Online Sorted Range Reporting and Approximating the Mode

Mark Greve

Progress Report

Department of Computer Science
Aarhus University
Denmark

January 4, 2010

Supervisor: Gerth Stølting Brodal

Online Sorted Range Reporting and Approximating the Mode

A Progress Report
Presented to the Faculty of Science
of the Aarhus University
in Partial Fulfilment of the Requirements for the
PhD Degree

by
Mark Greve
January 4, 2010

Contents

List of Figures

Chapter 1

Introduction

This progress report describes my research during the initial two years of my PhD studies. The work presented in this progress report was done in collaboration with the following colleagues: Gerth Stølting Brodal, Rolf Fagerberg, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, Alejandro López-Ortiz, and Jakob Truelsen. The work is presented in the following papers:

- Online Sorted Range Reporting
  Gerth Stølting Brodal, Rolf Fagerberg, Mark Greve and Alejandro López-Ortiz
  Proc. 20th Annual International Symposium on Algorithms and Computation, ISAAC '09 [8]

- Approximating the Mode and Determining Labels with Fixed Frequency
  Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, Jakob Truelsen
  Manuscript [22]

In my work I have mainly focused on the design of efficient data structures. The core body of my work is described in the two above papers, one of which is published and the other is a manuscript. My work deals with two distinct problems that have a common flavor. They both involve preprocessing a static array into a data structure that efficiently supports "range queries".

The first part of my work focuses on a problem called sorted range selection. In this problem we are to preprocess a static array A of n elements into a data structure that efficiently supports range queries of the following form: Report the k smallest elements in the subarray $A[i..j]$ in sorted order. We study several variants of this problem in the unit-cost RAM model.

The second part of my work focuses on the range mode problem. In this problem we must preprocess a static array A of n labels into a data structure that efficiently supports queries of the following form: Report the label of a most frequently occurring label in $A[i..j]$. In this work we mainly focus on efficiently approximating the answer to such queries, i.e. return a label that occurs nearly as often as that of a most frequently occurring label. We also study this problem in the unit-cost RAM model.

Outline. Chapter 2 is about my work on the online sorted range reporting problem [8]. This chapter consists of an extended version of the the paper that appeared at ISAAC '09. In this chapter the following one-dimensional range reporting problem is studied: On an array A

of n elements, support queries that given two indices $i \leq j$ and an integer k report the k smallest elements in the subarray $A[i..j]$ in sorted order. A data structure in the RAM model supporting such queries in optimal $O(k)$ time is presented. The structure uses $O(n)$ words of space and can be constructed in $O(n \log n)$ time. The data structure can be extended to solve the online version of the problem, where the elements in $A[i..j]$ are reported one-by-one in sorted order, in $O(1)$ worst-case time per element.

Chapter 3 deals with my work on the range mode problem [22]. In this chapter we consider data structures for approximate range mode and range k-frequency queries. The mode of a multiset of labels, is a label that occurs at least as often as any other label. The input to the range mode problem is an array A of size n. A range query $[i, j]$ must return the mode of the subarray $A[i], A[i + 1], \ldots, A[j]$. A c-approximate range mode query for any constant $c > 1$ must return a label that occurs at least $1/c$ times that of the mode. We describe a linear space data structure that supports 3-approximate range mode queries in constant time, and a data structure that uses $O(\frac{n}{\varepsilon})$ space and supports $(1 + \varepsilon)$-approximation queries in $O(\log \frac{1}{\varepsilon})$ time. Furthermore, we consider the related range k-frequency problem. The input is an array A of size n. A query $[i, j]$ must return whether there exists a label that occurs precisely k times in the subarray $A[i], A[i + 1], \ldots, A[j]$. We show that for any constant $k > 1$, this problem is equivalent to 2D orthogonal rectangle stabbing, and that for $k = 1$ this is no harder than four-sided 3D orthogonal range emptiness.

Finally in Chapter 4 the plans for the remaining time of my PhD studies are described. This includes the discussion of several open problems.

Chapter 2

Online Sorted Range Reporting

2.1 Introduction

In information retrieval, the basic query types are exact word matches, and combinations such as intersections of these. Besides exact word matches, search engines may also support more advanced query types like prefix matches on words, general pattern matching on words, and phrase matches. Many efficient solutions for these involve string tree structures such as tries and suffix trees, with query algorithms returning nodes of the tree. The leaves in the subtree of the returned node then represent the answer to the query, e.g. as pointers to documents.[1]

An important part of any search engine is the ranking of the returned documents. Often, a significant element of this ranking is a query-independent pre-calculated rank of each document, with PageRank [31] being the canonical example of such a query-independent rank. In the further processing of the answer to a tree search, it is beneficial to return the answer set ordered by the pre-calculated rank, and even better if it is possible to generate an increasing prefix of this ordered set on demand. The reasons for this will be given below when we elaborate on applications. In short, we would like a functionality similar to storing at each node in the tree a list of the leaves in its subtree sorted by their pre-calculated rank, but without the prohibitive space cost incurred by this solution.

Motivated by the above example, we in this chapter consider the following set of problems, listed in order of increasing generality. For each problem, an array $A[0..n-1]$ of n numbers is given, and the task is to preprocess A into a space-efficient data structure that efficiently supports the query stated.

Sorted range reporting: Given two indices $i \leq j$, report the elements in $A[i..j]$ in sorted order.

Sorted range selection: Given two indices $i \leq j$ and an integer k, report the k smallest elements in $A[i..j]$ in sorted order.

Online sorted range reporting: Given two indices $i \leq j$, report the elements in $A[i..j]$ one-by-one in sorted order.

[1] We will assume that each leaf of the tree (which for efficiency reasons must be in RAM, not on disk) corresponds to one document in the collection. Even for the collection sizes appearing in web search engines, this is realistic due to the massive partitioning and distribution of indices over many machines employed in these.

Note that if the leaves of a tree are numbered during a depth-first traversal, the leaves of any subtree form a consecutive segment of the numbering. By placing leaf number i at entry $A[i]$, and annotating each node of the tree by the maximal and minimal leaf number in its subtree, we see that the three problems above generalize our problems on trees.[2] The aim of this chapter is to present linear space data structures with optimal query bounds for each of the three problems.

We remark that the problems above also form generalizations of two well-studied problems, namely the range minimum query (RMQ) problem and three-sided planar range reporting.

The RMQ problem is to preprocess an array A such that given two indices $i \leq j$, the minimum element in $A[i..j]$ can be returned efficiently. This is generalized by the sorted range selection problem above. RMQ is equivalent to the lowest common ancestor (LCA) problem on trees, in the sense that there is a linear time transformation between the two problems [20]. The LCA problem is to preprocess a tree such that given two nodes, their lowest common ancestor can be returned efficiently. It is a fundamental combinatorial problem, and is used as a building block in many algorithms. It was shown to have a linear space, constant time query solution (in the RAM model) by Harel and Tarjan [23]. Later work on the LCA problem has focused on simplifying the solution and extending it to other models—for an overview, see [2]. Some recent algorithmic engineering work directly on the RMQ problem appears in [14] and [13], where the focus is on reducing the amount of space required to implement a suffix array based structure, a structure with a known optimal space bound of $n + o(n)$ [24].

Three-sided planar range reporting is the problem of given three parameters x_1, x_2 and y_1, report all points (x, y) with $x_1 \leq x \leq x_2$ and $y \geq y_1$. The online sorted range reporting problem generalizes this by returning the answer set in decreasing order of y. The classical solution to three-sided range queries are the priority search trees of McCreight [30], which uses linear space and $O(\log n + r)$ query time, where r is the size of the output. Versions for other models have been given, e.g. for integer data [19] and for external memory [3].

2.1.1 Applications: Top-k queries

A standard model in data retrieval is to return all tuples that satisfy a given query. However, with data collections growing in size, this model increasingly results in answer sets too large to be of use. In such cases, returning a limited sample from the answer set may be preferable. Depending on the application this can be a random subset, the first k results, or more importantly for our case, the top k results under some ranking or priority order. This is termed a *top-k* query. In what follows we give details of two distinct settings in which such a subset of results is desired. One is search engines for information retrieval, as introduced above. The other is relational databases.

Search engines for information retrieval Usually, search engines prioritize the returned documents using ranking algorithms, and return the top ten matches according to rank score. For simplicity of exposition, we are here assuming a ranking algorithm which is query-independent (such as PageRank).[3]

[2] For space efficiency, search engines may work with suffix arrays instead of suffix trees. In this case, the three problems even more directly generalize the corresponding problems on suffix arrays.

[3] More realistically, the search engine will scan answers to the query in decreasing order of query-independent rank, while factoring in query-dependent parts to obtain the final ranking values. It will stop when further answers better than the current kth best in final ranking are provably impossible (or just unlikely), based on

For exact word matches in search engines, a common data structure is inverted indices, which for each word in the document collection stores a list containing the IDs of the documents containing that word. In inverted files, top-k queries are easily accommodated by using ranks as IDs and storing lists sorted on ID.

In contrast, tries and their derivatives such as suffix trees, Pat trees and suffix arrays, which are needed for more advanced query types such as efficient phrase searches and general pattern searching, do not have a straightforward ranked variant. Baeza-Yates and Neto [4, page 202] point out this added difficulty: "[All these] queries retrieve a subtree of the suffix tree or an interval of the suffix array. The results have to be collected later, which may imply sorting them in ascending text order. This is a complication of suffix trees or arrays with respect to inverted indices." This raises the question of how to adapt these data structures to support top-k queries among the strings matching a given prefix, when leaves are annoted by ranks.

A motivating factor for returning the top-k set in sorted order is that if more than a single query term is given, then there is a set of documents associated to each query term, and the result set normally consists of the intersection of these sets. Fast methods to compute such intersection often are based on merging and rely on the sets being in sorted order. For example, Kaufmann and Schek [27] observe with regards to intersections on trie-based solutions that: "[A] negative aspect is that postings are returned unordered, i.e. not with ascending document IDs... [As an] intersection can only be carried out efficiently for ordered posting lists, [this makes] sorting (in $O(n \log n)$ steps) [of said lists] necessary, which cannot be accomplished in user-acceptable time in the case of large posting lists."

Note that to return the top-k result after an intersection, the merge process involved in the intersection only needs to be run as far as to produce an output of size k, hence online generation of the input streams may give significant savings in running time. This motivates the online version of our problem.

In short, the data structure we propose could take the role of sorted inverted lists in search engines, with the key difference that it now supports more queries efficiently. For still further details on search engine techniques, we refer the reader to [4, 21, 27, 29].

Relational Database Queries In the past, the query-answer model in relational databases was that an SQL query would return all the tuples matching the query. As databases grow in size, research has focused on a model in which every query has an implicit or explicit top-k qualifier attached to it, where top is defined in a query specific way. In particular the SQL operator "STOP AFTER" has been introduced to select the first k results from a traditional SELECT-FROM-WHERE query [9].

If the ranking function is completely arbitrary and unknown in advance it is not difficult to show that computing the top k results requires time at least linear on the entire set of tuples satisfying the query. On the other hand if the sorted order is given in advance and the aggregation function is well-behaved it is desirable to speed up the computation of the answer. How efficiently this can be achieved is known as the *braking distance* of the query [10].

In its full generality, top-k queries in the context of databases refer to the top-k ranked elements of a multidimensional dataset. A particular instance considered by Fagin et al. [12] is the case where a candidate set has been identified and has been sorted by rank on each of

how the final ranking value is composed. This only highlights that we are interested in top-k queries for k chosen online, as discussed below.

the ranking coordinates. The goal is then to compute the top-k candidates out of that set assuming a monotone ranking function. In general, the assumption in that work is that there is a sorted index on the result set alone and on each ranking dimension allowing for sequential access in descending order (see e.g. [36]).

In contrast, we consider the case in which producing a sorted list of the ranking dimension is non-trivial (under space constraints) and proposes a method to achieve this efficiently. Indeed, our algorithm can be used as a primitive by Fagin et al. in that it supports on-line sorted access to the top-k candidates on each coordinate, with the aggregation function being computed using Fagin et al.'s algorithm.

In more general terms, the main techniques normally used for computing top-k queries are either inserting the result set in a heap leading to time $O(k \log k + n)$ or using range partitioning. The algorithms are often made adaptive in that while they do not improve the worst-case complexity, they perform various optimizations on simpler instances.

Our algorithms improve the worst-case time-space complexity by a factor of $\log n$, and are relevant to the case where the basic query is a range selection, and the ranks of tuples are known in advance.

2.1.2 Contributions

We present data structures to support sorted range reporting queries in $O(j - i + 1)$ time, sorted range selection queries in $O(k)$ time, and online sorted range reporting queries in worst case $O(1)$ time per element reported. For all problems the solutions take $O(n)$ words of space and can be constructed in $O(n \log n)$ time.

We assume a unit-cost RAM whose operations include addition, subtraction, bitwise AND, OR, XOR, and left and right shifting and multiplication. Multiplication is not crucial to our constructions and can be avoided by the use of table lookup. By w we denote the word length in bits, and assume that $w \geq \log n$ and that w is a power of two. We use the convention that the least significant bit of an integer is the 0'th bit.

2.1.3 Overview of the chapter

In Section 2.2, we describe some simple constructions which are used extensively in the solution of all three problems. In Section 2.3, we give a simple solution to the sorted range reporting problem that illustrates some of the ideas used in our more involved solution of the sorted range selection problem. In Section 2.4, we present the main result of this chapter which is our solution to the sorted range selection problem. Building on the solution to the previous problem, we give a solution to the online sorted range reporting problem in Section 2.5.

2.2 Preliminaries

In this section, we describe three simple results used by our constructions. The main purpose is to state the lemmas and some definitions for later use.

2.2.1 Packing multiple items in a word

Essential for our constructions to achieve overall linear space, is the ability to pack multiple equally sized items into a single word.

Lemma 2.1 *Let X be a two-dimensional array of size $s \times t$ where each entry $X[i][j]$ consists of $b \leq w$ bits for $0 \leq i < s$ and $0 \leq j < t$. We can store this using $O(stb + w)$ bits, and access entries of X in $O(1)$ time.*

Proof. We use an array B of $\lceil stb/w \rceil$ words and store the y'th bit of $X[i][j]$ as the $((ti+j)b+y)$ mod w'th bit of $B\,[\lfloor((ti+j)b+y)/w\rfloor]$. To make a lookup of $z = X[i][j]$ we mask out all bits of $B\,[\lfloor((ti+j)b)/w\rfloor]$ except for the ones where z is stored, and shift a number of places to the right to retrieve z. However, if z is stored across two words, we get the remaining bits from the next word of B also by using a bit mask and shifting, and then using bitwise operations we concatenate the two numbers. All of these operations can be done in $O(1)$ time. The space bound follows from the fact that we waste at most $w-1$ bits on the last word of B, and that we need to know b to be able to access the entries. □

Concerning the word operations needed for the above packing, we can avoid the use of division and modulo since w is a power of two—instead we can perform this using bitwise operations. Note that if t and b are powers of two, we can replace the use of multiplication by shift operations in the computation of $(ti+j)b$. If t and b are not powers of two, we can replace them with their nearest larger power of two. This will at most quadruple the space usage.

In most cases we encounter, we have an array of secondary arrays, where the secondary arrays have variable length. It is easy to extend the above construction to allow this. If the largest secondary array has length t we pad those of length less than t with dummy elements, so that they all effectively get length t. For our applications we also need to be able to determine the length of a secondary array. Since a secondary array has length at most t, we need $\lfloor \log t \rfloor + 1$ bits to represent the length. By packing the length of each secondary array as described in Lemma 2.1, we get a total space usage of $O(stb + s \log t + w) = O(stb + w)$ bits of space, and lookups can still be performed in $O(1)$ time. We summarize this in a lemma.

Lemma 2.2 *Let X be an array of s secondary arrays, where each secondary array $X[i]$ contains up to t elements, and each entry $X[i][j]$ in a secondary array $X[i]$ takes up $b \leq w$ bits. We can store this using $O(stb + w)$ bits of space and access an entry or length of a secondary array in $O(1)$ time.*

2.2.2 Complete binary trees

Throughout this chapter $\mathcal{T}$ will denote a complete binary tree with n leaves where n is a power of two and the leaves are numbered from n to $2n-1$. The numbering of nodes is the one used for binary heaps [15], [38]. The root has index 1, and an internal node with index x has left child $2x$, right child $2x+1$ and parent $\lfloor x/2 \rfloor$. Below, we identify a node by its number.

We let $\mathcal{T}_u$ denote the subtree of $\mathcal{T}$ rooted at the node u, and $h(\mathcal{T}_u)$ the *height* of the subtree $\mathcal{T}_u$ where the height of a leaf is defined to be 0. The height of a node $h(u)$ is defined to be $h(T_u)$, and *level* ℓ of $\mathcal{T}$ is defined to be the set of nodes with height ℓ. The height $h(u)$ of a node u can be computed in $O(1)$ time as $h(\mathcal{T}) - d(u)$, where $d(u)$ is the *depth* of node

u. The depth $d(u)$ of a node u can be found in $O(1)$ time by computing the index of the most significant bit set in u (the root has depth 0). See [18] for a solution to the problem of finding the most significant bit set in a word in $O(1)$ time and space, assuming we can do multiplication in $O(1)$ time. Using a lookup table mapping every integer from $0 \ldots 2n-1$ to the index of its most significant bit set, we get an $O(1)$ time and $O(n)$ space solution avoiding the use of multiplication.

To navigate efficiently in $\mathcal{T}$, we explain a few additional operations that can be performed in $O(1)$ time. First we define $\text{anc}(u, \ell)$ as the ℓ'th ancestor of the node u, where $\text{anc}(u, 0) = u$ and $\text{anc}(u, \ell) = \text{parent}(\text{anc}(u, \ell - 1))$. Note that $\text{anc}(u, \ell)$ can be computed in $O(1)$ time by right shifting u ℓ times. Finally, we note that we can find the leftmost leaf in a subtree $\mathcal{T}_u$ in $O(1)$ time by left shifting u $h(u)$ times. Similarly we can find the rightmost leaf in a subtree $\mathcal{T}_u$ in $O(1)$ time by left shifting u $h(u)$ times, and setting the bits shifted to 1 using bitwise OR.

2.2.3 LCA queries in complete binary trees

An important component in our construction is finding the lowest common ancestor (LCA) of two nodes at the same depth in a complete binary tree in $O(1)$ time. We will briefly describe a standard solution to this problem. Given two nodes $u \neq v$ at the same depth in $\mathcal{T}$, we find the index y of the most significant bit where u and v differ (i.e. y is the index of the most significant bit set of u XOR v). This corresponds to where the two paths from the root towards u and v diverge, and this is exactly the definition of the LCA of u and v. To find the index of the LCA we right shift u (or v) y times. Thus, the essential part needed to compute the LCA in $O(1)$ time is finding the index of the most significant bit set in a word in $O(1)$ time.

2.2.4 Selection in sorted arrays

The following theorem due to Frederickson and Johnson [17] is essential to our construction in Section 2.4.4.

Theorem 2.1 *Given m sorted arrays, we can find the overall k smallest elements in time $O(m + k)$.*

Proof. By Theorem 1 in [17] with $p = \min(m, k)$ we can find the k'th smallest element in time $O(m + p \log(k/p)) = O(m + k)$ time. When we have the k'th smallest element x, we can go through each of the m sorted arrays and select elements that are $\leq x$ until we have collected k elements or exhausted all lists. This takes time $O(m + k)$. □

2.3 Sorted range reporting

In this section, we give a simple solution to the sorted range reporting problem with query time $O(j - i + 1)$. The solution introduces the concept of *local rank labellings* of elements, and shows how to combine this with radix sorting to answer sorted range reporting queries. These two basic techniques will be used in a similar way in the solution of the more general sorted range selection problem in Section 2.4.

We construct local rank labellings for each r in $0 \ldots \lceil \log \log n \rceil$ as follows (the rank of an element x in a set X is defined as $|\{y \in X \mid y < x\}|$). For each r, the input array is divided into $\lceil n/2^{2^r} \rceil$ consecutive subarrays each of size 2^{2^r} (except perhaps the last subarray), and for each element $A[x]$ the r'th local rank labelling is defined as its rank in the subarray $A[\lfloor x/2^{2^r} \rfloor 2^{2^r} .. (\lfloor x/2^{2^r} \rfloor + 1)2^{2^r} - 1]$. Thus, the r'th local rank for an element $A[x]$ consists of 2^r bits. Using Lemma 2.1 we can store all local rank labels of length 2^r using space $O(n2^r + w)$ bits. For all $\lceil \log \log n \rceil$ local rank labellings, the total number of bits used is $O(w \log \log n + n \log n) = O(nw)$ bits. All local rank labellings can be built in $O(n \log n)$ time using $O(n)$ extra words of space while performing mergesort on A (carried out level-by-level). The r'th structure is built by writing out the sorted lists, when we reach level 2^r.

Given a query for $k = j - i + 1$ elements, we find the r for which $2^{2^{r-1}} < k \leq 2^{2^r}$ by making a linear search for r in $O(r) = O(k)$ time. Since each subarray in the r'th local rank labelling contains 2^{2^r} elements, we know that i and j are either in the same or in two consecutive subarrays. If i and j are in consecutive subarrays, we compute the start index of the subarray where the index j belongs, i.e. $x = \lfloor j/2^{2^r} \rfloor 2^{2^r}$. We then radix sort the elements in $A[i..x-1]$ using the local rank labels of length 2^r. This can be done in $O(k)$ time using two passes by dividing the 2^r bits into two parts of 2^{r-1} bits each, since $2^{2^{r-1}} < k$. Similarly we radix sort the elements from $A[x..j]$ using the labels of length 2^r in $O(k)$ time. Finally, we merge these two sorted sequences in $O(k)$ time, and return the k smallest elements. If i and j are in the same subarray, we just radix sort $A[i..j]$.

2.4 Sorted range selection

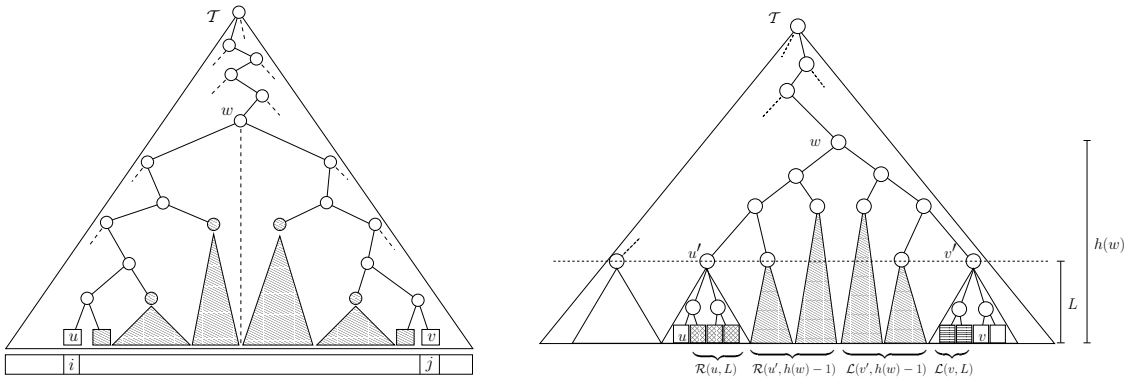
Before presenting our solution to the sorted range selection problem we note that if we do not require the output of a query to be sorted, it is possible to get a conceptually simple solution with $O(k)$ query time using $O(n)$ preprocessing time and space. First we build a data structure to support range minimum queries in $O(1)$ time using $O(n)$ preprocessing time and space [23, 5]. Given a query on $A[i..j]$ with parameter k, we lazily build the Cartesian tree [37] for the subarray $A[i..j]$ using range minimum queries. The Cartesian tree is defined recursively by choosing the root to be the minimum element in $A[i..j]$, say $A[x]$, and recursively constructing its left subtree using $A[i..x-1]$ and its right subtree using $A[x+1..j]$. By observing that the Cartesian tree is a heap-ordered binary tree storing the elements of $A[i..j]$, we can use the heap selection algorithm of Frederickson [16] to select the k smallest elements in $O(k)$ time. Thus, we can find the k smallest elements in $A[i..j]$ in *unsorted* order in $O(k)$ time.

In the remainder of this section, we present our data structure for the sorted range selection problem. The data structure supports queries in $O(k)$ time, uses $O(n)$ words of space and can be constructed in $O(n \log n)$ time. When answering a query we choose to have our algorithm return the indices of the elements of the output, and not the actual elements.

Our solution consists of two data structures for the cases where $k \leq \lfloor \log n/(2 \log \log n)^2 \rfloor$ and $k > \lfloor \log n/(2 \log \log n)^2 \rfloor$. The two data structures are described in Sections 2.4.3 and 2.4.4 respectively. In Sections 2.4.1 and 2.4.2, we present two simple techniques used heavily by both data structures. In Section 2.4.1, we show how to decompose sorted range selection queries into a constant number of smaller ranges, and in Section 2.4.2 we show how to answer a subset of the queries from the decomposition by precomputing the answers. The algorithms for constructing the data structures are described in Section 2.4.5.

2.4.1 Decomposition of queries

For both data structures described in Sections 2.4.3 and 2.4.4, we consider a complete binary tree $\mathcal{T}$ with the leaves storing the input array A. We assume without loss of generality that the size n of A is a power of two. Given an index i for $0 \leq i < n$ into A we denote the corresponding leaf in T as $\text{leaf}[i] = n + i$. For a node x in $\mathcal{T}$, we define the canonical subset C_x as the leaves in $\mathcal{T}_x$. For a query range $A[i..j]$, we let $u = \text{leaf}[i]$, $v = \text{leaf}[j]$, and $w = \text{LCA}(u, v)$. On the two paths from the two leaves to their LCA w we get at most $2 \log n$ disjoint canonical subsets, whose union represents all elements in $A[i..j]$, see Figure 2.1(a).



(a) The shaded parts and the leaves u and v cover the query range.

(b) Decomposition of a query range into four smaller ranges by cutting the tree at level L.

Figure 2.1: Query decomposition of a range.

For a node x we define the sets $\mathcal{R}(x, \ell)$ (and $\mathcal{L}(x, \ell)$) as the union of the canonical subsets of nodes rooted at the right (left for $\mathcal{L}$) children of the nodes on the path from x to the ancestor of x at level ℓ, but excluding the canonical subsets of nodes that are on this path, see Figure 2.1(a). Using the definition of the sets $\mathcal{R}$ and $\mathcal{L}$, we see that the set of leaves strictly between leaves u and v is equal to $\mathcal{R}(u, h(w) - 1) \cup \mathcal{L}(v, h(w) - 1)$. In particular, we will decompose queries as shown in Figure 2.1(b). Assume L is a fixed level in $\mathcal{T}$, and that the LCA w is at a level $> L$. Define the ancestors $u' = \text{anc}(u, L)$ and $v' = \text{anc}(v, L)$ of u and v at level L. We observe that the query range, i.e. the set of leaves strictly between leaves u and v can be represented as $\mathcal{R}(u, L) \cup \mathcal{R}(u', h(w) - 1) \cup \mathcal{L}(v', h(w) - 1) \cup \mathcal{L}(v, L)$. In the case that the LCA w is below or at level L, the set of leaves strictly between u and v is equal to $\mathcal{R}(u, h(w) - 1) \cup \mathcal{L}(v, h(w) - 1)$.

Hence to answer a sorted range selection query on k elements using the decomposition, we need only find the k smallest elements in sorted order of each of these at most four sets, and then select the k overall smallest elements in sorted order from these $O(1)$ sets including the two leaves u and v. Assuming we have a sorted list over the k smallest elements for each set, this can be done in $O(k)$ time by merging the sorted lists (including u and v), and extracting the k smallest of the merged list. Thus, assuming we have a procedure for finding the k smallest elements in each set in $O(k)$ time, we obtain a general procedure for sorted range selection queries in $O(k)$ time.

The above decomposition motivates the definition of *bottom* and *top* queries relative to a fixed level L. A *bottom* query on k elements is the computation of the k smallest elements in sorted order in $\mathcal{R}(x, \ell)$ (or $\mathcal{L}(x, \ell)$) where x is a leaf and $\ell \leq L$. A *top* query on k elements

is the computation of the k smallest elements in sorted order in $\mathcal{R}(x, \ell)$ (or $\mathcal{L}(x, \ell)$) where x is a node at level L. From this point on we only state the level L where we cut $\mathcal{T}$, and then discuss how to answer bottom and top queries in $O(k)$ time, i.e. implicitly assuming that we use the procedure described in this section to decompose the original query, and obtain the final result from the answers to the smaller queries.

2.4.2 Precomputing answers to queries

In this section, we describe a simple solution that can be used to answer a subset of possible queries, where a query is the computation of the k smallest elements in sorted order of $\mathcal{R}(x, \ell)$ or $\mathcal{L}(x, \ell)$ for some node x and a level ℓ, where $\ell \geq h(x)$. The solution works by precomputing answers to queries. We apply this solution later on to solve some of the cases that we split a sorted range selection query into.

Let x be a fixed node, and let y and K be fixed integer thresholds. We now describe how to support queries for the k smallest elements in sorted order of $\mathcal{R}(x, \ell)$ (or $\mathcal{L}(x, \ell)$) where $h(x) \leq \ell \leq y$ and $k \leq K$. We precompute the answer to all queries that satisfy the constraints set forth by K and y by storing two arrays R_x and L_x for the node x. In $R_x[\ell]$, we store the indices of the K smallest leaves in sorted order of $\mathcal{R}(x, \ell)$. The array L_x is defined symmetrically. We summarize this solution in a lemma, where we also discuss the space usage and how to represent indices of leaves.

Lemma 2.3 *For a fixed node x and fixed parameters y and K, where $y \geq h(x)$, we can store R_x and L_x using $O(Ky^2 + w)$ bits of space. Queries for the k smallest elements in sorted order in $\mathcal{R}(x, \ell)$ (or $\mathcal{L}(x, \ell)$) can be supported in time $O(k)$ provided $k \leq K$ and $h(x) \leq \ell \leq y$.*

Proof. By storing indices relative to the index of the rightmost leaf in $\mathcal{T}_x$, we only need to store y bits per element in R_x and L_x. We can store the two arrays R_x and L_x with a space usage of $O(Ky^2 + w)$ bits using Lemma 2.2. When reading an entry, we can add the index of the rightmost leaf in $\mathcal{T}_x$ in $O(1)$ time. The k smallest elements in $\mathcal{R}(x, \ell)$ can be reported by returning the k first entries in $R_x[\ell]$ (and similarly for $L_x[\ell]$). □

2.4.3 Solution for $k \leq \lfloor \log n/(2 \log\log n)^2 \rfloor$

In this section, we show how to answer queries for $k \leq \lfloor \log n/(2 \log\log n)^2 \rfloor$. Having discussed how to decompose a query into bottom and top queries in 2.4.1, and how to answer queries by storing precomputed answers in 2.4.2, this case is now simple to explain.

Theorem 2.2 *For $k \leq \lfloor \log n/(2 \log\log n)^2 \rfloor$, we can answer sorted range selection queries in $O(k)$ time using $O(n)$ words of space.*

Proof. Following the idea of decomposing queries, we cut $\mathcal{T}$ at level $2\lfloor \log\log n \rfloor$. A bottom query is solved using the construction in Lemma 2.3 with $K = \lfloor \log n/(2 \log\log n)^2 \rfloor$ and $y = 2\lfloor \log\log n \rfloor$. The choice of parameters is justified by the fact that we cut $\mathcal{T}$ at level $2\lfloor \log\log n \rfloor$, and by assumption $k \leq \lfloor \log n/(2 \log\log n)^2 \rfloor$. As a bottom query can be on any of the n leaves, we must store arrays L_x and R_x for each leaf as described in Lemma 2.3. All R_x structures are stored in one single array which is indexed by a leaf x. Using Lemma 2.3 the space usage for all R_x becomes $O(n(w + \lfloor \log n/(2 \log\log n)^2 \rfloor (2\lfloor \log\log n \rfloor)^2) = O(n(w + \log n)) = O(nw)$ bits (and similarly for L_x).

For the top query, we for all nodes x at level $2\lfloor \log\log n \rfloor$ use the same construction with $K = \left\lfloor \log n/(2\log\log n)^2 \right\rfloor$ and $y = \log n$. As we only have $n/2^{2\lfloor \log\log n \rfloor} = \Theta(n/(\log n)^2)$ nodes at level $2\lfloor \log\log n \rfloor$, the space usage becomes $O(\frac{n}{(\log n)^2}(w + \left\lfloor \log n/(2\log\log n)^2 \right\rfloor (\log n)^2)) = O(n(w + \log n)) = O(nw)$ bits (as before we store all the R_x structures in one single array, which is indexed by a node x, and similarly for L_x). For both query types the $O(k)$ time bound follows from Lemma 2.3. □

2.4.4 Solution for $k > \left\lfloor \log n/(2\log\log n)^2 \right\rfloor$

In this case, we build $O(\log\log n)$ different structures each handling some range of the query parameter k. The r'th structure is used to answer queries for $2^{2^r} < k \leq 2^{2^{r+1}}$. Note that no structure is required for r satisfying $2^{2^{r+1}} \leq \left\lfloor \log n/(2\log\log n)^2 \right\rfloor$ since this is handled by the case $k \leq \left\lfloor \log n/(2\log\log n)^2 \right\rfloor$ as described in Section 2.4.3.

The r'th structure uses $O(w + n(2^r + w/2^r))$ bits of space, and supports sorted range selection queries in $O(2^{2^r} + k)$ time for $k \leq 2^{2^{r+1}}$. The total space usage of the $O(\log\log n)$ structures becomes $O(w\log\log n + n\log n + nw)$ bits, i.e. $O(n)$ words, since $r \leq \lceil \log\log n \rceil$. Given a sorted range selection query, we find the right structure by either table lookup, computing r directly or using linear search for the right r. In all cases, finding the correct structure can be done in $o(k)$ time. Finally, we query the r'th structure in $O(2^{2^r} + k) = O(k)$ time, since $2^{2^r} < k$.

In the r'th structure, we cut $\mathcal{T}$ at level 2^r and again at level $7 \cdot 2^r$. By generalizing the idea of decomposing queries as explained in Section 2.4.1, we split the original sorted range selection query into three types of queries, namely *bottom*, *middle* and *top* queries. We define u' as the ancestor of u at level 2^r and u'' as the ancestor of u at level $7 \cdot 2^r$. We define v' and v'' in the same way for v. When the level of $w = \mathrm{LCA}(u, v)$ is at a level $> 7 \cdot 2^r$, we see that the query range (i.e. all the leaves strictly between the leaves u and v) is equal to

$$\mathcal{R}(u, 2^r) \cup \mathcal{R}(u', 7 \cdot 2^r) \cup \mathcal{R}(u'', h(w) - 1) \cup \mathcal{L}(v'', h(w) - 1) \cup \mathcal{L}(v', 7 \cdot 2^r) \cup \mathcal{L}(v, 2^r) \ .$$

In the case that w is below or at level $7 \cdot 2^r$, we can use the decomposition exactly as described in Section 2.4.1. In the following we focus on describing how to support each type of query in $O(2^{2^r} + k)$ time.

Bottom query A bottom query is a query on a leaf u for $\mathcal{R}(u, \ell)$ (or $\mathcal{L}(u, \ell)$) where $\ell \leq 2^r$. For all nodes x at level 2^r, we store an array S_x containing the canonical subset C_x in sorted order. Using Lemma 2.1 we can store the S_x arrays for all x using $O(n2^r + w)$ bits as each leaf can be indexed with 2^r bits (relative to the leftmost leaf in $\mathcal{T}_x$). Now, to answer a bottom query we make a linear pass through the array $S_{\mathrm{anc}(u,2^r)}$ discarding elements that are not within the query range. We stop once we have k elements, or we have no more elements left in the array. This takes $O(2^{2^r} + k)$ time.

Top query A top query is a query on a node x at level $7 \cdot 2^r$ for $\mathcal{R}(x, \ell)$ (or $\mathcal{L}(x, \ell)$) where $7 \cdot 2^r < \ell \leq \log n$. We use the construction in Lemma 2.3 with $K = 2^{2^{r+1}}$ and $y = \log n$. We have $\frac{n}{2^{7 \cdot 2^r}}$ nodes at level $7 \cdot 2^r$, so to store all structures at this level the total number of bits

of space used is

$$\begin{aligned} & O\left(\frac{n}{2^{7\cdot 2^r}}(w+2^{2^{r+1}}(\log n)^2)\right) \\ = \;& O\left(n\frac{w}{2^r}+\frac{n}{2^{5\cdot 2^r}}(\log n)^2\right) \\ = \;& O\left(n\frac{w}{2^r}+\frac{n}{\left\lfloor \log n/(2\log\log n)^2\right\rfloor^{5/2}}(\log n)^2\right) \\ = \;& O\left(n\frac{w}{2^r}\right)\,, \end{aligned}$$

where we used that $\left\lfloor \log n/(2\log\log n)^2\right\rfloor < k \leq 2^{2^{r+1}}$. By Lemma 2.3 a top query takes $O(k)$ time.

Middle query A middle query is a query on a node z at level 2^r for $\mathcal{R}(z,\ell)$ (or $\mathcal{L}(z,\ell)$) with $2^r < \ell \leq 7\cdot 2^r$. For all nodes x at level 2^r, let $\min_x = \min C_x$. The idea in answering middle queries is as follows. Suppose we could find the nodes at level 2^r corresponding to the up to k smallest $\min_x$ values within the query range. To answer a middle query, we would only need to extract the k overall smallest elements from the up to k corresponding sorted S_x arrays of the nodes, we just found. The insight is that both subproblems mentioned can be solved using Theorem 2.1 as the key part. Once we have the k smallest elements in the middle query range, all that remains is to sort them.

We describe a solution in line with the above idea. For each node x at levels 2^r to $7\cdot 2^r$, we have a sorted array $\mathcal{M}^r_x$ of all nodes x' at level 2^r in $\mathcal{T}_x$ sorted with respect to the $\min_{x'}$ values. To store the $\mathcal{M}^r_x$ arrays for all x, the space required is $O(\frac{n}{2^{2^r}}\cdot 6\cdot 2^r) = O(\frac{n}{2^r})$ words (i.e. $O(n\frac{w}{2^r})$ bits), since we have $\frac{n}{2^{2^r}}$ nodes at level 2^r, and each such node will appear $7\cdot 2^r - 2^r = 6\cdot 2^r$ times in an $\mathcal{M}^r_x$ array (and to store the index of a node we use a word).

To answer a middle query for the k smallest elements in $\mathcal{R}(z,\ell)$, we walk $\ell - 2^r$ levels up from z while collecting the $\mathcal{M}^r_x$ arrays for the nodes x whose canonical subset is a part of the query range (at most $6\cdot 2^r$ arrays since we collect at most one per level). Using Theorem 2.1 we select the k smallest elements from the $O(2^r)$ sorted arrays in $O(2^r+k) = O(k)$ time (note that there may not be k elements to select, so in reality we select up to k elements). This gives us the k smallest $\min_{x'}$ values of the nodes $x'_1, x'_2, \ldots, x'_k$ at level 2^r that are within the query range. Finally, we select the k overall smallest elements of the sorted arrays $S_{x'_1}, S_{x'_2}, \ldots, S_{x'_k}$ in $O(k)$ time using Theorem 2.1. This gives us the k smallest elements of $\mathcal{R}(z,\ell)$, but not in sorted order. We now show how to sort these elements in $O(k)$ time.

For every leaf u, we store its local rank relative to $C_{u''}$, where u'' is the the ancestor of u at level $7\cdot 2^r$. Since each subtree $\mathcal{T}_{u''}$ contains $2^{7\cdot 2^r}$ leaves, we need $7\cdot 2^r$ bits to index a leaf (relative to the leftmost leaf in $\mathcal{T}_{u''}$). We store all local rank labels of length $7\cdot 2^r$ in a single array, and using Lemma 2.1 the space usage becomes $O(n2^r + w)$ bits. Given $O(k)$ leaves from C_x for a node x at level $7\cdot 2^r$, we can use the local rank labellings of the leaves of length $7\cdot 2^r$ bits to radix sort them in $O(k)$ time (using 7 passes, and for the analysis we use that $2^{2^r} < k$). This completes how to support queries.

2.4.5 Construction

In this section, we show how to build the data structures in Sections 2.4.3 and 2.4.4 in $O(n\log n)$ time using $O(n)$ extra words of space during the construction.

The structures to be created for node x are a subset of the possible structures S_x, $\mathcal{M}_x^r$, $R_x[\ell]$, $L_x[\ell]$ (where ℓ is a level above x), and the local rank labellings. In total, the structures to be created store $O(n\frac{\log n}{\log\log n})$ elements which is dominated by the number of elements stored in the R_x and L_x structures for all leaves in Section 2.4.3.

The general idea in the construction is to perform mergesort bottom up on $\mathcal{T}$ (level-by-level) starting at the leaves. The time spent on mergesort is $O(n\log n)$, and we use $O(n)$ words of space for the mergesort as we only store the sorted lists for the current and previous level. Note that when visiting a node x during mergesort the set C_x has been sorted, i.e. we have computed the array S_x. The structures S_x and $\mathcal{M}_x^r$ will be constructed while visiting x during the traversal of $\mathcal{T}$, while $R_x[\ell]$ and $L_x[\ell]$ will be constructed at the ancestor of x at level ℓ. As soon as a set has been computed, we store it in the data structure, possibly in a packed manner.

For the structures in Section 2.4.3, when visiting a node x at level $\ell \leq 2\lfloor\log\log n\rfloor$ we compute for each leaf z in the right subtree of x the structure $R_z[\ell] = R_z[\ell-1]$ (where $R_z[0] = \emptyset$), and the structure $L_z[\ell]$ containing the (up to) $\lfloor \log n/(2\log\log n)^2 \rfloor$ smallest elements in sorted order of $L_z[\ell-1] \cup S_{2x}$. Both structures can be computed in time $O(\lfloor \log n/(2\log\log n)^2 \rfloor)$. Symmetrically, we compute the same structures for all leaves z in the left subtree of x.

In the case that x is at level $\ell > 2\lfloor\log\log n\rfloor$, we compute for each node z at level $2\lfloor\log\log n\rfloor$ in the right subtree of x the structure $R_z[\ell] = R_z[\ell-1]$ (where $R_z[2\lfloor\log\log n\rfloor] = \emptyset$), and the structure $L_z[\ell]$ containing the $\lfloor \log n/(2\log\log n)^2 \rfloor$ smallest elements in sorted order of $L_z[\ell-1] \cup S_{2x}$. Both structures can be computed in time $O(\lfloor \log n/(2\log\log n)^2 \rfloor)$. Symmetrically, we compute the same structures for all nodes z at level $2\lfloor\log\log n\rfloor$ in the left subtree of x.

For the structures in Section 2.4.4, when visiting a node x we first decide in $O(\log\log n)$ time if we need to compute any structures at x for any r.

In the case that x is a node at level 2^r, we store $S_x = C_x$ and $\mathcal{M}_x^r = \min C_x$. For x at level $2^r < \ell \leq 7\cdot 2^r$ we store $\mathcal{M}_x^r = \mathcal{M}_{2x}^r \cup \mathcal{M}_{2x+1}^r$. This can be computed in time linear in the size of $\mathcal{M}_x^r$. In the case that x is a node at level $7\cdot 2^r$, we store the local rank labelling for each leaf in $\mathcal{T}_x$ using the sorted C_x list.

For x at level $\ell > 7\cdot 2^r$, we compute for each z at level $7\cdot 2^r$ in the right subtree of x the structure $R_z[\ell] = R_z[\ell-1]$ (where $R_z[7\cdot 2^r] = \emptyset$), and the structure $L_z[\ell]$ containing the $2^{2^{r+1}}$ smallest elements in sorted order of $L_z[\ell-1] \cup S_{2x}$. Both structures can be computed in time $O(2^{2^{r+1}})$. Symmetrically, we compute the same structures for all nodes z at level $7\cdot 2^r$ in the left subtree of x.

Since all structures can be computed in time linear in the size and that we have $O(n\frac{\log n}{\log\log n})$ elements in total, the overall construction time becomes $O(n\log n)$.

2.5 Online sorted range reporting

In this section we describe how to extend the solution for the sorted range selection problem from Section 2.4 to a solution for the online sorted range reporting problem.

We solve the problem by performing a sequence of sorted range selection queries Q_y with indices i and j and $k = 2^y$ for $y = 0, 1, 2, \ldots$. The initial query to the range $A[i..j]$ is Q_0. Each time we report an element from the current query Q_y, we spend $O(1)$ time building part of the next query Q_{y+1} so that when we have exhausted Q_y, we will have finished building Q_{y+1}. Since we report the 2^{y-1} largest elements in Q_y (the 2^{y-1} smallest are reported for

$Q_0, Q_1, \ldots, Q_{y-1}$), we can distribute the $O(2^{y+1})$ computation time of Q_{y+1} over the 2^{y-1} reportings from Q_y. Hence the query time becomes $O(1)$ worst-case per element reported.

Chapter 3

Approximating the Mode

3.1 Introduction

In this chapter we consider the c-approximate range mode problem, and the range k-frequency problem. The frequency of a label l in a multiset S of labels, is the number of occurrences of l in S. The mode, M, of S is the most frequent label in S. In case of ties, any of the most frequent labels in S can be designated the mode.

The input to the following problems is an array A of length n containing labels. In the range mode problem, we must preprocess A into a data structure that given indices i and j, $1 \leq i \leq j \leq n$, returns the mode, $M_{i,j}$, in the subarray $A[i,j] = A[i], A[i+1], \ldots, A[j]$. We let $F_{i,j}$ denote the frequency of $M_{i,j}$ in $A[i,j]$. In the c-approximate range mode problem, a query is given indices i and j, $1 \leq i \leq j \leq n$, and returns a label that has a frequency that is at most a factor c from $F_{i,j}$. In the range k-frequency problem, a query is given indices i and j, $1 \leq i \leq j \leq n$, and returns whether there is a label occurring precisely k times in $A[i,j]$.

In this chapter we use the unit cost RAM, and let $w = \Theta(\log n)$ denote the word size. Furthermore, we use that $\frac{1}{\log(1+\varepsilon)} = O(\frac{1}{\varepsilon})$ for any $0 < \varepsilon \leq 1$.

Previous Results. The first data structure for the range mode problem achieving constant query time was developed in [28]. This data structure uses $O(n^2 \log\log n / \log n)$ words of space. This was subsequently improved to $O(n^2/\log n)$ words of space in [34] and finally to $O(n^2 \log\log n / \log^2 n)$ in [35].

For non-constant query time, the first developed data structure uses $O(n^{2-2\varepsilon})$ space and answers queries in $O(n^\varepsilon \log n)$ time, where $0 < \varepsilon \leq \frac{1}{2}$ is a query-space tradeoff constant [28]. The query time was later improved to $O(n^\varepsilon)$ without changing the space bound [34].

Given the rather large space bounds for the range mode problem, the approximate variant of the problem was considered in [7]. With constant query time, they solve 2-approximate range mode with $O(n \log n)$ space, 3-approximate range mode with $O(n \log\log n)$ space, and 4-approximate range mode with linear space. For $(1+\varepsilon)$-approximate range mode, they describe a data structure that uses $O(\frac{n}{\varepsilon})$ space and answers queries in $O(\log\log_{(1+\varepsilon)} n) = O(\log\log n + \log\frac{1}{\varepsilon})$ time. This data structure gives a linear space solution with $O(\log\log n)$ query time for c-approximate range mode when c is constant.

Our Results. In Section 3.2 we present a simple data structure for the 3-approximate range mode problem. The data structure uses linear space and answers queries in constant

time. This improves the best previous 3-approximate range mode data structures by a factor $O(\log\log n)$ either in space or query time. With linear space and constant query time, the best previous approximation factor was 4. In Section 3.3 we use our 3-approximate range mode data structure, to develop a data structure for $(1+\varepsilon)$-approximate range mode. This data structure uses $O(\frac{n}{\varepsilon})$ space and answers queries in $O(\log\frac{1}{\varepsilon})$ time. This removes the dependency on n in the query time compared to the previously best data structure, while matching the space bound. Thus, we have a linear space data structure with constant query time for the c-approximate range mode problem for any constant $c > 1$. We note that we get the same bound if we build on the 4-approximate range mode data structure from [7].

In Section 3.4 we consider the range k-frequency problem. To the best of our knowledge, we are the first to consider this problem. Here we show that 2D rectangle stabbing reduces to range k-frequency for any constant $k > 1$. This reduction gives a lower bound of $\Omega(\log n/\log\log n)$ query time for any range k-frequency data structure that uses $O(n\log^{O(1)} n)$ space [32, 33], for any constant $k > 1$. Secondly, we reduce range k-frequency to 2D rectangle stabbing. This reduction works for any k. This immediately gives a data structure for range k-frequency that uses linear space, and answers queries in optimal $O(\log n/\log\log n)$ time [26] (we note that 2D rectangle stabbing reduces to 2D range counting).

Finally we consider the restricted case where $k = 1$. This problem corresponds to determining whether there is a unique label in a subarray. The reduction from 2D rectangle stabbing only applies for $k > 1$, thus the lower bound breaks down in this restricted case. We show, somewhat surprisingly, that determining whether there is a label occurring exactly twice (or $k > 1$ times) in a subarray, is exponentially harder than determining if there is a label occurring exactly once. Specifically, we reduce range 1-frequency to four-sided 3D orthogonal range emptiness, which can be solved with $O(\log^2\log n)$ query time and $O(n\log n)$ space by a slight modification of the data structure presented in [1].

3.2 3-Approximate Range Mode

In this section, we construct a data structure that given a range $[i, j]$ computes a 3-approximation of $F_{i,j}$.

We use the following observation from [7]. If we can cover $A[i, j]$ with three disjoint subintervals $A[i, x], A[x+1, y]$ and $A[y+1, j]$ for which we know $F_{i,x}, F_{x+1,y}$ and $F_{y+1,j}$, then

$$\tfrac{1}{3}F_{i,j} \leq \max\{F_{i,x}, F_{x+1,y}, F_{y+1,j}\} \leq F_{i,j}.$$

First, we describe a data structure that uses $O(n\log\log n)$ space, and then we show how to reduce the space to $O(n)$. The data structure consists of a tree T of polynomial fanout where the i'th leaf stores $A[i]$, for $i = 1, \ldots, n$. For a node v let T_v denote the subtree rooted at v and let $|T_v|$ denote the number of leaves in T_v. The fanout of node v is $f_v = \lceil\sqrt{|T_v|}\rceil$. The height of T is $\Theta(\log\log n)$. Along with T, we store a lowest common ancestor (LCA) data structure, which given indices i and j, finds the LCA of the leaves corresponding to i and j in T in constant time [23, 6].

For every node $v \in T$, let $R_v = A[a, b]$ denote the consecutive range of entries stored in the leaves of T_v. The children $c_1, \ldots, c_{f_v}$ of v partition R_v into f_v disjoint subranges $R_{c_1} = A[a_{c_1}, b_{c_1}]$, $\ldots$, $R_{c_{f_v}} = A[a_{c_{f_v}}, b_{c_{f_v}}]$ each of size $O(\sqrt{|T_v|})$. For every pair of children c_r and c_s where $r < s-1$, we store $F_{a_{c_{r+1}}, b_{c_{s-1}}}$. Furthermore, for every child range R_{c_i} we store $F_{a_{c_i},k}$ and $F_{k,b_{c_i}}$ for every prefix and suffix range of R_{c_i} respectively. To compute a

3-approximation of $F_{i,j}$, we find the LCA of i and j. This is the node v in T for which i and j lie in different child subtrees, say T_{c_x} and T_{c_y} with ranges $R_{c_x} = [a_{c_x}, b_{c_x}]$ and $R_{c_y} = [a_{c_y}, b_{c_y}]$. We then lookup the frequency $F_{a_{c_{x+1}}, b_{c_{y-1}}}$ stored for the pair of children c_x and c_y, as well as the suffix frequency $F_{i,b_{c_x}}$ stored for the range $A[i, b_{c_x}]$ and the prefix frequency $F_{a_{c_y},j}$ stored for $A[a_{c_y}, j]$, and return the max of these.

Each node $v \in T$ uses $O(|T_v|)$ space for the frequencies stored for each of the $O(|T_v|)$ pairs of children, and $O(|T_v|)$ for all the prefix and suffix range frequencies. Since each node v uses $O(|T_v|)$ space and the LCA data structure uses $O(n)$ space, our data structure uses $O(n \log\log n)$ space. A query makes one LCA query and computes the max of three numbers which takes constant time.

We just need one observation to bring the space down to $O(n)$. Consider a node $v \in T$. The largest possible frequency that can be stored for any pair of children of v, or for any prefix or suffix range of a child of v is $|T_v|$, and each such frequency can be represented by $b = 1 + \lfloor \log |T_v| \rfloor$ bits. We divide the frequencies stored in v into chunks of size $\lfloor \frac{\log n}{b} \rfloor$ and pack each of them in one word. This reduces the total space usage of the nodes on level i to $O(n/2^i)$. We conclude that the data structure uses $O(n)$ words space and supports queries in constant time.

Theorem 3.2.1 *There exists a data structure for the* 3*-approximate range mode problem that uses $O(n)$ words of space and supports queries in constant time.*

3.3 $(1+\varepsilon)$-Approximate Range Mode

In this section, we describe a data structure using $O(\frac{n}{\varepsilon})$ space that given a range $[i, j]$, computes a $(1+\varepsilon)$-approximation of $F_{i,j}$ in $O(\log \frac{1}{\varepsilon})$ time.

Our data structure consists of two parts. The first part solves all queries $[i, j]$ where $F_{i,j} \leq \lceil \frac{1}{\varepsilon} \rceil$, and the latter solves the remaining. The first data structure also decides whether $F_{i,j} \leq \lceil \frac{1}{\varepsilon} \rceil$.

Small Frequencies For $i = 1, \ldots, n$ we store a table, Q_i, of length $\lceil \frac{1}{\varepsilon} \rceil$, where the value in $Q_i[k]$ is the largest integer $j \geq i$ such that $F_{i,j} = k$. To answer a query $[i, j]$ we do a successor search for j in Q_i. If j does not have a successor in Q_i then $F_{i,j} > \lceil \frac{1}{\varepsilon} \rceil$, and we query the second data structure. Otherwise, let s be the index of the successor of j in Q_i, then $F_{i,j} = s$. The data structure uses $O(\frac{n}{\varepsilon})$ space and supports queries in $O(\log \frac{1}{\varepsilon})$ time.

Large Frequencies For every index $1 \leq i \leq n$, define a list T_i of length $t = \lceil \log_{1+\varepsilon}(\varepsilon n) \rceil$, with the following invariant: For all j, if $T_i[k-1] < j \leq T_i[k]$ then $\lceil \frac{1}{\varepsilon}(1+\varepsilon)^k \rceil$ is a $(1+\varepsilon)$-approximation of $F_{i,j}$. The following assignment of values to the lists T_i satisfies this invariant:

Let $m(i,k)$ be the largest integer $j \geq i$ such that $F_{i,j} \leq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{k+1} \rceil - 1$. For T_1 we set $T_1[k] = m(1,k)$ for all $k = 1, \ldots, t$. For the remaining T_i we set

$$T_i[k] = \begin{cases} T_{i-1}[k] & \text{if } F_{i,T_{i-1}[k]} \geq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^k \rceil + 1 \\ m(i,k) & \text{otherwise} \end{cases}$$

The n lists are sorted by construction. For T_1, it is true since $m(i,k)$ is increasing in k. For T_i, it follows that $F_{i,T_i[k]} \leq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{k+1} \rceil - 1 < F_{i,T_i[k+1]}$, and thus $T_i[k] < T_i[k+1]$ for any k.

Let s be the index of the successor of j in T_i. We know that $F_{i,T_i[s]} \leq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{s+1} \rceil - 1$, $F_{i,T_i[s-1]} \geq \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{s-1} \rceil + 1$ and $T_i[s-1] < j \leq T_i[s]$. It follows that

$$\lceil \tfrac{1}{\varepsilon}(1+\varepsilon)^{s-1} \rceil + 1 \leq F_{i,j} \leq \lceil \tfrac{1}{\varepsilon}(1+\varepsilon)^{s+1} \rceil - 1 \tag{3.1}$$

and that $\lceil \frac{1}{\varepsilon}(1+\varepsilon)^{s} \rceil$ is a $(1+\varepsilon)$-approximation of $F_{i,j}$.

The second important property of the n lists, is that they only store $O(\frac{n}{\varepsilon})$ different indices, which allows for a space-efficient representation. If $T_{i-1}[k] \neq T_i[k]$ then the following $\lceil \frac{1}{\varepsilon}(1+\varepsilon)^{k+1} \rceil - 1 - \lceil \frac{1}{\varepsilon}(1+\varepsilon)^{k} \rceil - 1 \geq \lfloor (1+\varepsilon)^k \rfloor - 3$ entries, $T_{i+a}[k]$ for $a = 1, \ldots, \lfloor (1+\varepsilon)^k \rfloor - 3$, are not changed, hence we store the same index at least $\max\{1, \lfloor (1+\varepsilon)^k \rfloor - 2\}$ times. Therefore, the number of changes to the n lists, starting with T_1, is bounded by

$$\sum_{k=1}^{t} \frac{n}{\max\{1, \lfloor (1+\varepsilon)^k \rfloor - 2\}} = O(\tfrac{n}{\varepsilon}) \ .$$

This was observed in [7], where similar lists are maintained in a partially persistent search tree [11]. This data structure uses $O(\frac{n}{\varepsilon})$ space and supports queries in $O(\log\log_{1+\varepsilon} n)$ time.

We maintain these lists without persistence such that we can access any entry in any list T_i in constant time. Let $I = \{1, 1+t, \ldots, 1+\lfloor (n-1)/t \rfloor t\}$. For every $\ell \in I$ we store T_ℓ explicitly as an array S_ℓ. Secondly, for $\ell \in I$ and $k = 1, \ldots, \lceil \log_{1+\varepsilon} t \rceil$ we define a bit vector $B_{\ell,k}$ of length t and a change list $C_{\ell,k}$, where

$$B_{\ell,k}[a] = \begin{cases} 0 & \text{if } T_{\ell+a-1}[k] = T_{\ell+a}[k] \\ 1 & \text{otherwise} \end{cases}$$

Given a bit vector L, define $\mathrm{sel}(L, b)$ as the index of the b'th one in L. We set

$$C_{\ell,k}[a] = T_{i+\mathrm{sel}(B_{i,k},a)}[k] \ .$$

Finally, for every $\ell \in I$ and for $k = 1 + \lceil \log_{1+\varepsilon} t \rceil, \ldots, t$ we store $D_\ell[k]$ which is the smallest integer $z > \ell$ such that $T_z[k] \neq T_\ell[k]$. We also store $E_\ell[k] = T_{D_\ell[k]}[k]$. We store each bit vector in a rank and select data structure [25] that uses $O(\frac{n}{w})$ space for a bit vector of length n, and supports *rank*(i) in constant time. A rank(i) query returns the number of ones in the first i bits of the input.

Each change list, $C_{l,k}$ and every D_ℓ and E_ℓ list is stored as an array. The bit vectors indicate at which indices the contents of the first $\lceil \log_{1+\varepsilon} t \rceil$ entries of $T_\ell, \ldots, T_{\ell+t-1}$ change, and the change lists store what the entries change to. The D_ℓ and E_ℓ arrays do the same thing for the last $t - \lceil \log_{1+\varepsilon} t \rceil$ entries, exploiting that these entries change at most once in an interval of length t.

Observe that the arrays, $C_{\ell,k}$, $D_\ell[k]$ and $E_\ell[k]$, and the bit vectors, $B_{\ell,k}$ allow us to retrieve the contents of any entry, $T_i[k]$ for any i, k, in constant time as follows. Let $\ell = \lfloor i/t \rfloor t$. If $k > \lceil \log_{1+\varepsilon} t \rceil$ we check if $D_\ell[k] \leq i$, and if so we return $E_\ell[k]$, otherwise we return $S_\ell[k]$. If $k \leq \lceil \log_{1+\varepsilon} t \rceil$, we determine $r = \mathrm{rank}(i - \ell)$ in $B_{\ell,k}$ using the rank and select data structure. We then return $C_{\ell,k}[r]$ unless $r = 0$ in which case we return $S_\ell[k]$.

We argue that this correctly returns $T_i[k]$. In the case where $k > \lceil \log_{1+\varepsilon} t \rceil$, comparing $D_\ell[k]$ to i indicates whether $T_i[k]$ is different from $T_\ell[k]$. Since $T_z[k]$ for $z = \ell, \ldots, i$ can only change once, $T_i[k] = E_\ell[k]$ in this case. Otherwise, $S_\ell[k] = T_\ell[k] = T_i[k]$. If $k \leq \lceil \log_{1+\varepsilon} t \rceil$, the rank r of $i - \ell$ in $B_{\ell,k}$, is the number of changes that has occurred in the k'th entry from list

T_ℓ to T_i. Since $C_{\ell,k}[r]$ stores the value of the k'th entry after the r'th change, $C_{\ell,k}[r] = T_i[k]$, unless $r = 0$ in which case $T_i[k] = S_\ell[k]$.

The space used by the data structure is $O(\frac{n}{\varepsilon})$. We store $3\lceil\frac{n}{t}\rceil$ arrays, S_ℓ, D_ℓ and E_ℓ for $\ell \in I$, each using t space, in total $O(n)$. The total size of the change lists, $C_{\ell,k}$, is bounded by the number of changes across the T_i lists, which is $O(\frac{n}{\varepsilon})$ by the arguments above. Finally, the rank and select data structures, $B_{\ell,k}$, each occupy $O(\frac{t}{w}) = O(\frac{t}{\log n})$ words, and we store a total of $\lceil\frac{n}{t}\rceil\lceil\log_{1+\varepsilon} t\rceil$ such structures, thus the total space used by these is bounded by

$$\begin{aligned} & O\left(\frac{t}{\log n}\frac{n}{t}\log_{1+\varepsilon} t\right) \\ = \; & O\left(n\frac{\log_{1+\varepsilon} t}{\log n}\right) = O\left(\frac{n}{\varepsilon}\frac{\log t}{\log n}\right) \\ = \; & O\left(\frac{n}{\varepsilon}\frac{\log\frac{\log(\varepsilon n)}{\varepsilon}}{\log n}\right) = O\left(\frac{n}{\varepsilon}\frac{\log(n\log(\varepsilon n))}{\log n}\right) = O\left(\frac{n}{\varepsilon}\right) . \end{aligned}$$

In the last line we used that if $\lceil\frac{1}{\varepsilon}\rceil \geq n$ then we only store the small frequency data structure. We conclude that our data structures uses $O\left(\frac{n}{\varepsilon}\right)$ space.

To answer a query $[i, j]$, we first compute a 3-approximation of $F_{i,j}$ in constant time using the data structure from Section 3.2. Thus, we find $f_{i,j}$ satisfying $f_{i,j} \leq F_{i,j} \leq 3f_{i,j}$. Choose k such that $\lceil\frac{1}{\varepsilon}(1+\varepsilon)^k\rceil + 1 \leq f_{i,j} \leq \lceil\frac{1}{\varepsilon}(1+\varepsilon)^{k+1}\rceil - 1$ then the successor of j in T_i must be in one of the entries, $T_i[k], \ldots, T_i[k+O(\log_{1+\varepsilon} 3)]$. As stated earlier, the values of T_i are sorted in increasing order, and we find the successor of j using a binary search on an interval of length $O(\log_{1+\varepsilon} 3)$. Since each access to T_i takes constant time, we use $O(\log\log_{1+\varepsilon} 3) = O(\log\frac{1}{\varepsilon})$ time.

Theorem 3.3.1 *There exists a data structure for $(1+\varepsilon)$-approximate range mode that uses $O(\frac{n}{\varepsilon})$ space and supports queries in $O(\log\frac{1}{\varepsilon})$ time.*

The careful reader may have noticed that our data structure returns a frequency, and not a label that occurs approximately $F_{i,j}$ times. We can augment our data structure to return a label instead as follows.

We set $\varepsilon' = \sqrt{(1+\varepsilon)} - 1$, and construct our data structure from above. The *small frequency* data structure is augmented such that it stores the label $M_{i,Q_i[k]}$ along with $Q_i[k]$, and return this in a query. The *large frequency* data structure is augmented such that for every update of $T_i[k]$ we store the label that caused the update. Formally, let $a > 0$ be the first index such that $T_{i+a}[k] \neq T_i[k]$. Next to $T_i[k]$ we store the label $L_i[k] = A[i+a-1]$. In a query, $[i, j]$, let s be the index of the successor of j in T_i computed as above. If $s > 1$ we return the label $L_i[s-1]$, and if $s = 1$ we return $M_{i,Q_i[\lceil 1/\varepsilon'\rceil]}$, which is stored in the small frequency data structure.

In the case where $s = 1$ we know that $\lceil\frac{1}{\varepsilon'}\rceil \leq F_{i,j} \leq \lceil\frac{1}{\varepsilon'}(1+\varepsilon')^2\rceil - 1 = \lceil\frac{1}{\varepsilon'}(1+\varepsilon)\rceil - 1$ and we know that the frequency of $M_{i,Q_i[\lceil 1/\varepsilon'\rceil]}$ in $A[i, j]$ is at least $\lceil\frac{1}{\varepsilon'}\rceil$. We conclude that the frequency of $M_{i,Q_i[\lceil 1/\varepsilon'\rceil]}$ in $A[i, j]$ is a $(1+\varepsilon)$-approximation of $F_{i,j}$.

In the case where $s > 1$, we know that $\lceil\frac{1}{\varepsilon'}(1+\varepsilon')^{s-1}\rceil + 1 \leq F_{i,j} \leq \lceil\frac{1}{\varepsilon'}(1+\varepsilon')^{s+1}\rceil - 1$ by equation (3.1), and that the frequency, f_L, of the label $L_i[s-1]$ in $A[i, j]$ is at least $\lceil\frac{1}{\varepsilon'}(1+\varepsilon')^{s-1}\rceil + 1$. This means that $F_{i,j} \leq \frac{1}{\varepsilon'}(1+\varepsilon')^{s+1} \leq (1+\varepsilon')^2 f_L = (1+\varepsilon)f_L$, and we conclude that f_L is a $(1+\varepsilon)$-approximation of $F_{i,j}$.

The space needed for this data structure is $O(\frac{n}{\varepsilon'}) = O(\frac{n(\sqrt{1+\varepsilon}+1)}{\varepsilon}) = O(\frac{n}{\varepsilon})$, and a query takes $O(\log\frac{1}{\varepsilon'}) = O(\log\frac{1}{\varepsilon} + \log(\sqrt{1+\varepsilon}+1)) = O(\log\frac{1}{\varepsilon})$ time.

3.4 Range k-frequency

In this section, we consider the *range k-frequency* problem and its connection to classic geometric data structure problems. We show that the range k-frequency problem is equivalent to 2D rectangle stabbing for any fixed constant $k > 1$, and that for $k = 1$ the problem reduces to four-sided 3D orthogonal range emptiness.

In the 2D rectangle stabbing problem the input is n axis-parallel rectangles. A query is given a point, (x, y), and must return whether this point is contained[1] in at least one of the n rectangles in the input. A query lower bound of $\Omega(\log n/\log\log n)$ for data structures using $O(n\log^{O(1)} n)$ space is proved in [32], and a linear space static data structure with this query time can be found in [26].

In four-sided 3D orthogonal range emptiness, we are given a set P of n points in 3D, and must preprocess P into a data structure, such that given an open-ended four-sided rectangle $R = [-\infty, x] \times [y_1, y_2] \times [z, \infty]$, the data structure returns whether R contains a point $p \in P$. Currently, the best solution for this problem uses $O(n \log n)$ space and supports queries in $O(\log^2 \log n)$ time [1].

For simplicity, we assume that each coordinate is a unique integer between one and $2n$ (rank space).

Theorem 3.4.1 *The range k-frequency problem reduces to 2D rectangle stabbing.*

Proof. Let A be the input to the range k-frequency problem. We translate the ranges of A where there is a label with frequency k into $O(n)$ rectangles as follows. Fix a label $x \in A$, and let $s_x \geq k$ denote the number of occurrences of x in A. If $s_x < k$ then x is irrelevant and we discard it. Otherwise, let $i_1 < i_2 < \ldots < i_s$ be the position of x in A, and let $i_0 = 0$ and $i_{s+1} = n + 1$. Consider the ranges of A where x has frequency k. These are the subarrays, $A[a, b]$, where there exists an integer ℓ such that $i_\ell < a \leq i_{\ell+1}$ and $i_{\ell+k} \leq b < i_{\ell+k+1}$ for $0 \leq \ell \leq s_x - k$. This defines $s_x - k + 1$ two dimensional rectangles, $[i_\ell + 1, i_{\ell+1}] \times [i_{\ell+k}, i_{\ell+k+1} - 1]$ for $\ell = 0, \ldots, s_x - k$, such that x has frequency k in $A[i, j]$ if and only if the point (i, j) stabs one of the $s_x - k + 1$ rectangles defined by x. By translating the ranges of A where a label has frequency k into the corresponding rectangles for all distinct labels in A, we get a 2D rectangle stabbing instance with $O(n)$ rectangles. □

This means that we get a data structure for the range k-frequency problem that uses $O(n)$ space and supports queries in $O(\log n/\log\log n)$ time.

Theorem 3.4.2 *For $k = 1$, the range k-frequency problem reduces to four-sided orthogonal range emptiness queries in 3D.*

Proof. For each distinct label $x \in A$, we map the ranges of A where x has frequency one (it is unique in the range) to a 3D point. Let $i_1 < i_2 < \ldots < i_s$ be the positions of x in A, and let $i_0 = 0$ and $i_{s+1} = n+1$. The label x has frequency one in $A[a, b]$ if there exist an integer ℓ such that $i_{\ell-1} < a \leq i_\ell \leq b < i_{\ell+1}$. We define s points, $P_x = \{(i_{\ell-1} + 1, i_\ell, i_{\ell+1} - 1) \mid 1 \leq \ell \leq s\}$. The label x has frequency one in the range $A[a, b]$ if and only if the four-sided orthogonal range query $[-\infty, a] \times [a, b] \times [b, \infty]$ contains a point from P_x (we say that x is inside range $[x_1, x_2]$ if $x_1 \leq x \leq x_2$). Therefore, we let $P = \bigcup_{x \in A} P_x$ and get a four-sided 3D orthogonal range emptiness instance with $O(n)$ points. □

[1] points on the border of a rectangle are contained in the rectangle

Thus, we get a data structure for the range 1-frequency problem that uses $O(n \log n)$ space and supports queries in $O(\log^2 \log n)$ time.

Theorem 3.4.3 *Let k be a constant greater than one. The 2D rectangle stabbing problem reduces to the range k-frequency problem.*

Proof. We show the reduction for $k = 2$ and then generalize this construction to any constant value $k > 2$.

Let $R_1, \ldots, R_n$ be the input to the rectangle stabbing problem. We construct a range 2-frequency instance with n distinct labels each of which is duplicated exactly 6 times. Let R_ℓ be the rectangle $[x_{\ell_0}, x_{\ell_1}] \times [y_{\ell_0}, y_{\ell_1}]$. For each rectangle, R_ℓ, we add the pairs (x_{ℓ_0}, ℓ), (x_{ℓ_1}, ℓ) and (x_{ℓ_1}, ℓ) to a list X. Similarly, we add the pairs (y_{ℓ_0}, ℓ), (y_{ℓ_1}, ℓ), and (y_{ℓ_1}, ℓ) to a list Y. We sort X in descending order and Y in ascending order by their first coordinates. Since we assumed all coordinates are unique, the only ties are amongst pairs originating from the same rectangle, here we break the ties arbitrarily. The concatenation of X and Y is the range 2-frequency instance and we denote it A, i.e. the second component of each pair are the actual entries in A, and the first component of each pair is ignored.

We translate a 2D rectangle stabbing query, (x, y), into a query for the range 2-frequency instance as follows. Let p_x be the smallest index where the first coordinate of $X[p_x]$ is x, and let q_y be the largest index where the first coordinate of $Y[p_y]$ is y. If $A[p_x] = A[p_x + 1]$, two consecutive entries in A are defined by the right endpoint of the same rectangle, we set $i_x = p_x + 2$ (we move i_x to the right of the two entries), otherwise we set $i_x = p_x$. Similarly for the y coordinates, if $A[|X| + q_y] = A[|X| + q_y - 1]$ we set $j_y = q_y - 2$ (move j_y left of the two entries), otherwise we set $j_y = q_y$. Finally we translate (x, y) to the range 2-frequency query $[i_x, |X| + j_y]$ on A, see Figure 3.1. Notice that in the range 2-frequency queries that can be considered in the reduction, the frequency of a label is either one, two, three, four or six. The frequency of label ℓ in $A[i_x, |X|]$ is one if $x_{\ell_0} \leq x \leq x_{\ell_1}$, three if $x > x_{\ell_1}$ and zero otherwise. Similar, the frequency of ℓ in $A[|X| + 1, |X| + j_y]$ is one if $y_{\ell_0} \leq y \leq y_{\ell_1}$, three if $y > y_{\ell_1}$ and zero otherwise. We conclude that the point (x, y) stabs rectangle R_ℓ if and only if the label ℓ has frequency two in $A[i_x, |X| + j_y]$.

Since $x, y \in \{1, \ldots, 2n\}$, we can store a table with the translations from x to i_x and y to j_y. Thus, we can translate 2D rectangle stabbing queries to range 2-frequency queries in constant time.

For $k > 2$ we place $k - 2$ copies of each label between X and Y and translate the queries accordingly. □

We conclude that for data structures using $O(n \log^{O(1)} n)$ space, the range k-frequency problem is exponentially harder for $k > 1$ than for $k = 1$.

3.5 Summary and remarks

We have shown that using only linear space we can get any constant factor approximation of the mode in constant time. Secondly, we considered the range k-frequency problem and showed how this problem is strongly related to geometric data structure problems. We found matching upper and lower bounds for any constant $k > 1$, and showed that for $k = 1$ it is exponentially easier to solve for near linear space data structures. Unfortunately, we were not able to exploit this in our efforts to prove anything new regarding the range mode problem.

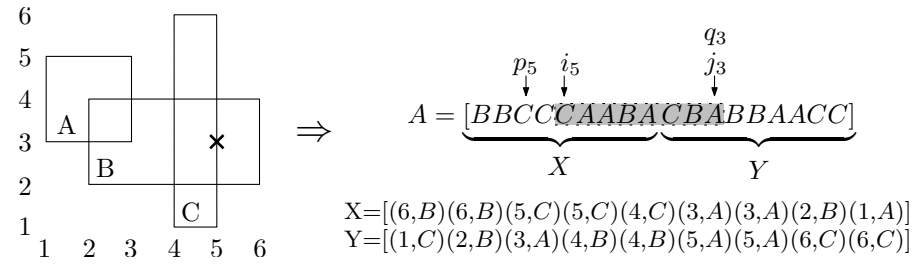


Figure 3.1: Reduction from 2D rectangle stabbing to range 2-frequency. The $\times$ marks a stabbing query, $(5, 3)$. This query is mapped to the range 2-frequency query $[i_5, |X| + j_3]$ in A, which is highlighted. Notice that $i_5 = p_5 + 2$ since $A[p_5] = A[p_5 + 1]$.

The range mode problem seems to be much harder than the range k-frequency problem, but we were not able to put any structure on the range mode problem that could be used to prove a lower bound as we did for the range k-frequency problem.

Chapter 4

Future Work

In this chapter my plans for the remaining two years of my PhD study are described. This is mainly about a number of open problems in connection with my previous work. I am planning on visiting a research institution abroad sometime around the middle of 2010.

4.1 Open Problems

Online Sorted Range Reporting in the I/O model

The motivation for studying this problem originated from its potential application in search engines. For this reason we were also mainly interested in obtaining solutions for the I/O model. While working on this problem we found it hard to develop non-trivial solutions in the I/O model, so we turned to the RAM model, and came up with the solution which was presented in Chapter 2.

We have recently turned our attention to this problem again in the I/O model, but so far we have not obtained results of any particular significance, and so I will omit describing them in detail. This is one of the problems that I plan to continue working on in my part B studies. In particular, we are interested in both lower and upper bounds for this problem. So far we have been looking a bit into lower bounds in terms of time-space tradeoffs. For instance we have been investigating the minimum number of I/Os needed for a query when we have linear space for the data structure, and the minimum space needed to get linear query time (i.e. $O(k/B)$ I/Os where k is the size of the output). Also note that by time I mean the number of I/Os needed to return the answer of a query. Upper bounds, i.e. solutions in the I/O model are also interesting for this problem, but unfortunately our solution in the RAM model is not easy to generalize to the I/O model. For instance, one of the major problems is that our solution in the RAM model returns the indices of the elements in the output, and also uses indirect addressing in several other places. This means that we need to use one I/O per output element in the worst case, which is not very appealing. In other words, we need to look into different approaches to attacking this problem. However, currently my focus is to work on the lower bounds, since this is the part where we have spent least time so far. Also, we are not aware of any research on this problem in the I/O model, so essentially most of the problems related to this are still open.

Range Mode Problem

Our initial goal for this problem was to get an $O(\text{polylog}(n))$ query time solution with $O(n\text{polylog}(n))$ words of space in the RAM model, but despite a lot of efforts this problem has yet resisted all our attacks. The current best upper bounds for the problem are described in Section 3.1. We have also attempted to prove some (fairly weak) lower bounds for the problem (i.e. a bound such as you need super-constant time for queries when you have $O(n\text{polylog}(n))$ words of space), but also without any success. Instead we were able to prove some lower bounds for a similar problem in Chapter 3. I plan to revisit this problem again during my part B studies. I am not yet convinced of the impossibility of a $O(n\text{polylog}(n))$ solution with $O(\text{polylog}(n))$ words of space despite the amount of time we have been working on this problem. I still plan to continue to look a bit into finding such a solution. Also, it would be interesting to perhaps find some connection between the range mode problem and the range k-frequency problem studied in Chapter 3 so that we can put a non-trivial lower bound on the range mode problem, even if the bound we would get is nowhere near the current best upper bound.

Bibliography

[1] Peyman Afshani. On dominance reporting in 3D. In *Proc. of the 16th Annual European Symposium on Algorithms*, pages 41–51, 2008.

[2] Stephen Alstrup, Cyril Gavoille, Haim Kaplan, and Theis Rauhe. Nearest common ancestors: A survey and a new algorithm for a distributed environment. *Theory of Computing Systems*, 37(3):441–456, 2004.

[3] Lars Arge, Vasilis Samoladas, and Jeffrey S. Vitter. On two-dimensional indexability and optimal range search indexing. In *Proceedings of the Eighteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '99)*, pages 346–357. Association for Computing Machinery, 1999.

[4] R. Baeza-Yates and B. Ribeiro Neto. *Modern Information Retrieval*. Addison-Wesley, 1999.

[5] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In Gaston H. Gonnet, Daniel Panario, and Alfredo Viola, editors, *LATIN*, volume 1776 of *Lecture Notes in Computer Science*, pages 88–94. Springer, 2000.

[6] Michael A. Bender and Martin Farach-Colton. The LCA problem revisited. In *Proc. of the 4th Latin American Symposium on Theoretical Informatics*, pages 88–94, 2000.

[7] Prosenjit Bose, Evangelos Kranakis, Pat Morin, and Yihui Tang. Approximate range mode and range median queries. In *Proc. 22nd Symposium on Theoretical Aspects of Computer Science*, pages 377–388, 2005.

[8] Gerth Stølting Brodal, Rolf Fagerberg, Mark Greve, and Alejandro López-Ortiz. Online sorted range reporting. In *Proc. 20th Annual International Symposium on Algorithms and Computation*, volume 5878 of *Lecture Notes in Computer Science*, pages 173–182. Springer Verlag, Berlin, 2009.

[9] Michael J. Carey and Donald Kossmann. On saying "enough already!" in SQL. *SIGMOD Rec.*, 26(2):219–230, 1997.

[10] Michael J. Carey and Donald Kossmann. Reducing the braking distance of an SQL query engine. In *VLDB '98: Proceedings of the 24rd International Conference on Very Large Data Bases*, pages 158–169. Morgan Kaufmann Publishers Inc., 1998.

[11] James R. Driscoll, Neil Sarnak, Daniel D. Sleator, and Robert E. Tarjan. Making data structures persistent. *Journal of Computer and System Sciences*, 38(1):86–124, 1989.

[12] Ronald Fagin, Amnon Lotem, and Moni Naor. Optimal aggregation algorithms for middleware. *J. Comput. Syst. Sci.*, 66(4):614–656, 2003.

[13] Johannes Fischer and Volker Heun. A new succinct representation of RMQ-information and improvements in the enhanced suffix array. In *ESCAPE*, volume 4614 of *Lecture Notes in Computer Science*, pages 459–470. Springer, 2007.

[14] Johannes Fischer and Volker Heun. Theoretical and practical improvements on the RMQ-problem, with applications to LCA and LCE. In *Combinatorial Pattern Matching, 17th Annual Symposium, CPM 2006, Proceedings*, volume 4009 of *Lecture Notes in Computer Science*, pages 36–48. Springer, 2006.

[15] Robert W. Floyd. Algorithm 245: Treesort3. *Communications of the ACM*, 7(12):701, 1964.

[16] Greg N. Frederickson. An optimal algorithm for selection in a min-heap. *Inf. Comput.*, 104(2):197–214, 1993.

[17] Greg N. Frederickson and Donald B. Johnson. The complexity of selection and ranking in $X + Y$ and matrices with sorted columns. *J. Comput. Syst. Sci.*, 24(2):197–208, 1982.

[18] Michael L. Fredman and Dan E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[19] Otfried Fries, Kurt Mehlhorn, Stefan Näher, and Athanasios K. Tsakalidis. A $\log \log n$ data structure for three-sided range queries. *Information Processing Letters*, 25(4):269–273, 1987.

[20] Harold Gabow, Jon Louis Bentley, and Robert Endre Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Annual ACM Symposium on Theory of Computing, STOC'84*, pages 135–143. ACM Press, 1984.

[21] Andrea Garratt, Mike Jackson, Peter Burden, and Jon Wallis. A survey of alternative designs for a search engine storage structure. *Information & Software Technology*, 43 (11):661–677, 2001.

[22] Mark Greve, Allan Grønlund Jørgensen, Kasper Dalgaard Larsen, and Jakob Truelsen. Approximating the mode and determining labels with fixed frequency. Manuscript.

[23] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM Journal on Computing*, 13(2):338–355, May 1984.

[24] Meng He, J. Ian Munro, and S. Srinivasa Rao. A categorization theorem on suffix arrays with applications to space efficient text indexes. In *SODA '05: Proceedings of the sixteenth annual ACM-SIAM symposium on Discrete algorithms*, pages 23–32. Society for Industrial and Applied Mathematics, 2005.

[25] Guy Joseph Jacobson. *Succinct static data structures.* PhD thesis, Carnegie Mellon University, 1988.

[26] Joseph JáJá, Christian Worm Mortensen, and Qingmin Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *Proc. 15th International Symposium on Algorithms and Computation*, pages 558–568, 2004.

[27] Helmut Kaufmann and Hans-Jörg Schek. Text search using database systems revisited - some experiments. In *BNCOD 13: Proceedings of the 13th British National Conference on Databases*, pages 204–225. Springer-Verlag, 1995.

[28] Danny Krizanc, Pat Morin, and Michiel H. M. Smid. Range mode and range median queries on lists and trees. *Nord. J. Comput.*, 12(1):1–17, 2005.

[29] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, July 2008.

[30] Edward M. McCreight. Priority search trees. *SIAM Journal on Computing*, 14(2):257–276, May 1985.

[31] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The PageRank citation ranking: Bringing order to the web. Technical Report 1999-66, Stanford InfoLab, 1999.

[32] Mihai Pătraşcu. (Data) STRUCTURES. In *Proc. 49th Annual IEEE Symposium on Foundations of Computer Science*, pages 434–443, 2008.

[33] Mihai Pătraşcu. Lower bounds for 2-dimensional range counting. In *Proc. 39th ACM Symposium on Theory of Computing*, pages 40–46, 2007.

[34] Holger Petersen. Improved bounds for range mode and range median queries. In *Proc. 34th Conference on Current Trends in Theory and Practice of Computer Science*, pages 418–423, 2008.

[35] Holger Petersen and Szymon Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Inf. Process. Lett.*, 109(4):225–228, 2008.

[36] Martin Theobald, Gerhard Weikum, and Ralf Schenkel. Top-k query evaluation with probabilistic guarantees. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 648–659. VLDB Endowment, 2004.

[37] Jean Vuillemin. A unifying look at data structures. *Commun. ACM*, 23(4):229–239, 1980.

[38] John William Joseph Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.