] = x$ could induce a new 1 in $A_x[l-1] = 1$. On the other hand, x is not a majority in $A[l-2,k]$. To be a majority in $A[i,k]$ with $i < l-2$, x has to be a majority in $A[i,l-3]$, and therefore only positions $A_x[l-2] = A_x[l-1] = 1$ could be new 1s induced by $A[k] = x$.

The consideration of the new induced segments of the form $[i,k+1] \in S_x$ is simpler, because we know that at this point $A[k+1] = \perp$. Therefore, if x is a majority in $A[i,k+1]$, it occurs more than $(k-i+2)/2$ times in $A[i,k+1]$, and thus it occurs more than $(k-i)/2$ times in $A[i,k-1]$, thus it is also a majority in $A[i,k-1]$. Therefore the only new 1 that can be added is $A_x[k+1] = 1$.

Finally, we consider the new induced segments of the form $[i,j] \in S_x$, with $i < k$ and $j > k+1$. We know that at this point $A[k+1,j] = \perp$. Therefore, if x is a majority in $A[i,j]$, it occurs more than $(j-i+1)/2$ times in $A[i,j]$, and thus it occurred more than $(j-i-1)/2$ times in $A[i,j]$ before setting $A[k] = x$. Thus x occurred more than $(j-i-1)/2$ times in $A[i,j-2]$ and thus it was already a majority in $A[i,j-2]$. Therefore the only new 1s that can be added by setting $A[k] = x$ are $A_x[j-1] = A_x[j] = 1$.

Overall, each new value $A[k] = x$ may induce up to five new 1s in A_x. □

The lemma shows that all the A_x bitmaps add up to $O(n)$ 1s, and the lengths of the M_x bitmaps adds up to $O(n)$ as well (recall that M_x has one position per 1 in A_x). Therefore, we can store all the M_x bitmaps within $O(n)$ bits of space. We cannot, however, store all the A_x bitmaps, as they may add up to $O(n^2)$ 0s (note there can be $O(n)$ distinct symbols x).

Instead, we will *coalesce* different bitmaps A_x into one, as long as their areas of contiguous 1s do not overlap or touch (that is, there must be at least one 0 between any two areas of 1s of two coalesced bitmaps). The bitmaps M_x are merged accordingly, in the same order of the areas. In our example, we can coalesce A_1 and A_2 into $A_{12} = \langle 1\ 0\ 1\ 0\ 1\ 1\ 1 \rangle$, with the corresponding $M_{12} = \langle 1\ 1\ 0\ 1\ 1 \rangle$.

Then, at query time, we check for the area $[i,j]$ of each coalesced bitmap using Lemma 4. We cannot confuse the areas of different symbols x because we force that there is at least one 0 between any two areas. If we find one majority in one coalesced bitmap, we know that there is a majority and can spot all of

its occurrences (or one, as the problem is defined), even if we cannot tell which particular symbol x is the majority.

This scheme will work well if we obtain just a few coalesced bitmaps overall. Next we show how to obtain only $O(\log n)$ coalesced bitmaps.

Lemma 6. *At most* $2 \lg n$ *distinct values of* x *can have* $A_x[k] = 1$ *for a given* k.

Proof. First, $A[k] = x$ is a majority in $A[k, k]$, thus $A_x[k] = 1$. Now consider any other element $x' \neq x$ such that $A_{x'}[k] = 1$. This means that x' is a majority in some $[i, j]$ that contains k. Since $A[k] \neq x'$, it must be that x' is a majority in $[i, k]$ or in $[k, j]$ (or in both). We say x' is a left-majority in the first case and a right-majority in the second. Let us call $y_1, y_2, \ldots$ the x' values that are left-majorities, and $i_1, i_2, \ldots$ the left endpoints of their segments (if they are majorities in several segments covering k, we choose one arbitrarily). Similarly, let $z_1, z_2, \ldots$ be the x' values that are right-majorities, and $j_1, j_2, \ldots$ the right endpoints of their segments. Assume the left-majorities are sorted by decreasing values of i_r and the right-majorities are sorted by increasing values of j_r. If a same value x' appears in both lists, we arbitrarily remove one of them. As an exception, we will start both lists with $y_0 = z_0 = x$, with $i_0 = j_0 = k$.

It is easy to see by induction that y_r must appear at least 2^r times in the interval $[i_r, k]$. This clearly holds for $y_0 = x$. Now, by the inductive hypothesis, values $y_0, y_1, \ldots, y_{r-1}$ appear at least $2^0, 2^1, \ldots, 2^{r-1}$ times within $[i_{r-1}, k]$ (which contains all the intervals), adding up to $2^r - 1$ occurrences. In order to be a left-majority, element y_r must appear at least 2^r times in $[i_r, k]$, to outweight all the $2^r - 1$ occurrences of the previous symbols. The case of right-majorities is analogous. This shows that there cannot be more than $\lg n$ left-majorities and $\lg n$ right-majorities. □

In the following it will be useful to define C_x as the set of maximal contiguous areas of 1s in A_x. That is, C_x is obtained by merging all the segments of S_x that touch or overlap. In our example, $C_1 = \{[1, 1], [5, 7]\}$, $C_2 = \{[3, 3]\}$, and $C_3 = \{[1, 6]\}$. Note that segments of C_x do not overlap, unlike those of S_x. Since a segment of C_x covers a position k iff some segment of S_x covers position k (and iff $A_x[k] = 1$), it follows by Lemma 6 that any position is covered by at most $2 \lg n$ segments of C_x of distinct symbols x. Clearly, a pair of consecutive positions is covered by at most $4 \lg n$ such segments (this is a crude upper bound).

We obtain $O(\log n)$ coalesced bitmaps as follows. We take the union of all the sets C_x of all the symbols x and sort the segments by their starting points. Then we start filling coalesced bitmaps. We check if the current segment can be added to an existing bitmap without producing overlaps (and leaving a 0 in between). If we can, we choose any appropriate bitmap, otherwise we start a new bitmap. If at some point we need more than $4 \lg n$ bitmaps, it is because all the last segments of the current $4 \lg n$ bitmaps overlap the starting point of the current segment or the previous position, a contradiction.

In our example, we take $C_1 \cup C_2 \cup C_3 = \{[1, 1], [1, 6], [3, 3], [5, 7]\}$, and the process produces precisely the coalesced bitmaps A_{12}, corresponding to the set $\{[1, 1], [3, 3], [5, 7]\}$ and A_3, corresponding to $\{[1, 6]\}$. Note that in general the

coalesced bitmaps may not correspond to the union of complete original bitmaps A_x, but areas of a bitmap A_x may end up in different coalesced bitmaps.

Therefore, the coalescing process produces $O(\log n)$ bitmaps. Consequently, we obtain $O(\log n)$ query time by simply checking the coalesced bitmaps one by one using Lemma 4.

Finally, representing the $O(\log n)$ coalesced bitvectors, which contain $O(n)$ 1s and have total length $O(n \log n)$, requires $O(n \log \log n)$ bits if we use a compressed bitmap representation [15] that still offers constant-time *rank* and *select* queries. This concludes the first part of our result.

4 An $O(n \log^* n)$ Bits Encoding for Range Majorities

We introduce a different representation of the coalesced bitmaps that allows us storing them in $O(n \log^* n)$ bits, while retaining all the mechanism and query time complexity. We will distinguish segments of C_x by their lengths, separating lengths by ranges between 2^ℓ and $2^{\ell+1} - 1$, for any ℓ. In the process of creating the coalesced bitmaps described in the previous section, we will have separate coalesced bitmaps for inserting segments within each range of lengths; these will be called bitmaps of level ℓ. There may be several bitmaps of the same level. It is important that, even with this restriction, our coalescing process will still generate $O(\log n)$ bitmaps, because only $O(1)$ coalesced bitmaps of each level ℓ will be generated.

Lemma 7. *There can be at most 8 segments of any C_x, of length between 2^ℓ and $2^{\ell+1} - 1$, covering a given position k, for any ℓ.*

Proof. Any such segment must be contained in the area $A[k - 2^{\ell+1}, k + 2^{\ell+1}]$, and if x is a majority in it, it must appear more than $2^{\ell-1}$ times. There can be only 8 different values of x appearing $2^{\ell-1}$ times in an area of length $2^{\ell+2}$. □

To represent any coalesced bitmap $B[1, n]$, we cut the universe $[1, n]$ into chunks of length $b = \lg n$. We store a string K of length $n/\lg n$, where for each position a 0 indicates that the chunk is all 0s, a 1 that the chunk is all 1s, and a 2 indicates that there are 0s and 1s in the chunk. We store explicitly only the chunks with value 2, concatenated one after the other. Let B_1 be a bitmap such that $B_1[k] = 1$ iff $K[k] = 1$, B_2 such that $B_2[k] = 1$ iff $K[k] = 2$, and C the bitmap where the explicit chunks are concatenated. Then it holds

$$\begin{aligned} rank(B, i) \;=\; & b \cdot rank(B_1, \lfloor (i-1)/b \rfloor) + \\ & rank(C, b \cdot rank(B_2, \lfloor i/b \rfloor) + [\textbf{if } B_2[1 + \lfloor i/b \rfloor] = 1 \textbf{ then } i \bmod b \textbf{ else } 0]), \end{aligned}$$

which takes constant time. Operation $select(B, j)$ can be done by binary search on *rank*, which takes $O(\log n)$ time but has to be done once per query, hence retaining the $O(\log n)$ query time. Note that K is not explicitly stored, but it is represented with B_1 and B_2.

In our example, we would have three coalesced bitmaps: $B^0 = \langle 1\ 0\ 1\ 0\ 0\ 0\ 0 \rangle$, of level $\ell = 0$, for the segments $[1, 1]$ and $[3, 3]$; $B^1 = \langle 0\ 0\ 0\ 0\ 1\ 1\ 1 \rangle$, of level

$\ell = 1$, for the segment $[5, 7]$; and $B^2 = \langle 1\ 1\ 1\ 1\ 1\ 1\ 0 \rangle$, of level $\ell = 2$, for the segment $[1, 6]$. Assume $b = 2$. Then, for B^0 we would have $K^0 = \langle 2\ 2\ 0\ 0 \rangle$, $B_1^0 = \langle 0\ 0\ 0\ 0 \rangle$, $B_2^0 = \langle 1\ 1\ 0\ 0 \rangle$, and $C^0 = \langle 1\ 0\ 1\ 0 \rangle$. For B^1 we would have $K^1 = \langle 0\ 0\ 1\ 1 \rangle$, $B_1^1 = \langle 0\ 0\ 1\ 1 \rangle$, $B_2^1 = \langle 0\ 0\ 0\ 0 \rangle$, and $C^1 = \langle\rangle$. Finally, for B^2 we would have $K^2 = \langle 1\ 1\ 1\ 0 \rangle$, $B_1^2 = \langle 1\ 1\ 1\ 0 \rangle$, $B_2^2 = \langle 0\ 0\ 0\ 0 \rangle$, and $C^2 = \langle\rangle$.

Consider a fixed bitmap B of some level ℓ, which has been formed by adding n' segments. We store at most $2n' \lg n$ bits in the explicit chunks of C, as there are only n' transitions from 0 to 1 and n' from 1 to 0 in B. For any level $\ell \geq \lg \lg n$, there are at least $n' \lg n$ 1s, because the segments have length at least $2^\ell \geq \lg n$. Therefore, in those levels, the number of bits stored in C bitmaps is of the same order of the total number of 1s in the corresponding bitmaps B. Thus we store only $O(n)$ bits over all the chunks of all coalesced bitmaps of levels $\ell \geq \lg \lg n$. As for the sequences B_1 and B_2 describing the chunks, they are of length $n/\lg n$, so they add up to $O(n)$ bits over all the possible $O(\log n)$ levels.

Now, for the levels up to $\lg \lg n$, we use chunk size $b = \lg \lg n$, storing a sequence of length $n/\lg \lg n$. The explicitly stored chunks C add up to $n' \lg \lg n$ bits, and for any level $\ell \geq \lg \lg \lg n$, the total number of 1s is over $n' \lg \lg n$, thus the total number of stored bits is of the same order of the 1s. The sequences B_1 and B_2 describing the chunks add up to $O(n)$, because there are only $O(\log \log n)$ levels where this is applied.

We continue with the remaining (lowest) $\lg \lg \lg n$ levels, and so on. Then the total number of stored bits is $O(n \log^* n)$, dominated by the sequences B_1 and B_2. This proves Theorem 1.

5 Extension to τ-Majorities

We first consider the case where τ is fixed at the time the data structure is built, and then move on to the case of τ given at query time. For lack of space we only sketch the results, which follow relatively easily from our results on majorities. First, Lemmas 3 and 4 hold verbatim if we define S_x as the segments where x is a τ-majority. Lemma 5 can be extended to this case, so that any position $A[k] = x$ induces $O(1/\tau)$ 1s in A_x. As a consequence, there are $O(n/\tau)$ 1s in all the A_x bitmaps. Lemma 6 can also be extended, so that $O(\log_{1/(1-\tau)} n) = O((1/\tau) \log n)$ distinct values of x can have $A_x[k] = 1$ for a given k. Therefore, the coalescing process produces $O((1/\tau) \log n)$ bitmaps, and this is the query time. Lemma 7 can be extended similarly, so that there can be only $O(1/\tau)$ coalesced bitmaps of any given level, and there are $\lg n$ levels. Thus the mechanism of Section 4 can be applied verbatim, so that the total number of bits used is $O((n/\tau) \log^* n)$. Therefore we obtain the following result.

Theorem 2. *For a fixed threshold $0 < \tau \leq 1/2$, there exists an encoding using $O((n/\tau) \log^* n)$ bits answering range τ-majority queries in time $O((1/\tau) \log n)$.*

In order to allow τ to be specified at query time, we build the encoding of Theorem 2 for values $\tau = 1/2, 1/4, 1/8, \ldots, 1/2^{\lceil \lg 1/\mu \rceil}$, where μ is the minimum τ value to support. Then, given a τ-majority query, we run the query on the

structure built for $\tau' = 1/2^{\lceil \lg 1/\tau \rceil}$. Note that $\tau/2 < \tau' \leq \tau$, therefore the query time is $O((1/\tau')\log n) = O((1/\tau)\log n)$. For each possible answer to the τ'-majority query, we use *rank* on the coalesced M_x bitmaps to find out whether the answer is actually a τ-majority. This verification does not change the worst-case time complexity. As for the space, the factor multiplying $O(n \log^* n)$ is $2 + 4 + 8 + \ldots + 2^{\lceil \lg 1/\mu \rceil} = O(1/\mu)$. Therefore we obtain the following result.

Theorem 3. *For a fixed threshold $0 < \mu \leq 1/2$, there exists an encoding using $O((n/\mu)\log^* n)$ bits answering range τ-majority queries, for any $\mu \leq \tau \leq 1/2$ given at query time, in $O((1/\tau)\log n)$ time.*

6 Construction

The most complex part of the construction of our encoding is to build the sets C_x; once these are built, the construction of the structure of Section 4 can be easily carried out in $O(n \log^* n)$ additional time.

We separate the set of increasing positions P_x where x appears in A, for each x. The P_x sets are easily built in $O(n \log n)$ time. Now we build C_x from each P_x using a divide and conquer approach, in $O(|P_x| \log |P_x|)$ time, for a total construction time of $O(n \log n)$.

We pick the middle element $k \in P_x$ and compute in linear time the segment $[l, r] \in C_x$ that contains k. To compute l, we find the leftmost element $p_l \in P_x$ such that x is a majority in $[p_l, k_r]$, for some $k_r \in P_x$ with $k_r \geq k$.

To find p_l, we note that it must hold $(w(p_l, k-1) + w(k, k_r))/(k_r - p_l + 1) > 1/2$, where $w(i, j)$ is the number of occurrences of x in $A[i, j]$. The condition is equivalent to $2w(p_l, k-1) + p_l - 1 > k_r - 2w(k, k_r)$. Thus we compute in linear time the minimum value v of $k_r - 2w(k, k_r)$ over all those $k_r \in P_x$ to the right of k, and then traverse all those $p_l \in P_x$ to the left of k, left to right, to find the first one that satisfies $2w(p_l, k-1) + p_l + 1 > v$, also in linear time. Once we find the proper p_l and its corresponding k_r, the starting position of the segment is slightly adjusted to the left of p_l, to be the smallest value that satisfies $w(p_l, k_r)/(k_r - l + 1) > 1/2$, that is, l satisfies $l > -2w(p_l, k_r) + k_r + 1$, that is, $l = k_r - 2w(p_l, k_r) + 2$.

Once p_r and then r are computed analogously, we insert $[l, r]$ into C_x and continue recursively with the elements of P_x to the left of p_l and to the right of p_r. Upon return, it might be necessary to join $[l, r]$ with the rightmost segment of the left part and/or with the leftmost segment of the right part, in constant time. The total construction time is $T(n) = O(n) + 2T(n/2) = O(n \log n)$. The construction for τ-majorities is similar, although for τ given at query time we must build $O(\log(1/\mu))$ similar structures.

7 Final Remarks

We have obtained the first result about encodings for answering range majority queries, that is, data structures that use less space than the data and do not need

to access it. We have proved that $\Omega(n)$ bits are necessary for any such encoding, and have presented a particular encoding that uses $O(n \log^* n)$ bits and $O(\log n)$ time. It can be built in $O(n \log n)$ time. An open question is whether it is possible to reach $O(n)$ bits of space and/or constant query time.

We have also extended our result to range τ-majorities, where we have proved a lower bound of $O(\tau \log(1/\tau)n)$ bits and presented an encoding using $O((n/\tau) \log^* n)$ bits and $O((1/\tau) \log n)$ query time. An intriguing aspect of this result is that our lower bound suggests that τ-majorities require less space for smaller τ, whereas our upper bound uses more space (and time) for smaller τ, in line with previous work on data structures that are not encodings. It is an interesting problem to determine which is the case.

References

1. D. Belazzougui, T. Gagie, and G. Navarro. Better space bounds for parameterized range majority and minority. In *WADS*, pages 121–132, 2013.
2. P. Bose, E. Kranakis, P. Morin, and Y. Tang. Approximate range mode and range median queries. In *STACS*, pages 377–388, 2005.
3. G. Brodal, R. Fagerberg, M. Greve, and A. López-Ortiz. Online sorted range reporting. In *ISAAC*, pages 173–182, 2009.
4. T. Chan, S. Durocher, K. Larsen, J. Morrison, and B. Wilkinson. Linear-space data structures for range mode query in arrays. In *STACS*, pages 290–301, 2012.
5. T. Chan, S. Durocher, M. Skala, and B. Wilkinson. Linear-space data structures for range minority query in arrays. In *SWAT*, pages 295–306, 2012.
6. S. Durocher, M. He, I. Munro, P. Nicholson, and M. Skala. Range majority in constant time and linear space. *Inf. Comput.*, 222:169–179, 2013.
7. J. Fischer and V. Heun. Space-efficient preprocessing schemes for range minimum queries on static arrays. *SIAM J. Comput.*, 40(2):465–492, 2011.
8. T. Gagie, M. He, I. Munro, and P. Nicholson. Finding frequent elements in compressed 2d arrays and strings. In *SPIRE*, pages 295–300, 2011.
9. M. Greve, A. Jørgensen, K. D. Larsen, and J. Truelsen. Cell probe lower bounds and approximations for range mode. In *ICALP*, pages 605–616, 2010.
10. R. Grossi, J. Iacono, G. Navarro, R. Raman, and S. Rao Satti. Encodings for range selection and top-k queries. In *ESA*, pages 553–564, 2013.
11. M. Karpinski and Y. Nekrich. Searching for frequent colors in rectangles. In *CCCG*, 2008.
12. I. Munro. Tables. In *FSTTCS*, pages 37–42, 1996.
13. H. Petersen. Improved bounds for range mode and range median queries. In *SOFSEM*, pages 418–423, 2008.
14. H. Petersen and S. Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Inf. Process. Lett.*, 109(4):225–228, 2009.
15. R. Raman, V. Raman, and S. Srinivasa Rao. Succinct indexable dictionaries with applications to encoding k-ary trees, prefix sums and multisets. *ACM Trans. Alg.*, 3(4):article 43, 2007.
16. M. Skala. Array range queries. In *Space-Efficient Data Structures, Streams, and Algorithms*, pages 333–350, 2013.