Adaptive Range Counting and Other Frequency-Based Range Query Problems

by

Bryan T. Wilkinson

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

Abstract

We consider variations of range searching in which, given a query range, our goal is to compute some function based on frequencies of points that lie in the range. The most basic such computation involves counting the number of points in a query range. Data structures that compute this function solve the well-studied range counting problem. We consider adaptive and approximate data structures for the 2-D orthogonal range counting problem under the w-bit word RAM model. The query time of an adaptive range counting data structure is sensitive to k, the number of points being counted. We give an adaptive data structure that requires $O(n \log\log n)$ space and $O(\log\log n + \log_w k)$ query time. Non-adaptive data structures on the other hand require $\Omega(\log_w n)$ query time (Pătraşcu, 2007). Our specific bounds are interesting for two reasons. First, when $k = O(1)$, our bounds match the state of the art for the 2-D orthogonal range emptiness problem (Chan et al., 2011). Second, when $k = \Theta(n)$, our data structure is tight to the aforementioned $\Omega(\log_w n)$ query time lower bound.

We also give approximate data structures for 2-D orthogonal range counting whose bounds match the state of the art for the 2-D orthogonal range emptiness problem. Our first data structure requires $O(n \log\log n)$ space and $O(\log\log n)$ query time. Our second data structure requires $O(n)$ space and $O(\log^{\epsilon} n)$ query time for any fixed constant $\epsilon > 0$. These data structures compute an approximation k' such that $(1-\delta)k \leq k' \leq (1+\delta)k$ for any fixed constant $\delta > 0$.

The range selection query problem in an array involves finding the k^{th} lowest element in a given subarray. Range selection in an array is very closely related to 3-sided 2-D orthogonal range counting. An extension of our technique for 3-sided 2-D range counting yields an efficient solution to adaptive range selection in an array. In particular, we present an adaptive data structure that requires $O(n)$ space and $O(\log_w k)$ query time, exactly matching a recent lower bound (Jørgensen and Larsen, 2011).

We next consider a variety of frequency-based range query problems in arrays. We give efficient data structures for the range mode and least frequent element query problems and also exhibit the hardness of these problems by reducing Boolean matrix multiplication to the construction and use of a range mode or least frequent element data structure. We also give data structures for the range α-majority and α-minority query problems. An α-majority is an element whose frequency in a subarray is greater than an α fraction of the size of the subarray; any other element is an α-minority. Surprisingly, geometric insights prove to be useful even in the design of our 1-D range α-majority and α-minority data structures.

Acknowledgements

First and foremost, I would like to thank my supervisor, Timothy Chan, for his support and patience throughout my studies. If I had not taken his algorithms course during my undergraduate degree, I may never have caught his infectious enthusiasm for the wonderful world of theoretical computer science. Parts of this thesis result from collaborations with Stephane Durocher, Kasper Green Larsen, Jason Morrison, and Matthew Skala. I thank all of my collaborators for very enjoyable collaborative experiences. I also thank my thesis readers Anna Lubiw and Alex López-Ortiz. I am indebted to my parents for, among many other things, introducing me to computer science and for their continued support and encouragement as I explore problems that are decreasingly easy to appreciate. I thank my friends for frequent reminders that there exists a world beyond this thesis. Finally, I acknowledge the Natural Sciences and Engineering Research Council of Canada for much appreciated funding.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Range Searching. *Range searching* is one of the fundamental problems in computational geometry. In this problem, we must preprocess a set P of n points from $\mathbb{R}^d$ so that given a query range $Q \subseteq \mathbb{R}^d$ we can efficiently report all points in $P \cap Q$. Efficient data structures for range searching problems only exist if the query ranges are restricted to some special class of ranges. The most important of these classes are halfspaces and axis-aligned boxes. When query ranges are restricted to the former class, the specialized problem is called *halfspace range searching*. When query ranges are restricted to the latter class, the specialized problem is called *orthogonal range searching*. Orthogonal range searching in particular is a very practical tool that can support database queries with inequality filters on d columns. The range tree [Ben80] is the standard textbook solution to orthogonal range searching, requiring $O(n \log^{d-1} n)$ space and $O(\log^d n + k)$ query time, where $k = |P \cap Q|$. The query time can be reduced to $O(\log^{d-1} n + k)$ via downpointers [Wil85], a special case of fractional cascading [CG86].

Word RAM Model. We consider the *w-bit word RAM model* [FW93]. Under this model, the coordinates of the points of P and the vertices of Q are taken from a fixed-size universe $[U] = [1, U] = \{1, \ldots, U\}$. Furthermore, operations on single words take constant time. We make two standard assumptions on w, the number of bits in a word. First, $w = \Omega(\log U)$ so that any coordinate fits in a single word. Second, $w = \Omega(\log n)$ so that an index into the input array representing P fits in a single word. Bounds on the space requirements of data structures are given in words, unless stated otherwise.

It is a common misconception that the assumption $w = \Omega(\log n)$ indicates that data structures under the word RAM model require a computer with a dynamic word size that somehow varies with the size of the input. In reality, the assumption is very natural and

Space	Query Time
$O(n)$	$O((1+k)\log^{\epsilon} n)$
$O(n \log\log n)$	$O((1+k)\log\log n)$
$O(n \log^{\epsilon} n)$	$O(\log\log n + k)$

Table 1.1: Bounds of Chan et al. [CLP11] for 2-D orthogonal range searching

practical. One of the first lines of C++ or Java code that a programmer writes when solving almost any problem includes a loop that iterates through the input to the problem. This loop includes an integer counter variable, usually named i, which is stored in a single word. The fact that few programmers are worried about this counter variable overflowing as it is incremented to n is due to an implicit assumption that $w = \Omega(\log n)$. In the case that a real-world programmer is indeed worried about such overflow, it is likely that a data structure designed under an external memory model [AV88] or a streaming model [AMS99] is more appropriate.

Rank Space. For the purposes of orthogonal range searching, we can operate in *rank space* [GBT84]. Instead of working with the coordinates of points, we work with the ranks of their coordinates, reducing the size of the fixed-size universe from U to n at the expense of d predecessor searches during each query to convert the query into rank space. Under the word RAM model, there are linear-space data structures for predecessor search that require $O(\log\log U)$ [Wil83], $O(\log_w n)$ [FW93], and $O(\sqrt{\log n/\log\log n})$ [AT07] query time. These data structures are all optimal [PT06]. Unless stated otherwise, the data structures that we consider are for points in rank space. Such a data structure can handle points in $[U]^d$ only after including an additive $O(\log\log U)$, $O(\log_w n)$, or $O(\sqrt{\log n/\log\log n})$ term to its query time. Under the word RAM model, the state of the art for 2-D orthogonal range searching amongst points in rank space is a space-time tradeoff given by Chan et al. [CLP11]. Some important instances of the tradeoff are described in Table 1.1.

Range-Aggregate Queries. Given the massive scale of modern data sets, range searching may not even be feasible since the running time of a query must depend linearly on k, as each point in $P \cap Q$ must be individually reported. However, in applications where k may be so large that range searching is not even feasible, the result of a query is clearly never intended to be presented to a human in its raw form. Instead, the result of the query must be summarized in some way. We can encapsulate this idea of summarizing $P \cap Q$ with an aggregation function f with domain $\mathcal{P}(P)$. The result of a query is then $f(P \cap Q)$. Data structures that compute such functions solve *range-aggregate query problems*. The running times of range-aggregate queries need not depend linearly on k as the points in $P \cap Q$ need not be individually reported. For example, consider the Boolean function $f(X)$ that is true if and only if X is empty. The resulting range-aggregate query problem is called the *range emptiness problem*. The best bounds for the 2-D orthogonal range emptiness problem are achieved by the same data structures of Chan et al. [CLP11] that solve 2-D orthogonal range searching, by setting $k = 1$ in their query times. Despite a simple Boolean output, the range emptiness problem still captures the hardness of range searching.

Range Counting. We consider aggregation functions that are based on frequencies of points. The most basic of these functions is $f(X) = |X|$, which gives rise to the *range counting problem*. The data structure of JaJa et al. [JMS05] for 2-D orthogonal range counting requires $O(n)$ space and $O(\log_w n)$ query time. Pătraşcu [Păt07] gives a matching $\Omega(\log_w n)$ lower bound on query time, which holds for data structures using up to $n \log^{O(1)} n$ space. We consider *adaptive* data structures for the 2-D orthogonal range counting problem. When $k = \Theta(n)$, these data structures perform similarly to non-adaptive data structures. However, when $k = o(n)$, adaptive data structures can achieve faster query times than their non-adaptive counterparts. The query times of these adaptive data structures are sensitive to the number of points being counted. Our main result is an adaptive data structure for 2-D orthogonal range counting requiring $O(n \log \log n)$ space and $O(\log \log n + \log_w k)$ query time. These specific bounds for 2-D orthogonal range counting are interesting for two reasons. First, our bounds match the state of the art for the 2-D orthogonal range emptiness problem. That is, our data structure solves the 2-D orthogonal range emptiness problem in $O(n \log \log n)$ space and $O(\log \log n)$ query time, by setting $k = 1$ in the query time. Second, when $k = \Theta(n)$, our data structure is tight to the aforementioned $\Omega(\log_w n)$ query time lower bound for 2-D orthogonal range counting.

The first step that our adaptive data structure takes when answering a query is to determine a multiplicative constant-factor approximation of k. Thus, we also consider the *approximate range counting problem*, which involves computing an approximation k' such that $(1-\delta)k \leq k' \leq (1+\delta)k$. We give two approximate data structures for 2-D orthogonal range counting, both of which match the state of the art for the range emptiness problem. The first data structure requires $O(n \log \log n)$ space and $O(\log \log n)$ query time. The second data structure requires $O(n)$ space and $O(\log^\epsilon n)$ query time.

Our adaptive and approximate data structures combine shallow cuttings [Mat92] with various techniques for 2-D orthogonal range searching, in addition to new techniques inspired by the realm of succinct data structures.

Range Queries in Arrays. An important class of aggregation functions are those that operate on weights that are assigned to the points of P. For each point $p \in P$, let $w(p) \in [U]$ be its weight. This class of aggregation functions is especially important in the 1-D setting where range-aggregate query problems in rank space are exactly equivalent to range query problems in arrays. For example, given an array A of n elements from $[U]$, the well-known *range minimum query problem* involves preprocessing A so that we can efficiently identify the least element in any subarray $A[\ell : r]$ of A. This problem is equivalent to a 1-D range-aggregate query problem in rank space where $f(X) = \min\{w(p) \mid p \in X\}$. Replacing min with mean or median results in the *range mean* and *range median query problems*, which are also well-studied in the context of arrays. There are known optimal data structures for the range minimum, mean, and median query problems [HT84, BJ09].

Range Selection. The range minimum and median query problems are generalized by the aggregation function $f(X) = v$ such that $|\{p \in X \mid w(p) \leq v\}| = k$, where k is given at query time. This generalization is the *range selection query problem.* The range minimum query problem is obtained by fixing $k = 1$. The range median query problem is obtained by fixing $k = \lceil (r - \ell + 1)/2 \rceil$. Jørgensen and Larsen [JL11] give an adaptive data structure for the range selection query problem in an array that is sensitive to the query parameter k. Their data structure requires $O(n)$ space and $O(\log\log n + \log_w k)$ query time. They also give a query time lower bound of $\Omega(\log_w k)$, to which their data structure is tight for all but small values of k. Range selection in an array is very closely related to 3-sided 2-D orthogonal range counting. An extension of our technique for 3-sided 2-D range counting yields an efficient solution to adaptive range selection in an array. In particular, we present an adaptive data structure that requires $O(n)$ space and $O(\log_w k)$ query time, exactly matching the lower bound of Jørgensen and Larsen [JL11].

Range Mode. There is considerably less progress in the literature for the *range mode query problem*, which is obtained by similarly replacing min with mode, the final and often overlooked measure of central tendency. Continuing our investigation of aggregation functions related to frequencies of points, we turn our attention to the range mode query problem. A mode of a multiset is a most frequently occurring element in the multiset. The mode is an especially important measure when considering categorical data. For example, if we interpret the elements of A as colours with no inherent ordering, then the mode is the only well-defined measure of central tendency. We first give a data structure for the range mode query problem requiring $O(n)$ space and $O(\sqrt{n})$ query time, thus eliminating an $O(\log\log n)$ factor from the query time of the data structure of Krizanc et al. [KMS05]. We show that a similar approach solves the *range least frequent element problem.* Using succinct data structures and bit-packing, we are able to achieve $o(\sqrt{n})$ query time while keeping space consumption linear. In particular, we achieve $O(\sqrt{n/w})$ query time. We note that the same optimization does not seem to be applicable for the range least frequent element problem.

Our improvement over the data structure of Krizanc et al. [KMS05] may seem modest, given that our query time is still very close to $O(\sqrt{n})$. However, we also give evidence that suggests that $O(\sqrt{n})$ is the critical running time around which we should be optimizing for logarithmic factors. In particular, we show a reduction from Boolean matrix multiplication to both the range mode and range least frequent element query problems. This reduction suggests that the best query time that is likely to be achieved via purely combinatorial means is around $O(\sqrt{n})$.

Range Majority. The mode of a multiset is interesting due to its exceptionally high frequency. Other elements with high frequency in a multiset may also be interesting, which motivates searching for elements with frequencies that surpass some threshold. In particular, we consider searching for *α-majority* elements. These α-majority elements have frequencies greater than αk. We also consider searching for an *α-minority* element: any element with frequency no greater than αk. We consider both data structures that allow α to be specified at query time and data structures that fix the parameter during preprocessing. To distinguish the latter case, we instead use the parameter β. We consider both 1-D and 2-D data structures. Surprisingly, geometric insights prove to be useful even in the design of our 1-D data structures. In particular, one of the tools that we use is Chazelle's hive graph data structure [Cha86]. In the design of our 2-D data structure, we get the opportunity to apply our own approximate data structure for 2-D orthogonal range counting, thus exemplifying the cohesion of our results.

Outline. In Chapter 2, we discuss related work. In Chapter 3, we describe all of the important tools that we use in the design of our data structures. This chapter may be best used as a reference. In Chapter 4, we describe our adaptive and approximate data structures for 2-D orthogonal range counting. In Chapter 5, we extend our techniques for adaptive range counting to give an optimal adaptive range selection data structure. In Chapter 6, we give data structures for and discuss the hardness of the range mode and range least frequent element problems. In Chapter 7, we apply techniques from computational geometry to the design of data structures for the range majority and range minority query problems. We conclude with open problems and directions for future research in Chapter 8.

Notation. All logarithms without an explicit base have base 2. We denote by $[i, j]$ the set of integers $\{i, i+1, \ldots, j-1, j\}$ and we denote by $[i]$ the set of integers $\{1, 2, \ldots, i-1, i\}$. Given an array A, $A[i]$ represents the array element at index i and $A[i : j]$ represents the subarray from index i to index j. We describe that we are working in d dimensions with the short form "d-D." All occurrences of ϵ, δ, and any decorated versions of these letters represent arbitrary constants greater than zero.

Chapter 2

Related Work

Range Searching. The range tree [Ben80] is a simple solution to 2-D orthogonal range searching that operates via binary divide-and-conquer on the x-coordinates of points. Most subsequent solutions, summarized in Table 2.1, are variations on the range tree. Chazelle [Cha88] introduces a tradeoff between the space of a data structure and the time required to report each point. Essentially, he designs compressed range trees which require time to decompress each reported point. Chan et al. [CLP11] refine this idea, resulting in the best known data structures for range emptiness.

Range Counting. The range tree [Ben80] is also a simple solution to 2-D orthogonal range counting. The x-interval of a query rectangle can be partitioned into $O(\log n)$ subintervals such that each subinterval is the x-interval of a node in the range tree. In this way, we can decompose the 2-D query into 1-D queries in these $O(\log n)$ nodes of the range tree. Counting points in 1-D ranges reduces to predecessor search. Under the comparison model, predecessor search requires $\Omega(\log n)$ time and is solved in $O(\log n)$ time via binary search. Thus the total 2-D query time is $O(\log^2 n)$. Since the height of a range tree is $O(\log n)$ and each point is present in exactly one node of each level of the tree, the total space requirement is $O(n \log n)$. The downpointers of Willard [Wil85], a special case of fractional cascading [CG86], is a technique that can be used to avoid the full costs of repeated predecessor searches at each level of the range tree. Assume we have performed predecessor search in a node of the range tree and we now wish to perform the same search in one of the node's children. For each element in the parent, we can store pointers to the element's predecessor in the child without an asymptotic increase in space. Then, during a 2-D range counting query, we only need to perform one full binary search with cost $O(\log n)$ at the root of the range tree. The rest of the $O(\log n)$ predecessor searches take constant time each for a total of $O(\log n)$ time. Chazelle's compressed range tree [Cha88] reduces

Reference	Space	Query Time
[Ben80]	$O(n \log n)$	$O(\log^2 n + k)$
[Wil85]	$O(n \log n)$	$O(\log n + k)$
[Cha88]	$O(n)$	$O(\log n + k \log^\epsilon n)$
	$O(n \log \log n)$	$O(\log n + k \log \log n)$
	$O(n \log^\epsilon n)$	$O(\log n + k)$
[Ove88]	$O(n \log n)$	$O(\log \log n + k)$
[ABR00]	$O(n \log \log n)$	$O((\log \log n + k) \log \log n)$
	$O(n \log^\epsilon n)$	$O(\log \log n + k)$
[Nek09b]	$O(n)$	$O(\log_w n + k \log^\epsilon n)$
[CLP11]	$O(n)$	$O((1 + k) \log^\epsilon n)$
	$O(n \log \log n)$	$O((1 + k) \log \log n)$

Table 2.1: Data structures for 2-D orthogonal range searching

space to $O(n)$ in the word RAM model by representing the downpointers succinctly in $O(n)$ bits at each node. JaJa et al. [JMS05] use a w^ϵ-ary range tree and fusion trees [FW93] for predecessor search in order to reduce the query time to $O(\log_w n)$.

To the best of the author's knowledge, there are no previous adaptive data structures explicitly designed for 2-D orthogonal range counting. However, the adaptive range selection data structure of Jørgensen and Larsen [JL11] can be adapted to solve 3-sided 2-D orthogonal range counting. Such a data structure would require $O(n)$ space and $O(\log \log n + \log_w k)$ query time.

Recent research in the area of approximate range counting focuses on halfspace queries. In the context of halfspace range counting, the best known exact counting data structure requires $O(n)$ space and $O(n^{1-1/d})$ query time [Mat93]. Emptiness queries can be answered in $O(n^{1-1/\lfloor d/2 \rfloor})$ time [Mat92]. It turns out that the complexity of approximate halfspace range counting lies close to that of the halfspace range emptiness problem [AS10, AHP08]. Afshani et al. [AHZ10] give a general framework for approximate range counting that applies to any type of range that admits shallow cuttings. They give linear-space data structures for 3-D halfspace and 3-D orthogonal dominance counting that require optimal $O(\log_B(n/k))$ query time under the external memory model. Applying their technique under the word RAM model seems to require $\Omega((\log \log n)^2)$ query time and thus we cannot approach optimal $O(\log \log n)$ query time without further ideas.

Reference	Problem	Space	Query Time
[CLP11]	Emptiness	$O(n \log\log n)$	$O(\log\log n)$
[CLP11]	Emptiness	$O(n)$	$O(\log^\epsilon n)$
[JMS05]	Standard	$O(n)$	$O(\log_w n)$
new	Adaptive	$O(n \log\log n)$	$O(\log\log n + \log_w k)$
[Nek09a]	Approximate	$O(n \log^2 n)$	$O(\log\log n)$
new	Approximate	$O(n \log\log n)$	$O(\log\log n)$
new	Approximate	$O(n)$	$O(\log^\epsilon n)$

Table 2.2: Comparison of 2-D orthogonal range counting results

Nekrich [Nek09a] gives data structures for approximate orthogonal range counting using a more precise notion of approximation than in the literature for halfspace range counting. In particular, his data structures compute an approximation k' such that $k - \delta k^p \leq k' \leq k + \delta k^p$, where $\delta > 0$ and $0 < p < 1$. He gives a 2-D data structure that requires $O(n \log^4 n)$ space and $O((1/p) \log\log n)$ query time, if p can be specified at query time. If p is fixed during the preprocessing of the data structure, the space requirement can be reduced to $O(n \log^2 n)$. In either case, these data structures require much more space than the 2-D orthogonal range emptiness data structure of Chan et al. [CLP11].

We compare our results to these previous results in Table 2.2.

Range Selection. Research on range query problems in arrays tends to focus on data structures that require linear space or constant query time. Research on linear-space data structures is interesting since there is often an $\Omega(n)$ lower bound on space for range query problems. On the other hand, there are at most $O(n^2)$ different subarrays, so there are trivial data structures that precompute and store the answer for every possible query in order to support constant-time queries using $O(n^2)$ space. Research on data structures with constant query time involves reducing space from the $O(n^2)$ bound as much as possible.

In the context of data structures with constant query time, Krizanc et al. [KMS05] give a data structure for the range median query problem that requires $O(n^2 \log\log n / \log n)$ space. Petersen [Pet08] improves the space requirement to $O(n^2 \log^{(c)} n / \log n)$, where $\log^{(c)} n$ is the iterated logarithm with a constant number of iterations. Finally, Petersen and Grabowski [PG09] improve the space requirement to $O((n \log\log n / \log n)^2)$.

In the context of linear space data structures, initial results for range selection require $O(\log n)$ query time [GPT09, GS09]. Brodal and Jørgensen [BJ09] reduce query time to $O(\log_w n)$. Jørgensen and Larsen [JL11] give an adaptive data structure that requires $O(\log\log n + \log_w k)$ query time. They also prove an $\Omega(\log_w k)$ lower bound, showing that

Reference	Problem	Space	Query Time
[BJ09]	Standard	$O(n)$	$O(\log_w n)$
[JL11]	Adaptive	$O(n)$	$O(\log\log n + \log_w k)$
new	Adaptive	$O(n)$	$O(\log_w k)$

Table 2.3: Comparison of range selection results

Reference	Problem	Space	Query Time
[KMS05]	Mode	$O(n)$	$O(\sqrt{n}\log\log n)$
[Pet08]	Mode	$O(n^{1+\epsilon})$	$O(\sqrt{n})$
new	Mode	$O(n)$	$O(\sqrt{n/w})$
new	LFE	$O(n)$	$O(\sqrt{n})$

Table 2.4: Comparison of range mode and least frequent element results

their data structure is optimal for all but small values of k. We compare our result to these previous results in Table 2.3.

Range Mode. In the context of data structures with constant query time, Krizanc et al. [KMS05] give a data structure for the range mode query problem requiring the same space consumption as their data structure for the range median query problem. Petersen and Grabowski [PG09] also give a data structure for the range mode query problem that requires $O((n/\log n)^2 \log\log n)$ space, which is less space than required by their data structure for the range median query problem by an $O(\log\log n)$ factor.

Krizanc et al. [KMS05] also give a space-time tradeoff by describing a data structure that requires $O(n^{2-2s})$ space and $O(n^s \log n)$ query time, for any $0 < s \leq 1/2$. At the linear-space end of this tradeoff, the data structure requires $O(\sqrt{n}\log n)$ query time. The logarithmic factor arises due to predecessor search in a universe of size n. Substituting an efficient predecessor search data structure designed under the word RAM model [Wil83] yields a query time of $O(\sqrt{n}\log\log n)$. Petersen [Pet08] eliminates the logarithmic factor from the space-time tradeoff. However, his data structure is only valid for $0 \leq s < 1/2$, as the number of levels in a hierarchical set of tables and hash functions approaches ∞ as s approaches 1/2. Thus, his data structure always requires $\omega(n)$ space. We compare our results to these previous results in Table 2.4.

Finally, Greve et al. [GJLT10] prove a lower bound of $\Omega(\log n/\log(Sw/n))$ query time for any data structure that uses S memory cells of w bits in the cell probe model. There is evidently a huge gap between the upper and lower bounds.

Reference	Problem	Dimensions	Space	Query Time
[DHM$^+$11]	β-majority	1	$O(n \log(1/\beta))$	$O(1/\beta)$
[GHMN11]	α-majority	1	$O(n \log n)$	$O(1/\alpha)$
new	α-minority	1	$O(n)$	$O(1/\alpha)$
alternative	α-majority	1	$O(n \log n)$	$O(1/\alpha)$
[KN08]	β-majority	2	$O((1/\beta) n \log n)$	$O((1/\beta) \log^2 n)$
new	β-majority	2	$O(\log(1/\beta) n \log^\epsilon n)$	$O((1/\beta) \log n)$

Table 2.5: Comparison of range majority and minority results

Range Majority. There are previous results for both the α-majority problem, in which different values of α can be specified during each query, and the β-majority problem, in which the value of β is fixed during preprocessing. Durocher et al. [DHM$^+$11] give a data structure for the range β-majority problem in an array requiring $O(n \log(1/\beta))$ space and $O(1/\beta)$ query time. Gagie et al. [GHMN11] consider the α-majority problem in 2-D matrices and in 1-D arrays. Note that problems in 2-D matrices with n elements are easier then problems over n 2-D points in rank space as the "positions" of elements are predetermined. Their 2-D data structure requires $O(n \log^2 n)$ space and $O(1/\alpha)$ query time. Their 1-D data structure requires $O(n \log n)$ space and $O(1/\alpha)$ query time. Karpinski and Nekrich [KN08] consider the problem of searching for τ-dominating colours amongst coloured 2-D points. Their problem is equivalent to the 2-D orthogonal β-majority range query problem. They give a data structure that requires $O((1/\beta)n \log n)$ space and $O((1/\beta) \log^2 n)$ query time. We compare our results to these previous results in Table 2.5.

Chapter 3

Toolbox

This chapter contains descriptions of various existing data structures and techniques that we apply in the design of our own data structures. The sections of this chapter are mostly self-contained and are referenced in later chapters, so it is not necessary to process all of the contents of this chapter before proceeding to the descriptions of our data structures.

3.1 Succinct Data Structures

Classic analysis of a data structure in the word RAM model involves giving an asymptotic upper bound on space consumption in words. The study of *succinct* data structures [Jac88] involves refining this analysis of space consumption. Consider a data structure problem in which there is an optimal number of bits $\mathcal{B}(n)$ required to be able to represent all possible problem instances of size n, based on information-theoretic arguments. The goal of a designer of a succinct data structure is to create a data structure that requires only $\mathcal{B}(n) + o(\mathcal{B}(n))$ bits of space, while still supporting efficient queries. As we do not intend to design succinct data structures, our motivation for using succinct data structures is primarily that we can use bit-packing to squeeze a succinct data structure into $O(\mathcal{B}(n)/w)$ words of space.

3.1.1 Succinct Rank and Select

A common technique used to store information compactly is to represent the information as a string and extract it using *rank* and *select* queries. Assume we are given a string

$a_1 a_2 \dots a_n$ of n elements from the alphabet $\Sigma = \{1, \dots, \sigma\}$. Then, $\text{rank}_x(i)$ for $x \in \Sigma$ and $i \in [n]$ is the number of occurrences of x in the substring $a_1 a_2 \dots a_i$. Also, $\text{select}_x(i)$ for $x \in \Sigma$ and $i \in \mathbb{Z}^+$ is the index of the i^{th} occurrence of x, if such an occurrence exists. Succinct data structures for this problem were first obtained for a binary alphabet $\Sigma = \{0, 1\}$.

Theorem 3.1 (Clark and Munro [CM96])**.** *There exists a data structure for rank and select queries in a binary string of length n requiring $O(n)$ bits of space and $O(1)$ query time.*

Golynski et al. [GMR06] give a data structure for the general problem of any arbitrary alphabet size that requires $O(n \log \sigma)$ bits of space, $O(\log \log \sigma)$ time for rank queries, and $O(1)$ time for select queries. Supporting constant-time rank queries in the same space bound is possible, if there is a restriction on the alphabet size.

Theorem 3.2 (Ferragina et al. [FMM04])**.** *There exists a data structure for rank and select queries in a string with alphabet size $\sigma = \log^{O(1)} n$ requiring $O(n \log \sigma)$ bits of space and $O(1)$ query time.*

3.1.2 Succinct Trees

The use of trees to store information is ubiquitous in computer science. The structure of a static tree on n nodes can be stored succinctly while still supporting a wide array of constant-time navigation queries.

Theorem 3.3 (e.g., Sadakane and Navarro [SN10])**.** *There exists a data structure for a static tree on n nodes that requires $O(n)$ bits of space and supports LCA operations in $O(1)$ time.*

3.1.3 Succinct Predecessor Search

Given a set S of n elements from $\mathbb{R}$, the *predecessor* of some element $x \in \mathbb{R}$ is the greatest element of S that is less than x. Binary search in a sorted array representation of S is a simple solution to the predecessor search problem requiring $O(\log n)$ time, which is optimal under the comparison model. Under the word RAM model, where elements are take from the fixed-size universe $[U]$, data structures exist with query times that depend only on U. The van Emde Boas tree [vEBKZ76] was the first data structure to achieve $O(\log \log U)$

query time. The y-fast trie of Willard [Wil83] supports the same query time and requires $O(n)$ space. Pătraşcu and Thorup [PT06] give a matching lower bound.

However, it turns out that the requirement of $O(n)$ space to achieve $O(\log\log U)$ query time is simply due to a need for access to the input elements. Storing the input elements explicitly requires $O(n)$ space. If we instead allow oracle access to the input elements, more succinct predecessor search data structures exist.

Lemma 3.4 (Grossi et al. [GORR09]). *There exists a data structure for predecessor search requiring $O(n \log\log U)$ bits of space and $O(A(n)\log_w n)$ query time, where $A(n)$ is the time required for oracle access to an input element given its rank.*

Lemma 3.5 (Grossi et al. [GORR09]). *There exists a data structure for predecessor search requiring $O(n \log\log U)$ bits of space and $O(\log\log U + A(n))$ query time, where $A(n)$ is the time required for oracle access to an input element given its rank.*

Data structures that support rank and select queries in a binary string (see Section 3.1.1) provide an alternative solution to the predecessor search problem. We construct a binary string $a_1 a_2 \dots a_U$ such that $a_x = 1$ if and only if $x \in S$ for each $x \in [U]$. Then, if $a_x = 1$ the predecessor of x is $\text{select}_1(\text{rank}_1(x) - 1)$. Otherwise, the predecessor of x is $\text{select}_1(\text{rank}_1(x))$. Theorem 3.1 thus gives a predecessor search data structure with constant query time.

Lemma 3.6. *There exists a data structure for predecessor search requiring $O(U)$ bits of space and $O(1)$ query time.*

In many cases, $O(U)$ bits of space may be prohibitive. However, when working in rank space (i.e., when $U = n$), Lemma 3.6 is very efficient in terms of both space and query time.

3.2 Range Trees and Ball Inheritance

The *range tree* [Ben80] is a very common tool used to solve orthogonal range searching and its *decomposable* variants. Let (X_1, X_2) be a partition of a set of points X. A range-aggregate query problem with aggregation function f is decomposable if we can compute $f(X)$ in constant time given $f(X_1)$ and $f(X_2)$. The standard binary range tree is constructed by dividing the set P of n points in $[U]^d$ into two sets of size $n/2$ and recursing on each of these sets so that the leaves of the range tree contain only one point each. The points are divided based on x_i-coordinate for some fixed i. All points with x_i-coordinate less than or equal to the median x_i-coordinate become the left child of the current node of the range tree. All points with x_i-coordinate greater than the median become the right child. The height of the range tree is $O(\log n)$.

One purpose of the range tree is to decompose one d-D query into $O(\log n)$ $(d-1)$-D queries: at most two queries in nodes of each level of the range tree. Another purpose is to decompose an s-sided query into $(s-1)$-sided queries. If any two sides are parallel, then one s-sided query becomes two $(s-1)$-sided queries: one query in each of the children of the deepest node that contains both of the parallel sides.

At each node of the range tree, we store a data structure D for $(d-1)$-D queries or $(s-1)$-sided queries, as required. The space required by the range tree as a whole is typically $O(S_D(n)\log n)$. One way to obtain a range tree requiring $o(n\log n)$ space is to increase its fan-out. In particular, a B-ary range tree has height $O(\log_B n)$ and thus requires space $O(S_D(n)\log_B n)$. However, in a B-ary range tree, the decomposition of a query becomes more complex.

Another way to obtain a range tree requiring $o(n\log n)$ space is to use a succinct data structure D that requires $o(n)$ space. In such little space, D cannot explicitly store the points over which it supports queries. Providing this access is a problem solved by the *ball inheritance* data structure of Chan et al. [CLP11]. Given a range tree, a ball inheritance query involves determining the coordinates of a point p given a node v of the range tree and the x_j-rank of p in v, for any j. Chan et al. [CLP11] give a space-time tradeoff for the ball inheritance problem in rank space.

Lemma 3.7 (Chan et al. [CLP11]). *For any $B \in [2, \lfloor \log^\epsilon n \rfloor]$, there exists a data structure for the ball inheritance problem requiring either $O(nB\log\log n)$ space and $O(\log_B \log n)$ query time, or $O(n\log_B \log n)$ space and $O(B\log\log n)$ query time.*

Corollary 3.8. *There exists a data structure for the ball inheritance problem requiring $O(n\log\log n)$ space and $O(\log\log n)$ query time.*

Corollary 3.9. *There exists a data structure for the ball inheritance problem requiring $O(n)$ space and $O(\log^\epsilon n)$ query time.*

Corollary 3.10. *There exists a data structure for the ball inheritance problem requiring $O(n\log^\epsilon n)$ space and $O(1)$ query time.*

By combining this ball inheritance data structure that provides access to input elements with the succinct predecessor search data structure of Lemma 3.5 which requires oracle access to input elements, Chan et al. [CLP11] give an *augmented range tree* that supports efficient predecessor search along the x_j-axis at every node, for any j.

Lemma 3.11 (Chan et al. [CLP11])**.** *Given a data structure D for the ball inheritance problem, there exists an augmented range tree that supports predecessor search at every node requiring $O(n + S_D(n))$ space and $O(\log\log n + Q_D(n))$ query time.*

3.3 Cartesian Trees

Given a set P of n points from $[U]^2$, the *Cartesian tree* for P is constructed by creating a node containing the lowest point $p \in P$ and recursing on all points that lie to the left of p and on all points that lie to the right of p. Unlike a range tree, the height of a Cartesian tree is not guaranteed to be logarithmic since p does not necessarily lie in the middle of P along the x-axis. In fact, the height of the tree can be linear if, for example, the points of P are organized along a diagonal line.

The key property of the Cartesian tree is that, given a query range of the form $Q = [\ell, r] \times [U]$ (i.e., a vertical slab), the lowest point in Q is the lowest common ancestor of the leftmost and rightmost points in Q. If the x-coordinate of this point is m, then we can find all other points in Q by recursing in query ranges $[\ell, m-1] \times [U]$ and $[m+1, r] \times [U]$. Finding the leftmost and rightmost points in Q is a 1-D problem: predecessor search along the x-axis. So, a Cartesian tree reduces range searching in a vertical slab to predecessor search queries and LCA queries. Using the succinct predecessor search data structure of Lemma 3.6 and the succinct tree data structure of Theorem 3.3, we obtain a succinct Cartesian tree.

Lemma 3.12. *There exists a Cartesian tree that requires $O(U)$ bits of space and $O(1)$ time to find the x-rank of the lowest point in a query range of the form $[\ell, r] \times [U]$.*

The Cartesian tree can also handle a 3-sided query range of the form $[\ell, r] \times [1, t]$ by terminating its recursion whenever reaching a point with y-coordinate greater than t. The data structure of Chan et al. [CLP11] for 2-D orthogonal range searching uses the augmented range tree of Lemma 3.11 to transform a general 4-sided query into two 3-sided queries and uses succinct Cartesian trees at each node of the range tree to handle the 3-sided queries.

3.4 Shallow Cuttings

Let H be a set of n d-D hyperplanes. A $(1/r)$-*cutting* [Mat91] is a decomposition of $\mathbb{R}^d$ into simplices such that each simplex intersects at most $O(n/r)$ hyperplanes of H. The size of a $(1/r)$-cutting is the number of simplices into which $\mathbb{R}^d$ is decomposed. Chazelle and Friedman [CF90] show that there exist $(1/r)$-cuttings of optimal size $O(r^d)$. Cuttings are often applied recursively to efficiently solve problems in computational geometry in which we are given query points. For example, a data structure first does some processing for a query point at the current node of recursion. The data structure then determines in which of the $O(r^d)$ simplices the query point lies. It finally recurses in this simplex, thus reducing the size of the problem to $O(n/r)$.

Let the *K-level* of H be the set of points p on the hyperplanes of H such that the hyperplane containing p has the K^{th} lowest x_i-coordinate for some fixed i. The *$(\leq K)$-level* of H is the set of points of $\mathbb{R}^d$ that lie on or under the K-level of H, based again on x_i-coordinate. The shallow cutting lemma of Matoušek [Mat92] shows that, if a cutting need only decompose the $(\leq K)$-level of H instead of all of $\mathbb{R}^d$, smaller cuttings exist. In particular, there exists a (K/n)-cutting of the $(\leq K)$-level of H of size $O((n/K)^{\lfloor d/2 \rfloor})$. We call this (K/n)-cutting a *K-shallow cutting*. A K-shallow cutting thus consists of $((n/K)^{\lfloor d/2 \rfloor})$ simplices, each of which intersects $O(K)$ hyperplanes of H. This result is particularly nice for $d = 3$ since $\lfloor d/2 \rfloor = 1$. Thus, in 3-D there are K-shallow cuttings of size $O(n/K)$.

Agarwal et al. [AES99] show that 3-D shallow cuttings exist for arrangements of more general classes of surfaces than hyperplanes. Afshani [Afs08] observes that shallow cuttings are thus applicable to 3-D orthogonal problems involving a set P of n points by showing the equivalence (in this context) of shallow cuttings and the approximate boundaries of Vengroff and Vitter [VV96]. We opt to use the terminology of shallow cuttings rather than that of approximate boundaries. Assume queries are dominance regions of the form $(-\infty, x] \times (-\infty, y] \times (-\infty, z]$. A K-shallow cutting then consists of $O(n/K)$ cells, each of which is a subset of P. If a query Q contains no more than K points, then there exists a shallow cutting cell C such that $C \cap Q = P \cap Q$. Finding such a shallow cutting cell reduces to 2-D orthogonal point location.

Consider a set P of n points from $[U]^2$. There exist shallow cuttings for 3-sided orthogonal queries of the form $[\ell, r] \times [1, t]$, by a simple mapping of the points to 3-D points and queries to 3-D dominance regions. Jørgensen and Larsen [JL11] give an alternative construction of shallow cuttings for range selection queries, which are closely related to 3-sided 2-D orthogonal range counting queries. In their construction, each shallow cutting cell is assigned a key, which is a horizontal line segment. Finding a shallow cutting cell C that can answer a range selection query reduces to finding the key with the least y-coordinate contained in the vertical slab $[\ell, r] \times [U]$. This problem again reduces to 2-D orthogonal point location.

3.5 Relative (p, ϵ)-Approximations

A standard technique for approximate range counting amongst a set P of n points involves taking a random sample $R \subseteq P$ and showing that for some positive probability, $|R \cap Q|$ is a scaled-down approximation of $|P \cap Q|$ for all queries Q. An *absolute ϵ-approximation* is a set $R \subseteq P$ such that

$$\left| \frac{|R \cap Q|}{|R|} - \frac{|P \cap Q|}{|P|} \right| < \epsilon.$$

So, if we calculate $k' = (|P|/|R|)|R \cap Q|$ given a query Q, we are guaranteed that $k - \epsilon n < k' < k + \epsilon n$. There exist constant-size absolute ϵ-approximations for orthogonal ranges [PA95]. However, the accuracy of approximation degrades for small k as the error term always depends on n. Fixing this problem, a *relative ϵ-approximation* is a set $R \subseteq P$ such that

$$(1-\epsilon)\frac{|P \cap Q|}{|P|} \leq \frac{|R \cap Q|}{|R|} \leq (1+\epsilon)\frac{|P \cap Q|}{|P|}.$$

This time if we calculate $k' = (|P|/|R|)|R \cap Q|$, we are guaranteed that $(1-\epsilon)k \leq k' \leq (1+\epsilon)k$. However, this guarantee is hard to achieve for small k. In particular, if we determine that $k' = 0$, then k must also be 0. This observation implies that there exists only one relative ϵ-approximation and it is $R = P$, which is useless. A *relative (p, ϵ)-approximation* [CKMS06] masks out this problem by providing the same guarantee as a relative ϵ-approximation, but only when $|P \cap Q|/|P| \geq p$ for a fixed $0 < p < 1$.

Theorem 3.13 (Har-Peled and Sharir [HPS11])**.** *There exist relative (p, ϵ)-approximations of size $O((1/p)\log(1/p))$ for orthogonal ranges. Furthermore, for a sufficiently large constant c depending on ϵ and the number of dimensions d, any random sample of size $c(1/p)\log(1/p)$ is a relative (p, ϵ)-approximation for orthogonal ranges with constant positive probability.*

3.6 Hive Graphs

Given n horizontal line segments, consider the problem of finding every segment that intersects a query vertical line segment. Given a data structure D for 2-D orthogonal point location, the *hive graph* data structure of Chazelle [Cha86] solves this problem in

$O(n + S_D(n))$ space and $O(Q_D(n) + k)$ query time, where k is the number of segments reported. In fact, the hive graph solves the more general problem of reporting the horizontal line segments that intersect a query vertical ray in sorted order. Finding the first horizontal line intersecting a vertical ray requires a 2-D orthogonal point location query; however, subsequent intersections can be found in constant time each. Using the point location data structure of Chan [Cha11] yields an efficient linear-space hive graph.

Lemma 3.14. *There exists a data structure for vertical ray shooting queries through horizontal line segments requiring $O(n)$ space, $O(\log \log U)$ time to find the first intersection, and $O(1)$ time to find each subsequent intersection in sorted order along the query ray.*

Chapter 4

Range Counting

We consider 2-D orthogonal range counting. In this problem, we must preprocess a set P of n points from $[n]^2$ so that given a query rectangle $Q = [\ell, r] \times [b, t] \subseteq [n]^2$ we can efficiently compute $k = |P \cap Q|$. JaJa et al. [JMS05] give an efficient linear-space data structure for this problem.

Theorem 4.1 (JaJa et al. [JMS05])**.** *There exists a data structure for 2-D orthogonal range counting requiring $O(n)$ space and $O(\log_w n)$ query time.*

This data structure is optimal as Pǎtraşcu [Pǎt07] gives a matching $\Omega(\log_w n)$ query time lower bound that holds even for data structures requiring up to $n \log^{O(1)} n$ space. We seek adaptive data structures with query times that depend on k so that for $k = o(n)$, we can beat the query time lower bound of $\Omega(\log_w n)$. We also seek approximate data structures for 2-D orthogonal range counting. An approximate data structure need only compute an approximation k' of k such that $(1 - \delta)k \leq k' \leq (1 + \delta)k$. We consider approximate data structures for two reasons. First, we use our approximate data structure as a building block for our adaptive data structure. Second, approximate range counting is of independent interest.

For both our adaptive and approximate range counting data structures, our goal is to match the best known bounds for range emptiness data structures. In linear space, Chan et al. [CLP11] support emptiness queries in $O(\log^\epsilon n)$ time. Query time can be reduced to $O(\log \log n)$ if space consumption is increased to $O(n \log \log n)$. We give approximate data structures that match these bounds exactly. We also give an adaptive data structure requiring $O(n \log \log n)$ space and $O(\log \log n + \log_w k)$ query time. These bounds

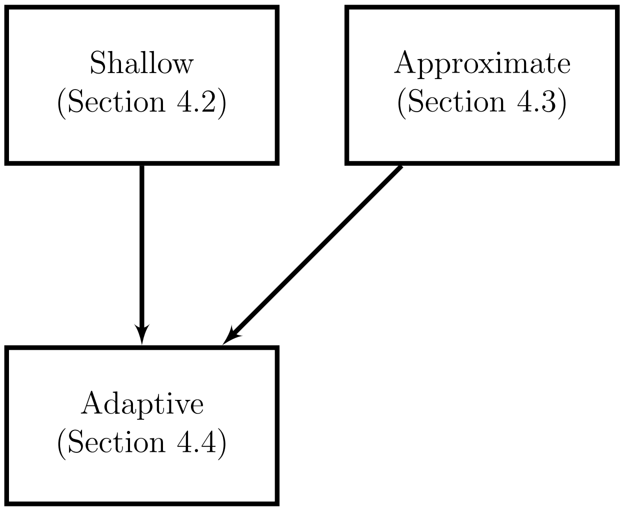


Figure 4.1: Overview of our adaptive range counting data structure

match those of the range emptiness problem when $k = O(1)$, as well as Pătraşcu's lower bound [Păt07] when $k = \Theta(n)$.

We introduce and give efficient solutions to a problem called K*-shallow range counting.* In this problem, our goal is again to compute $|P \cap Q|$ given a query Q. However, if $|P \cap Q| > K$, we may output an error.

In Section 4.1 we begin by considering the decomposability of shallow, approximate, and adaptive range counting queries. In Section 4.2, we design data structures for K-shallow range counting. The query time for each of these data structures includes an $O(\log_w K)$ term. Our intent is to construct $O(\log \log n)$ copies of such a data structure for double exponentially increasing values of K. Then, for any standard range counting query, one of the copies must be able to answer the query such that the $O(\log_w K)$ term in its query time is also $O(\log_w k)$. Determining which of the copies to query reduces to approximate range counting, which we consider in Section 4.3. In Section 4.4, we combine our shallow and approximate data structures, as briefly outlined in this paragraph, to create our final adaptive data structure. We summarize our high-level plan in Figure 4.1.

4.1 Decomposability

As is common in the study of orthogonal range searching problems, we start by considering the special case of 3-sided queries of the form $[\ell, r] \times [1, t]$ and then we find some way to generalize to 4-sided queries. In the case of standard range counting, generalizing from 3-sided queries to 4-sided queries is trivial. Range counting is an instance of a more general problem in which each point is assigned a weight and we must compute the group sum of the weights of the points in a given range. So, we can handle a 4-sided query by subtracting the sum/count for the 3-sided range $[\ell, r] \times [1, b-1]$ from the sum/count for the 3-sided range $[\ell, r] \times [1, t]$. It is important to note that this technique is not applicable in the context of shallow, approximate, or adaptive range counting. In the context of K-shallow range counting, even if the count for the 4-sided query is no more than K, the number of points in the range $[\ell, r] \times [1, b-1]$ may exceed K. In the context of approximate range counting, the error term would depend on the number of points in the range $[\ell, r] \times [1, b-1]$, which may be greater than k. In the context of adaptive range counting, the running time of a query would depend on the number of points in the range $[\ell, r] \times [1, b-1]$, which may be greater than k.

Although we cannot take advantage of techniques for the range group sum problem, we can take advantage of techniques for the range semigroup sum problem. For example, we can efficiently decompose a query Q into two queries Q_1 and Q_2 such that (Q_1, Q_2) is a partition of Q.

Lemma 4.2 (Shallow Decomposability)**.** *Given a method to obtain the K-shallow count for Q_1 in $f_1(n, K)$ time and for Q_2 in $f_2(n, K)$ time, the K-shallow count for Q can be computed in $O(f_1(n, K) + f_2(n, K))$ time.*

Proof. We compute $k_1 = |P \cap Q_1|$ in $f_1(n, K)$ time. If there is an error, then $k_1 > K$. Thus, $k = k_1 + k_2 > K$ so we output an error. We similarly compute k_2, outputting an error if there is an error in computing it. If there are no errors, then we compute and output $k = |P \cap Q| = |P \cap Q_1| + |P \cap Q_2| = k_1 + k_2$. □

Lemma 4.3 (Approximate Decomposability)**.** *Given a method to obtain the $(1+\delta)$-approximate count for Q_1 in $f_1(n)$ time and for Q_2 in $f_2(n)$ time, the $(1+\delta)$-approximate count for Q can be computed in $O(f_1(n) + f_2(n))$ time.*

Proof. Let $k_1 = |P \cap Q_1|$ and $k_2 = |P \cap Q_2|$, so that $k = |P \cap Q| = |P \cap Q_1| + |P \cap Q_2| = k_1 + k_2$. We compute and return $k' = k'_1 + k'_2$, where k'_1 is the $(1+\delta)$-approximate count for Q_1 and k'_2 is the $(1+\delta)$-approximate count for Q_2. Since $(1-\delta)k_1 \leq k'_1 \leq (1+\delta)k_1$ and $(1-\delta)k_2 \leq k'_2 \leq (1+\delta)k_2$,

$$\begin{aligned} (1-\delta)(k_1 + k_2) &\leq k'_1 + k'_2 & \qquad k'_1 + k'_2 &\leq (1+\delta)(k_1 + k_2) \\ (1-\delta)k &\leq k' & k' &\leq (1+\delta)k. \end{aligned}$$

□

Lemma 4.4 (Adaptive Decomposability)**.** *Given a method to obtain the exact count for Q_1 in $f_1(n, k_1)$ time and for Q_2 in $f_2(n, k_2)$ time, the exact count for Q can be computed in $O(f_1(n, k) + f_2(n, k))$ time, for non-decreasing functions f_1 and f_2.*

Proof. Let $k_1 = |P \cap Q_1|$ and $k_2 = |P \cap Q_2|$, so that $k = |P \cap Q| = |P \cap Q_1| + |P \cap Q_2| = k_1 + k_2$. We compute k_1 in $f_1(n, k_1)$ time and k_2 in $f_2(n, k_2)$ time. We compute and return $k = k_1 + k_2$. Since $k_1 \le k$, then $f_1(n, k_1) = O(f_1(n, k))$. Similarly, $f_2(n, k_2) = O(f(n, k))$. So, the total running time is $O(f_1(n, k) + f_2(n, k))$. □

4.2 Shallow Range Counting

In this section, we give data structures for K-shallow 2-D orthogonal range counting. We begin with a simple linear-space 3-sided data structure that uses a K-shallow cutting for 3-sided ranges (see Section 3.4). We next create a data structure that solves the same problem but requires only $O(n \log(K \log n))$ bits of space by using a space-saving technique from the realm of succinct data structures (see Section 3.1). We use a range tree (see Section 3.2) to support 4-sided queries using our succinct 3-sided data structure. In order to save space, we use a range tree with fan-out of $\Theta(K \log n)$. However, using a $\Theta(K \log n)$-ary range tree requires, at each node, a data structure for range counting queries on a narrow $[\Theta(K \log n)] \times [n]$ grid. Using known techniques for range counting on a narrow grid, the running time of such queries matches the running time of our 3-sided queries. We summarize our high-level plan in Figure 4.2.

Lemma 4.5. *There exists a data structure for K-shallow 3-sided 2-D orthogonal range counting requiring $O(n)$ space and $O(\log \log n + \log_w K)$ query time.*

Proof. We construct a K-shallow cutting of P. In each cell of the shallow cutting we build the standard range counting data structure of Theorem 4.1. Each of the $O(n/K)$ cells thus requires $O(K)$ space, for a total of $O(n)$ space.

We can determine in which cell a 3-sided query rectangle Q lies, if any, using a 2-D orthogonal point location query. We build the linear-space point location data structure of Chan [Cha11] which answers queries in $O(\log \log n)$ time. If Q does not lie in a shallow cutting cell, then $k > K$, so we output an error. If Q lies in shallow cutting cell C, then we forward Q onto the standard range counting data structure stored for C. Since $|C| = O(K)$, the standard query takes $O(\log_w K)$ time. □

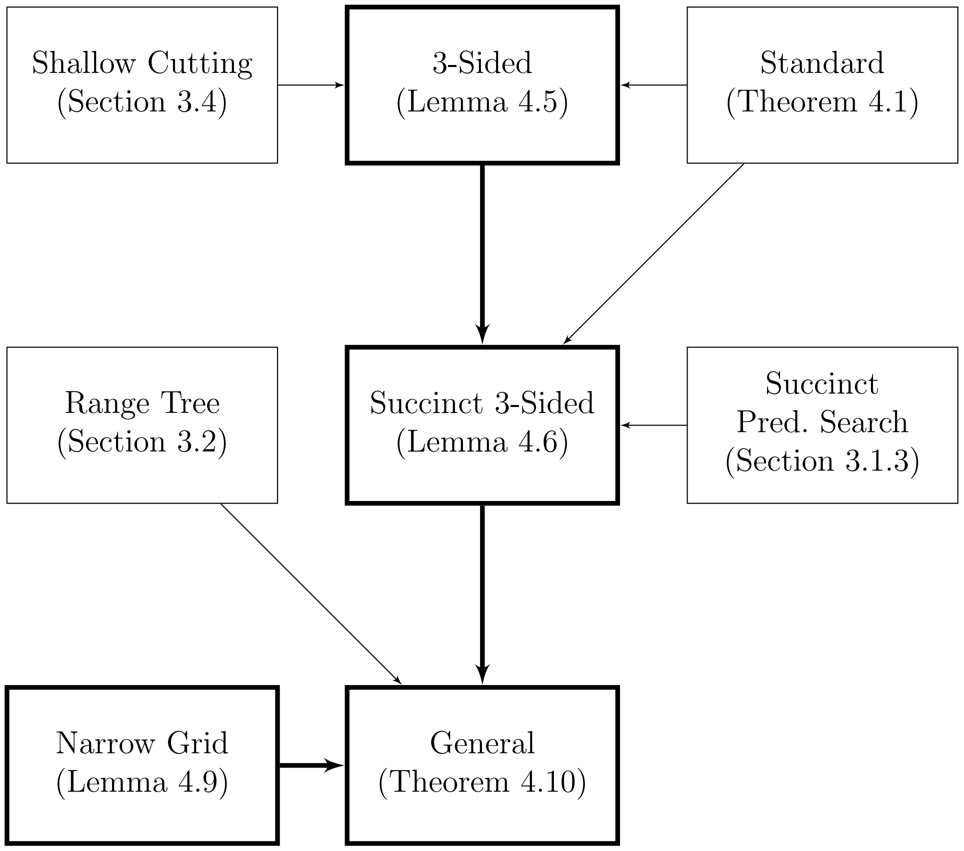


Figure 4.2: Overview of our shallow range counting data structure

In the design of succinct data structures, an important technique involves finding ways to store pointers to input elements in fewer than $O(\log n)$ bits. An easy way to achieve this is to divide the input into blocks of size $\log^{O(1)} n$. Then, local pointers within a block require only $O(\log \log n)$ bits of space. We improve the space bound of Lemma 4.5 with this idea in mind. We also apply existing succinct data structures for predecessor search (see Section 3.1.3).

Lemma 4.6. *There exists a data structure for K-shallow 3-sided 2-D orthogonal range counting requiring $O(n \log(K \log n))$ bits of space and $O(\log \log n + \log_w K + A(n))$ query time, where $A(n)$ is the time required for oracle access to a point given its x-rank in P.*

Proof. We partition P by x-coordinate into n/B vertical *slabs* $S_1, S_2, \ldots, S_{n/B}$, each containing $B = K \log n$ points. As allowed by Lemma 4.2, we decompose a 3-sided query of the form $[\ell, r] \times [1, t]$ into at most two 3-sided queries within slabs and at most one *slab-aligned* query: a 3-sided query whose boundaries along the x-axis align with slab boundaries.

In each slab S_i we build the standard range counting data structure of Theorem 4.1 for the points in S_i's rank space, which requires $O(B \log B)$ bits of space. We also build the succinct predecessor search data structure of Lemma 3.5 for the points of S_i along both the x- and y-axes. These succinct data structures require only $O(B \log \log n)$ bits of space, but during a search they require oracle access to the full $(\log n)$-bit coordinates of $O(1)$ points, given their x- or y-ranks within S_i. For now, we simply convert oracle access given x- or y-rank within S_i to oracle access given x-rank within P. Given an x-rank j within S_i, the corresponding x-rank within P is $(i-1)B + j$, which we can easily compute in $O(1)$ time. Given a y-rank j within S_i, it is now sufficient to find the corresponding x-rank within S_i. We create an array requiring $O(B \log B)$ bits to map y-ranks to x-ranks within S_i in constant time. Across all slabs we require $O(n \log B) = O(n \log(K \log n))$ bits of space. Given a query Q within S_i, we first convert Q into a query Q' in S_i's rank space via predecessor search in $O(\log \log n + A(n))$ time, where $A(n)$ is the time required for oracle access to a point given its x-rank in P. We then forward Q' onto the standard range counting data structure stored for S_i. Since $|S_i| = B = K \log n$, the standard query takes $O(\log_w |S_i|) = O(\log_w K)$ time.

It remains to handle slab-aligned queries. Assume we are given one such query Q. If $|Q \cap S_i| > K$, then we know that the $K + 1$ points in S_i with the least y-coordinates lie in Q. The converse is also trivially true. Let S'_i be the set of $K + 1$ points in S_i with the least y-coordinates. Then, for the purposes of K-shallow range counting, computing $|Q \cap S'_i|$ instead of $|Q \cap S_i|$ is sufficient for a single slab S_i: if $|S_i \cap Q| \leq K$ then $|S'_i \cap Q| = |S_i \cap Q|$ and if $|S_i \cap Q| > K$ then $|S'_i \cap Q| > K$. To handle multiple slabs simultaneously, we construct a set of points $P' = \bigcup_{i=1}^{n/B} S'_i$ and build the data structure of Lemma 4.5 over these points. Since $|P'| = (n/B)K = n/\log n$, this data structure requires only $O(n)$ bits of space. We compute $|P' \cap Q|$ in $O(\log\log n + \log_w K)$ time. If $|P' \cap Q| \leq K$ then $k = |P \cap Q| = |P' \cap Q|$. If $|P' \cap Q| > K$ then $k = |P \cap Q| > K$ and we output an error. □

Our intention is to use a range tree in combination with Lemma 4.6 in order to support 4-sided queries. In order to ensure that space consumption does not increase by a full multiplicative $O(\log n)$ factor, we use instead a B-ary tree, for some appropriate fan-out of B. Answering a query then involves a 4-sided query on a narrow $[B] \times [n]$ grid. An important component of the optimal standard range counting data structure of JaJa et al. [JMS05] is an efficient solution to the range counting problem on a very narrow grid.

Lemma 4.7 (JaJa et al. [JMS05])**.** *There exists a data structure for 2-D orthogonal range counting on a $[w^\epsilon] \times [n]$ grid requiring $O(n \log w)$ bits of space and $O(1)$ query time.*

Both JaJa et al. [JMS05] and Nekrich [Nek09b] apply w^ϵ-ary range trees in order to support queries on grids that are less narrow. The following is a simple generalization of their technique that reduces a query on the $[n] \times [n]$ grid to multiple queries on $[w^\epsilon] \times [n]$ grids.

Lemma 4.8 (JaJa et al. [JMS05], Nekrich [Nek09b])**.** *Given a data structure D for 2-D orthogonal range counting on a $[w^\epsilon] \times [n]$ grid, for any $B \in [n]$, there exists a data structure for the same problem on a $[B] \times [n]$ grid requiring $O(S_D(n) \log_w B)$ space and $O(Q_D(n) \log_w B)$ query time.*

Lemma 4.9. *For any $B \in [n]$, there exists a data structure for 2-D orthogonal range counting on a $[B] \times [n]$ grid requiring $O(n \log B)$ bits of space and $O(\log_w B)$ query time.*

Proof. By Lemmata 4.8 and 4.7. □

Theorem 4.10. *Given a data structure D for the ball inheritance problem, there exists a data structure for K-shallow 2-D orthogonal range counting requiring $O(n + S_D(n))$ space and $O(\log\log n + \log_w K + Q_D(n))$ query time.*

Proof. Assume for simplicity of presentation that n is a power of 2. We build a B-ary range tree, where B is the first power of 2 no less than $K \log n$. Then, B is a power of 2 and is also $\Theta(K \log n)$.

Given a 4-sided query, we find the node v of the range tree that contains the two vertical sides of the query in different child nodes. Finding v reduces to finding the LCA of two leaves of the range tree. There exists a linear-space data structure for computing LCAs in a tree in only $O(1)$ time [HT84]. Then, as allowed by Lemma 4.2, we decompose the query into at most two 3-sided query in child nodes of v and at most one *slab-aligned* query in v: a 4-sided query whose boundaries along the x-axis align with the boundaries of children of v.

At each node of the range tree, we build the data structure of Lemma 4.6 to handle 3-sided queries of the form $[1, r] \times [b, t]$ and $[\ell, 1] \times [b, t]$. These data structures require oracle access to points given their y-ranks. Since B is a power of 2, every node in our range tree corresponds to some node in a binary range tree. Lemma 3.7, a solution to the ball inheritance problem in a binary tree, is thus exactly what we need to implement the oracle access required by Lemma 4.6. We also augment this binary range tree to allow predecessor search at every node as in Lemma 3.11. Excluding the augmented binary tree and the ball inheritance data structure, this component of our data structure requires $O(n \log(K \log n))$ bits of space at each node of the B-ary range tree, for a total of $O(n)$ words of space since the tree has height $O(\log_B n) = O(\log n/\log(K \log n))$. Given a 3-sided query Q at a child u of node v, we first convert Q into a query Q' in u's rank space using the augmented range tree. We then forward Q' to the 3-sided data structure stored for u. The query time is thus $O(\log\log n + \log_w K + Q_D(n))$, where D is the data structure for the ball inheritance problem.

It remains to handle slab-aligned queries. At each node we construct a point set P' by rounding the x-coordinates of all of the points of P to the boundaries of child nodes. Then, for any query Q whose boundaries align with those of child nodes, $|Q \cap P'| = |Q \cap P|$, so computing $|Q \cap P'|$ is sufficient. Since the points of P' lie on a narrow $[B] \times [n]$ grid, we construct the data structure of Lemma 4.9 to handle these queries in $O(\log_w B) = O(\log_w K)$ time. Again, the space requirement at every node is $O(n \log B) = O(n \log(K \log n))$ bits, for a total of $O(n)$ words of space across the entire B-ary range tree. □

Corollary 4.11. *There exists a data structure for K-shallow 2-D orthogonal range counting requiring $O(n \log\log n)$ space and $O(\log\log n + \log_w K)$ query time.*

Proof. By Corollary 3.8. □

Corollary 4.12. *There exists a data structure for K-shallow 2-D orthogonal range counting, requiring $O(n)$ space and $O(\log^\epsilon n + \log_w K)$ query time.*

Proof. By Corollary 3.9. □

4.3 Approximate Range Counting

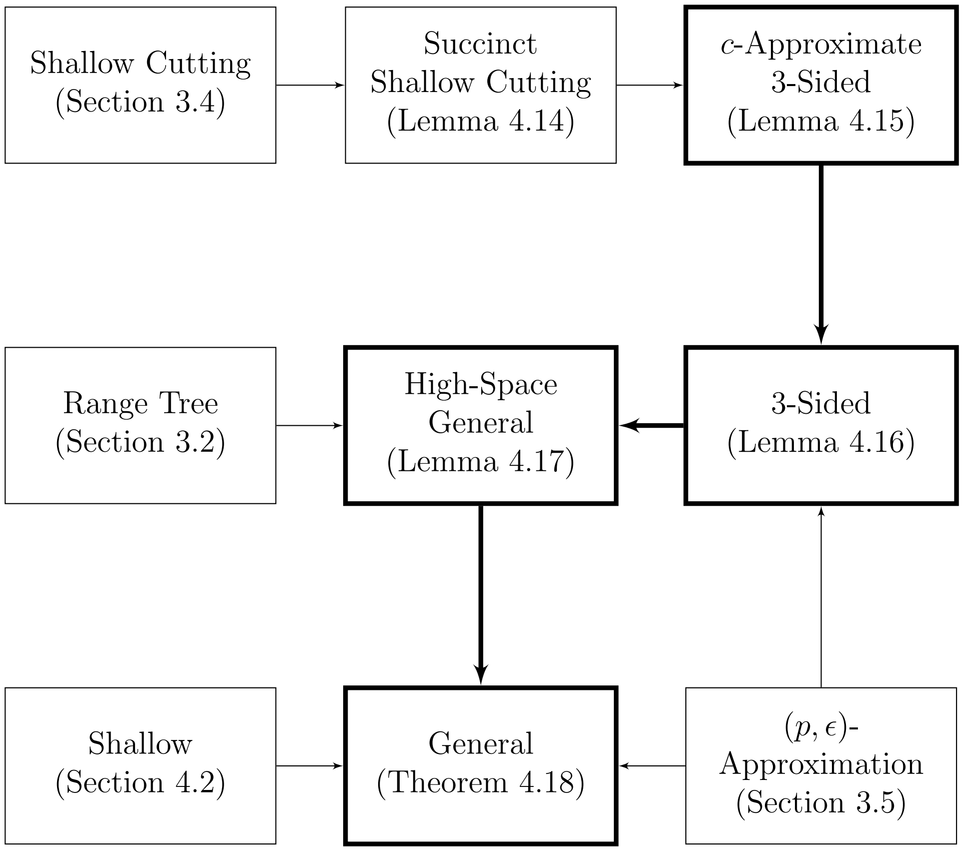
In this section, we give approximate data structures for 2-D orthogonal range counting. Our key ingredient is a succinct data structure representation of shallow cuttings for 3-sided ranges (see Section 3.4). By building K-shallow cuttings for exponentially increasing values of K, we can find, for 3-sided queries, a multiplicative constant-factor approximation of k in $O(\log\log n)$ time via a binary search over the $O(\log n)$ shallow cuttings. We refine the approximation k' such that $(1-\delta)k \leq k' \leq (1+\delta)k$ using relative (p,ϵ)-approximations (see Section 3.5) of the points in each shallow cutting cell. We apply a range tree to support 4-sided queries using our 3-sided data structure, at the expense of a logarithmic increase in space consumption. In order to reduce space, we build the range tree on a relative (p,ϵ)-approximation of P so that we can only compute accurate approximations for ranges containing at least $\log^{O(1)} n$ points. We build a $(\log^{O(1)} n)$-shallow data structure to handle ranges containing fewer than $\log^{O(1)} n$ points. We summarize our high-level plan in Figure 4.3.

We temporarily consider points that have not been reduced to rank space. Thus, for now, the points of P are from $[U]^2$ instead of from $[n]^2$. We do so to highlight how our data structures use space and to highlight the necessity for rank space reduction in a later data structure.

We begin by describing a succinct data structure representation of a shallow cutting for 3-sided queries. This data structure can find shallow cutting cells containing 3-sided queries in constant time without the need for $O(\log\log n)$-time 2-D orthogonal point location queries.

In the construction of the shallow cuttings of Jørgensen and Larsen [JL11], a horizontal sweep line passes from $y=1$ to $y=U$. Throughout the sweep, we maintain a partition of the plane into vertical *slabs*, initially containing a single slab $(-\infty,\infty)\times[U]$. During the sweep, if any slab contains $2K+2$ points on or below the sweep line $y=y_0$, this slab is split in two at the median m of the x-coordinates of the $2K+2$ points. Let (m,y_0) be a *split point*. Throughout the sweep, we build a set $S=\{s_1,s_2,\ldots,s_{|S|}\}$ of all split points sorted in order of x-coordinate.

Let $X=\{x_1,x_2,\ldots,x_{|X|}\}$ be the x-coordinates of all split points immediately following the insertion of a new split point with x-coordinate x_i. Assume the sweep line is at $y=y_0$. After the insertion of the split point, we construct two shallow cutting cells. The first cell contains all points of P that lie in $[x_{i-2},x_{i+1}]\times[1,y_0]$. The second cell contains all points of P that lie in $[x_{i-1},x_{i+2}]\times[1,y_0]$. Each cell is assigned a *key*, which is a horizontal segment used to help determine in which cell a query range lies. The key for the cell defined by the



Shallow Cutting (Section 3.4)
Succinct Shallow Cutting (Lemma 4.14)
c-Approximate 3-Sided (Lemma 4.15)
Range Tree (Section 3.2)
High-Space General (Lemma 4.17)
3-Sided (Lemma 4.16)
Shallow (Section 4.2)
General (Theorem 4.18)
(p, ϵ)-Approximation (Section 3.5)

Figure 4.3: Overview of our approximate range counting data structure

range $[x_{i-2}, x_{i+1}] \times [1, y_0]$ is the segment with x-interval $[x_{i-1}, x_i]$ at height y_0. The key for the cell defined by the range $[x_{i-1}, x_{i+2}] \times [1, y_0]$ is the segment with x-interval $[x_i, x_{i+1}]$ at height y_0. For each pair of adjacent slabs in the final partition of the plane, we create an additional *keyless* shallow cutting cell containing all points in both slabs. We highlight the following property of the resulting shallow cutting for later use.

Observation 4.13. *Every cell in the shallow cuttings of Jørgensen and Larsen [JL11] is the intersection of some range of the form $[\ell, r] \times [1, t]$ with P.*

Note that an invariant of the sweep is that all slabs contain at most $O(K)$ points on or below the sweep line. All cells (keyless or not) also contain $O(K)$ point as each cell overlaps a constant number of slabs (at the time of its creation) and only contains points on or below the sweep line. A new slab is created only when the number of points in an existing slab on or under the sweep line grows by at least $K+1$. Thus, the the shallow cutting contains $O(n/K)$ cell.

Assume we are given a 3-sided query $Q = [\ell, r] \times [1, t]$. If the slab $[\ell, r] \times [U]$ does not contain any key, then Q lies in one of the keyless shallow cutting cells. Otherwise, consider the lowest key contained in $[\ell, r] \times [U]$ and let $X = \{x_1, x_2, \ldots, x_{|X|}\}$ be the x-coordinates of all split points at the time of its creation. Assume without loss of generality that the key has x-interval $[x_i, x_{i+1}]$ and height y_0. Assume that $\ell < x_{i-1}$. Then, there is a key in $[\ell, r] \times [U]$ with x-interval $[x_{i-1}, x_{i+1}]$ and height less than y_0: a contradiction. Thus, $\ell \geq x_{i-1}$. Similarly, $r \leq x_{i+2}$. So, the x-interval of Q lies in the x-interval of the key's cell. If t is no less than the y_0, then Q contains at least $K+1$ points since there are exactly $K+1$ points on or under every key. In this case, Q does not need to lie in a shallow cutting cell as it is not K-shallow. If t is less than y_0, then Q lies in the key's cell.

Lemma 4.14. *There exists a data structure representation of a K-shallow cutting that requires $O(U + (n/K) \log U)$ bits of space and that can find a shallow cutting cell containing a 3-sided query, or determine that no such cell exists, in $O(1)$ time.*

Proof. Assume we are given a 3-sided query $Q = [\ell, r] \times [1, t]$. In order to determine whether or not Q lies in a keyless cell, it is sufficient to count the number of split points in $Q' = [\ell, r] \times [U]$. If there are fewer than two, then Q lies in a keyless cell. We can count the number of split points in Q' as well as determine the keyless cell containing Q via predecessor search in S. We build the predecessor search data structure of Lemma 3.6, which requires $O(U)$ bits of space and $O(1)$ query time.

If Q' contains at least one key, it is sufficient to find the lowest key in Q'. Consider first the lowest split point s_i in Q'. There are two keys that share s_i as an endpoint. The other endpoint of each of these keys must either be directly above another split point or at negative or positive infinity along the x-axis. In the former case, since s_i is the lowest split point in Q', the key must extend to the x-coordinate of some split point that is outside of Q'. In the latter case, the key is infinitely long. In either case, Q' cannot contain the key.

Consider the second highest split point s_j in Q' and assume without loss of generality that it lies to the left of s_i. Then, there is a key whose left endpoint is s_j and whose right endpoint lies above s_i. This key is thus contained in Q'. It is also the lowest key in Q' since neither of the keys associated with s_i are in Q'. Thus, we have reduced finding the lowest key in Q' to finding the second lowest split point in Q'.

We build a Cartesian tree over the points of S. We encode the tree using the succinct data structure of Theorem 3.3 which allows constant-time LCA queries and requires $O(n/K)$ bits of space. Finding the second lowest point in $S \cap Q'$ using a Cartesian tree reduces to a constant number of predecessor searches and LCA queries (see Section 3.3). Since we have succinct data structures for both of these problems with constant query time, we can find the second lowest point in constant time.

In order to be able to compare t with the y-coordinate of the lowest key, we must store all of the y-coordinates of all of the keys. Storing these coordinates requires a total of $O((n/K)\log U)$ bits. □

A *c-approximate* range counting data structure is permitted to overestimate the number of points in a given range by a constant multiplicative factor. In particular, if the data structure outputs a count k', then it must hold that $k \leq k' \leq ck$.

Lemma 4.15. *There exists a data structure for c-approximate 3-sided 2-D orthogonal range counting requiring $O(U)$ space and $O(\log\log n)$ query time.*

Proof. We construct 2^i-shallow cuttings for $i \in [\lceil \log n \rceil]$ and represent each with the data structure of Lemma 4.14. In total, the space requirement of these data structures in bits is

$$\begin{aligned} S(n) &= \sum_{i=1}^{\lceil \log n \rceil} (U + (n/2^i)\log U) \\ &= O(U \log U). \end{aligned}$$

Additionally, for each cell of the 2-shallow cutting, we store all of the points in the cell in an array. Since this shallow cutting has $O(n)$ cells and each cell contains $O(1)$ points, these arrays require $O(n) \subseteq O(U)$ space.

Given a query Q, we perform a binary search amongst our $O(\log n)$ shallow cuttings to find the shallowest shallow cutting with a cell C that contains Q. This binary search takes $O(\log\log n)$ time, since for each shallow cutting we can determine in $O(1)$ time whether or not it has a cell containing Q. Assume C is a cell of the 2^i-shallow cutting. If $i = 1$, we determine $k' = k$ exactly in $O(1)$ time by iterating through the $O(1)$ points of C which are stored for the 2-shallow cutting only. Since the size of each cell in a K-shallow cutting is $O(K)$, there exists some constant c' such that all cells in all of our 2^j-shallow cuttings have size no greater than $c'2^j$. If $i > 1$, we output the approximate count $k' = c'2^i$. Since C contains Q and since $|C| \leq c'2^i$, we have $k \leq c'2^i = k'$. There must not be a cell in the 2^{i-1}-shallow cutting that contains Q. So, $k > 2^{i-1}$ and we have $k' < 2c'k$. Thus, $k \leq k' \leq ck$, where $c = 2c'$. □

We now refine the approximation of the data structure of Lemma 4.15 by building relative (p, ϵ)-approximations of the points in each shallow cutting cell. Since we have K-shallow cuttings for exponentially increasing values of K, for the shallowest cell C containing a query range Q, it must be that $|C \cap Q| = O(|C|)$. Thus, we only need a relative (p, ϵ)-approximation of constant size in each shallow cutting cell.

Lemma 4.16. *There exists a data structure for* $(1+\delta)$*-approximate* 3*-sided* 2*-D orthogonal range counting requiring* $O(U)$ *space and* $O(\log\log n)$ *query time.*

Proof. We extend the c-approximate data structure of Lemma 4.15. For each cell C of each 2^i-shallow cutting for $i > 1$ we store a relative (p, ϵ)-approximation C' of C for $p = 1/c$ and $\epsilon = \delta$. By Theorem 3.13 there exists such a relative (p, ϵ)-approximation such that $|C'| = O(1)$. Space remains linear in U since there are a total of $O(n) \subseteq O(U)$ cells across all $O(\log n)$ shallow cuttings.

As in Lemma 4.15, during a query we find the shallowest shallow cutting that contains Q in some cell C. We output the approximate count $k' = (|C|/|C'|)|C' \cap Q|$. We compute $|C' \cap Q|$ in constant time by iterating through the $O(1)$ points of C'. Since $|C \cap Q| > 2^{i-1}$ and since $|C| \leq c'2^i$, we have $|C \cap Q|/|C| \geq 1/(2c') = 1/c = p$. Then, by the definition of a relative (p, ϵ)-approximation, we have

$$(1-\delta)\frac{|C \cap Q|}{|C|} \leq \frac{|C' \cap Q|}{|C'|} \leq (1+\delta)\frac{|C \cap Q|}{|C|}.$$

Since $|C \cap Q| = |P \cap Q| = k$, we have $(1-\delta)k \leq k' \leq (1+\delta)k$. □

We now return to considering points in rank space. Thus, the points of P are from $[n]^2$. The next step is to support 4-sided queries using a range tree.

Lemma 4.17. *There exists a data structure for* $(1+\delta)$*-approximate* 2*-D orthogonal range counting requiring* $O(n \log n)$ *space and* $O(\log \log n)$ *query time.*

Proof. We build a range tree, storing the data structure of Lemma 4.16 at each node to handle 3-sided queries of the form $[1, r] \times [b, t]$ and $[\ell, 1] \times [b, t]$. We store each data structure in the rank space of its node so that its space requirement is linear in the size of the node instead of linear in n. At each node we also store a linear-space data structure for predecessor search that supports queries in $O(\log \log n)$ time [Wil83]. The size of the range tree is thus $O(n \log n)$.

Given a 4-sided query, we find the pair of sibling nodes in the range tree that each contain only one of the two vertical sides of the query. As allowed by Lemma 4.3, we decompose the 4-sided query into a 3-sided query in each of these sibling nodes. Finding the sibling nodes reduces to finding the LCA of two nodes of the range tree. There exists a linear-space data structure for computing LCAs in a tree in only $O(1)$ time [HT84]. For a 3-sided query Q at node v, we convert Q into a query Q' in v's rank space via predecessor search in $O(\log \log n)$ time. We then forward Q' to the 3-sided data structure stored for v, which outputs an approximate count in $O(\log \log n)$ time. We output the sum of the approximate counts for both 3-sided queries. □

Finally, we reduce space by building the data structure of Lemma 4.17 only for a relative (p, ϵ)-approximation of P. For ranges containing few points, we need an alternative data structure: our shallow data structure is sufficient!

Theorem 4.18. *Given a data structure* D *for the ball inheritance problem, there exists a data structure for* $(1+\delta)$*-approximate* 2*-D orthogonal range counting requiring* $O(n+S_D(n))$ *space and* $O(\log \log n + Q_D(n))$ *query time.*

Proof. We build the data structure of Theorem 4.10, setting $K = \log^2 n$. This data structure requires $O(n + S_D(n))$ space and computes exact counts for queries containing up to $\log^2 n$ points in $O(\log \log n + Q_D(n))$ time. However, if the data structure may output an error if the count is greater than $\log^2 n$. To handle queries such that $k > \log^2 n$, we build a relative (p, ϵ)-approximation P' of P setting $p = (\log^2 n)/n$ and $\epsilon = \delta_1$. By Theorem 3.13, $|P'| = O((1/p) \log(1/p)) = O(n/ \log n)$. We construct the $(1 + \delta_2)$-approximate data structure of Lemma 4.17 on the points of P', which requires $O(n)$ space. Given a query Q such that $k > \log^2 n$, we first compute the $(1 + \delta_2)$-approximate count k'' for $P' \cap Q$ in $O(\log \log n)$ time. We then output $k' = (|P|/|P'|)k''$. Since $k > \log^2 n$, we have $|P \cap Q|/|P| > (\log^2 n)/n = p$. By the definition of a relative (p, ϵ)-approximation, we have

$$(1-\delta_1)\frac{|P\cap Q|}{|P|} \le \frac{|P'\cap Q|}{|P'|} \le (1+\delta_1)\frac{|P\cap Q|}{|P|}.$$

We thus have $(1-\delta)k \le k' \le (1+\delta)k$ for $\delta = \delta_1 + \delta_2 + \delta_1\delta_2$. □

Corollary 4.19. *There exists a data structure for $(1+\delta)$-approximate 2-D orthogonal range counting requiring $O(n \log\log n)$ space and $O(\log\log n)$ query time.*

Proof. By Corollary 3.8. □

Corollary 4.20. *There exists a data structure for $(1+\delta)$-approximate 2-D orthogonal range counting requiring $O(n)$ space and $O(\log^{\epsilon} n)$ query time.*

Proof. By Corollary 3.9. □

4.4 Adaptive Range Counting

We finally combine our shallow and approximate data structures for 2-D orthogonal range counting to create an adaptive data structure for the same problem.

Theorem 4.21. *There exists a data structure for 2-D orthogonal range counting requiring $O(n \log\log n)$ space and $O(\log\log n + \log_w k)$ query time.*

Proof. We build the K-shallow data structure of Theorem 4.10 for $K = 2^{2^i}$ for $i \in [\lceil \log\log n \rceil]$. We reuse the ball inheritance data structure of Corollary 3.8 across all $O(\log\log n)$ of these data structures. The total space requirement is $O(n \log\log n)$. We also build the $(1+\delta)$-approximate data structure of Corollary 4.19, which also requires $O(n \log\log n)$ space.

Given a query, we first compute a $(1+\delta)$-approximate count k' in $O(\log\log n)$ time. We compute $k'' = (1/(1-\delta))k'$ so that $k'' \ge k$ and $k'' = O(k)$. We find the least i such that $2^{2^i} \ge k''$. Since $k \le k'' \le 2^{2^i}$, we can query the 2^{2^i}-shallow data structure to obtain an accurate count in $O(\log\log n + \log_w 2^{2^i})$ time. By our definition of i we know $k'' > 2^{2^{i-1}}$. Thus $k = \Omega(2^{2^{i-1}})$ and $\log_w k = \Omega(\log_w 2^{2^i})$. □

Chapter 5

Range Selection

Given an array A of n elements from $[U]$, we consider the problem of preprocessing A so that we can efficiently compute the k^{th} lowest element in subarray $A[\ell : r]$, given $k, \ell, r \in [n]$ such that $\ell \leq r$ and $k \leq r - \ell + 1$. We immediately note that we can reduce the elements of A to rank space without incurring the query time cost of predecessor search, since none of the query parameters are elements of A. Thus, if a query in rank space outputs the rank k', then we can select in constant time the element from $[U]$ with rank k' in A via an array representation of the elements of A indexed by rank. Going forward, we assume that A is in rank space and thus each of the n elements is from $[n]$.

The range selection query problem is very closely related to 3-sided 2-D range counting. Consider the 2-D point set $P = \{(i, A[i]) \mid i \in [n]\}$. Finding the k^{th} lowest element in $A[\ell : r]$ is equivalent to finding the point $p_i = (i, A[i])$ with the k^{th} lowest y-coordinate in $P \cap Q$, where $Q = [\ell, r] \times [n]$. Equivalently, p_i is the point for which $|P \cap Q'| = k$, where Q' is the 3-sided range $[\ell, r] \times [1, A[i]]$. Going forward, we work with this geometric interpretation of the range selection query problem. Similarly to 2-D orthogonal range counting, there is an efficient linear-space data structure that answers range selection queries in $o(\log n)$ time.

Theorem 5.1 (Brodal and Jørgensen [BJ09])**.** *There exists a data structure for the range selection query problem requiring $O(n)$ space and $O(\log_w n)$ query time.*

Jørgensen and Larsen [JL11] give a matching lower bound, but also investigate adaptive data structures with query times that are sensitive to k. They give a linear-space data structure that requires $O(\log \log n + \log_w k)$ query time and an $\Omega(\log_w k)$ query time lower bound. Their data structure is thus optimal for all but small values of k.

A key difference between the range selection query problem and 2-D orthogonal range counting is that the latter is decomposable while the former is not. If we are given the sizes of two multisets we can easily compute the size of the union of the multisets in $O(1)$ time via addition. However, if we are given, for example, the elements with rank k in two multisets we cannot efficiently compute the element with rank k in the union of the multisets. This difference is the reason that we cannot directly apply the techniques of Section 4.2 to solve the range selection query problem.

Shallow cuttings are a tool that we still can use, since reducing a query over the points of P to a query over the points of a single shallow cell does not decompose the query into multiple parts. However, in the context of the range selection query problem, a K-shallow cutting requires an additional property in order to be useful. In particular, if a query Q of the form $[\ell, r] \times [n]$ contains at least $k \leq K$ points, then there must exist a cell C such that the point with the k^{th} lowest y-coordinate in $C \cap Q$ is the point with the k^{th} lowest y-coordinate in $P \cap Q$. We say shallow cutting cell C resolves query Q. The shallow cuttings of Jørgensen and Larsen [JL11] satisfy this property and play an important role in their data structure. Their data structure applies shallow cuttings recursively. Rank space reduction is applied at each level of recursion in order to reduce space.

We take a different approach to reduce space and instead use another succinct data structure representation of a shallow cutting. We then follow the approach of Chapter 4 by first creating a succinct data structure for K-shallow range selection and then building such a data structure for double exponentially increasing values of K in order to create an adaptive data structure. The query parameter k of a K-shallow query is restricted such that $k \leq K$. We summarize our high-level plan in Figure 5.1.

Lemma 5.2. *There exists a data structure representation of a K-shallow cutting that requires $O(n \log(K \log n) + (n/K) \log n)$ bits of space and, given a query Q of the form $[\ell, r] \times [n]$, can:*

- *find a shallow cutting cell that resolves Q, or determine that no such cell exists, in $O(1)$ time, and*
- *access the full $(\log n)$-bit coordinates of a point with a given x-rank in a given cell in $O(A(n))$ time, where $A(n)$ is the time required for oracle access to a point given its x-rank in P.*

Proof. The data structure of Lemma 4.14 requires $O(n + (n/K) \log n)$ bits of space and supports finding cells that resolve queries in $O(1)$ time. In order to give access to points by x-rank in a cell, we adapt the technique of Lemma 4.6.

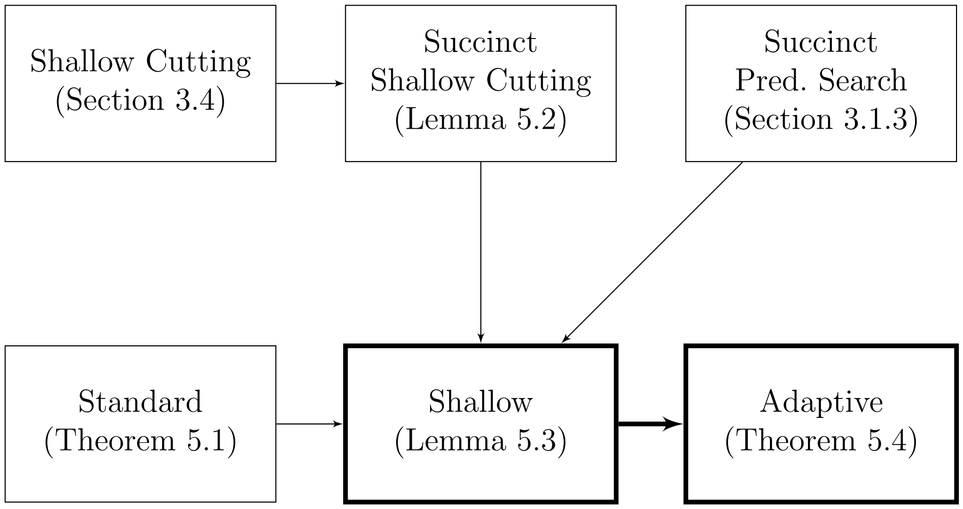


Figure 5.1: Overview of our adaptive range selection data structure

We partition P by x-coordinate into n/B vertical *slabs* $S_1, S_2, \ldots, S_{n/B}$, each containing $B = K \log n$ points. Since the size of each cell in our K-shallow cutting is $O(K)$, there exists some constant c such that each cell of our K-shallow cutting has size no greater than cK. Let S'_i be the set of cK points in S_i with the least y-coordinates. Let $P' = \bigcup_{i=1}^{n/B} S'_i$ so that $|P'| = O(n/\log n)$. We construct a cK-shallow cutting of P'. This shallow cutting has $O(n/(K \log n))$ cells, each containing $O(K)$ points.

Consider a cell C of our K-shallow cutting of P. By Observation 4.13, the points of C are exactly $P \cap R$ for some 3-sided range R of the form $[\ell, r] \times [1, t]$. Let S_a be the slab that contains the left vertical side of R and let S_b be the slab that contains the right vertical side of R. Finally, consider the points $C' = C \setminus (S_a \cup S_b)$. Assuming C' is not empty, these points are exactly $P \cap R'$, where R' is a 3-sided rectangle whose vertical sides are aligned with the slab boundaries of S_a and S_b and whose top side is at the same height as R. Since $C' \subseteq C$, we have $|C'| \leq cK$. Assume towards contradiction that there is a point $p \in C'$ such that $p \notin P'$. Let S_i be the slab containing p. Since $p \notin P'$, it must also be that $p \notin S'_i$. Since the vertical sides of R' are aligned with slab boundaries, C' must contain all points in S_i that are lower than p, including all points of S'_i. Since $|S'_i| = cK$, we have that $|C'| > cK$, a contradiction. Therefore, $C' \subseteq P'$ and $C' = P' \cap R'$. Since $|C'| \leq cK$, R' must lie in one of the cK-shallow cutting cells of P'. Let this cell be C^*. Each point $p \in C$ must either be in S_a, S_b, or C^*.

We store pointers to S_a, S_b, and C^* for C, which requires $O(\log n)$ bits. Across all cells of the K-shallow cutting of P, the space requirement is $O((n/K) \log n)$ bits. For each point $p \in C$, we store in a constant number of bits which of these three sets contains p. Let this set be X. We also store the x-rank of p within X. Since $|X| \leq \max\{|S_a|, |S_b|, |C^*|\} = O(K \log n)$, we require $O(\log(K \log n))$ bits per point. We store this information in an array T_C indexed by x-rank in C. Across all points of all cells of the K-shallow cutting of P, the space requirement is $O((n/K) \cdot K \cdot \log(K \log n)) = O(n \log(K \log n))$ bits. Given a point $p \in C$ and its x-rank i in C, we can then lookup $T_C[i]$, the x-rank of p in X, in constant time.

We store an array F_{C^*} containing the full $(\log n)$-bit coordinates of all of the points of C^* indexed by x-rank in C^*. Across all points of all cells of the cK-shallow cutting of P', the space requirement is $O((n/K \log n) \cdot K \cdot \log n) = O(n)$ bits. Assume we need access to the full $(\log n)$-bit coordinates of the point $p \in C$ with x-rank i in C^*. We then lookup $F_{C^*}[i]$, the full $(\log n)$-bit coordinates of p, in constant time.

Without loss of generality, assume we need access to a point $p \in C$ with x-rank i in S_a. We cannot afford to store the full $(\log n)$-bit coordinates of all points of all slabs, as doing so would require $O(n \log n)$ bits of space. Instead we use an oracle that gives access to, in $A(n)$ time, the full $(\log n)$-bit coordinates of a point given its x-rank in P. The x-rank of p within P is $(a-1)B + i$, which we can easily compute in $O(1)$ time. $\square$

The key new feature of the data structure of Lemma 5.2 is the ability to access a point in a shallow cutting cell by its rank without requiring linear space. We can then build succinct predecessor search data structures (see Section 3.1.3) in each cell and the total space requirement is still sublinear. Using predecessor search in a cell, we can reduce a query to rank space inside the cell. Thus, any linear-space data structure we build in the cell to handle a query requires only $O(K \log K)$ bits instead of $O(K \log n)$ bits.

Lemma 5.3. *There exists a data structure for the K-shallow range selection query problem that requires $O(n \log(K \log n) + (n/K) \log n)$ bits of space and $O(A(n) \log_w K)$ query time, where $A(n)$ is the time required for oracle access to a point given its x-rank in P.*

Proof. We build a K-shallow cutting of P and represent it with the data structure of Lemma 5.2. For each cell C of the shallow cutting, we build the succinct predecessor search data structure of Lemma 3.4 to search amongst the points of C along the x-axis. Each of the $O(n/K)$ cells requires $O(K \log \log n)$ bits of space for a total of $O(n \log \log n)$ bits. The succinct predecessor search data structure in each cell C requires oracle access to the full $(\log n)$-bit coordinates of $O(\log_w K)$ points, given their x-ranks within C. We implement this oracle access via the second operation of Lemma 5.2.

In each cell C, we also build the standard range selection data structure of Theorem 5.1 in the rank space of C. This data structure requires $O(K \log K)$ bits. Across all $O(n/K)$ cells, the space requirement is $O(n \log K)$ bits.

Given a query range Q of the form $[\ell, r] \times [n]$ and a query rank $k \leq K$, we first find a cell C that resolves Q in $O(1)$ time via the data structures of Lemma 5.2. Next, we reduce Q to a query range Q' in the rank space of C in $O(A(n) \log_w K)$ time, via the succinct predecessor search data structure for C. We forward the query range Q' and the query rank k on to the standard range selection data structure for C, which requires $O(\log_w K)$ time. The result is a point in C's rank space, which we convert to a point in P in $O(A(n))$ time via the second operation of Lemma 5.2. □

In the context of adaptive range counting, in order to obtain an adaptive data structure we need a K-shallow data structures for double exponentially increasing values of K as well as an approximate data structure. We need the approximate data structure to determine which K-shallow data structure to query. In the context of adaptive range selection, we are given k exactly as part of the query, thus we do not need any approximate data structure.

Theorem 5.4. *There exists a data structure for the range selection query problem that requires $O(n)$ space and $O(\log_w k)$ query time.*

Proof. We build the K-shallow data structure of Lemma 5.3 for $K = 2^{2^i}$ for $i \in [\lceil \log\log n \rceil]$. Each data structure requires the same oracle access to a point in P given its x-rank in P. We implement this oracle access by sorting the points of P in an array by x-rank. This implementation requires $O(n)$ space and $O(1)$ time for an access. The total space in bits required by all of our succinct shallow data structures is

$$\begin{aligned} S(n) &= \sum_{i=1}^{\lceil \log\log n \rceil} (n \log(2^{2^i} \log n) + (n/2^{2^i}) \log n) \\ &= O(n \log n). \end{aligned}$$

Given a query rank k, we forward the query to the 2^{2^i}-shallow data structure with least i such that $k \leq 2^{2^i}$. The query runs in $O(\log_w 2^{2^i})$ time. Since $k > 2^{2^{i-1}}$, we have $\log_w k = \Omega(\log_w 2^{2^i})$ and thus the query time is $O(\log_w k)$. □

Chapter 6

Range Mode and Least Frequent Element

The *frequency* of an element x in a multiset S, denoted $\mathrm{freq}_S(x)$, is the number of occurrences (i.e., the multiplicity) of x in S. A *mode* of S is an element $a \in S$ such that for all $x \in S$, $\mathrm{freq}_S(x) \leq \mathrm{freq}_S(a)$. A multiset S may have multiple distinct modes with equal frequencies. Similarly, a *least frequent element* of S is an element $a \in S$ such that for all $x \in S$, $\mathrm{freq}_S(x) \geq \mathrm{freq}_S(a)$.

Along with the mean and median, the mode is a fundamental statistic in data analysis. Given a sequence of n elements in an array A, a range query seeks to compute the corresponding statistic on the multiset determined by a subinterval of the list $A[\ell : r]$. The objective is to preprocess A to construct a data structure that supports efficient response to one or more subsequent range queries, where the corresponding input parameters (ℓ, r) are provided at query time. A range mean query is equivalent to a normalized range sum query, for which a precomputed prefix-sum array provides a linear-space static data structure with constant query time [KMS05]. Range median queries have been analyzed extensively in recent years, resulting in an optimal data structure [BJ09] and our optimal range selection data structure of Theorem 5.4 which supports range median queries as a special case. In contrast, range mode queries appear more challenging than range mean and median. As expressed recently by Brodal et al. [BGJS11, page 2]: "The problem of finding the most frequent element within a given array range is still rather open."

The best previous linear-space data structure for the range mode query problem is by Krizanc et al. [KMS05], who obtain a query time of $O(\sqrt{n} \log \log n)$. No better approach has been discovered in over five years, which leads one to suspect that a query time around

$O(\sqrt{n})$ might be the best one could hope for.

Indeed, we present strong evidence that purely combinatorial approaches cannot avoid the $\sqrt{n}$ effect in the preprocessing or query costs, up to polylogarithmic factors. The method of Krizanc et al. [KMS05] has preprocessing time around $O(n^{3/2}) = O(n\sqrt{n})$. More specifically, we show in Section 6.1 that Boolean matrix multiplication (matrix multiplication on $\{0, 1\}$-matrices with addition and multiplication replaced by OR and AND, respectively) of two $n \times n$ matrices reduces to n^2 range mode queries in an array of size $O(n^2)$. We show a similar reduction to the construction and use of a data structure for the range least frequent element problem. These reductions imply that any data structure for the range mode or least frequent element query problems must have either $\Omega(n^{\omega/2})$ preprocessing time or $\Omega(n^{\omega/2-1})$ query time in the worst case, where ω denotes the matrix multiplication exponent. Since the current best matrix multiplication algorithm has exponent 2.3727 [Wil11], we cannot obtain preprocessing time better than $n^{1.18635}$ and query time better than $n^{0.18635}$ simultaneously without a major breakthrough. Moreover, since the current best combinatorial algorithm for Boolean matrix multiplication (which avoids algebraic techniques as in Strassen's algorithm) has running time only a polylogarithmic factor better than cubic [BW09], we cannot obtain preprocessing time better than $O(n^{3/2})$ and query time better than $O(\sqrt{n})$ simultaneously by purely combinatorial techniques with current knowledge, except for a speedup by a polylogarithmic factor.

In view of the above hardness result, it is therefore worthwhile to pursue more modest improvements for the range mode query problem. Notably, can the extra $O(\log \log n)$ factor in the query time bound of Krizanc et al. [KMS05] be eliminated?

In Section 6.2, we give a data structure that accomplishes just that: with $O(n)$ space, we can answer range mode queries in $O(\sqrt{n})$ time. The data structure is based on—and in some ways simplifies—that of Krizanc et al. [KMS05], since we use only rudimentary structures (mostly arrays) and no complex predecessor search techniques. We also show that, with some additional ideas, the same technique can solve the range least frequent element query problem.

In the case of the range mode query problem only, we go beyond eliminating a mere $O(\log \log n)$ factor: in Section 6.5, we present a linear-space data structure that answers range mode queries in $o(\sqrt{n})$ time! The precise worst-case time bound is $O(\sqrt{n/w}) \subseteq O(\sqrt{n/\log n})$. As one might guess, bit packing tricks are used to achieve the speedup, but in addition we need a nontrivial combination of ideas, including partitioning elements into two sets (one with small maximum frequency and another with a small number of distinct elements), each handled by a different method, and an interesting application of succinct rank and select data structures.

Throughout this chapter, let $m \leq n$ denote the maximum frequency (i.e., the frequency of the mode of A), and let $\Delta \leq n$ denote the number of distinct elements in A.

6.1 Reductions from Boolean Matrix Multiplication

In this section, we show that Boolean matrix multiplication of two $n \times n$ matrices reduces to n^2 range mode queries in an array of size $O(n^2)$. In our proof we use two simple observations of Greve et al. [GJLT10]:

Observation 6.1 (Greve et al. [GJLT10])**.** *Let S be a multiset with elements from universe U. Then adding one of each element in U to S increases the frequency of the mode of S by one.*

Observation 6.2 (Greve et al. [GJLT10])**.** *Let S_1 and S_2 be two sets (not multisets) and let S be the multiset union of S_1 and S_2. Then the frequency of the mode of S is two if and only if $S_1 \cap S_2 \neq \emptyset$.*

Now, let L and R be two $n \times n$ Boolean matrices for which we are to compute the product $P = LR$. The entry $p_{i,j}$ in P must be 1 precisely if there exists at least one index k, where $1 \leq k \leq n$, such that $\ell_{i,k} = r_{k,j} = 1$. Our goal is to determine whether this is the case using one range mode query for each entry $p_{i,j}$. Our first step is to transform each row of L and each column of R into a set. For the i^{th} row of L, we construct the set L_i containing all those indices k for which $\ell_{i,k} = 1$, i.e., $L_i = \{k \mid \ell_{i,k} = 1\}$. Similarly we let $R_j = \{k \mid r_{k,j} = 1\}$. Clearly $p_{i,j} = 1$ if and only if $L_i \cap R_j \neq \emptyset$. By Observation 6.2, this condition can be tested if we can determine the frequency of the mode in the multiset union of L_i and R_j. Our last step is thus to embed all the sets L_i and R_j into an array, such that we can use range mode queries to perform these intersection tests for every pair (i, j). Our constructed array A has two parts, a left part A_L and a right part A_R. The array A is then simply the concatenation of A_L and A_R. The array A_L represents all the sets L_i. It consists of n *blocks* of n entries. The i^{th} block (entries $A_L[(i-1)n+1 : in]$) represents the set L_i, and it consists of the elements $[n] \setminus L_i$ in some arbitrary order, followed by the elements of L_i in some arbitrary order. The array A_R similarly represents the sets R_j and it also consists of n blocks of n entries. The j^{th} block represents the set R_j and it consists of the elements of R_j in some arbitrary order, followed by the elements $[n] \setminus R_j$ in some arbitrary order.

Now assume that $|L_i|$ and $|R_j|$ are known for each set L_i and R_j. We can now determine whether $L_i \cap R_j \neq \emptyset$ (i.e., whether $p_{i,j} = 1$) from the result of the range mode query on $A[\text{start}(i) : \text{end}(j)]$, where

$$\begin{aligned}\text{start}(i) &= in - |L_i| + 1\\ \text{end}(j) &= n^2 + (j-1)n + |R_j|.\end{aligned}$$

To see this, first observe that the index start(i) is the first index in A of the elements in L_i, and that end(j) is the last index in A of the elements in R_j. In addition to a suffix of the block representing L_i and a prefix of the block representing R_j, the subarray $A[\text{start}(i) : \text{end}(j)]$ contains $n-i$ complete blocks from A_L and $j-1$ complete blocks from A_R. Since a complete block contains one occurrence of each element of $[n]$, it follows from Observations 6.1 and 6.2 that $L_i \cap R_j \neq \emptyset$ (i.e., $p_{i,j} = 1$) if and only if the frequency of the mode in $A[\text{start}(i) : \text{end}(j)]$ is $2 + (n-i) + (j-1)$. The answer to the range mode query $(\text{start(i)}, \text{end}(j))$ thus allows us to determine whether $p_{i,j} = 1$ or not. The array A and the values $|L_i|$ and $|R_j|$ can clearly be computed in $O(n^2)$ time when given matrices L and R, thus we have the following result:

Theorem 6.3. *Given a data structure for the range mode query problem 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 $O(P(n^2) + n^2 Q(n^2) + n^2)$ time.*

We can also give a similar reduction from Boolean matrix multiplication to the range least frequent element query problem. Let S_1 and S_2 be two subsets of $[n]$ and let S be the multiset union of $[n] \setminus S_1$, $[n] \setminus S_2$, and $[n]$. Similarly to Observation 6.2, the frequency of the least frequent element of S is one if and only if $S_1 \cap S_2 \neq \emptyset$. Thus, when we construct the block corresponding to L_i in A_L we instead include the elements of L_i in some arbitrary order, followed by the elements $[n] \setminus L_i$ in some arbitrary order. In this way, we swap the elements L_i and $[n] \setminus L_i$ in the block. We perform a symmetric modification to A_R. We also include one additional block between A_L and A_R containing all elements of $[n]$ in some arbitrary order. We can then determine whether $L_i \cap R_j \neq \emptyset$ from the result of the range least frequent element query on $A[\text{start}(i) : \text{end}(j)]$, where

$$\begin{aligned}\text{start}(i) &= (i-1)n + |L_i| + 1\\ \text{end}(j) &= n^2 + (j+1)n - |R_j|.\end{aligned}$$

In particular, by a similar argument to that of our reduction to the range mode query problem, $L_i \cap R_j \neq \emptyset$ if and only if the frequency of the least frequent element in $A[\text{start}(i) :$

end(j)] is $(n-i)+(j-1)+1$. The reason that we include the additional block between A_L and A_R is simply to ensure that every element in $[n]$ occurs in each query range; our definition of least frequent element requires that the element has frequency greater than zero.

Theorem 6.4. *Given a data structure for the least frequent element query problem 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 $O(P(n^2)+n^2Q(n^2)+n^2)$ time.*

6.2 First Method: $O(\sqrt{n})$ Query Time

We begin by presenting a linear-space data structure with $O(\sqrt{n})$ query time, improving the query time of the data structure of Krizanc et al. [KMS05] by an $O(\log\log n)$ factor. We build on their data structure and introduce a different technique that avoids the need for predecessor search, which is the origin of the $O(\log\log n)$ factor in their query time. In Section 6.2.1 we describe our data structure for the range mode query problem. In Section 6.2.2 we adapt our technique to solve the range least frequent element problem.

6.2.1 Range Mode

We actually establish the following time-space tradeoff—the linear-space result follows by setting the parameter $s = \lceil\sqrt{n}\rceil$.

Theorem 6.5. *For any $s \in [n]$, there exists a data structure for the range mode query problem in an array requiring $O(n+s^2)$ space, $O(ns)$ preprocessing time, and $O(n/s)$ query time.*

The following observation will be useful:

Observation 6.6 (Krizanc et al. [KMS05])**.** *Let S_1 and S_2 be any multisets. If element e is a mode of $S_1 \cup S_2$ and $e \notin S_1$, then e is a mode of S_2.*

We are given an array A of n elements. In general, these elements are from $[U]$. However, we can work instead in rank space, representing each element by its rank in A instead of its actual value. Since there are Δ distinct elements in A, the ranks of elements are from $[\Delta]$. Thus, if a query in rank space outputs the rank k, then we can select in constant time the distinct element with rank k in A via an array representation of the distinct elements of A indexed by rank. Reducing A to rank space and constructing the sorted array of distinct elements requires $O(n \log \Delta)$ time during preprocessing. Going forward, we assume that A is in rank space and thus each of the n elements is from $[\Delta]$. We also assume that we know Δ.

Preprocessing. For each $x \in [\Delta]$, let $I_x = \{i \mid A[i] = x\}$. That is, I_x is the set of indices i such that $A[i] = x$. For any element x, a range counting query for element x in $A[\ell : r]$ can be answered by searching for the predecessors of ℓ and r, respectively, in the set I_x; the difference of the indices of the two predecessors is the frequency of x in $A[\ell : r]$ [KMS05]. Such a range counting query can be implemented using an efficient linear-space predecessor search data structure [Wil83] in $\Theta(\log \log n)$ time in the worst case.

The following related decision problem, however, can be answered in constant time by a linear-space data structure: does $A[\ell : r]$ contain at least k instances of element $A[\ell]$? This question can be answered by a "select" query that returns the index of the k^{th} occurrence of $A[\ell]$ in $A[\ell : n]$. For each $x \in [\Delta]$, we store the set I_x as a sorted array (also denoted I_x for simplicity). Define a "rank" array $R[1 : n]$ such that for all i, $R[i]$ denotes the multiplicity of $A[i]$ in $A[1 : i]$. Given any k, ℓ, and r, to determine whether $A[\ell : r]$ contains at least k instances of $A[\ell]$ it suffices to check whether $I_{A[\ell]}[R[\ell] + k - 1] \leq r$. Since array $I_{A[\ell]}$ stores the sequence of indices of instances of element $A[\ell]$ in A, looking ahead $k - 1$ positions in $I_{A[\ell]}$ returns the index of the k^{th} occurrence of element $A[\ell]$ in $A[\ell : n]$; if this index is at most r, then the frequency of $A[\ell]$ in $A[\ell : r]$ is at least k. If the index $R[\ell] + k - 1$ exceeds the size of the array $I_{A[\ell]}$, then the query returns a negative answer. The arrays $I_1, \ldots, I_\Delta$ and R can be constructed in $O(n)$ total time in a single scan of array A. They require a total of $O(n)$ space as each element of A has a corresponding entry in only one of the index arrays.

Lemma 6.7. *Given an array $A[1 : n]$, there exists a data structure requiring $O(n)$ space that can determine in constant time for any $1 \leq \ell \leq r \leq n$ and any k whether $A[\ell : r]$ contains at least k instances of element $A[\ell]$.*

Following Krizanc et al. [KMS05], given any $s \in [n]$ we partition array A into s *blocks* of size $t = \lceil n/s \rceil$. That is, for each $b_i \in [s]$, the b_i^{th} block spans $A[(b_i - 1)t + 1 : b_i t]$. We precompute tables $M[1 : s, 1 : s]$ and $F[1 : s, 1 : s]$, each of size $\Theta(s^2)$, such that for any $1 \leq b_\ell \leq b_r \leq s$, $M[b_\ell, b_r]$ stores a mode of $A[(b_\ell - 1)t + 1 : b_r t]$ and $F[b_\ell, b_r]$ stores the frequency of this mode.

The tables M and F require $O(s^2)$ space. The following lemma is useful for the construction of M and F. It is slightly more powerful than necessary, but we will use it to its full potential when we consider the range least frequent element problem.

Lemma 6.8. *There exists a data structure maintaining an initially empty multiset S of elements from $[\Delta]$. 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.*
- *MostFrequent/LeastFrequent(S, k): Returns the k most or 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 $N[1 : \Delta]$ 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 most or least frequent elements, we simply iterate through L starting from the head or tail until we have reported k elements or until there are no more elements to report. □

We construct M and F in $O(ns)$ time by repeatedly passing through A, starting at each of the s block boundaries. During each pass we incrementally build a multiset using the data structure of Lemma 6.8. At every block boundary (i.e., every t elements) we obtain the mode of the multiset and its frequency in $O(1)$ time.

Query Algorithm. Given a query range $A[\ell : r]$, let $b_\ell = \lceil \ell/t \rceil + 1$ and $b_r = \lceil r/t \rceil - 1$ denote the respective indices of the first and last blocks completely contained within $A[\ell : r]$. Let $A[(b_\ell - 1)t + 1 : b_r t]$ be the *span* of the query range. Let $A[\ell : \min\{(b_\ell - 1)t, r\}]$ be the *prefix* of the query range and let $A[\max\{b_r t + 1, \ell\} : r]$ be the *suffix* of the query range. One or more of the prefix, span, and suffix may be empty; in particular, if $b_\ell > b_r$, then the span is empty.

The value $c = M[b_\ell, b_r]$ is a mode of the span with corresponding frequency $f_c = F[b_\ell, b_r]$. If the span is empty, then let $f_c = 0$. By Observation 6.6, either c is a mode of $A[\ell : r]$ or some element of the prefix or suffix is a mode of $A[\ell : r]$. Thus, to identify a mode of $A[\ell : r]$, we verify for every element in the prefix and suffix whether its frequency in $A[\ell : r]$ exceeds f_c and, if so, we identify this element as a candidate mode and count its additional occurrences in $A[\ell : r]$. We present the details of this procedure for the prefix; an analogous procedure is applied to the suffix.

We now describe how to compute the frequency of all candidate elements in the prefix over the range $A[\ell : r]$, storing the value and frequency of the current best candidate in

c and f_c. We sequentially scan the items in the prefix starting at the leftmost index, ℓ, and let i denote the index of the current item. If $I_{A[i]}[R[i]-1] \geq \ell$, then an instance of element $A[i]$ appears in $A[\ell : i-1]$, and its frequency has been counted already; in this case, simply skip $A[i]$ and increment i. Otherwise, check whether the frequency of $A[i]$ in $A[\ell : r]$ (which is equal to the frequency of $A[i]$ in $A[i : r]$) is at least f_c by Lemma 6.7 (i.e., by testing whether $I_{A[i]}[R[i]+f_c-1] \leq r$). If not, we again skip $A[i]$ and increment i. Otherwise, $A[i]$ is a candidate, and the exact frequency of $A[i]$ in $A[\ell : r]$ can be counted by a linear scan of $I_{A[i]}$, starting at index $R[i]+f_c-1$ and terminating at the last index j such that $I_{A[i]}[j] \leq r$. That is, $I_{A[i]}[j]$ denotes the index of the rightmost occurrence of element $A[i]$ that lies inside query range $A[\ell : r]$. Consequently, the frequency of $A[i]$ in $A[\ell : r]$ is $f_i = y - R[i] + 1$. We update the current best candidate: $c \leftarrow A[i]$ and $f_c \leftarrow f_i$.

After all elements in the prefix and suffix have been processed, a mode of $A[\ell : r]$ and its frequency are stored in c and f_c, respectively. Excluding the linear scans of $I_{A[i]}$, the query cost is clearly bounded by $O(t)$. For each candidate $A[i]$ encountered during the processing of the prefix, the cost of the linear scan of $I_{A[i]}$ is $O(f_i - f_c)$. Since f_c is at least the frequency of the mode of the span, at least $f_i - f_c$ instances of $A[i]$ must occur in the prefix or suffix. We can thus charge the cost of the scan to these instances. Since each element $A[i]$ is considered a candidate at most once (during its first appearance) in the prefix, we conclude that the total time required by all the linear scans is proportional to the total number of elements in the prefix (i.e., $O(t)$ time). An analogous argument holds for the cost of processing the suffix. Therefore, a query requires $O(t) = O(n/s)$ total time. We conclude our proof of Theorem 6.5.

6.2.2 Range Least Frequent Element

The range least frequent element query problem has significant differences when compared to the range mode query problem. For example, the frequencies of the respective modes of $A[\ell : r]$ and $A[\ell : r+1]$ differ by either zero or one. Also, a mode of $A[\ell : r+1]$ is either a mode of $A[\ell : r]$ or it is $A[r+1]$. On the other hand, the frequencies of respective least frequent elements of $A[\ell : r]$ and $A[\ell : r+1]$ can differ by any value in $\{\ell - r, \ldots, 0, 1\}$. Also, if the addition of an element to a multiset changes the least frequent element of the multiset, the new least frequent element has no relationship to the newly added element.

In this section we present a linear-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 prove the following theorem that implies the above result when $s = \lceil \sqrt{n} \rceil$:

Theorem 6.9. *For any $s \in [n]$, there exists a data structure for the range least frequent element problem requiring $O(n + s^2)$ space, $O(ns)$ preprocessing time, and $O(n/s)$ query time.*

We begin with a lemma that describes a data structure that can determine the frequencies of every element, including that of the least frequent element, in a query range in time proportional to the size of the query range.

Lemma 6.10. *Given an array $A[1 : n]$, there exists a linear-space data structure that computes in $O(r - \ell + 1)$ time for any $1 \le \ell \le r \le n$ the frequencies of all elements in $A[\ell : r]$. In particular, a least frequent element in $A[\ell : r]$ and its frequency can be computed in $O(r - \ell + 1)$ time.*

Proof. No actual preprocessing is necessary other than initializing an array $C[1 : \Delta]$ to zero. The query algorithm is similar to counting sort: for each element $A[i]$ in $A[\ell : r]$ we increment $C[A[i]]$. Then, for every element x, $C[x]$ corresponds to the frequency of x in $A[\ell : r]$. We iterate through $A[\ell : r]$ again to find the element x with minimum $C[x]$. We iterate through $A[\ell : r]$ one last time to reset all entries in C that we modified to zero. □

Preprocessing. We construct the data structure of Lemma 6.7, including arrays $I_1, \ldots, I_\Delta$ and R. We also construct the data structure of Lemma 6.10. As in Section 6.2.1, we divide A into s *blocks* of size $t = \lceil n/s \rceil$. We compute tables M, F, M', and F', so that for each table entry $[b_\ell, b_r]$ for $1 \le b_\ell \le b_r \le s$ contains information associated with the span from block b_ℓ to block b_r. Table M contains a least frequent element of each span. Table F contains the frequencies of these least frequent elements in their spans. Table M' contains a least frequent element of each span, excluding elements that appear in blocks immediately adjacent to the span. Table F' contains the associated frequencies of the elements stored in M'.

All four tables require $O(s^2)$ space. We construct these tables in $O(ns)$ time by repeatedly passing through A, starting at each of the s block boundaries. During each pass we incrementally build a multiset using the data structure of Lemma 6.8. 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 adjacent 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 the query range $Q = A[\ell : r]$. We define the *prefix*, *suffix*, and *span* of the query range as in Section 6.2.1. By the data structure of Lemma 6.10, if $r - \ell + 1 < 2t$, then the range query can be answered in $O(t) = O(n/s)$ time. Now consider the case $r - \ell + 1 \geq 2t$. 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 Q, S, P_1, P_2, P_1', and 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 distinct elements of Q into four sets and proceed to find an element of minimum frequency amongst the distinct elements in each case:

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

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 $A[i]$ at index i. Assume without loss of generality that it appears in P_1'. The next occurrence of element $A[i]$ is at index $j = I_{A[i]}[R[i] + 1]$, which we compute in $O(1)$ time. Thus, S contains an occurrence of element $A[i]$ if and only if j lies inside S.

The least frequent element in Q is given by the least frequent of the least frequent elements for each of the cases defined above.

Case 1. By Lemma 6.10, we compute the frequencies of all elements in P_1 in $O(t)$ time, omitting the final step of resetting the entries of array C to zero. We then repeat for P_2 so that that C contains aggregate data for all of P. Consider all elements that occur in P but not in S. For each such element x, $\text{freq}_Q(x) = \text{freq}_P(x)$. So, the least frequent of these elements in Q is the element with minimum non-zero entry in C. We find this element via another iteration through both P_1 and P_2.

Case 2. Let c be the least frequent element in S stored in M and let f_c be its frequency in S stored in F. The minimum frequency in Q of any element present in both S and P is at least f_c and at most $f_c + 2t$. For each element x that occurs in both S and P_1, we find the leftmost occurrence of x within P_1 in a scan through P_1. We repeat in a symmetric fashion in P_2. Then, by Lemma 6.7, we can check in $O(1)$ time whether an element x in both S and P has frequency in Q less than some threshold. We begin with a threshold of $f_c + 2t + 1$. If an element x has frequency less than the threshold, we find its actual frequency by iterating through I_x (forward or backwards depending on whether we are considering an element in P_1 or P_2) until reaching an index within Q. 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 furthest to which the threshold can decrease is f_c, 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 Q are bounded between f_c and $f_c + 2t$. 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 6.7, 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 x, $\mathrm{freq}_Q(x) = \mathrm{freq}_S(x)$. The least frequent of these elements has been precomputed and stored in table M'. The frequency of this element is stored in table F'.

The running time of each case is bounded by $O(t) = O(n/s)$. We conclude our proof of Theorem 6.9.

6.3 Second Method: $O(\sqrt{n/w})$ Query Time when $m \leq \sqrt{nw}$

Our second method is a refinement of the first method (from Section 6.2), in which we store the tables M and F more compactly by an encoding scheme that enables efficient retrieval of the relevant information, using techniques from succinct data structures, specifically, for rank and select operations. We show how to reduce a query to four rank and select operations. These new ideas allow us to improve the space bound in Theorem 6.5 by a factor of w, which enables us to use a slightly larger number of blocks, s, which in turn leads to an improved query time. However, there is one important caveat: our space-saving technique only works when the maximum frequency is small (i.e., when $m \leq s$). Specifically, we prove the following theorem in this section: choosing $s = \lceil\sqrt{nw}\rceil$ gives $O(n)$ space and $O(\sqrt{n/w})$ query time for $m \leq \sqrt{nw}$.

Theorem 6.11. *For any $s \in [m, n]$, there exists a data structure for the range mode query problem requiring $O(n + s^2/w)$ space and $O(n/s)$ query time.*

Preprocessing. Recall that for a span from block b_ℓ to block b_r, $M[b_\ell, b_r]$ is a mode of the span and $F[b_\ell, b_r]$ is the frequency of the mode of the span. As we will show, a mode of the span can be computed efficiently if its frequency is known; consequently, we omit table M. Also, instead of storing the frequency of the mode explicitly, we store column-to-column frequency deltas (i.e., differences of adjacent frequency values); observe that frequency values are monotone increasing across each row. We encode the frequency deltas for a single row as a bit string, where a zero bit represents an increment in the frequency of the mode (i.e., each frequency delta is encoded in unary) and a one bit represents a former cell boundary. In any row, the number of ones is at most the number of blocks, s, and the number of zeroes is at most $m \leq s$. We construct the succinct rank and select data structure of Theorem 3.1 that requires a linear number of bits to support constant-time rank and select operations on each row. Thus, each row of the table uses $O(s)$ bits of space. The table has s rows and requires $O(s^2)$ bits of space in total. We pack these bits into words, resulting in a table that requires $O(s^2/w)$ words of space.

Query Algorithm. Assuming we know a mode of the span and its frequency, we can process the prefix and suffix ranges in $O(t)$ time as before. Our attention turns now to determining a mode of the span and its frequency. We first obtain the frequency of the mode of the span; this is not difficult using rank and select queries on the bit string of the b_ℓ^{th} row, in $O(1)$ time:

$$\begin{aligned} pos_{b_r} &= \text{select}_1(b_r - b_\ell + 1) \\ \mathit{freq} &= \text{rank}_0(pos_{b_r}). \end{aligned}$$

Having found the frequency of the mode, identifying a mode itself is still a tricky problem. We proceed in two steps. We first determine the block in which the last occurrence of a mode lies, in $O(1)$ time, as follows:

$$\begin{aligned} pos_{\text{last}} &= \text{select}_0(\mathit{freq}) \\ b_{\text{last}} &= \text{rank}_1(pos_{\text{last}}). \end{aligned}$$

Next, we find a mode of the span by iteratively examining each element in block b_{last}, using a technique analogous to that for processing a suffix from Section 6.2. By Lemma 6.7

(reversed with $r \leq \ell$), we can check whether each element $A[i]$ in b_{last} has frequency *freq* in $A[(b_\ell - 1)t + 1 : i]$, in $O(1)$ time per element. If the mode occurs multiple times in block b_{last}, its last occurrence will be successfully identified. Processing block b_{last} requires $O(t)$ total time. We conclude that the total query time is $O(t) = O(n/s)$ time. We conclude our proof of Theorem 6.11.

6.4 Third Method: $O(\Delta)$ Query Time

In this section, we take a quick detour and consider a third method that has query time sensitive to Δ, the number of distinct elements in A; this "detour" turns out to be essential in assembling our final solution. We show the following:

Theorem 6.12. *There exists a data structure for the range mode query problem that requires $O(n)$ space and $O(\Delta)$ query time.*

The proof is simple: to answer a range mode query, the approach is to compute the frequency (in the query range) for each of the Δ possible elements explicitly, and then just compute the maximum in $O(\Delta)$ time.

Preprocessing. We divide A into blocks of size $t = \Delta$. For each $i \in [\lfloor n/\Delta \rfloor]$, and for every $x \in [\Delta]$, store the frequency $C_i[x]$ of x in the range $A[1 : i\Delta]$. The total size of all these arrays is $O((n/\Delta)\Delta) = O(n)$. The preprocessing time required is $O(n)$.

Query Algorithm. Given a query range $A[\ell : r]$, as mentioned, it suffices to compute the frequency of x in $A[\ell : r]$ for every $x \in [\Delta]$. Let $b_r = \lceil r/\Delta \rceil - 1$. We can compute the frequency $C(x)$ of x in the suffix $A[b_r\Delta + 1 : r]$ for every $x \in [\Delta]$ by a linear scan, in $O(\Delta)$ time since the suffix has size at most Δ. Then the frequency of x in $A[1, r]$ is given by $C_{b_r}[x] + C(x)$. The frequency of x in $A[1, \ell - 1]$ can be similarly computed. The frequency of x in $A[\ell, r]$ is just the difference of these two frequencies. The total query time is clearly $O(\Delta)$. We conclude our proof of Theorem 6.12.

6.5 Final Method: $O(\sqrt{n/w})$ Query Time

We are finally ready to present our improved linear-space data structure with $O(\sqrt{n/w})$ query time. Our final idea is simple: if the elements all have small frequencies, the second method (Section 6.3) already works well; otherwise, the number of distinct elements with large frequencies is small, and so the third method (Section 6.4) can be applied instead.

More precisely, let s be any fixed value in $[n]$. We partition the elements of A into those with low frequencies (i.e., frequencies at most s), and those with high frequencies (i.e., frequencies greater than s). A mode of all low-frequency elements has frequency at most s. Thus we can apply Theorem 6.11 to build a data structure that requires $O(n + s^2/w)$ space and $O(n/s)$ query time. On the other hand, there are at most n/s distinct high-frequency elements. Thus, we can apply Theorem 6.12 to build a linear-space data structure on the high-frequency elements supporting $O(n/s)$ query time. The following simple "decomposition" lemma allows us to combine the two structures:

Lemma 6.13. *Given an array $A[1 : n]$ and any ordered partition of A into two arrays $B_1[1 : n']$ and $B_2[1 : n-n']$ such that no element in B_1 occurs in B_2 nor vice versa, if there exist respective $S_1(n)$- and $S_2(n)$-space data structures that support range mode queries on B_1 and B_2 in $Q_1(n)$ and $Q_2(n)$ time, then there exists an $O(n+S_1(n)+S_2(n))$-space data structure that supports range mode query on A in $O(Q_1(n) + Q_2(n))$ time.*

Proof. For each $a \in \{1, 2\}$ and $i \in [n]$, precompute $I_a[i]$, the index in the B_a array of the first element in A to the right of $A[i]$ that lies in B_a; and precompute $J_a[i]$, the index in the B_a array of the first element in A to the left of $A[i]$ that lies in B_a. Given a range query $A[\ell : r]$, we can compute the mode in the range $B_1[I_1[\ell], J_1[r]]$ and the mode in the range $B_2[I_2[\ell], J_2[r]]$ and determine which has larger frequency; this is a mode of $A[\ell : r]$. $\square$

We have thus completed the proof of our main theorem concerning the range mode query problem:

Theorem 6.14. *Given any $s \in [n]$, there exists a data structure for the range mode query problem requiring $O(n + s^2/w)$ space and $O(n/s)$ query time. In particular, by setting $s = \lceil\sqrt{nw}\rceil$, there exists a data structure requiring $O(n)$ space and $O(\sqrt{n/w})$ query time.*

Chapter 7

Range Majority and Minority

Given some $0 < \alpha < 1$, an element x is an α-*majority* in a multiset S if $\mathrm{freq}_S(x) > \alpha|S|$. Otherwise, x is an α-*minority* in S. An α-majority range query in an array A specifies some $0 < \alpha < 1$ and a pair of indices (ℓ, r) into A such that $1 \leq \ell \leq r \leq n$. The result of the query is the set of all α-majorities of the multiset defined by subarray $A[\ell : r]$. These are all elements with frequency greater than $\alpha(r - \ell + 1)$ in $A[\ell : r]$. An α-minority range query in A instead outputs a single α-minority element: any element with frequency no greater than $\alpha(r - \ell + 1)$. 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 during the preprocessing of the data structure. We do so to differentiate from the more challenging case in which different parameter values can be specified at query time.

In Section 7.1 we give a linear-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. [DHM+11] for β-majority range queries and of Gagie et al. [GHMN11] for α-majority range queries, which have worse space bounds ($O(n \log(1/\beta + 1)$ and $O(n \log n)$, respectively).

In Section 7.2 we apply a variation of our technique to give a data structure for the α-majority range query problem in an array requiring $O(n \log n)$ space and $O(1/\alpha)$ query time. These space and time bounds match those achieved by a recent α-majority data structure of Gagie et al. [GHMN11].

Both our data structures in Sections 7.1 and 7.2 make interesting use of existing tools from computational geometry. Notably, we apply Chazelle's hive graphs [Cha86], 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 (see Section 3.6).

Finally, in Section 7.3, we give a data structure for the 2-D orthogonal range β-majority problem. In this setting, we are given a set P of n coloured points from $[n]^2$. Given a query rectangle Q, we must find either all β-majority colours amongst the colours of the points of $P \cap Q$. Interestingly, we make use of one of our approximate data structures for 2-D orthogonal range counting from Section 4.3.

7.1 1-D Range α-Minority

In this section we describe a linear-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 7.1. *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. We construct the linear-space data structure of Lemma 6.7. With this data structure, we can check in $O(1)$ time whether there are at least k instances of $A[\ell]$ in the range $A[\ell : r]$ for any $k \geq 0$ and $r \geq \ell$.

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 $A[\ell : r]$, we use data structure D to find the leftmost occurrences of the $1/\alpha$ leftmost distinct elements on or to the right of index ℓ in $Q_D(n, 1/\alpha)$ time. Some of these leftmost occurrences may lie to the right of index r; we can ignore these elements as none of their occurrences lie in $A[\ell : r]$. There are $O(1/\alpha)$ remaining leftmost occurrences of leftmost distinct elements. Consider such an occurrence at index i. Since this occurrence is the first of $A[i]$ on or after index ℓ, the frequency of $A[i]$ in $A[i : r]$ is equal to the frequency of $A[i]$ in $A[\ell : r]$. We can then check whether or not $A[i]$ is an α-minority in $A[\ell : r]$ in $O(1)$ time by setting $k = \alpha(r-\ell+1)+1$ in Lemma 6.7. 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 $A[\ell : r]$ so there cannot be an α-minority. $\square$

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 7.2, we map this problem to a 2-D problem in computational geometry that can be solved by Lemma 3.14, Chazelle's hive graph data structure [Cha86] (see Section 3.6). 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 7.2. *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, $\sum_{i=1}^{n} |L_i|$ 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 $[0, \ell]$ and with y-value ℓ. We thus have n horizontal segments. We build the data structure of Lemma 3.14 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 a query to the hive graph of Lemma 3.14 is $O(\log\log n + k)$ time, since all of our segments have coordinates from the universe $[0, n]$.

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 at which we perform 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. □

Corollary 7.3. *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 Lemmata 7.1 and 7.2. □

7.2 1-D Range α-Majority

We now consider the α-majority range query problem. Recently, Gagie et al. [GHMN11] describe an $O(n \log n)$-space data structure that supports α-majority queries 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 with the same bounds. Previous work by Durocher et al. [DHM$^+$11] 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 consider first 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 7.2 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 6.8 in which all updates are provided offline.

Lemma 7.4. *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 $A[1 : r]$ for $1 \le r \le n$. 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 $A[1 : i]$ through $A[1 : j]$ 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 $f \geq 0$ and arbitrarily perturb the y-values for any segments that overlap.

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 the hive graph of Lemma 3.14 on these segments.

For every distinct element e in $A[1:r]$ there is a horizontal segment with x-interval $[i, j+1)$ intersecting the vertical line $y = r$ with $A[i] = e$ and $\text{freq}_{A[1:r]}(e) = f$. These horizontal segments are sorted along the vertical line in the order of frequency. To find the k most frequent elements in $A[1:r]$, we query the hive graph for the first k horizontal segments intersecting the vertical ray from (r, n) to $(r, -\infty)$. As in Lemma 7.2, 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 $A[i]$ where i is the left x-coordinate of the segment. $\square$

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 $A[1:r]$. 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. However, α-majority queries are decomposable in this way.

Observation 7.5 (Karpinski and Nekrich [KN08])**.** *Assume R is a multiset and (R_1, R_2) is a partition of R. Every α-majority of R is an α-majority of at least one of R_1 and R_2.*

Proof. Let x be an α-majority of R so that $\text{freq}_R(x) > \alpha|R|$. Assume towards contradiction that x is neither an α-majority of R_1 nor of R_2. Then, $\text{freq}_{R_1}(x) \leq \alpha|R_1|$ and $\text{freq}_{R_2}(x) \leq \alpha|R_2|$. Since $\text{freq}_R(x) = \text{freq}_{R_1}(x) + \text{freq}_{R_2}(x)$ and since $|R| = |R_1| + |R_2|$, we have $\text{freq}_R(x) \leq \alpha|R|$: a contradiction. $\square$

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 a range tree (see Section 3.2) to support two-sided α-majority queries.

Theorem 7.6. *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 7.4 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 represent the range tree with the succinct tree data structure of Theorem 3.3. We use this data structure to decompose a two-sided query into one-sided queries in two nodes of the range tree via an LCA operation. We also build the arrays required to support the queries of Lemma 6.7.

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 Observation 7.5, 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 6.7 to check which of the $O(1/\alpha)$ most frequent elements are in fact α-majorities in constant time each. $\square$

7.3 2-D Range β-Majority

We turn our attention to a 2-D generalization of the β-majority range query problem. In this generalization, we are given a set P of n coloured points from $[n]^2$. We must preprocess P so that given any query rectangle Q, we can efficiently find all β-majority colours in the multiset of the colours of the points of $P \cap Q$.

Our high-level plan is to use a $\Theta(\log^\epsilon n)$-ary range tree to decompose query Q into $O(\log n/ \log\log n)$ 1-D queries. By Observation 7.5, the β-majorities of Q must be in the union of the β-majorities of the all of these 1-D queries. We take the union of the results of the 1-D queries to obtain $O((1/\beta) \log n/ \log\log n)$ candidates. We can determine which of these candidates are actually β-majorities in Q by counting, for each candidate, the number of points of the candidate colour in Q. All of these $\Theta(\log_w n)$-time range counting queries (see Theorem 4.1) would require a total of $O((1/\beta)(\log n/ \log\log n)^2)$ time. We can do better, by first filtering out candidates using approximate range counting. We can determine a multiplicative constant-factor approximation for each candidate in $O(\log\log n)$ time each via our data structure for approximate range counting. In only $O((1/\beta) \log n)$ time, we are thus able to filter our candidates down to a set of size $O(1/\beta)$. We can then afford to perform the expensive exact range counting queries for this smaller set of candidates without increasing the overall query time beyond $O((1/\beta) \log n)$. Since we are performing counting in 2-D to verify which candidates are α-majorities, we are able to save space by using a modified 1-D data structure that does not verify that the $O(1/\beta)$ colours that it outputs are all β-majorities.

Before we dive in, we consider a generalization of the augmented range tree of Chan et al. [CLP11] (see Section 3.2) for $\Theta(\log^\epsilon n)$-ary range trees. This generalization will be useful in our decomposition of 2-D queries into 1-D queries. The augmented range tree of Lemma 3.11 equipped with the ball inheritance data structure of Corollary 3.10 requires $O(n \log^\epsilon n)$ space, $O(1)$ time for ball inheritance queries, and $O(\log \log n)$ time for predecessor search queries in any node. In our generalization, a ball inheritance query specifies the rank of a point amongst those of a sequence of adjacent siblings in the range tree instead of amongst those of a single node. Similarly, a generalized predecessor search query searches amongst the points of a sequence of adjacent siblings.

Lemma 7.7. *There exists an augmented $\Theta(\log^\epsilon n)$-ary range tree that requires $O(n \log^{2\epsilon} n)$ space and that supports generalized ball inheritance queries in $O(1)$ time and generalized predecessor search queries in $O(\log \log n)$ time.*

Proof. Let b be the first power of 2 no less than $\log^\epsilon n$. Then, b is a power and 2 and is also $\Theta(\log^\epsilon n)$. Every node in a b-ary range tree contains exactly the same points as some node in a standard binary range tree. Thus, using the augmented range tree of Lemma 3.11 equipped with the ball inheritance data structure of Corollary 3.10, ball inheritance queries involving only a single node can be answered in constant time using $O(n \log^\epsilon n)$ space by querying the corresponding node in the standard binary augmented range tree.

Consider a sequence of adjacent children $c_i, \ldots, c_j$ of the root of the b-ary range tree. We build a bit vector $A_{i,j}[1:n]$ where $A_{i,j}[k]$ is 1 if and only if the point at index k in the root node is in one of $c_i, \ldots, c_j$. We build the succinct rank and select data structure of Theorem 3.1 for each of these bit vectors. We build this data structure for all $O(b^2)$ sequences of adjacent children of the root node and also recursively build the data structures in each child node. The additional space required is given by the recurrence

$$\begin{aligned} & S(n) = bS(n/b) + O(nb^2/w) \\ \Rightarrow \quad & S(n) = O((nb^2/w) \log_b n) \\ & = O(n \log^{2\epsilon} n). \end{aligned}$$

Assume we are given a generalized ball inheritance query consisting of a sequence of children $c_i, \ldots, c_j$ of a node p and a rank k to a point within this sequence. We select, in constant time, the k^{th} one bit in $A_{i,j}$, resulting in an index k' into the bit vector. The point with rank k' in p is the point with rank k in $c_i, \ldots, c_j$. A standard ball inheritance query can now, in constant time, determine the coordinates of the point with rank k' in p.

Now, assume we are given a generalize predecessor search query consisting of a sequence of children $c_i, \ldots, c_j$ of a node p and a value v from which to search. We perform a standard predecessor search query in p with value v, resulting in a rank k' in p. We determine k, the number of one bits up to index k' in $A_{i,j}$, via a constant-time rank query. The point with rank k in $c_i, \ldots, c_j$ is the predecessor of the point with rank k' in p, and is thus also the predecessor of value v. $\square$

We now consider the range β-majority problem in an array with slightly modified requirements in order to save space. We relax the requirements of the data structure in two ways. First, the data structure reports a colour by giving its index in A. Second, the data structure may report indices that correspond to colours that are not β-majorities in $A[\ell : r]$, as long as it reports all β-majorities in $A[\ell : r]$.

Lemma 7.8. *There exists a data structure for an array that, given a subarray $A[\ell : r]$, reports a set I of indices such that $\{A[i] \mid i \in I\}$ contains all β-majorities in $A[\ell : r]$. The data structure requires $O(n \log(1/\beta))$ bits of space and $O(1/\beta)$ query time.*

Proof. We begin with the range β-majority data structure of Durocher et al. [DHM+11]. The data structure is conceptually a range tree formed by recursing on the left and right halves of A. At each level of the tree, consecutive sequences of four tree nodes are grouped into *quadruples*. There are $O(n)$ such quadruples. Consider a quadruple (a, b, c, d) consisting of a total of m elements. If a query within this quadruple contains either b or c, then the query has size at least $m/4$. Then, any β-majority must have multiplicity greater than $\beta(m/4)$. However, there can be at most $4/\beta = O(1/\beta)$ elements with such high multiplicity in the quadruple. For our purposes, it is sufficient to output these $O(1/\beta)$ candidates. Durocher et al. [DHM+11] show that for every query, there is a quadruple (a, b, c, d) that contains the query such that the query contains either b or c. Furthermore, such a quadruple can be found in constant time by an $O(n)$-bit data structure.

For quadruples of size $m > 1/\beta$, we encode the set of $O(1/\beta)$ candidates by an array of indices into the quadruple pointing to each candidate. Each index requires $O(\log m)$ bits. For quadruples of size $m \leq 1/\beta$ we do not store anything as we can simply output every element in the query range in $O(1/\beta)$ time in this case. The total space requirement in bits is given by the recurrence

$$
\begin{aligned}
& S(n) = \begin{cases} 2S(n/2) + O((1/\beta)\log n) & \text{for } n > 1/\beta \\ O(1) & \text{for } n \leq 1/\beta \end{cases} \\
\Rightarrow \quad & S(n) = O(n \log(1/\beta)).
\end{aligned}
$$

During a query we can convert indices into a quadruple to indices in A in constant time based on the offset of the quadruple in A, which can be computed in constant time. □

We now have the tools required to tackle our 2-D results.

Theorem 7.9. *There exists a data structure for the 2-D orthogonal range β-majority query problem requiring $O(\log(1/\beta)n\log^{\epsilon} n)$ space and $O((1/\beta)\log n)$ query time.*

Proof. We build the generalized augmented $\Theta(\log^{\epsilon'} n)$-ary range tree of Lemma 7.7. For every sequence of adjacent siblings in the range tree, we map the union of the points of these siblings to an array such that the i^{th} entry contains the colour of the point with y-rank i. For each such array, we build the data structure of Lemma 7.8. The total space required by all of these data structures is $O(\log(1/\beta)n\log^{\epsilon} n)$ where $\epsilon = 2\epsilon'$.

For each colour, we also build the approximate 2-D orthogonal range counting data structure of Corollary 4.19, as well as the exact 2-D orthogonal range counting data structure of Theorem 4.1. The exact counting data structures require $O(n)$ space across all colours and answer queries in $O(\log_w n)$ time. The approximate range counting data structures require $O(n\log\log n)$ space across all colours and answer queries in $O(\log\log n)$ time. Finally, we build the exact counting data structure of Theorem 4.1 over all of P, requiring linear space and $O(\log_w n)$ query time.

Given a query range Q, we use our range tree to decompose Q into $O(\log n/\log\log n)$ vertical *slabs*, where each slab corresponds to some sequence of adjacent siblings in the range tree. We use generalized predecessor search data structures to map the top and bottom boundaries of Q into y-ranks in each slab. All of these predecessor searches require $O(\log n)$ time. In each slab we then find $O(1/\beta)$ β-majority candidates using the data structure of Lemma 7.8. These candidates are represented as ranks of points in each slab. We can determine the colours of these points via generalized ball inheritance queries. By Observation 7.5, every β-majority of Q must be a β-majority in one of the slabs that partition Q. Thus, we have $O((1/\beta)\log n/\log\log n)$ candidates for β-majority of Q.

We determine $|P \cap Q|$ in $O(\log_w n)$ time using the exact counting data structure for P. For each candidate of colour c, we use the approximate range counting data structure for c to find a multiplicative constant-factor approximation k of the number of points with colour c in Q. Then, for some sufficiently small constant $f < 1$, it is safe to exclude candidates for which $k < f\beta|P \cap Q|$. Each remaining colour has multiplicity $\Omega(\beta|P \cap Q|)$ in $P \cap Q$; there can be at most $O(1/\beta)$ such colours. Finally, we use the exact counting data structures for each remaining colour to determine which of these remaining candidates are actually the β-majorities. The query time is dominated by the cost of

$O((1/\beta)\log n/\log\log n)$ approximate range counting queries that require $O(\log\log n)$ time each for a total of $O((1/\beta)\log n)$ time. $\square$

Chapter 8

Conclusion

Range Counting. We have given an adaptive data structure for 2-D orthogonal range counting that requires $O(n \log\log n)$ space and $O(\log\log n + \log_w k)$ query time. When when $k = O(1)$, the space and query time bounds for this data structure match the best known bounds for the 2-D orthogonal range emptiness query problem. However, there is also a data structure for the 2-D orthogonal range emptiness query problem that requires only $O(n)$ space and $O(\log^\epsilon n)$ query time. An interesting open problem is whether or not there is an adaptive data structure for 2-D orthogonal range counting that similarly matches these bounds, modulo an $O(\log_w k)$ term in the query time.

Open Problem 8.1. *Does there exist a data structure for* 2*-D orthogonal range counting requiring $O(n)$ space and $O(\log^\epsilon n + \log_w k)$ query time?*

Such a data structure would be a strict improvement over the linear-space data structure of JaJa et al. [JMS05] that supports 2-D orthogonal range counting queries in $O(\log_w n)$ time. In Corollary 4.12, we give a linear-space K-shallow data structure for 2-D orthogonal range counting that requires $O(\log^\epsilon n + \log_w K)$ query time. However, we cannot follow the approach of building such a data structure for double exponentially increasing values of K to create an adaptive data structure, as the space requirement would increase by an $O(\log\log n)$ factor. It seems that new ideas are necessary.

Since K-shallow cuttings of size $O(n/K)$ exist for 3-D dominance regions, it is natural to wonder whether our techniques can be applied to 3-D orthogonal range counting. The best known data structure for the 3-D orthogonal range emptiness query problem requires $O(n \log^{1+\epsilon} n)$ space and $O(\log\log n)$ query time [CLP11]. The best known data structure for 3-D orthogonal range counting requires $O(n \log_w n)$ space and $O((\log_w n)^2)$ query time [JMS05]. It turns out that our succinct data structure representations of shallow cuttings are key to keeping both our space and query time bounds low. These succinct shallow cuttings depend on the structure of shallow cuttings for 3-sided 2-D ranges and cannot be generalized to shallow cuttings for 3-D dominance regions. Then, to decide in which cell a 3-D dominance region lies requires 2-D orthogonal point location queries that run in $O(\log \log n)$ time. Thus, if we have K-shallow cuttings for various values of K, we must overlay them in order to avoid increasing query time beyond $O(\log \log n)$. However, overlaying two shallow cuttings can result in $O(n^2)$ cells and our space bound would suffer.

Open Problem 8.2. *Does there exist a data structure for 3-D orthogonal range counting requiring $O(n \log^{1+\epsilon} n)$ space and $O(\log \log n + (\log_w k)^2)$ query time?*

Open Problem 8.3. *Does there exist a data structure for $(1+\delta)$-approximate 3-D orthogonal range counting requiring $O(n \log^{1+\epsilon} n)$ space and $O(\log \log n)$ query time?*

Range Selection. We have given an adaptive data structure for the range selection query problem in an array that requires $O(n)$ space and $O(\log_w k)$ query time to select the k^{th} lowest element in a given subarray. Our data structure is optimal as it matches the lower bound of Jørgensen and Larsen [JL11]. Our data structure can answer range minimum queries in optimal $O(1)$ time and range median queries in optimal $O(\log_w n)$ time. Other than potential generalizations or simplifications, our result ends its line of research.

Range Mode. We have demonstrated the hardness of the range mode and least frequent element problems by reducing Boolean matrix multiplication to the construction and use of a range mode or least frequent element data structure. This argument is not a rigorous lower bound argument as the complexity of Boolean matrix multiplication is not known. Since no current techniques seem capable of proving unconditional super-polylogarithmic cell probe lower bounds, the following open problem is likely very difficult to solve.

Open Problem 8.4. *What is the optimal query time required by any linear-space data structure for the range mode query problem in an array?*

Using succinct rank and select data structures and bit packing, we reduce the query time for the range mode query problem from $O(\sqrt{n})$ to $O(\sqrt{n/w})$ without increasing the data structure's space consumption beyond $O(n)$. Unlike the frequency of the mode of a multiset, the frequency of the least frequent element of a multiset does not vary monotonically as elements are added to the multiset. Furthermore, when the least frequent element changes, the new element of minimum frequency has no relationship with the newly added element. Consequently, our techniques do not seem immediately applicable to the least frequent element range query problem.

Open Problem 8.5. *Does there exist a data structure for the range least frequent element query problem in an array requiring $O(n)$ space and $o(\sqrt{n})$ query time?*

An interesting generalization of the range mode and least frequent element query problems is to compute the k^{th} most frequently occurring element (or the top k most frequent elements) in a query range. This problem generalizes the range least frequent element query problem similarly to how the range selection query problem generalizes the range minimum query problem. Generalizations of our techniques for the range mode and least frequent element query problems seem unlikely without a significant increase in space when k is large.

Open Problem 8.6. *Does there exist a data structure that can identify the k^{th} most frequently occurring element (or the top k most frequent elements) in a given subarray of an array while requiring $O(n)$ space and $O(n^{1-\epsilon})$ (or $O(n^{1-\epsilon} + k)$) query time for some constant $\epsilon > 0$.*

We note that Lemma 7.4 is a linear-space data structure that solves the variant of reporting the top k most frequent elements, but only for one-sided ranges of the form $A[1 : r]$.

Range Majority. We have given a data structure achieving $O(1/\alpha)$ query time in $O(n)$ space for the range α-minority problem. This data structure is efficient despite allowing α to be specified at query time. We also give a data structure achieving $O(1/\alpha)$ query time in $O(n \log n)$ space for the range α-majority problem. These bounds match those of Gagie et al. [GHMN11], but our technique is significantly different. The greater space required by our α-majority data structure in comparison to our α-minority data structure suggests that further improvements may be possible.

Open Problem 8.7. *Does there exist a data structure for the range α-majority query problem in an array requiring $o(n \log n)$ space and $O(1/\alpha)$ query time?*

Our data structure for the 2-D orthogonal range β-majority problem requires significantly more space and query time than data structures for the 2-D orthogonal range emptiness problem, suggesting again that further improvements may be possible. Also, investigation into supporting α-majority queries is warranted.

References

[ABR00] S. Alstrup, G. S. Brodal, and T. Rauhe. New data structures for orthogonal range searching. In *Proceedings of the 41st Symposium on Foundations of Computer Science (FOCS'00)*, pages 198–207. IEEE Computer Society, 2000.

[AES99] P. K. Agarwal, A. Efrat, and M. Sharir. Vertical decomposition of shallow levels in 3-dimensional arrangements and its applications. *SIAM J. Comput.*, 29(3):912–953, 1999.

[Afs08] P. Afshani. On dominance reporting in 3D. In *Proceedings of the 16th European Symposium on Algorithms (ESA'08)*, volume 5193 of *LNCS*, pages 41–51. Springer-Verlag, 2008.

[AHP08] B. Aronov and S. Har-Peled. On approximating the depth and related problems. *SIAM J. Comput.*, 38(3):899–921, 2008.

[AHZ10] P. Afshani, C. Hamilton, and N. Zeh. A general approach for cache-oblivious range reporting and approximate range counting. *Comput. Geom. Theory Appl.*, 43(8):700–712, 2010.

[AMS99] N. Alon, Y. Matias, and M. Szegedy. The space complexity of approximating the frequency moments. *J. Comput. Syst. Sci.*, 58(1):137–147, 1999.

[AS10] B. Aronov and M. Sharir. Approximate halfspace range counting. *SIAM J. Comput.*, 39(7):2704–2725, 2010.

[AT07] A. Andersson and M. Thorup. Dynamic ordered sets with exponential search trees. *J. ACM*, 54(3), 2007.

[AV88] A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, 1988.

[Ben80] J. L. Bentley. Multidimensional divide-and-conquer. *Commun. ACM*, 23(4):214–229, 1980.

[BGJS11] G. S. Brodal, B. Gfeller, A. G. Jørgensen, and P. Sanders. Towards optimal range medians. *Theor. Comput. Sci.*, 412(24):2588–2601, 2011.

[BJ09] G. S. Brodal and A. G. Jørgensen. Data structures for range median queries. In *Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC'09)*, volume 5878 of *LNCS*, pages 822–831. Springer-Verlag, 2009.

[BW09] N. Bansal and R. Williams. Regularity lemmas and combinatorial algorithms. In *Proceedings of the 50th Symposium on Foundations of Computer Science (FOCS'09)*, pages 745–754. IEEE Computer Society, 2009.

[CF90] B. Chazelle and J. Friedman. A deterministic view of random sampling and its use in geometry. *Combinatorica*, 10(3):229–249, 1990.

[CG86] B. Chazelle and L. Guibas. Fractional cascading: I. A data structuring technique. *Algorithmica*, 1(1):133–162, 1986.

[Cha86] B. Chazelle. Filtering search: A new approach to query-answering. *SIAM J. Comput.*, 15(3):703–724, 1986.

[Cha88] B. Chazelle. Functional approach to data structures and its use in multidimensional searching. *SIAM J. Comput.*, 17(3):427–462, 1988.

[Cha11] T. M. Chan. Persistent predecessor search and orthogonal point location on the word RAM. In *Proceedings of the 22nd Symposium on Discrete Algorithms (SODA'11)*, pages 1131–1145. SIAM, 2011.

[CKMS06] E. Cohen, H. Kaplan, Y. Mansour, and M. Sharir. Approximations with relative errors in range spaces of finite VC-dimension. Manuscript, 2006.

[CLP11] T. M. Chan, K. G. Larsen, and M. Pătraşcu. Orthogonal range searching on the RAM, revisited. In *Proceedings of the 27th Symposium on Computational Geometry (SoCG'11)*, pages 1–10. ACM, 2011.

[CM96] D. R. Clark and J. I. Munro. Efficient suffix trees on secondary storage. In *Proceedings of the 27th Symposium on Discrete Algorithms (SODA'96)*, pages 383–391. Society for Industrial and Applied Mathematics, 1996.

[DHM$^+$11] S. Durocher, M. He, J. I. Munro, P. K. Nicholson, and M. Skala. Range majority in constant time and linear space. In *Proceedings of the 38th International Colloquium on Automata, Languages and Programming (ICALP'11)*, volume 6755 of *LNCS*, pages 244–255. Springer-Verlag, 2011.

[FMM04] P. Ferragina, G. Manzini, and V. Makinen. Succinct representation of sequences. Technical Report TR/DCC-2004-5, Department of Computer Science, University of Chile, 2004.

[FW93] M. L. Fredman and D. E. Willard. Surpassing the information theoretic bound with fusion trees. *J. Comput. Syst. Sci.*, 47(3):424–436, 1993.

[GBT84] H. N. Gabow, J. L. Bentley, and R. E. Tarjan. Scaling and related techniques for geometry problems. In *Proceedings of the 16th Symposium on Theory of Computing (STOC'84)*, pages 135–143. ACM, 1984.

[GHMN11] T. Gagie, M. He, J. I. Munro, and P. K. Nicholson. Finding frequent elements in compressed 2D arrays and strings. In *Proceedings of the 18th Symposium on String Processing and Information Retrieval (SPIRE'11)*, volume 7024 of *LNCS*, pages 295–300. Springer-Verlag, 2011.

[GJLT10] M. Greve, A. G. Jørgensen, K. G. Larsen, and J. Truelsen. Cell probe lower bounds and approximations for range mode. In *Proceedings of the 37th International Colloquium on Automata, Languages and Programming (ICALP'10)*, volume 6198 of *LNCS*, pages 605–616. Springer-Verlag, 2010.

[GMR06] A. Golynski, J. I. Munro, and S. S. Rao. Rank/select operations on large alphabets: A tool for text indexing. In *Proceedings of the 17th Symposium on Discrete Algorithms (SODA'06)*, pages 368–373. ACM, 2006.

[GORR09] R. Grossi, A. Orlandi, R. Raman, and S. S. Rao. More haste, less waste: Lowering the redundancy in fully indexable dictionaries. In *Proceedings of the 26th Symposium on Theoretical Aspects of Computer Science (STACS'09)*, volume 3 of *Leibniz International Proceedings in Informatics*, pages 517–528. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2009.

[GPT09] T. Gagie, S. J. Puglisi, and A. Turpin. Range quantile queries: Another virtue of wavelet trees. In *Proceedings of the 16th Symposium on String Processing and Information Retrieval (SPIRE'09)*, volume 5721 of *LNCS*, pages 1–6. Springer-Verlag, 2009.

[GS09] B. Gfeller and P. Sanders. Towards optimal range medians. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming (ICALP'09)*, volume 5555 of *LNCS*, pages 475–486. Springer-Verlag, 2009.

[HPS11] S. Har-Peled and M. Sharir. Relative (p, ϵ)-approximations in geometry. *Discrete Comput. Geom.*, 45(3):462–496, 2011.

[HT84] D. Harel and R. E. Tarjan. Fast algorithms for finding nearest common ancestors. *SIAM J. Comput.*, 13(2):338–355, 1984.

[Jac88] G. J. Jacobson. *Succinct static data structures.* PhD thesis, Carnegie Mellon University, 1988.

[JL11] A. G. Jørgensen and K. G. Larsen. Range selection and median: Tight cell probe lower bounds and adaptive data structures. In *Proceedings of the 22nd Symposium on Discrete Algorithms (SODA'11)*, pages 805–813. SIAM, 2011.

[JMS05] J. JaJa, C. W. Mortensen, and Q. Shi. Space-efficient and fast algorithms for multidimensional dominance reporting and counting. In *Proceedings of the 15th International Conference on Algorithms and Computation (ISAAC'04)*, volume 3341 of *LNCS*, pages 1755–1756. Springer-Verlag, 2005.

[KMS05] D. Krizanc, P. Morin, and M. Smid. Range mode and range median queries on lists and trees. *Nordic J. of Computing*, 12(1):1–17, 2005.

[KN08] M. Karpinski and Y. Nekrich. Searching for frequent colors in rectangles. In *Proceedings of the 20th Canadian Conference on Computational Geometry (CCCG'08)*, 2008.

[Mat91] J. Matoušek. Cutting hyperplane arrangements. *Discrete Comput. Geom.*, 6(5):385–406, 1991.

[Mat92] J. Matoušek. Reporting points in halfspaces. *Comput. Geom. Theory Appl.*, 2(3):169–186, 1992.

[Mat93] J. Matoušek. Range searching with efficient hierarchical cuttings. *Discrete Comput. Geom.*, 10(1):157–182, 1993.

[Nek09a] Y. Nekrich. Data structures for approximate orthogonal range counting. In *Proceedings of the 20th International Symposium on Algorithms and Computation (ISAAC'09)*, volume 5878 of *LNCS*, pages 183–192. Springer-Verlag, 2009.

[Nek09b] Y. Nekrich. Orthogonal range searching in linear and almost-linear space. *Comput. Geom. Theory Appl.*, 42(4):342–351, 2009.

[Ove88] M. H. Overmars. Efficient data structures for range searching on a grid. *J. Algorithms*, 9(2):254–275, 1988.

[PA95] J. Pach and P. K. Agarwal. *Combinatorial Geometry.* Wiley-Interscience Series in Discrete Mathematics and Optimization. John Wiley & Sons, Inc., Hoboken, NJ, USA, 1995.

[Păt07] M. Pătraşcu. Lower bounds for 2-dimensional range counting. In *Proceedings of the 39th Symposium on Theory of Computing (STOC'07)*, pages 40–46. ACM, 2007.

[Pet08] H. Petersen. Improved bounds for range mode and range median queries. In *Proceedings of the 34th Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'08)*, volume 4910 of *LNCS*, pages 418–423. Springer-Verlag, 2008.

[PG09] H. Petersen and S. Grabowski. Range mode and range median queries in constant time and sub-quadratic space. *Inf. Process. Lett.*, 109(4):225–228, 2009.

[PT06] M. Pătraşcu and M. Thorup. Time-space trade-offs for predecessor search. In *Proceedings of the 38th Symposium on Theory of Computing (STOC'06)*, pages 232–240. ACM, 2006.

[SN10] K. Sadakane and G. Navarro. Fully-functional succinct trees. In *Proceedings of the 21st Symposium on Discrete Algorithms (SODA'10)*, pages 134–149. Society for Industrial and Applied Mathematics, 2010.

[vEBKZ76] P. van Emde Boas, R. Kaas, and E. Zijlstra. Design and implementation of an efficient priority queue. *Theory of Comput. Syst.*, 10(1):99–127, 1976.

[VV96] D. E. Vengroff and J. S. Vitter. Efficient 3-d range searching in external memory. In *Proceedings of the 28th Symposium on Theory of Computing (STOC'96)*, pages 192–201. ACM, 1996.

[Wil83] D. E. Willard. Log-logarithmic worst-case range queries are possible in space $\Theta(N)$. *Inf. Process. Lett.*, 17(2):81–84, 1983.

[Wil85] D. E. Willard. New data structures for orthogonal range queries. *SIAM J. Comput.*, 14(1):232–253, 1985.

[Wil11] V. Vassilevska Williams. Breaking the Coppersmith-Winograd barrier. `http://www.cs.berkeley.edu/~virgi/matrixmult.pdf`, 2011.