Linear-Space Data Structures for Range Minority Query in Arrays[⋆]

Timothy M. Chan[1], Stephane Durocher[2], Matthew Skala[2], and Bryan T. Wilkinson[1]

[1] Cheriton School of Computer Science, University of Waterloo, Waterloo, Canada, {tmchan,b3wilkin}@uwaterloo.ca

[2] Department of Computer Science, University of Manitoba, Winnipeg, Canada, {durocher,skala}@cs.umanitoba.ca

Abstract. We consider range queries in arrays that search for low-frequency elements: least frequent elements and α-minorities. An α-minority of a query range has multiplicity no greater than an α fraction of the elements in the range. Our data structure for the least frequent element range query problem requires $O(n)$ space, $O(n^{3/2})$ preprocessing time, and $O(\sqrt{n})$ query time. A reduction from boolean matrix multiplication to this problem shows the hardness of simultaneous improvements in both preprocessing time and query time. Our data structure for the α-minority range query problem requires $O(n)$ space, supports queries in $O(1/\alpha)$ time, and allows α to be specified at query time.

1 Introduction

The *frequency* of an element x in a multiset stored as an array $A[0 : n-1]$, denoted $\mathrm{freq}_A(x)$, is the number of occurrences (i.e., the multiplicity) of x in A. Given $\alpha \in [0,1]$, an element x is an *α-minority* in A if $1 \leq \mathrm{freq}_A(x) \leq \alpha n$, whereas x is an *α-majority* if $\mathrm{freq}_A(x) > \alpha n$.

We examine two problems which involve preprocessing a given array A to construct a data structure that can efficiently find low-frequency elements in query ranges. A least frequent element range query specifies a pair of indices (i, j) and returns a least frequent element that occurs in $A[i : j]$. An α-minority range query specifies some $\alpha \in [0, 1]$ and a pair of indices (i, j), and returns an element whose frequency in $A[i : j]$ is at least 1 and at most $\alpha|j - i + 1|$. If no such element exists, the query must not return any element. Whenever we discuss a data structure with a parameter β instead of α, β is fixed before preprocessing. We do so to differentiate from the more challenging case in which different parameter values can be specified at query time.

Several recent results examine the minimum, selection (including median), mode (i.e., the most frequent element), β-majority, and α-majority range query problems on arrays (e.g., [1–3, 5, 7–14, 16–18]). Most relevant to our low-frequency

⋆ Work supported in part by the Natural Sciences and Engineering Research Council of Canada (NSERC).

query problems are results for their high-frequency analogues: an $O(n)$-space data structure that supports range mode queries in $O(\sqrt{n/\log n})$ time [5], an $O(n\log(1/\beta+1))$-space data structure that supports β-majority range queries in $O(1/\beta)$ time [9], and a $O(n\log n)$-space data structure that supports α-majority range queries in $O(1/\alpha)$ time [11]. Related generalizations include examinations of the the β-majority range query problem in the dynamic setting [10] and the α-majority range query problem in two dimensions [11]. Greve et al. [13] give a lower bound of $\Omega(\log n/\log(s\cdot w/n))$ on the range mode query time for any data structure that uses s memory cells of w bits in the cell probe model; they show the same bound applies to the problem of determining whether any element in a given query range has frequency exactly k, for any k given at query time. Consequently, no $O(n)$-space data structure can support constant-time (independent of α) α-minority queries.

Our low-frequency query problems have significant differences when compared to their high-frequency analogues. For example, for any (i,j), the frequencies of respective modes of $A[i:j]$ and $A[i:j+1]$ differ by either zero or one. The frequency of the mode of a set increases monotonically with the addition of new elements into the set. Conversely, the frequencies of respective least frequent elements of $A[i:j]$ and $A[i:j+1]$ can differ by any value in $\{i-j,\ldots,0,1\}$. Similarly, if x is a mode of $A[i:k]$ and $A[k+1:j]$, then x is a mode of $A[i:j]$, whereas the analogous property does not hold for least frequent elements.

In Section 2 we consider the least frequent element range query problem. We describe an $O(n)$-space data structure that identifies a least frequent element in a query range in $O(\sqrt{n})$ time. This data structure is a variant of a previous data structure of Chan et al. [5] for the range mode problem (which in turn was an improvement of a previous data structure of Krizanc et al. [16]). In addition, using an argument similar to that of Chan et al. [5], we present a reduction from boolean matrix multiplication to the least frequent element range query problem, showing the hardness of simultaneously improving our preprocessing and query time bounds.

Section 3 contains the main result of this paper: an $O(n)$-space data structure that supports α-minority range queries in $O(1/\alpha)$ time. Our technique is quite different from the previous techniques of Durocher et al. [9] for β-majority range queries and Gagie et al. [11] for α-majority range queries, which have worse space bounds ($O(n\log(1/\beta+1)$ and $O(n\log n)$, respectively).

In Section 4 we apply a variation of our technique to give an $O(n\log n)$-space data structure that supports α-majority range queries in $O(1/\alpha)$ time. These space and time bounds match those achieved by a recent α-majority data structure of Gagie et al. [11].

Both our data structures in Sections 3 and 4 make interesting use of existing tools from computational geometry. Notably, we apply Chazelle's *hive graphs* [6], which were designed for a seemingly unrelated two-dimensional searching problem: preprocess a set of horizontal line segments so that we can report segments intersecting a given vertical line segment or ray.

2 Finding a Least Frequent Element

2.1 $O(\sqrt{n})$-Time Data Structure

In this section we present an $O(n)$-space data structure that identifies a least frequent element in a query range in $O(\sqrt{n})$ time and requires $O(n^{3/2})$ preprocessing time. Specifically, we will prove the following theorem that implies the above result when $s = \sqrt{n}$:

Theorem 1. *Given an array $A[0 : n-1]$ and any fixed $s \in [1, n]$, there exists an $O(n + s^2)$-space data structure that supports least frequent range query on A in $O(n/s)$ time and requires $O(ns)$ preprocessing time.*

Preprocessing. Given an arbitrary input array $A[0 : n-1]$, we begin by building an array $B[0 : n-1]$ such that $B[x]$ is the rank of $A[x]$ amongst the distinct elements of A. We find the ranks of all the elements by sorting A. Thus, all elements in B are in the range $\{0, \ldots, \Delta - 1\}$, where Δ denotes the number of distinct elements in A. Furthermore, $B[x]$ is a least frequent element in $B[i : j]$ if and only if $A[x]$ is a least frequent element in $A[i : j]$, for any i, j, and x. Following Krizanc et al. [16] and Chan et al. [5], for each $x \in \{0, \ldots, \Delta - 1\}$, we define an array Q_x such that $Q_x[k]$ stores the index of the kth instance of x in B. Since each element in B is represented exactly once in $Q_0, \ldots, Q_{\Delta-1}$, the total space required by $Q_0, \ldots, Q_{\Delta-1}$ is $\Theta(n)$. We also define a rank array $B'[0 : n-1]$ such that for all b, $B'[b]$ denotes the rank (i.e., the index) of b in $Q_{B[b]}$. Therefore, $Q_{B[b]}[B'[b]] = b$. Using these arrays, Chan et al. observe the following lemma (which follows by comparing $Q_{B[i]}[B'[i] + q - 1]$ with j):

Lemma 1 (Chan et al. [5, Lemma 3]). *Given an array $B[0 : n-1]$, there exists an $O(n)$-space data structure that determines in $O(1)$ time for any $0 \leq i \leq j \leq n-1$ and any q whether $B[i : j]$ contains at least q instances of element $B[i]$.*

We also require the following lemma:

Lemma 2. *Given an array $B[0 : n-1]$, there exists an $O(n)$-space data structure that computes in $O(j - i + 1)$ time for any $0 \leq i \leq j \leq n-1$ the frequencies of all elements in $B[i : j]$. In particular, a least frequent element in $B[i : j]$ and its frequency can be computed in $O(j - i + 1)$ time.*

Proof. No actual preprocessing is necessary other than initializing an array $C[0 : \Delta - 1]$ to zero. The query algorithm is similar to counting sort: compute a frequency table for $B[i : j]$ stored in C (i.e., for every x, $C[x]$ corresponds to the frequency of x in $B[i : j]$), then find a minimum element in C. The time required to find the minimum is bounded by $O(j - i + 1)$ by comparing all frequencies $C[x]$ were x corresponds to an element in $B[i : j]$ (these are exactly the elements of C that have non-zero values). This procedure is repeated after identifying the minimum to reset C to zero. $\square$

We divide the array B into s blocks of size $t = \lceil n/s \rceil$. A query range $B[i:j]$ spans between 0 and s complete blocks. Let the *span* of $B[i:j]$ be the sequence of complete blocks contained within $B[i:j]$. Let the *prefix* and *suffix* of $B[i:j]$ be the elements of $B[i:j]$ that respectively precede and succeed the span of $B[i:j]$. We precompute the following data for each possible span S:

i. an element of minimum frequency and its frequency in S, among all elements in S, and
ii. an element of minimum frequency and its frequency in S, among all elements (if any) that appear in S but not in the blocks immediately adjacent to the left and right of S.

Since $s(s+1)/2$ spans are possible, these data can be stored in a table D of size $\Theta(s^2)$. We construct this table in $O(ns)$ time by repeatedly passing through the entire array, starting at each of the s block boundaries. We will use the following lemma:

Lemma 3. *There exists a data structure maintaining an initially empty multiset S of elements from $\{0, \ldots, \Delta - 1\}$. It requires $O(\Delta)$ space and preprocessing time and supports the following operations:*

– *Insert(S, e): Inserts element e into multiset S in $O(1)$ time.*
– *LeastFrequentElement(S, k): Returns the k least frequent elements in S, along with their frequencies, in $O(k)$ time.*

Proof. We construct a doubly-linked list L, where each node contains a frequency f and a doubly-linked sublist of all distinct elements with frequency f. The nodes of L are sorted in the ascending order of frequency. Nodes for the sublists are taken from an array $N[0 : \Delta - 1]$ of nodes for each distinct element. Each of these sublist nodes contains a pointer to its containing sublist. It can be verified that an insertion of an element e causes only local changes around $N[e]$ that run in $O(1)$ time. To find the k least frequent elements, we simply iterate through L and its sublists until we have reported k elements or there are no more elements to report. □

During each pass we incrementally build a multiset using the data structure of Lemma 3. At every block boundary (i.e., every t elements) we obtain the least frequent element of the multiset in $O(1)$ time. We must also find the least frequent element excluding the elements contained in two blocks. This set of excluded elements has size $O(t)$ and so the element for which we are searching must appear amongst the $O(t)$ least frequent elements of the multiset, which we can find in $O(t)$ time. The total cost of a single pass is thus $O(n + st) = O(n)$ time. Therefore, the s passes altogether require $O(ns)$ time.

Query Algorithm. Consider arbitrary indices $0 \le i \le j \le n-1$ and the corresponding query range $R = B[i:j]$. If the prefix and suffix are empty, then the query can be answered in $O(1)$ time by referring to table D. By Lemma 2, if $j - i + 1 < 2t$, then the range query can be answered in $O(t) = O(n/s)$ time.

Now consider the case $j - i + 1 \geq 2t$. In this case, the span, denoted S, must be non-empty. We denote the prefix by P_1 and the suffix by P_2. Let P'_1 and P'_2 denote the respective blocks that contain P_1 and P_2. We now treat R, S, P_1, P_2, P'_1, P'_2 as multisets. Let P denote the union of P_1 and P_2. Similarly, let P' denote the union of P'_1 and P'_2. We partition the elements of R into four groups (see Figure 1) and find an element of minimum frequency among those in each group:

Case 1. elements of R that are in P but not S,
Case 2. elements of R that are in S and P,
Case 3. elements of R that are in S and P', but not P, and
Case 4. elements of R that are in S but not P'.

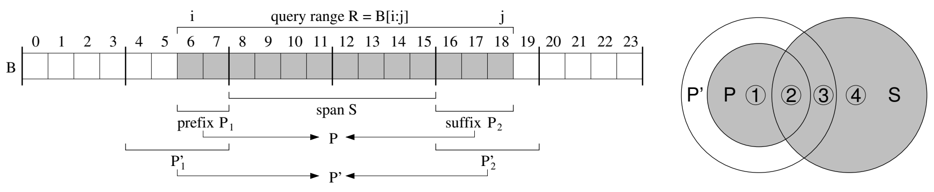

Fig. 1. Every element in the query range R (shaded) is in P or S, partitioned into sets 1–4.

We first show how to determine which elements of P' fall into Cases 1, 2, and 3. It suffices to determine for each element of P' whether or not the element appears in P and whether or not the element appears in S. We determine which elements appear in P by simply iterating through P. To determine which elements appear in S, we first find the closest occurrence of each element to S in a scan through P'. Assume that we have one such closest element $B[x]$ at index x. Assume without loss of generality that it appears in P'_1. The next occurrence of element $B[x]$ is at index $x' = Q_{B[x]}[B'[x] + 1]$, which we compute in $O(1)$ time. Thus, S contains an occurrence of element $B[x]$ if and only if x' lies inside S.

The least frequent element in R is given by the least frequent of those found in each of the four cases defined above:

Case 1. By Lemma 2, we compute the frequencies of all elements in P_1 in $O(t)$ time, omitting the final step of resetting the frequency table to zero. We then repeat for P_2 so that the frequency table contains aggregate data for all of P. Consider all elements that occur in P but not in S. For each such element e, $\text{freq}_R(e) = \text{freq}_P(e)$. So, the least frequent of these elements in R is the element with minimum non-zero entry in the frequency table.

Case 2. Let f denote the precomputed minimum frequency of any element in S, which is stored in table D. The minimum frequency in R of any element present in both S and P is at least f and at most $f + 2t$. For each element e that occurs in both S and P_1, we find the leftmost occurrence of e within P_1 in a scan through P_1. We repeat in a symmetric fashion in P_2. Then, by Lemma 1, we can check in $O(1)$ time whether an element e in both S and P has frequency in R less than some threshold. We begin with a threshold of $f+2t+1$. If an element e has frequency less than the threshold, we find its actual frequency by iterating through Q_e (forward or backwards depending on whether we are considering an element in P_1 or P_2) until reaching an index within R. This frequency becomes our new threshold. We repeat with all other elements that occur in both S and P. The last element to change the threshold is the least frequent of these elements. Since the threshold can decrease to at most f, the total time spent finding exact frequencies is $O(t)$.

CASE 3. Consider all elements that occur in both S and P' but not in P. As in Case 2, their frequencies in R are bounded between f and $f+2t$. We can thus apply the same technique as in Case 2. However, for each element, instead of finding the leftmost occurrence in P_1 or the rightmost occurrence in P_2 from which to base the queries of Lemma 1, we find the rightmost occurrence in P'_1 or the leftmost occurrence in P'_2.

CASE 4. Consider all elements that occur in S but not in P'. For each such element e, $\mathrm{freq}_R(e) = \mathrm{freq}_S(e)$. The least frequent of these elements has been precomputed and can be found in table D in $O(1)$ time.

Analysis. In addition to the arrays A, B, and B' (each $O(n)$ space), the data structure stores the tables $Q_0, \ldots, Q_{\Delta-1}$ ($O(n)$ total space), the tables D ($O(s^2)$ space), and a frequency table ($O(\Delta) \subseteq O(n)$ space). Populating, scanning, and resetting the frequency table during a query requires $O(t) = O(n/s)$ time. The query algorithm involves a constant number of scans of the blocks P'_1 and P'_2. Each element is processed in $O(1)$ amortized time, resulting in $O(t)$ total time. Thus, the data structure has space $O(n+s^2)$ and supports queries in $O(t) = O(n/s)$ time in the worst case. This completes the proof of Theorem 1.

2.2 Reduction from Boolean Matrix Multiplication

We follow the technique of Chan et al. [5] to multiply two $n \times n$ boolean matrices L and R via least frequent element range queries. In particular, we build an array A of size $n' \in O(n^2)$, and after preprocessing the array in $P(n')$ time we perform n^2 least frequent element queries, each in $Q(n')$ time, to calculate $M = LR$. The result is Theorem 2.

Theorem 2. *Given a data structure for least frequent element query in an array of n elements with $P(n)$ preprocessing time and $Q(n)$ query time, there exists an algorithm for boolean matrix multiplication of two $n \times n$ matrices that runs in $O(P(n^2) + n^2 Q(n^2))$ time.*

Thus, a data structure for least frequent element with $P(n) \in o(n^{3/2-\epsilon})$ and with $Q(n) \in o(n^{1/2-\epsilon})$ would yield an algorithm for boolean matrix multiplication that runs in $o(n^{3-2\epsilon})$ time, via purely combinatorial means.

The technique of Chan et al. [5] first reduces boolean matrix multiplication to set disjointness queries between sets encoding the rows of L and the columns of R. Let $U = \{1, \ldots, n\}$ be our ground set. We are left with the following problem: given sets $L_1, \ldots, L_n \subseteq U$ and $R_1, \ldots, R_n \subseteq U$, determine whether $L_i \cap R_j = \emptyset$ for all $i, j \in \{1, \ldots, n\}$.

Our construction of A involves creating $2n+1$ blocks of n elements: a block for each set L_i, followed by a block containing each element of U, followed by a block for each set R_j. The block for set L_i contains all elements of L_i followed by all elements of $U \setminus L_i$. The block for set R_j contains all elements of $U \setminus R_j$ followed by all elements of R_j.

We determine whether or not L_i and R_j are disjoint via a single least frequent element query from the leftmost element of $U \setminus L_i$ to the rightmost element of $U \setminus R_j$. This query range contains $i + j - 1 > 0$ complete blocks, each containing some permutation of U. If L_i and R_j are disjoint, then every element of U occurs either in $U \setminus L_i$ or $U \setminus R_j$. Thus, in this case, the least frequent element has frequency greater than $i + j - 1$. If L_i and R_j are not disjoint then some element occurs in neither $U \setminus L_i$ nor $U \setminus R_j$, and thus has the lowest possible frequency of $i + j - 1$. Thus, $L_i \cap R_j = \emptyset$ if and only if the frequency of the least frequent element in the range is exactly $i + j - 1$.

In total we must preprocess A, which has size $O(n^2)$ and perform n^2 least frequent element queries in this array, resulting in an algorithm that requires $O(P(n^2) + n^2 Q(n^2))$ time. This completes the proof of Theorem 2.

3 Range Minority

In this section we describe an $O(n)$-space data structure that identifies an α-minority element, if any exists, in a query range in $O(1/\alpha)$ time. We first reduce this α-minority range query problem to the problem of identifying the leftmost occurrences of the k leftmost distinct elements on or to the right of a given query index. We call the latter problem *distinct element searching* and we require that k can be specified at query time.

Lemma 4. *Given a data structure D for distinct element searching that requires $S_D(n)$ space and $Q_D(n, k)$ query time to report k elements, there exists a data structure for the α-minority range query problem that requires $O(S_D(n) + n)$ space and $O(Q_D(n, 1/\alpha) + 1/\alpha)$ query time.*

Proof. As described in Section 2.1, suppose we store in an array B', for each index i of A, a count of the number of times $A[i]$ occurs previously in A, and for each distinct element $x \in \{0, \ldots, \Delta - 1\}$, a sorted array Q_x of all the indices where x occurs in A. These arrays require $O(n)$ space. By Lemma 1, we can check in $O(1)$ time whether there are at least q instances of $A[i]$ in the range $A[i:j]$ for any $q \geq 0$ and $j \geq i$.

Observe that any element in a range is either an α-majority or an α-minority for the range and fewer than $1/\alpha$ distinct elements can be α-majorities. Thus, if we can find $1/\alpha$ distinct elements in a range, then at least one of them must be an α-minority.

Given a query range $A[i:j]$, we use data structure D to find the leftmost occurrences of the $1/\alpha$ leftmost distinct elements on or to the right of index i in $Q(n, 1/\alpha)$ time. Some of these leftmost occurrences may lie to the right of index j; we can ignore these elements as no occurrence of these elements lies in $A[i:j]$. There are $O(1/\alpha)$ remaining leftmost occurrences of leftmost distinct elements. Consider such an occurrence at index ℓ. Since this is the first occurrence of $A[\ell]$ on or after index i, the frequency of $A[\ell]$ in $A[\ell:j]$ is equal to the frequency of $A[\ell]$ in $A[i:j]$. We can then check whether or not $A[\ell]$ is an α-minority in $A[i:j]$ in $O(1)$ time by setting $q = \alpha(j-i+1)+1$ in Lemma 1. Repeating for all leftmost occurrences requires $O(1/\alpha)$ time.

If we find an α-minority we are done. If we do not find an α-minority, then there must not have been $1/\alpha$ distinct elements to check. In that case, we checked all distinct elements in $A[i:j]$ so there cannot be an α-minority. □

We can now focus on distinct element searching. If all queries use a common fixed k (as is the case if our goal is to solve just the range β-minority problem), there is a simple data structure that requires $O(n)$ space and $O(k)$ query time: for each i that is a multiple of k, store the k leftmost distinct elements to the right of index i; then for an arbitrary index i, we can answer a query by examining the k elements stored at $j' = \lceil i/k \rceil k$ in addition to the $O(k)$ elements in $A[i:j']$. However, it is not obvious how to extend this solution to the general problem for arbitrary k, without increasing the space bound.

In Lemma 5, we will map this problem to a 2-dimensional problem in computational geometry that can be solved by Chazelle's hive graph data structure [6]. Given n horizontal line segments, the hive graph allows efficient intersection searching along vertical rays. Finding the first horizontal line intersecting a vertical ray requires an orthogonal planar point location query; however, subsequent intersections can be found in sorted order in constant time each. The hive graph requires $O(n)$ space.

Lemma 5. *There exists a data structure for distinct element searching that requires $O(n)$ space and $O(k)$ query time.*

Proof. Let L_i be the set of indices in A that are associated with the leftmost occurrence of an element on or after index i. We can find the leftmost occurrences of the k leftmost distinct elements on or after index i by iterating through L_i in sorted order. However, $\sum_{i=0}^{n-1} |L_i|$ can be $\Omega(n^2)$ so we cannot afford to explicitly store all these sets.

Consider an index ℓ. Clearly, $\ell \in L_\ell$ and $\ell \notin L_i$ for $i > \ell$. Consider the first occurrence of $A[\ell]$ to the left of index ℓ at index ℓ', if it exists. Then $\ell \notin L_i$ for $i \leq \ell'$. However, for $\ell' < i \leq \ell$, $\ell \in L_i$. We associate ℓ with a horizontal segment with x-interval $(\ell', \ell]$ and with y-value ℓ. If no such index ℓ' exists, then we associate ℓ with a horizontal segment with x-interval $(-\infty, \ell]$ and with y-value ℓ. We thus have n horizontal segments. We build Chazelle's hive graph data structure [6] on these segments.

By the construction of the x-intervals of our segments, a segment intersects the vertical line $y = i$ if and only if it is associated with an index ℓ such that $\ell \in L_i$. Since the y-value of a segment associated with ℓ is ℓ, the segments are sorted along the vertical line in the order of their associated indices. Thus, to find the k leftmost indices in L_i, we query the hive graph for the horizontal segments with a vertical ray from $(i, 0)$ to (i, ∞). The cost of Chazelle's query algorithm is $O(t_{\mathrm{PL}}(n)+k)$ time, where $t_{\mathrm{PL}}(n)$ denotes the cost of a point location query in an orthogonal subdivision of size $O(n)$. The overall query time would then be $O(\log\log n + k)$ if we use the best known linear-space data structure for orthogonal point location of Chan [4].

To reduce the query time to $O(k)$, our key idea is to observe that there are only n distinct vertical rays with which we query the hive graph, and hence only n distinct points with which we do point location. Thus, we can perform the orthogonal point location component of each query during preprocessing and store each resulting node in the hive graph in a total of $O(n)$ space. (In fact, since all the query rays originate from points on the x-axis, the batched point locations are one-dimensional and can be handled easily in our application.) $\square$

Corollary 1. *There exists a data structure for the α-minority range query problem that requires $O(n)$ space and $O(1/\alpha)$ query time.*

Proof. By Lemmas 4 and 5. $\square$

4 Range Majority

We now consider the α-majority range query problem. Recently, Gagie et al. [11] describe an $O(n \log n)$-space data structure that supports α-majority in $O(1/\alpha)$ time, where α is specified at query time. In this section we describe a different α-majority range query data structure whose asymptotic space and time costs match those of Gagie et al. Previous work by Durocher et al. [9] considers the β-majority range query problem, where β is specified during preprocessing; their data structure requires $O(n \log(1/\beta + 1))$ space and supports queries in $O(1/\beta)$ time.

We begin by noting that a β-majority data structure can be adapted to support α-majority at the cost of increased space. Consider $\log n$ instances of the β-majority data structure of Durocher et al. [9], each with respective values $\beta = 2^{-i}$, for $i = 1, \ldots, \log n$, for a total of $O(n \log^2 n)$ space. For any query with parameter α, there is a data structure for which $1/\alpha \leq 1/\beta$ but $1/\beta \in O(1/\alpha)$. Querying this data structure results in a superset of the α-majorities of size $O(1/\alpha)$. The data structure, having counted the frequencies of each of these elements, can then filter out the α-minorities in $O(1/\alpha)$ time. Our effort now turns to solving the problem in $O(n \log n)$ space and $O(1/\alpha)$ query time using an entirely different approach.

Next we consider a related problem: reporting the top k most frequent elements in a query range where k is specified at query time. We call this problem the *top-k range query problem* while warning the reader not to confuse it with reporting the top k highest valued elements. We use a variation on the technique of Lemma 5 in order to support one-sided queries in $O(n)$ space and $O(k)$ query time. We note that the resulting data structure is a persistent version of Lemma 3 in which all updates are provided offline.

Lemma 6. *There exists a data structure for the one-sided top-k range query problem that requires $O(n)$ space and $O(k)$ query time.*

Proof. Assume our one-sided queries take the form $A[0:i]$ for $0 \leq i \leq n-1$. Consider the frequencies of the elements as we enlarge the one-sided range from left to right. Say an element has frequency f for ranges $A[0:i]$ through $A[0:j]$ and this range of ranges is maximal. We construct a horizontal segment with x-interval $[i, j+1)$ and with y-value f. We repeat for all elements and for all $f > 0$ and arbitrarily perturb the y-values for any segments that overlap.

In total, we construct $\Delta \leq n$ segments with y-value 0: one segment corresponding to each distinct element having frequency 0 in a vacuous subarray. Each element of A causes a single change in frequency of a single element, which results in one additional segment. So, in total we construct $O(n)$ segments. We build Chazelle's hive graph data structure [6] on these segments.

For every distinct element e in $A[0:i]$ there is a horizontal segment with x-interval $[\ell, r+1)$ intersecting the vertical line $y = i$ with $A[\ell] = e$ and $\text{freq}_{A[0:i]}(e) = f$. These horizontal segments are sorted along the vertical line in the order of frequency. To find the k most frequent elements in $A[0:i]$, we query the hive graph for the first k horizontal segments intersecting the vertical ray from (i, n) to $(i, -\infty)$. As in Lemma 5, there are only n distinct queries to the hive graph, so we can perform the orthogonal point location component of each query during preprocessing at a cost of $O(n)$ space to store the resulting nodes of the hive graph. For each segment that the hive graph reports, we report $A[\ell]$ where ℓ is the left x-coordinate of the segment. □

Observe also that the index of the leftmost endpoint of the horizontal segment associated with a reported element is the index of the rightmost occurrence of the element in $A[0:i]$. Top-k queries are not decomposable in the sense that, given a partition of a range R into two subranges R_1 and R_2, there is no relationship between the top k most frequent elements in R_1, R_2, and R. As observed by Karpinski and Nekrich [15], given the same partition of R, an α-majority in R must either be an α-majority in R_1 or R_2. Since α-majority queries are decomposable in this way, and since all α-majorities are amongst the top $1/\alpha$ most frequent elements, we can now apply the range tree to support two-sided α-majority queries.

Theorem 3. *There exists a data structure for the α-majority range query problem that requires $O(n \log n)$ space and $O(1/\alpha)$ query time.*

Proof. We build the data structure of Lemma 6 on array A. We divide A into two halves and recurse in both halves to create a range tree. The total space consumption of all top-k data structures is thus $O(n \log n)$. We also include a data structure for lowest common ancestor queries in the range tree. We use this data structure to decompose a two-sided query into one-sided queries in two nodes of the range tree. There are succinct data structures for LCA that require only $O(n)$ bits of space and $O(1)$ time (e.g., [19]). We also build the arrays required to support the queries of Lemma 1.

We decompose a two-sided query into one-sided queries in two nodes of the range tree in $O(1)$ time. For each one-sided query we find the $1/\alpha$ most frequent elements using the top-k data structures in $O(1/\alpha)$ time. By the decomposability of α-majority queries as observed by Karpinski and Nekrich [15], our $O(1/\alpha)$ most frequent elements in both one-sided ranges are a superset of the α-majorities of the original two-sided query. Since the top-k data structures report for each element occurrences that are closest to one of the boundaries of the two-sided range, we can apply Lemma 1 to check which of the $O(1/\alpha)$ most frequent elements are in fact α-majorities in constant time each. $\square$

5 Discussion

Using binary rank and select data structures and bit packing, Chan et al. [5] reduce the range mode query time from $O(\sqrt{n})$ to $O(\sqrt{n/\log n})$ without increasing the data structure's space beyond $O(n)$. Unlike the frequency of the mode, the frequency of the least frequent element does not vary monotonically over a sequence of elements. Furthermore, unlike the mode, when the least frequent element changes, the new element of minimum frequency is not necessarily located in the block in which the change occurs. Consequently, the techniques of Chan et al. do not seem immediately applicable to the least frequent range query problem; it remains open whether $o(\sqrt{n})$ query time is possible in $O(n)$ space.

We have described a data structure for the range least frequent element problem achieving $O(\sqrt{n})$ query time with $O(n^{3/2})$ preprocessing time, and given a lower bound by reduction from boolean matrix multiplication under which least frequent element with $o(n^{1/2-\epsilon})$ query time and $o(n^{3/2-\epsilon})$ preprocessing time would imply matrix multiplication in $o(n^{3-2\epsilon})$ time by purely combinatorial means. We have also given a data structure achieving $O(1/\alpha)$ query time in $O(n)$ space on the range α-minority problem; and one achieving $O(1/\alpha)$ query time in $O(n \log n)$ space on the range α-majority problem, matching the bounds achieved by that of Gagie et al. [11]. The greater space required by current α-majority data structures compared to that required by current α-minority data structures suggests that further improvement may be possible; whether α-majority range queries can be supported in $o(n \log n)$ space and $O(1/\alpha)$ query time remains open.

Acknowledgements

The authors thank Patrick Nicholson for insightful discussion of the α-majority range query problem.

References

1. M. A. Bender and M. Farach-Colton. The LCA problem revisited. In *Proc. LATIN*, volume 1776 of *LNCS*, pages 88–94. Springer, 2000.
2. P. Bose, E. Kranakis, P. Morin, and Y. Tang. Approximate range mode and range median queries. In *Proc. STACS*, volume 3404 of *LNCS*, pages 377–388. Springer, 2005.
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. T. M. Chan. Persistent predecessor search and orthogonal point location on the word RAM. In *Proc. ACM-SIAM SODA*, pages 1131–1145, 2011.
5. T. M. Chan, S. Durocher, K. G. Larsen, J. Morrison, and B. T. Wilkinson. Linear-space data structures for range mode query in arrays. In *Proc. STACS*, volume 14, pages 291–301, 2012.
6. B. Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comp.*, 15(3):703–724, 1986.
7. E. D. Demaine, G. M. Landau, and O. Weimann. On Cartesian trees and range minimum queries. In *Proc. ICALP*, volume 5555 of *LNCS*, pages 341–353. Springer, 2009.
8. S. Durocher. A simple linear-space data structure for constant-time range minimum query. *CoRR*, abs/1109.4460, 2011.
9. S. Durocher, M. He, J. I. Munro, P. K. Nicholson, and M. Skala. Range majority in constant time and linear space. In *Proc. ICALP*, volume 6755 of *LNCS*, pages 244–255. Springer, 2011.
10. A. Elmasry, M. He, J. I. Munro, and P. Nicholson. Dynamic range majority data structures. In *Proc. ISAAC*, volume 7074 of *LNCS*, pages 150–159. Springer, 2011.
11. T. Gagie, M. He, J. I. Munro, and P. Nicholson. Finding frequent elements in compressed 2D arrays and strings. In *Proc. SPIRE*, volume 7024 of *LNCS*, pages 295–300. Springer, 2011.
12. T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proc. SPIRE*, volume 5721 of *LNCS*, pages 1–6. Springer, 2009.
13. M. Greve, A. G. Jørgensen, K. D. Larsen, and J. Truelsen. Cell probe lower bounds and approximations for range mode. In *Proc. ICALP*, volume 6198 of *LNCS*, pages 605–616. Springer, 2010.
14. A. G. Jørgensen and K. D. Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proc. ACM-SIAM SODA*, pages 805–813, 2011.
15. M. Karpinski and Y. Nekrich. Searching for frequent colors in rectangles. In *Proc. CCCG*, pages 11–14, 2008.
16. D. Krizanc, P. Morin, and M. Smid. Range mode and range median queries on lists and trees. *Nordic J. Computing*, 12:1–17, 2005.
17. H. Petersen. Improved bounds for range mode and range median queries. In *Proc. SOFSEM*, volume 4910 of *LNCS*, pages 418–423. Springer, 2008.
18. H. Petersen and S. Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Inf. Proc. Let.*, 109:225–228, 2009.
19. K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proc. ACM-SIAM SODA*, pages 134–149, 2010.