Faster 3SUM-INDEXING and Related Results

Isaac Hair

Abstract

Given three input lists of values X, Y, and Z, each of size n, the 3SUM problem asks whether there exists $(x, y, z) \in X \times Y \times Z$ such that $x + y + z = 0$. The current fastest algorithms for 3SUM only achieve polylogarithmic factor speedups over the folklore $\Theta(n^2)$ time algorithm despite decades of research [Chan SODA'18; Baran, Demaine, and Pătraşcu WADS'05]. In this paper, we study a variant of 3SUM called 3SUM-INDEXING (this was first introduced by Goldstein, Kopelowitz, Lewenstein, and Porat [WADS'17]): the algorithm preprocesses two lists of values X and Y, each of size n, and then receives one or more input query values z; for each, the goal is to determine whether there exists $(x, y) \in X \times Y$ such that $x+y+z = 0$. In contrast to previous publications, our paper focuses on time (instead of space) as a limiting factor: Given slightly subquadratic preprocessing time, how fast can we evaluate queries?

We first establish a baseline by surveying the fastest algorithms for 3SUM-INDEXING that can be obtained using current techniques in literature. These allow us to achieve a query time of $n/2^{c \log^{1/3} n}$ for any $c = o(1)$ after using $o(n^2)$ preprocessing time. Our main contribution is a new technique for sorting large blocks that can achieve query time as low as $n/2^{c \log^{2/5} n / \log^{4/5} \log n}$ for any $c = o(1)$ after using $o(n^2)$ preprocessing time, which improves previous results substantially. In order to demonstrate the versatility of our approach to 3SUM-INDEXING, we show that our techniques give new results for another 3SUM variant and natural extensions of classic problems such as detecting target weight cycles in digraphs and $X + Y$ selection.

1 Introduction

The starting point for this paper is the **3SUM** problem, which is defined:

> Given three lists of numbers X, Y, and Z, each of size n, determine whether there exists a triple $(x, y, z) \in X \times Y \times Z$ such that $x + y + z = 0$.

3SUM presents a window into the nuances of fine-grained analysis; it has a folklore $\Theta(n^2)$ time algorithm, but, with significant effort, it is possible to construct an algorithm that improves upon this running time by approximately two $\log n$ factors [Cha18]. The fact that it is possible to outperform $\Theta(n^2)$ was not known until 2014, when Grønlund and Pettie gave subquadratic algorithms for the problem. Their deterministic version has a running time of $O((n^2/\log^{2/3} n)\log^{2/3}\log n)$, and their randomized version has a running time of $O((n^2/\log n)\log^2\log n)$ [GP14]. Freund and then Gold and Sharir independently improved the deterministic running time to $O((n^2/\log n)\log\log n)$ [Fre15, GS15]. More recently, Chan gave an $O((n^2/\log^2 n)\log^{O(1)}\log n)$ time algorithm [Cha18].

These improvements are particularly valuable because 3SUM reduces to and from multitudes of problems. Gajentann and Overmars gave the first set of reductions from 3SUM. Their work focused on geometric problems in 2D and 3D [GO95]. Since then, publications have given reductions from 3SUM and its variations to problems such as polygon containment [BHP01], graph feature detection [Pat10, KPP16, CWX22], data compression [CHC09], and string alignment [AWW14, ACLL14]. This gives a class of 3SUM-hard problems. In the other direction, some pattern matching problems [AC22] reduce *to* 3SUM.

Because 3SUM is such a fundamental problem, many different formulations of it are studied in literature, and finding faster algorithms for these variants is important. In this paper, we study an existing online variant and introduce a new asymmetric variant. Both are related to an array of problems in an analogous way to standard 3SUM and shed light on 3SUM itself. We hope that calling attention to these variants will encourage other researchers to explore them further.

1.1 3SUM-INDEXING

In 2017, Goldstein, Kopelowitz, Lewenstein, and Porat pioneered the study of 3SUM in an online context by introducing the problem **3SUM-INDEXING** [GKLP17], which is defined as follows:

> The algorithm is given two lists of values X and Y, each of size n, and allowed to preprocess them. After preprocessing is finished, one or more input query values z are given sequentially; for each, the goal is to determine whether there exists $(x, y) \in X \times Y$ such that $x + y + z = 0$.

Let us define the ***query time*** as the time to evaluate a single z value after preprocessing is finished.

Recent works have shown the connection between 3SUM-INDEXING and time/space tradeoffs for set disjointedness, set interseciton, reachability, APSP, and gapped string indexing [GKLP17, GLP19, BGL+22]. Other works have given results for lower and upper bounds on the space required during the preprocessing stage to evaluate queries in a given amount of time [KP19, CGL23, GGH+20]. While the best 3SUM algorithms only shave polylogarithmic factors from the folklore running time, there are algorithms for 3SUM-INDEXING that shave polynomial factors from the preprocessing space and from the query time as compared to folklore algorithms [KP19]. The theme between all of these papers is to examine the *space* consumed by the data structure created during the preprocessing phase and the time/space taken to evaluate queries.

In this paper, we examine the trade-off between the *time* for preprocessing and the time to evaluate queries. Investigating 3SUM-INDEXING in this context allows us to attack the problem from a different angle: Given slightly subquadratic preprocessing time, how fast can we evaluate input queries? Making progress on this question gives us results for 3SUM with preprocessing. Any

instance of 3Sum with lists X, Y, and Z (all of length n) can be converted into an instance of 3Sum-Indexing with n queries: just preprocess the lists X and Y, then individually query each value in list Z. Large speedups over $O(n)$ for a single query translate to large speedups over $O(n^2)$ for 3Sum after taking mildly subquadratic time to preprocess lists X and Y.

Note, however, that examining 3Sum-Indexing from this perspective has two important differences from the standard 3Sum problem: 1) we separate the time complexity for preprocessing and for evaluating queries, and 2) we are focusing on evaluating individual query values, whereas the standard 3Sum problem allows us to consider an entire list of Z values all at once. This means that the applications derived from our 3Sum-Indexing techniques (see Section 1.3) are distinct from the applications that are possible with standard 3Sum techniques.

To further motivate the study of 3Sum-Indexing in the context of time complexity, we highlight the work of Chan and Lewenstein concerning 3Sum with preprocessing [CL15]. For a specific 3Sum variant, they get an algorithm with strongly subquadratic preprocessing time and strongly sublinear query time. The catch is that the algorithm only works on special integer input sets. They also consider the problem of preprocessing a small universe of integers so that 3Sum on subsets of the universe can be solved in strongly subquadratic time (but with expensive preprocessing time). We see our approach to 3Sum-Indexing as a follow-up to Chan and Lewenstein's work where we achieve a balance between universe generality and preprocessing time: we take all real numbers as the universe *and* we constrict ourselves to slightly subquadratic preprocessing time.

Folklore Algorithms. Before discussing our results, we give a couple folklore algorithms for 3Sum-Indexing with real-valued inputs (we focus on deterministic algorithms for the real RAM). Given two lists X and Y, each of size n, we could simply sort X and Y separately in time $O(n \log n)$ during the preprocessing stage and then use the folklore "two pointers method" for 3Sum to evaluate each query value in $O(n)$ time. We could also directly calculate and sort all sums in $X + Y$ during the preprocessing stage in time $O(n^2 \log n)$ (the ***Cartesian sum*** $X + Y$ is the set $\{x + y : (x, y) \in X \times Y\}$). This allows us to achieve $O(\log n)$ query time by binary searching for each desired value. Thus, the most interesting cases to study are when we achieve subquadratic preprocessing time and sublinear query time concurrently.

Our Results. The first contribution of this paper is to observe that, given $o(n^2)$ preprocessing time, it is indeed possible to evaluate 3Sum-Indexing queries in time superpolylogarithmically faster than $O(n)$ by using modified versions of existing techniques. Then, using some new techniques, we develop an even faster algorithm for 3Sum-Indexing. We present our result in terms of a parameter m that allows us to adjust time spent during preprocessing versus time spend to evaluate queries. Our algorithm is deterministic and operates on real values. The full algorithm is presented in Sections 5 and 6.

Theorem 1. 3Sum-Indexing *has an algorithm with* $\min\{O((n^2 \log^{5/3}(m \log n)/\log^{2/3} n) \log^{4/3} \log n), O((n^2 \log^3(m \log n)/\log n) \log \log n)\}$ *preprocessing time and* $O(n/m)$ *query time for any desired* $m \leq n^{O(1/\log^{1/2} \log n)}$.

By substituting $m = 2^{c \log^{2/5} n/\log^{4/5} \log n}$, we get the following corollary:

Corollary 1. *For any* $c = o(1)$*, there is an algorithm for* 3Sum-Indexing *with* $o(n^2)$ *preprocessing time that has* $n/2^{c \log^{2/5} n/\log^{4/5} \log n}$ *query time.*

As explained earlier, this gives an algorithm for 3Sum with running time just $n^2/2^{c \log^{2/5} n/\log^{4/5} \log n}$ for any $c = o(1)$ that uses $o(n^2)$ time to preprocess lists X and Y.

To contextualize our new algorithm, we give baseline summarizing the fastest algorithms for 3Sum-Indexing that follow from extensions of previous work (see Section 3.1 for a full explanation). These baseline results are presented in Figure 1. We separate methods that rely on AKS sorting [AKS83] from those that don't because, while the big-O complexity of these methods is desirable, their actual running times have exceptionally large constant factors. In other words, AKS sorting gives these algorithms an unfair advantage when evaluating their big-O complexity.

Query Time Given $o(n^2)$ Preprocessing Time

Source	With AKS sorting allowed	Without AKS sorting
Folklore	$\Theta(n)$	$\Theta(n)$
Grønlund and Pettie [GP14]	$n\log^{2/3}\log n/(c\log^{2/3} n)$	$n\log^{2/3}\log n/(c\log^{2/3} n)$
Freund [Fre15]	$n\log\log n/(c\log n)$	$n\log\log n/(c\log n)$
Gold and Sharir [GS15]	$n\log\log n/(c\log n)$	$n\log\log n/(c\log n)$
Chan [Cha18]	$n/2^{c\log^{1/3} n}$	$n/2^{c\log^{1/4} n}$
This Paper	$n/2^{c\log^{2/5} n/\log^{4/5}\log n}$	$n/2^{c\log^{2/5} n/\log^{4/5}\log n}$

Figure 1: *These are for deterministic methods with real-valued inputs, and they hold for any $c = o(1)$.*

The speedups for Grønlund and Pettie, Freund, and Gold and Sharir exactly match the speedups they achieve for their respective 3Sum algorithms. We observe, however, that Chan's bit packing techniques for 3Sum can be adapted with some work to yield an algorithm for 3Sum-Indexing that has superpolylogarithmically faster query time. For our results, we use improved techniques to handle bit packed lists, and we introduce the technique of grouping lists that contain self-similar structures in a specific manner that allows us to sort them in multiple stages. This gives us a faster algorithm and allows us to stop relying on the AKS sorting network. Our new algorithm outperforms previous methods by a similar margin even when allowing these methods to use randomization and an integer universe. Our speedups for 3Sum-Indexing are immediately important because 3Sum-Indexing is a fundamental problem in fine-grained complexity.

1.2 Long-3Sum

For our next 3Sum variant, we make a simple observation: research on the 3Sum problem could be made more illuminating if we consider cases where X, Y, and Z are not necessarily all equal in size. Numerous other problems are studied in the context of multiple parameters specifying input size, such as those related to rectangular matrix multiplication [Cop82, GU18] and APSP-like problems on sparse graphs [LWW18, AR18, DJWW22]. These works have led to important results for fundamental problems; for example, Coppersmith's rectangular matrix multiplication algorithms [Cop82] led to a significantly faster APSP algorithm [Wil14] and faster algorithms to detect cycles in graphs [YZ04]. To study 3Sum in a similar context, we introduce **Long-3Sum**:

> Given three lists of numbers X, Y, and Z such that $|X| = |Y| = n \leq |Z| = kn$, determine whether there exists $(x, y, z) \in X \times Y \times Z$ such that $x + y + z = 0$.

We choose to study this variant in particular because instances of other variants, such as ones where $|X| \neq |Y| \neq |Z|$ or $|X| = |Y| > |Z|$, can either be solved quickly via direct sorting methods or reduced to Long-3Sum instances efficiently (see Section 3.3 for details). In contrast, Long-3Sum

does *not* reduce to 3Sum without incurring a huge slowdown when using known techniques.

Our Results. To get a fast algorithm for Long-3Sum, we can perform a simple reduction to 3Sum-Indexing. Specifically, Long-3Sum on lists X, Y, and Z where $|X| = |Y| = n$ and $|Z| = kn \geq n$ reduces to 3Sum-Indexing on lists X and Y with kn query values. Using Theorem 1 with $m = k \log n$, we achieve $O(n/m)$ time per query value or $O(n^2/\log n)$ time over all queries. Combining this with the time required for preprocessing, we get:

Corollary 2. Long-3Sum *has an algorithm running in time* $\min\{O((n^2 \log^{5/3}(k \log n)/\log^{2/3} n) \log^{4/3} \log n), O((n^2 \log^3(k \log n)/\log n) \log \log n)\}$ *for* $k \leq n^{O(1/\log^{1/2} \log n)}$.

Time Complexities with AKS Sorting Allowed (Left) and Not Allowed (Right)

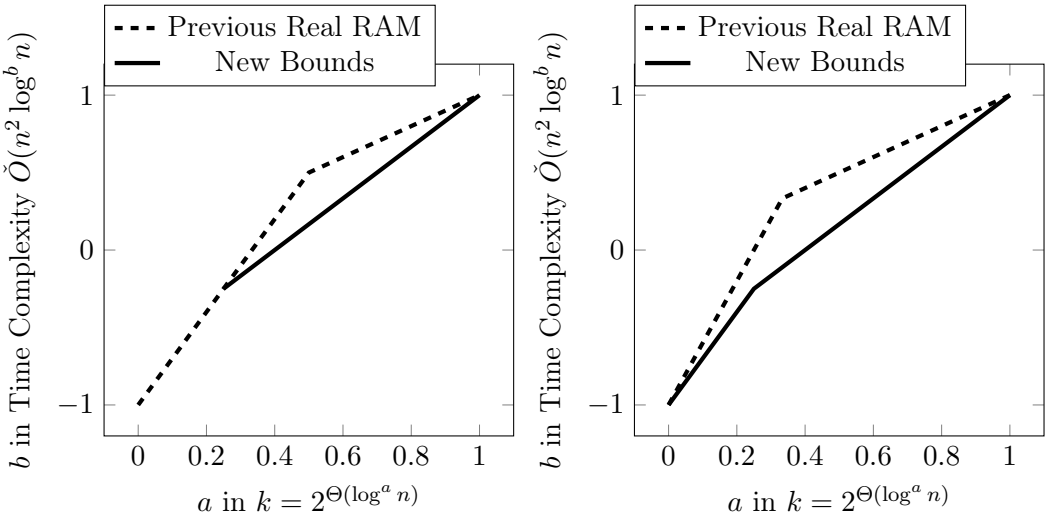

Figure 2: *These charts apply for all* $0 < a < 1$*, and all bounds here allow randomization (although our new bounds are for a deterministic algorithm on real values).* $\check{O}$ *omits* $\log \log n$ *factors.*

Figure 2 graphs the time complexity of our Long-3Sum algorithm alongside adaptations of the baseline algorithms discussed for 3Sum-Indexing (see Section 3.2 for the derivation). Our algorithm outperforms all baseline algorithms on the real RAM when $2^{\omega(\log^{1/4} n \log \log n)} < k < n^{o(1/\log^2 \log n)}$ by a factor of up to $O(\log^{1/3} n/\log^{4/3} \log n)$, and it finally improves over the naive $O(n^2 \log k)$ algorithm for larger k. When avoiding AKS sorting (AKS sorting, as explained previously, gives an unfair advantage during big-O calculations), we can outperform all previous algorithms on the real RAM when $\log^{\omega(1)} n < k < n^{o(1/\log^2 \log n)}$, which covers nearly all $k = n^{o(1)}$.

1.3 Applications

In this section, we describe how our techniques for 3Sum-Indexing can be used to get fast algorithms to find small target weight cycles in digraphs and to select rankings within Cartesian sums. All of our algorithms are deterministic and for the real RAM. We also outline some reductions.

Target Weight Cycle Detection. A common problem studied alongside 3Sum is the zero-weight triangle problem [GP14, Fre15, Cha18]: given an edge-weighted digraph with n vertices, does it contain a zero-sum triangle? This problem is tied to 3Sum because the current techniques

for 3SUM also give fast algorithms to compute a variant of tropical matrix multiplication called (TARGET,+)-MATRIX-MULTIPLICATION on copies of a graph's adjacency matrix, and this can be used to detect the desired triangles. Using our techniques for 3SUM-INDEXING, we get an algorithm for (TARGET,+)-MATRIX-MULTIPLICATION in the online setting with query time superpolylogarithmically faster than previous algorithms, allowing us to get faster procedures for a variety of natural online target weight cycle problems.

Theorem 2. *Given an n vertex digraph with real edge weights and* $\min\{O((n^3 \log^{5/3}(m \log n)/\log^{2/3} n) \log^{4/3} \log n), O((n^3 \log^3(m \log n)/\log n) \log\log n)\}$ *preprocessing time (for any desired $m \leq n^{O(1/\log^{1/2}\log n)}$), we can answer the following queries deterministically in $O(n^3/m)$ time:*

- *Determine whether there exists a triangle with edge weights summing to a query target value.*
- *Determine whether there exists a zero-sum 4-cycle containing a specific query vertex.*
- *Determine whether there exists a zero-sum 5-cycle containing a specific query edge.*

The general idea behind this algorithm is to perform Four Russians style preprocessing [ADKF70] on copies of the graph's adjacency matrix, except we process superpolylogarithmic size blocks and use bit packed lists representing sorted orderings instead of truth tables. See Section 7 for details.

Using Theorem 2 and substituting either $m = 2^{c\log^{2/5} n/\log^{4/5}\log n}$ or $m = k \log n$, we get:

Corollary 3. *Given an n vertex digraph with real edge weights, each query from Theorem 2 can be evaluated deterministically in time $n^3/2^{c\log^{2/5} n/\log^{4/5}\log n}$ for any $c = o(1)$ after using $o(n^3)$ preprocessing time.*

Corollary 4. *Given an n vertex digraph with real edge weights and a series of k queries from Theorem 2, all queries can be evaluated deterministically in time* $\min\{O((n^3 \log^{5/3}(k \log n)/\log^{2/3} n) \log^{4/3} \log n), O((n^3 \log^3(k \log n)/\log n) \log\log n)\}$ *for $k \leq n^{O(1/\log^{1/2}\log n)}$.*

All of these algorithms can be trivially adapted to report witnesses and/or give the *closest* cycle to each target value. They improve upon previous results analogously to our 3SUM-INDEXING and LONG-3SUM results; for Corollary 3, we improve the query time by a superpolylogarithmic factor, and for Corollary 4, we improve previous running times for most $k = n^{o(1)}$. These results have direct implications for the fine-grained analysis of problems involving target weight cycles. Some applications include electrical circuit modeling, periodic scheduling, and networking [KLM+09].

Cartesian Selection Problems. Cartesian selection problems are also tied to fast 3SUM algorithms, although this connection appears to be less explored in literature. Using our 3SUM-INDEXING techniques, we obtain faster algorithms for Frederickson and Johnson's $X + Y$ selection problem in the online setting [FJ82]. Here, we call their problem **SQUARE-SELECT**:

> The algorithm is given two lists of real numbers X and Y (each of size n) and is allowed to preprocess them. After preprocessing is finished, one or more input query rankings r are given; for each, the goal is to select the r^{th} smallest sum from $X + Y$ and report this value.

Naively, we could explicitly calculate and sort all sums in $X+Y$ during preprocessing, which takes $O(n^2 \log n)$ time. This allows us to report any query ranking in $O(1)$ time. We could also just sort X and Y independently during preprocessing, which takes $O(n \log n)$ time, then use Frederickson and Johnson's sorted matrix selection algorithm [FJ84] to evaluate each query in $O(n)$ time. As

with 3Sum-Indexing, the most interesting case is when we achieve subquadratic preprocessing time and sublinear query time concurrently.

Our main algorithm for the problem works by preprocessing $X + Y$ using the exact same techniques as 3Sum-Indexing algorithm, converting $X + Y$ into a rectangular matrix of sums with sorted rows and columns, and then selecting in this new matrix rapidly. See section 8 for details.

Theorem 3. Square-Select *has a deterministic algorithm with* $\min\{O((n^2 \log^{5/3}(m \log n)/\log^{2/3} n) \log^{4/3} \log n), O((n^2 \log^3(m \log n)/\log n) \log \log n)\}$ *preprocessing time and* $O(n/m)$ *query time for any desired* $m \leq n^{O(1/\log^{1/2} \log n)}$.

By substituting either $m = 2^{c \log^{2/5} n/\log^{4/5} \log n}$ or $m = k \log n$, we get the following corollaries:

Corollary 5. *For any* $c = o(1)$, Square-Select *has a deterministic algorithm with* $o(n^2)$ *preprocessing time that can evaluate queries in time* $n/2^{c \log^{2/5} n/\log^{4/5} \log n}$.

Corollary 6. *There is a deterministic algorithm for instances of* Square-Select *with* kn *queries that has total running time* $\min\{O((n^2 \log^{5/3}(k \log n)/\log^{2/3} n) \log^{4/3} \log n), O((n^2 \log^3(k \log n)/\log n) \log \log n)\}$.

As with 3Sum-Indexing, our online results improve upon previous algorithms by a superpolylogarithmic factor, and Corollary 6 improves previous results for most $k = n^{o(1)}$. One important application of our online algorithm is finding the optimum distribution of effort for concave functions given several different effort quotas; see Frederickson and Johnson's paper for details [FJ82].

When implemented in an offline manner, Square-Select reduces from $X + Y$ sorting. To reduce, we simply give lists X and Y as input and then select for the 0^{th} through $(n^2 - 1)^{\text{th}}$ smallest. Whether we can outperform the folklore $O(n^2 \log n)$ time $X + Y$ sorting algorithm is a major open problem [BCD+06, KLM19]. Our results from Corollary 6 come close, yielding improved running times over previous algorithms when selecting for the exact values for $2^{O(\log^{1-\epsilon} n)} n$ of the rankings in $X + Y$ for any constant $\epsilon > 0$ (having $\epsilon = 0$ and a sufficiently large big-O constant would allow us to sort the entirety of $X + Y$).

Reductions. One of the reasons 3Sum is so important is because it reduces to and from multitudes of problems. 3Sum-Indexing and Long-3Sum reduce to and from many of the same problems as 3Sum, just with an online or asymmetric twist, respectively. For example, consider the general position testing problem: Given a set of n red, n blue, and n green points in the plane, does there exist a red/blue/green triple that is collinear? Gajentaan and Overmars [GO95] gave a simple reduction from 3Sum to this problem. Using the same methods, 3Sum-Indexing reduces to an online version of the problem where the set of red and blue points are given, the algorithm is allowed to perform some preprocessing, and then the algorithm must determine whether query points are collinear with a red and blue point pair. Long-3Sum follows the same reduction as 3Sum except there are n red points, n blue points, and kn green points. Many other examples can be derived easily; for the sake of brevity, we do not mention them here.

1.4 Technical Overview

Here, we will discuss previous methods for getting subquadratic-time 3Sum algorithms, explore the challenges that arise when applying these techniques to 3Sum-Indexing, and then introduce our new algorithm for 3Sum-Indexing. Note that the Cartesian sum $X + Y$ usually refers to the case that X and Y are sorted. If represented graphically, the sums would be displayed in a grid with X addends nondecreasing and Y addends fixed when moving right and with Y addends nondecreasing and X addends fixed when moving up. $[n]$ refers to the set $\{0, 1, \ldots n - 1\}$.

Prior Work. Algorithm 1 gives an outline of a subquadratic time 3SUM procedure often used in literature. It may be summarized as follows: divide sorted X and sorted Y into contiguous segments $X_0 \dots X_{n/d-1}$ and $Y_0 \dots Y_{n/d-1}$ of size d values each, pre-sort the Cartesian sum for every combination of an X and Y segment, and then rapidly search for the negative of each z value in the pre-sorted Cartesian sums. For a more comprehensive explanation, see Freund's paper [Fre15].

Algorithm 1 A subquadratic time algorithm for 3SUM.

Given: lists X, Y, and Z of n values each.
Sort X, Y, and Z into nondecreasing order using merge sort. Negate every value in Z.
Pre-sort every Cartesian sum in the multiset $\{X_i + Y_j : i \in [n/d] \text{ and } j \in [n/d]\}$
for each $z \in Z$ **do**
    $p \leftarrow n/d - 1$; $q \leftarrow 0$
    while $p \geq 0$ and $q \leq n/d - 1$ **do**
        Binary search for z in $X_p + Y_q$ If a solution is found, return "Yes" and halt.
        if $\min(X_p) + \max(Y_q) > z$ **then**
            $p \leftarrow p - 1$
        else
            $q \leftarrow q + 1$
        end if
    end while
end for
Return "No solution exists" and halt.

For each Z value, we only have to spend $O((n \log d)/d)$ time to determine whether it participates in a 3SUM solution (assuming that the Cartesian sums are already sorted). This improves over the folklore algorithm by a factor of $\Theta(d/\log d)$ because the folklore algorithm is only able to check for solutions individually, whereas the fast algorithm can binary search for solutions inside sorted Cartesian sums (we will refer to these Cartesian sums as ***blocks*** from now on).

The difficulty with the above procedure lies in preprocessing all of the blocks. To tackle this, the first subquadratic algorithms for 3SUM [GP14, Fre15, GS15] all used a subroutine solving instances of the ***dominance merge*** problem. Roughly speaking, this subroutine allows us to test all blocks at once to determine if they contain a specific exact sub-ordering. Unfortunately, because we must guess and check all sub-orderings, the algorithm becomes impractically slow if d exceeds $\Theta(\log n)$. The most recent 3SUM algorithm [Cha18] uses a different subroutine that computes ***hyperplane cuttings*** in order to preprocess blocks. Combining this geometric technique with bit packing, the algorithm can efficiently simulate checking for solutions in sorted blocks of size $\check{\Theta}(\log^2 n)$. However, this algorithm is incapable of sorting blocks much larger than this in subquadratic time.

We've highlighted that the above algorithms struggle with larger blocks to show that they do not translate well to 3SUM-INDEXING. To achieve superpolylogarithmic improvements in query time, we must sort blocks that are superpolylogarithmic in size using subquadratic preprocessing time.

An Initial Algorithm. Here, use Chan's bit packing techniques [Cha18] to get a time efficient procedure for 3SUM-INDEXING. It works by sorting all blocks in (sorted X) + (sorted Y) during the preprocessing stage and then applying a variant of Algorithm 1 to evaluate each input query. As before, sorting large blocks is the biggest challenge. We present a technique below that is capable of sorting blocks of size $2^{c \log^{1/3} n}$ for any $c = o(1)$ using $o(n^2)$ time. For details, see Section 3.1.

The general idea is to represent the sorted ordering of each $d \times d$ block as a ***bit packed list*** of ***local coordinates*** in $[d] \times [d]$. In the final ordering, each successive local coordinate represents the

position of the next smallest sum relative to the bottom corner of the block in question. To create this ordering, we enumerate the local coordinates for all sums in the block, then permute them into sorted order using a sorting network. This procedure is subquadratic because we operate on entire *words* of our bit packed lists at once, allowing us to effectively perform a superconstant number of coordinate manipulations in constant time. Unfortunately, we cannot hope to pack multiple real values into each word, so it appears that we must sort by operating on the coordinates alone. However, there is a trick: if we could replace the local coordinates with short integers that encode the appropriate real sums, we would be able to pack several integers per word and sort using them.

For inspiration, let us consider Fredman's trick, which is the following observation: $X[i]+Y[j] > X[i']+Y[j'] \leftrightarrow X[i]-X[i'] > Y[j']-Y[j]$ [Fre76]. In other words, comparing the sums represented by two coordinates is equivalent to comparing the difference between two X list values and the difference between two Y list values. When sorting blocks, there are many possible sums but few possible differences, so we can afford to manually sort groupings of differences and then assign the members of each group integer ***difference rankings*** based on their position in the ordering. Clearly, comparing two difference rankings from the same group is equivalent to comparing two real differences, and this is equivalent to comparing the values of two local coordinates by Fredman's trick. These difference rankings are short enough that we can pack several of them per word as desired. However, we can only use them if we perform pre-determined swaps; *this* is why we have to use a sorting network. A major downside to this algorithm is that we must perform many rounds of packed integer sorting because the algorithm relies on grouping together identical comparisons as an intermediate step (we haven't mentioned this step explicitly).

Our Algorithm. Our new algorithm improves significantly upon the above adaptation. While it still uses a variant of Algorithm 1 to evaluate queries, it is faster because it can partition $X+Y$ into blocks of side length $d = 2^{c\log^{2/5} n/\log^{4/5}\log n}$ for any $c = o(1)$ and sort these in $o(n^2)$ time. We also find our methods for sorting large blocks interesting in their own right; while preprocessing polylogarithmic-size blocks has a rich history dating back to the Four Russians algorithm for Boolean matrix multiplication [ADKF70], relatively little research has focused on larger blocks.

A summary of our new techniques is as follows. We start by restructuring how blocks are sorted: instead of using sorting networks to compute the ordering directly, we implement partitioning tricks to more rapidly perform macroscopic-level sorting and then only use sorting networks on small sub-problems. Our next improvement is that we operate on entire lists of comparisons at once (as opposed to individual comparisons). This allows us to substitute most of the packed integer sorting for pigeonhole sorting, which is much faster in this context. However, operating in this manner makes it much more difficult to produce short difference rankings to represent the coordinate comparisons we want to make. Our final contribution is a technical multi-level grouping procedure to handle these comparisons. We operate on a large strip of blocks at once and group query lists from distant blocks that have a specific desirable quality. Using these groupings, we produce the difference rankings in multiple steps, which gives us a chance to shorten the rankings significantly before they are finally used. We feel that our multi-level grouping techniques are particularly elegant, and we expect them to apply to other problems involving Cartesian sums such as variants of APSP.

Block Sorting Scheme. Our algorithm sorts the local coordinates for an entire block by merging together larger and larger sorted sections of the block and using a subroutine for sorting polylogarithmic-size portions of the block. As a whole, this procedure will look similar to merge sort, but there are several technical details involved. Let us consider merging two sorted sections. Because we are operating on lists of coordinates *representing* real values, we cannot efficiently merge the

sorted bit packed lists using a modified scan. We can, however, quickly select for polylogarithmically-spaced quantiles between the two lists using direct comparisons and a complex helper algorithm. Once we have our quantiles, we can rapidly partition the two lists into segments of local coordinates that must fall between each pair of consecutive quantiles, giving us a polylogarithmic-size ***block buckets*** for each quantile range. Sorting each block bucket lets us produce the final ordering.

Our goal now is to sort the block buckets. As with the adaptation of Chan's algorithm, we will use a sorting network to accomplish this task, but we implement the network using a completely different strategy. The difference in methods allows us to avoid many of the technical sorting aspects of Chan's algorithm, making our implementation more efficient. Also, we end up using the bitonic sorting network [Bat68] instead of the AKS sorting network, which eliminates the huge constant factors in the final time complexity. A brief description of our strategy is as follows. Sorting networks operate by performing a large number of pre-determined comparisons at once, swaping values based on these comparisons, and then repeating this process a small number of times. To sort our block buckets using a sorting network, we will use truth tables to produce a list of *pairs* of local coordinates representing the comparisons we want to make, have a subroutine evaluate these comparisons, and then use truth tables to swap local coordinates based on these comparisons. At first, this procedure might sound impossible to perform efficiently because we might have to compare/swap values across multiple words, meaning that we can do no better than individually comparing/swapping values. However, due to technical details of the bitonic sorting network of the block buckets' construction, we can group comparisons together and group swaps together, allowing us to perform them in *batches*. This is where the efficiency of our algorithm comes from; we can perform a superconstant number of comparison operations and swap operations in constant time.

Query Lists. We now describe our general scheme for partitioning and re-arranging lists of local coordinate comparisons to make them easier to handle. In the adaptation of Chan's algorithm we don't rearrange in this manner and just evaluate coordinate comparisons using a few rounds of packed integer sorting. Our first step is to partition the lists of coordinate comparisons into ***query lists***; these are lists of local coordinate pairs that fit into less than a word. We then partition $X+Y$ into ***provinces***, which are strips p blocks high that span the full width of $X+Y$ (we will discuss the variable p in the next subsection). Using provinces, we group our query lists into ***families***; two query lists are in the same family if and only if they have identical local coordinate pairs in identical order and come from the same province. It is possible to have query lists with identical coordinate pairs in identical order because the coordinate pairs only indicate position relative to their respective block, and we allow the source blocks for each identical query list to be different.

The above grouping procedure is rapid because we can simply read the contents of each query list in $O(1)$ time and then assign it to the correct family. Once the query lists have been evaluated, we can rapidly extract them from their families using a similar pigeonhole procedure. The key is that we are never forced to perform any sort of comparison-based sorting, allowing us to avoid the packed integer sorting bottleneck present in the adaptation of Chan's work.

Multi-Level Grouping. Using a series of novel multi-level grouping techniques, we determine the results for all coordinate comparisons in each query list once they are grouped into families. This type of grouping was absent in all previous publications. Our end goal is to evaluate each query list in nearly constant time using difference rankings. Recall that comparing an X difference ranking and a Y difference ranking is equivalent to comparing a real X difference and a real Y difference. By Fredman's trick, this is also equivalent to comparing two coordinates' sums. We note: naively constructing difference rankings yields integers $O(\log n)$ bits long (because this corresponds

to computing all possible differences across all of $X + Y$), smaller difference rankings allow our overall algorithm to run faster (because more can be packed into a word), and longer query lists allow our overall algorithm to run faster (because we are using pigeonhole sorting).

We now re-examine the height of the provinces used to create families. We set p just large enough so that, on average, each query list will come from a block with the same X position as a few other query lists in its family (the same ***absolute X position***). The shape of provinces becomes important when generating integer difference rankings for the whole province, which is why we keep p small. Each province may have up to $\Theta(nd)$ possible X differences for its blocks, so, if we were to naively assign integer difference rankings, they would be of size $\Theta(\log n)$. This is far too large. However, because each province is very thin, the number of Y differences required for each province is only about $O(pd^2)$. We only ever compare X differences against Y differences, so we can give all differences a ranking of size $O(\log p)$ bits and still support all possible comparisons.

Each family has many query lists occupying each row of blocks and each column of blocks on average. This means that, if we make the list of difference rankings for a representative query list in every row and column for a family, we could use those ranking lists to populate every query list in the family via simple copy operations, seemingly allowing us to quickly generate the results for all query lists. This method is fast because it allows us to copy *all* difference rankings for a query list at once, avoiding the need to individually meddle with the rankings. Unfortunately, with this method, no combination of query list length and p value will allow us to sort blocks larger than possible with the baseline methods in subquadratic time.

There is a trick we can use. As stated, the average family member (query list) will have the same absolute X position as a few other members of its family, but it also has the same absolute Y position as *polynomially* many members due to the thin shape of provinces. This allows us to partition almost all family members into small ***sparse squares***. These are groupings of query lists where each member query list has the same absolute Y position as about $\log n / \log p$ members and the same absolute X position as about $\log n / \log p$ members. See Figure 3 for an example.

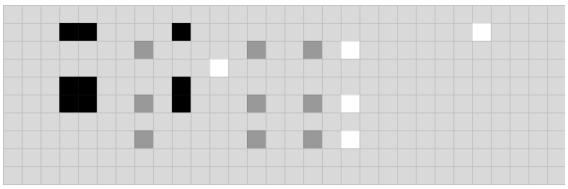

Figure 3: *The black and dark gray boxes each represent a single sparse square. The white boxes represent straggler lists that will be processed individually (there are asymptotically few stragglers).*

We then create representative ranking lists for each row and column, but we only have to populate the query lists on the bottom and left edge of each sparse square with the difference rankings, which means that there are approximately $\Theta(\log n / \log p)$ less copy operations than in the method from a couple paragraphs ago. We only populate the bottom and left edge with rankings because these rankings contain all the information required to evaluate the difference queries for *all* query lists in the sparse square. Because each sparse square only uses very few difference rankings, we can sort the rankings along the bottom and left edge of each sparse square again and then replace them with tiny difference rankings that will be functionally equivalent. Now, we can rapidly populate all query lists inside the sparse squares with their rankings and evaluate the queries.

We shorten the rankings twice: first because of the shape of provinces and second because of the ability to make sparse squares. This saves us a time complexity factor of $\check{\Theta}((\log n / \log p)^2)$, which is enough for our method to sort larger blocks than all previous methods could in subquadratic time.

2 Preliminaries

2.1 Notations, Definitions, and Assumptions

The k^{th} smallest member of a multiset is greater than exactly k other members. Equal values are sorted/ranked lexicographically based on additional information. ***Lexicographical order according to*** **(A, B)** indicates that members should be listed according to these rules:

- Members with a larger A value should come later in the list.
- If two members share the same A value, the one with a larger B value should come later.
- If two members share the same A and B values, they are be listed in a fixed arbitrary order.

Multiset A ***dominates*** another multiset B if all values in A are greater than or equal to all values in B. Recall that we represent the Cartesian sum of two sorted lists as a matrix of sums where the bottom left sum corresponds to the sum of the smallest entries in each list and sums in a row or column all have a common addend from one list and consecutive addends from the other list. For $X + Y$ specifically, the X addends only change when moving horizontally and the Y addends only change when moving vertically.

Whenever we use the word ***conceptual***, it indicates that a statement is merely included for expositional purposes. The names of problems refer both to the problem itself and the algorithm for the problem. Unless otherwise stated, all parameters are in $\Omega(1)$. We assume that, whenever required, a constant fraction of space is allotted as padding between values in memory. This makes data copy operations easier and doesn't impact the asymptotic behavior of our algorithms.

2.2 Model of Computation

Our new algorithms operate entirely on the real RAM model of computation, which is defined as follows. RAM stores real numbers and integers of word length w bits. Only addition, subtraction, and comparison of real numbers is allowed. Each integer may be manipulated in constant time using one of a constant number of arbitrarily-defined functions (these are decided upon and fixed before computation starts). For this paper, we add the extra stipulation that every such function must be computable in AC^0. AC^0 operations have an input of size $O(w)$ bits, and they are computable by a circuit of constant depth and $w^{O(1)}$ size (unlimited fan-in is permitted). They are often used as a baseline for what a simple computer can process in constant time [BDP05].

Note that AC^0 circuits can't compute general multiplication and division. We get around this by implying that all parameters (but not indexes or inputs, of course) are appropriately adjusted to be of the form $x2^e$ where e is a positive integer in $[w]$ and x is a positive integer in $[c_1]$ for a fixed constant c_1. These constraints mean that multiplication and division reduce to the corresponding operation on a ***constant*** number of bits and then shifting operations.

Words may also be manipulated using truth tables; however, these truth tables must be generated using a combination of AC^0 operations. Truth tables allow some nonstandard word operations in $O(1)$ time so long as $O(2^{\epsilon w})$ preprocessing time is allowed.

2.3 Structural Observation

We now make a structural observation:

Lemma 2.1. *Generate $X+Y$ from sorted X and sorted Y. If all members in a multiset A of sums in $X + Y$ are above and to the right of all members in another multiset B of sums in $X + Y$, then*

A dominates B. If $X + Y$ is partitioned into squares of side length d, and all squares are sorted, this operation implies $O(n/d)$ sorted lists of size $O(nd)$ each. Every sum in $X + Y$ is assigned to one of these lists during this process.

Let us call the $O(n/d)$ sorted lists ***long sorted diagonals***.

Proof. Consider an arbitrary member of A and an arbitrary member of B (call these a and b respectively). Recall that a and b each equal the sum of one value from X and one value from Y. Because a is above b, a's addend from Y is at least as large as b's addend from Y. A similar argument applies to the X addends, meaning that the value of a must be at least as large as the value of b. If we partition $X + Y$ (formed from sorted X and sorted Y) into contiguous squares of equal size, the square directly above and to the right of an arbitrary square will dominate the arbitrary square via the same argument, and we can chain squares in this way to produce the long sorted diagonals described in the lemma. □

3 Baseline Algorithms

In this section, we will adapt methods from previous papers to attack 3Sum-Indexing and then use these algorithms to get a baseline for Long-3Sum. We also briefly discuss naive algorithms for other variants of 3Sum. Baselines for our graph cycle and Cartesian selection problems follow via similar techniques. All algorithms in this section operate on the real RAM.

3.1 3Sum-Indexing

Let us consider a 3Sum-Indexing instance on lists X and Y of size n each. To achieve a query time of $O(n/m)$, the following preprocessing times are required when using different adaptations of Chan's methods [Cha18]:

1. $O(n^2 \log^3(m \log n)/\log n)$ (for $m < n^{o(1)}$): This is a direct extension of Theorem 3.4 in Chan's paper [Cha18]. Conceptually, divide $X+Y$ into contiguous squares of $(m \log^2 n)/\log(m \log n)$ by $(m \log^2 n)/\log(m \log n)$ sums. For each square, create a bit packed list of $((m \log^2 n)/\log(m \log n))^2$ index pairs, each pair in $[(m \log^2 n)/\log(m \log n)] \times [(m \log^2 n)/\log(m \log n)]$, to represent all the indexes in the square in lexicographical order (every square will have the same starting bit packed index list). Append to each list an identifier occupying $O(1)$ words indicating which block is being considered. Sort the bit packed lists of indexes using Theorem 3.4 from Chan's paper and the AKS sorting network [AKS83] so that the sums corresponding to the indexes in each list are in nondecreasing order. This yields $O((n \log(m \log n))/(m \log^2 n))$ long sorted diagonals as per Lemma 2.1 and takes $O(n^2 \log^3(m \log n)/\log n)$ time. Note that this step has a huge constant factor in the complexity due to the use of the AKS sorting network. To evaluate queries, we just binary search for the negative of the query value in each long sorted diagonal; the time across all binary searches is $O((n \log(m \log n))/(m \log n)) \leq O(n/m)$. We binary search for the negative of the query value because this allows us to identify instances where $x + y = -z$, which is equivalent to $x + y + z = 0$.

2. $O(n^2 \log^4(m \log n)/\log n)$ (for $m < n^{o(1)}$): This is identical to the previous method, but it uses the bitonic sorting network [Bat68] instead. This eliminates the huge constant factor in the preprocessing time.

Historic 3Sum-Indexing Bounds. To get the bounds in Figure 1 for Grønlund and Pettie, Gold and Sharir, and Freund, we simply use their 3Sum algorithms during the preprocessing stage to sort small blocks in the Cartesian sum $X + Y$. We can thens use the same procedure as in their 3Sum algorithms to search for query values. No adaptations appear to improve over this performance.

3.2 Long-3Sum

To get our baseline for Long-3Sum, we first convert our 3Sum-Indexing algorithms derived from Chan's work into algorithms for Long-3Sum by substituting $m = k \log n$. We are left with the following bounds for an instance of Long-3Sum on lists X, Y, and Z where $|X| = |Y| = n \leq |Z| = kn$:

- $O(n^2 \log^3(k \log n)/\log n)$ (for $k < n^{o(1)}$). This uses AKS sorting.
- $O(n^2 \log^4(k \log n)/\log n)$ (for $k < n^{o(1)}$). This doesn't use AKS sorting.

Assuming $k \leq O(n)$, we also have the following Long-3Sum bounds:

- $O(n^2 \log k)$: Populate an n by n grid with all sums in $X + Y$. Partition this into contiguous squares of size $\min\{k, n\}$ by $\min\{k, n\}$ sums and sort every square using merge sort in time $O(n^2 \log k)$. By Lemma 2.1, this forms $O(n/\min\{k, n\})$ long sorted diagonals, and these contain all of the sums in $X + Y$. Create a list of the negative of each value from Z and sort the list using a comparison sort in time $O(kn \log n)$. Merge this list with each of the long sorted diagonals; if a value from the negative Z list and a value from a sorted $X + Y$ list are equal, this is a solution to the Long-3Sum instance. The time to merge is only $O(n^2 + nk)$.
- $O((kn^2/\log^2 n) \log^{O(1)} \log n)$: In Chan's paper [Cha18], the main result is an algorithm for standard 3Sum with three lists of length n each running in time $O((n^2 \log^{O(1)} \log n)/\log^2 n)$. This applies without requiring the AKS sorting network. Long-3Sum may be reduced to $O(k)$ instances of standard 3Sum by just partitioning Z into $O(k)$ lists of length n each.

3.3 Other 3Sum Variants

As stated in the introduction, we don't study other 3Sum variants because they either efficiently reduce to Long-3Sum or have fast naive algorithms. General 3Sum has lists of length $|X| = n \leq |Y| = k_0 n \leq |Z| = kn$ (this is equivalent to all other cases by re-labeling the inputs). We assume that all parameters are integers since padding the inputs to make this the case incurs no big-O complexity increase. In $O(k_0 kn)$ time, this problem reduces to k_0 instances of Long-3Sum by partitioning $|Y|$ into k_0 segments of length n each, invoking Long-3Sum with X, Z, and a single Y segment as inputs, and repeating this for all Y segments. We don't preprocess the Cartesian sum $X + Y$ all at once because, when using the methods in this paper and in prior work, this would not be any faster. Another algorithm for real-valued inputs where k might be much larger than n is to just sort all of the sums in $X + Y$ at once and then binary search for the negative of every Z value in the sorted sums. The time complexity is $O(n^2 k_0 \log(nk_0) + nk \log(nk_0))$.

4 Useful Lemmas

In this section, we introduce a couple lemmas that will be helpful when proving our main theorems. Lemmas similar to these have appeared in literature before; however, we still include full proofs

for the sake of completeness and to introduce a couple new ideas. If desired, the reader may skip directly to Sections 5 and 6 for a description of our 3SUM-INDEXING algorithm without reading this section.

4.1 Selection in a Multiset of Sorted Lists

We will describe an algorithm to solve instances of problem **SELECT-INTERVALS**, which is defined as follows:

> Given parameter k_g and e sorted input lists $\{L_0, L_1, \ldots L_{e-1}\}$ of values in $\mathbb{R} \cup \{\infty, -\infty\}$ (or pointers to such values) with average length s, return lists $\{R_0, R_1, \ldots R_{e-1}\}$ such that, for all $j \in [es/k_g]$ and $i \in [e]$, $R_i[j]$ is the largest value (or points to the largest value) in L_i greater than at most $k_g(j+1) - 1$ values from the multiset of all values from lists $\{L_0, L_1, \ldots L_{e-1}\}$. Each $R_i[j]$ is set to NULL if no such value exists.

The time complexity of this problem is $T_{SI}(e, s, k_g)$.

Our algorithm for this problem is inspired by Frederickson and Johnson's paper [FJ84]. It does not actually read the entirety of each input list; instead, it requests different single values within each list and uses these to conduct binary searches. It can be adapted to return any multiset of desired quantiles (as opposed to quantiles spaced apart by exactly k_g values) with the same time complexity as long as the arithmetic average gap between adjacent quantiles is k_g.

Lemma 4.1. $T_{SI}(e, s, k_g) = O((e^2 s/k_g) \log(k_g/e + 2))$

Proof. We now describe a witness algorithm to prove the lemma. Assume without loss of generality that es/k_g is a power of two. We may also assume $es \geq k_g$; otherwise, our algorithm just reports the maximum from each list and then terminate. We describe an algorithm for the value variant of the problem; the pointer variant is very similar.

Before describing our overall algorithm, we introduce a helper algorithm: Algorithm 2. Consider one application of this algorithm to the input lists $L_0, L_1, \ldots L_{e-1}$. For the median ranking $k = es/2 - 1$, it selects the k^{th} smallest of e input lists of total size es during the "while" loop. This is accomplished by dividing the lists into smaller and smaller contiguous cells and throwing away cells that contain values definitely greater than the k^{th} smallest and definitely less than the k^{th} smallest. Correctness follows from the fact that, if a cell's minimum is greater than x other cell's minimums, it must dominate at least $(x - e) \cdot (\text{CELLSIZE})$ values because the smaller minimums, except for the first e minimums, are all greater than or equal to CELLSIZE new values from their corresponding list. An analogous statement can be made for maximums, except that, because the uppermost section of each list might be smaller than CELLSIZE, a maximum less than x other maximums is only guaranteed to be less than $(x - 2e) \cdot (\text{CELLSIZE})$ values. The starting cell size is such that at most $O(e)$ cells are partitioned at first, meaning that we start with cells of size $\Theta(s)$. $O(e)$ cells remain at the end of each round, and cells are halved in size with every round. The time complexity for each selection and partition step is proportional to the number of cells. The binary searches in the last step are most expensive when cells are divided equally among the maximum number of remaining lists. Therefore, the overall time complexity of this single application is $O(e \log(s))$ if $s \geq 2$ or $O(e)$ otherwise.

Algorithm 2 Finding the k^{th} smallest in a multiset of sorted lists.

Given: sorted lists $\{M_0, M_1, \dots M_{e-1}\}$ of size $n := |M_0| + |M_1| + \cdots + |M_{e-1}|$ and parameter k
$k' \leftarrow k$
CellSize $\leftarrow n/(4e)$ rounded up to the nearest power of 2
while CellSize > 1 **do**
    Partition each list into contiguous cells of size CellSize. If a list does not evenly divide by CellSize, the one shorter segment should be the one with the largest values.
    Select the $(e + \lceil k'/\text{CellSize} \rceil)^{\text{th}}$ smallest value from the multiset containing every minimum from the cells (or select the largest minimum if this value is out of bounds) using a linear time selection algorithm. Discard all portions of $\{M_0, M_1, \dots M_{e-1}\}$ corresponding to cells with a minimum greater than this.
    Select the $(-2e + \lfloor k'/\text{CellSize} \rfloor)^{\text{th}}$ smallest value from the multiset containing every maximum from the cells (or select the smallest maximum if this value is out of bounds) using a linear time selection algorithm. Discard all portions of $\{M_0, M_1, \dots M_{e-1}\}$ corresponding to cells with a maximum less than this. For every list that was completely discarded, report its maximum.
    CellSize $\leftarrow$ CellSize/2
    $k' \leftarrow k' - \text{CellSize} \cdot \max\{0, -2e + \lfloor k'/\text{CellSize} \rfloor\}$
end while
$v \leftarrow$ the k'^{th} smallest element from the values in the remaining lists. Find v using a linear time selection algorithm. Report v if desired.
For each remaining portion of each list, binary search for the largest value less than or equal to v and report this value.

We now describe the overall algorithm proving the lemma. First, apply Algorithm 2 as described in the previous paragraph. Once execution terminates, it reports the largest value from each list less than or equal to the median (the ***middle values***), or it reports no value from a list because all values in the list are definitely greater than the median. Record these values in the output arrays as $L_i[es/(2k_g) - 1]$ for each respective i, and write NULL for each list that didn't have a value reported. Divide each list into a ***lower half*** (whose largest value is the corresponding ***middle value***) and an ***upper half*** with the remaining values. Repeat the whole median finding process on the resulting upper and lower halves independently. Note that each of these halves might have lists of varying lengths; this is okay. Continue selecting for medians in smaller and smaller collections of lists until the resulting quantiles are k_g values apart. Because the number of applications of Algorithm 2 doubles every time the lists are divided into smaller segments, the complexity of this entire procedure is dominated by the time required to evaluate the $O(es/k_g)$ final quantiles. Each evaluation starts with cells of size $O(k_g/e)$, so $O(\log(k_g/e))$ rounds of the "while" loop occur (assuming $k_g/e \geq 2$), so the time required is $O(e \log(k_g/e))$ if $k_g/e \geq 2$ or $O(e)$ time otherwise. This is equivalent to stating that each evaluation requires $O(e \log(k_g/e + 2))$ time. Multiplying by $O(es/k_g)$ gives the overall time complexity. □

Remark. This algorithm need not select for equally spaced quantiles. Given a sorted list of target quantiles, the algorithm could select for the median from this list, then the first and third quartiles from this list, and so on. The special case where quantiles are equally spaced (as analyzed above) is the most expensive due to the fact that $\log(n/2) + \log(n/2) \geq \log(n-i) + \log(i)\ \forall i \in \{1, 2, \dots n-1\}$.

4.2 Packed Integer Sorting

Considered here is a type of sorting we call **Packed-Integer-Sort**:

Given a bit packed list of n integers, each in $[b]$, permute them into nondecreasing order.

Note that each integer only occupies $\Theta(\log b)$ bits, meaning that $\Theta(w/\log b)$ integers fit into a single word. We assume $w \geq \Omega(\log b)$. It is assumed that the input list of values is given in ***bit packed form*** where each word stores the maximum number of integers (or a constant factor less than this). Let the time complexity of this problem be $T_{PIS}(b, n)$.

The idea is to store the result of merging all possible combinations of two segments of integers occupying a constant fraction of a word per segment using truth tables. The use of such tables hampers the running time for some applications; however, this does not affect the time complexity of this paper's results. See Albers and Hagerup's paper [AH97] for a more versatile set of algorithms.

Lemma 4.2. *$T_{PIS}(b, n) = O(\lceil (n \log b)/w \rceil \log((n \log b)/w + 2) + 2^{\epsilon w + O(\log w)})$ for constant ϵ such that $0 < \epsilon \leq 1$ and $b \leq O(w)$*

Proof. We give a witness algorithm. If $\log b = \Omega(w)$, an $O(n \log n)$ time algorithm is achieved by using a comparison sort to operate on the input integers, and this trivially supports the lemma. Therefore, we will assume $\log b < o(w)$ for the remainder of this proof.

Choose a sufficiently small constant ϵ and create a table $f : [b]^{\epsilon w/(2 \log b)} \times [\epsilon w/(2 \log b)] \times [b]^{\epsilon w/(2 \log b)} \times [\epsilon w/(2 \log b)] \rightarrow [b]^{\epsilon w/\log b} \times \{0, 1\} \times [\epsilon w/(2 \log b)]$. f takes two input segments of integers in $[b]$, each containing $\epsilon w/(2 \log b)$ integers (these are assigned indexes $\{0, 1, \ldots \epsilon w/(2 \log b) - 1\}$ in the order that they appear in memory). Note that each segment occupies only a small constant fraction of a word ($\epsilon w/2$ bits in total). For each segment, f also takes an index to one of its integers. Denote this table application $f[L_0, p_0, L_1, p_1]$, where L_0 is the first segment, p_0 is the pointer for the first segment, L_1 is the second segment, and p_1 is the pointer for the second segment. The value of $f[L_0, p_0, L_1, p_1]$ is set to equal the segment of integers F resulting from following construction:

1. Let F be a bit packed list containing all integers from L_0 and L_1 in nondecreasing order.
2. Discard every integer in F that came from L_0 and had an index less than p_0 in L_0.
3. Discard every integer in F that came from L_1 and had an index less than p_1 in L_1.
4. Record whether the last integer in F came from L_0 or L_1. Reading backwards from the end of F, discard every integer that came from this input segment. Stop discarding as soon as an integer that came from the opposite input segment is encountered (don't discard this integer).

f also has an option to access two specifiers, which are listed below:

- $f^{(0)}[L_0, p_0, L_1, p_1]$: Whichever input segment the last integer in $f[L_0, p_0, L_1, p_1]$ came from (a value in $\{0, 1\}$). If this value is 0, this indicates L_0; if it is 1, this indicates L_1.
- $f^{(1)}[L_0, p_0, L_1, p_1]$: The index to the smallest integer from the input segment *not* indicated by $f^{(0)}[L_0, p_0, L_1, p_1]$ that does not appear in $f[L_0, p_0, L_1, p_1]$ (because it was discarded during Item 4 when constructing F).

The total time to construct table f is $O(2^{\epsilon w + O(\log w)})$.

We are ready to sort a given bit packed list of n integers in $[b]$. Take the bit packed list and partition it into contiguous segments of size $\epsilon w/\log b$ integers. Permute each segment of integers into sorted order using truth tables (for this step, we do *not* use table f). These tables also take $O(2^{\epsilon w + O(\log w)})$ time to construct and can permute each of the segments into sorted order in time $O(1)$ each. The total time is thus $O(\lceil (n \log b)/w \rceil + 2^{\epsilon w + O(\log w)})$ because ϵ is constant.

Create pairs of sorted segments of integers. We will use table f to merge every pair of sorted segments to produce new sorted segments of double the length (see the next paragraph for a detailed description). Repeating this process $O(\log((n \log b)/w) + 2)$ times will allow us to permute all of the integers into nondecreasing order (the "+2" prevents subconstant logarithm values). Assume without loss of generality that all segments may be partitioned evenly and paired evenly.

Algorithm 3 Merging two sorted bit packed segments of integers.

Given: sorted input segments of integers M_0, M_1
Initialize array N to be empty
Partition M_0 and M_1 into contiguous segments of $\epsilon w/(2 \log b)$ integers. Label these segments $M_0^{(0)}, M_0^{(1)}, \ldots M_0^{(l_0-1)}$ and $M_1^{(0)}, M_1^{(1)}, \ldots M_1^{(l_1-1)}$, respectively.
$q_0 \leftarrow 0$; $q_1 \leftarrow 0$; $p_0 \leftarrow 0$; $p_1 \leftarrow 0$
while $q_0 < l_0$ and $q_1 < l_1$ **do**
    Append $f[M_0^{(q_0)}, p_0, M_1^{(q_1)}, p_1]$ to N
    $temp \leftarrow f^{(1)}[M_0^{(q_0)}, p_0, M_1^{(q_1)}, p_1]$
    if $f^{(0)}[M_0^{(q_0)}, p_0, M_1^{(q_1)}, p_1] = 0$ **then**
        $q_1 \leftarrow temp$; $q_0 \leftarrow q_0 + 1$; $p_0 \leftarrow 0$
    else
        $q_0 \leftarrow temp$; $q_1 \leftarrow q_1 + 1$; $p_1 \leftarrow 0$
    end if
end while
Only one of M_0 or M_1 has had all of its integers appended to N. For the input segment that hasn't, append all of its remaining integers to N in order.
Return N

Algorithm 3 describes how we use table f to merge two sorted segments. The main idea is to start with the lowest portions of each segment (each portion of size $\epsilon w/(2 \log b)$ integers), merge them using the table, and then proceed to the next portions as appropriate. Every time we merge two portions using f, the last value in one portion (call this portion A) will be smaller than some number of values from the other portion B (equal values can be ordered lexicographically). We must retain these extra values from B and compare them to A's succeeding portion to determine exactly where they fall in the sorted ordering. This process is orchestrated using the output specifiers from each application. The algorithm is fast because, at the end of each "while" loop iteration, we always read another portion of one of the input segments. For a single merge round from the previous paragraph, we will read a total of $O(\lceil (n \log b)/w \rceil)$ portions of size $\epsilon w/(2 \log b)$ integers. Each portion is only read a constant number of times and can be processed in $O(1)$ time. Therefore, the time complexity for a single merge round from the previous paragraph is $O(\lceil (n \log b)/w \rceil)$.

The overall time complexity to make the initial sorted segments and then merge them is $O(\lceil (n \log b)/w \rceil \log((n \log b)/w + 2) + 2^{\epsilon w + O(\log w)})$. We keep the ceiling function because it becomes important when working with several separate invocations of Packed-Integer-Sort that each operate on very few integers. □

Remark. $O(\log b)$ extra bits can be appended to each value without affecting the time complexity by simply appending them as the least significant digits of each value and sorting with them attached.

5 Fast 3Sum-Indexing

Recall the definition of 3Sum-Indexing:

> Given are two lists of values X and Y, each of size n, and a certain amount of preprocessing time. After preprocessing is finished, one or more input query values z are given sequentially; for each, the goal is to determine whether there exists $(x, y) \in X \times Y$ such that $x + y + z = 0$.

Reminder. ***Theorem 1:*** 3Sum-Indexing *has an algorithm with* $\min\{O((n^2 \log^{5/3}(m \log n)/\log^{2/3} n) \log^{4/3} \log n), O((n^2 \log^3(m \log n)/\log n) \log\log n)\}$ *preprocessing time and* $O(n/m)$ *query time for any desired* $m \leq n^{O(1/\log^{1/2} \log n)}$.

Towards proving Theorem 1, we will present and analyze our main 3Sum-Indexing algorithm. Note that we can assume $m = \omega(1)$; if not, the folklore algorithm from Section 1.1 will give the desired preprocessing and query time. This algorithm will prove the first time bound; we show that, with $O((n^2 \log^{5/3}(m \log n) \log^{4/3} \log n)/\log^{2/3} n)$ preprocessing time and a word size $w = \Theta(\log n)$, this algorithm can achieve $O(n/m)$ query time for all $m \leq n^{O(1/\log^{1/2} \log n)}$. Later, we will provide another algorithm to prove the second time bound. These two algorithms may be executed in parallel during preprocessing, with preprocessing ending once either of them terminates; this results in the claimed preprocessing time.

We will give our algorithm for 3Sum-Indexing as a chain of reductions with each step occupying its own subsection. All reduction steps implicitly involve passing the sorted list of real X values, the sorted list of real Y values, and parameter w (word size) as arguments. Restrictions on parameters are inherited between subsections. During our analysis, we will simply assume that $\Omega(\log n) \leq w < n^{o(1)}$; we only assign w the intended value at the very end. As explained before, we assume that, whenever required, a constant fraction of space is allotted as padding between values in memory.

Preprocessing. For the preprocessing stage, we will sort large contiguous blocks in $X + Y$. Formally, this stage proceeds by solving a single instance of **Block-Sort**, which is defined as follows:

> Given sorted lists X and Y of n real numbers each, divide them into contiguous segments $X_0, X_1, \ldots X_{n/d-1}$ and $Y_0, Y_1, \ldots Y_{n/d-1}$ of size d values each and then report the sums from each Cartesian sum in the multiset $\{X_{i_b} + Y_{j_b} : i_b \in [n/d] \text{ and } j_b \in [n/d]\}$ arranged into nondecreasing order. This ordering may be represented using pointers instead of the actual real values.

The time complexity of this problem is $T_{BS}(d, n)$. Note that Block-Sort can run in subquadratic time if we pack several pointers in each word (we will explain how to do this in the next reduction step).

Evaluating Queries. Once the blocks are sorted, evaluating queries is a simple matter. Assuming that we have already preprocessed $X + Y$ by solving an instance of Block-Sort, let $T_{Q3SUM}(d, n)$ be the time required to determine whether a query z value participates in a 3Sum solution in $X + Y$ (that is, whether there exists $(x, y) \in X \times Y$ such that $x + y + z = 0$).

Lemma 5.1. $T_{Q3SUM}(d, n) = O((n \log n)/d)$ *for* $d \leq n$ *where* d *is a function of* n *and* w

Proof. There is a simple witness algorithm. We already solved an instance of Block-Sort on $X + Y$, which produced sorted contiguous square blocks of side length d values. By Lemma 2.1,

this creates $O(n/d)$ long sorted diagonals. For the desired query value z, binary search in each long sorted diagonal for $-z$. If such a value is found, we have found a solution for the 3SUM-INDEXING query; if not, no solution exists. The time to conduct all binary searches is $O((n \log n)/d)$. Note that this procedure works with the same time complexity even if we are operating on sorted bit packed lists of indices instead of lists of values as long as the time to access an entry is $O(1)$. □

5.1 Sorting Blocks

Our goal now is to reduce the problem of sorting blocks to the problem of sorting small components of each block. Specifically, we reduce to problem **SORT-BLOCK-BUCKETS**, which is defined as follows:

> We are given n^2/q bit packed local index lists $\langle (x_0, y_0), (x_1, y_1), \ldots (x_{q-1}, y_{q-1}) \rangle$ of q ***local index pairs*** in $[d] \times [d]$, each list with an associated pair $(i_b, j_b) : i_b, j_b \in [n/d]$ such that the sequence of values $\langle (X[x_0 + i_b d] + Y[y_0 + j_b d]), (X[x_1 + i_b d] + Y[y_1 + j_b d]), \ldots (X[x_{q-1} + i_b d] + Y[y_{q-1} + j_b d]) \rangle$ is the concatenation of two nondecreasing sequences (with each sequence explicitly marked). We want to permute each index list such that, when its local index pairs are read sequentially from memory in their new order, $\langle (X[x_0 + i_b d] + Y[y_0 + j_b d]), (X[x_1 + i_b d] + Y[y_1 + j_b d]), \ldots (X[x_{q-1} + i_b d] + Y[y_{q-1} + j_b d]) \rangle$ forms a single nondecreasing sequence.

The time complexity of this problem is $T_{SBB}(q, d, n)$. This sorting procedure should be applied to all lists, and different lists should be outputted in the same order that they were inputted. The (i_b, j_b) pair appended to each list is called the ***global block index*** or ***block identifier***. Together, the list of index pairs and block identifier form a ***block bucket***; the reason for this name will become clear in the reduction itself. During input/output, each block bucket occupies $O(\lceil (q \log d + \log(n/d))/w \rceil)$ words because we use bit packing but store separate block buckets in separate words.

The block buckets are stored in ***data structure*** $\mathbf{C}$ as follows:

> Each block bucket is indexed as $\{C_i^{(i_b, j_b)} : i \in [d^2/q] \text{ and } i_b, j_b \in [n/d]\}$ where $C_i^{(i_b, j_b)}$ is the i^{th} block bucket in the block indicated by (i_b, j_b). $C_i^{(i_b, j_b)}[a]$ represents the a^{th} local index pair in that block bucket's list. Block buckets are stored in lexicographical order according to (i_b value, j_b value). A few bits of information are appended to each block bucket to indicate the boundary between its two nondecreasing sequences.

With these definitions, we can now prove the following lemma.

Lemma 5.2. *$T_{BS}(d, n) = O(\log d((n^2 \log d)/w + T_{SBB}(q, d, n)))$ for $q \leq d \leq n$ and where $q := w \log w$*

Proof. We describe an algorithm to reduce from BLOCK-SORT to SORT-BLOCK-BUCKETS. Independently sort lists X and Y (of n real numbers each) into nondecreasing order using a fast comparison sort in time $O(n \log n)$ (we omit this from the final running time because $w < n^{o(1)}$). Divide each the lists into contiguous segments $X_0, X_1, \ldots X_{n/d-1}$ and $Y_0, Y_1, \ldots Y_{n/d-1}$ of size d values each. Divide each block (Cartesian sum) in the multiset $\{X_{i_b} + Y_{j_b} : i_b, j_b \in [n/d]\}$ into d rectangular strips of height 1 and width d. We observe:

1. Each strip is already sorted in nondecreasing order from left to right.
2. The sorted ordering of each strip may be produced by creating the bit packed list of local index pairs in $[d] \times [d]$ for the corresponding $y \in [d]$. This would look like $\langle (0, y), (1, y), \ldots (d-1, y) \rangle$.

We produce the sorted ordering of each strip using these observations and then append a single coordinate in $[n/d] \times [n/d]$ that encodes the (i_b, j_b) index of the block the strip came from (its block identifier). The indexes are defined such that the bottom left sum in this block is $(X[i_b d + 0] + X[j_b d + 0])$. Strips are constructed a constant fraction of a word at a time (this doesn't require truth tables). All strips from a block are grouped together in memory, and different blocks are stored in lexicographical order according to (X global block index, Y global block index). This entire process requires $O((n^2 \log d)/w)$ time.

For each pair of adjacent strips from the same block, apply Select-Intervals to mark the local indexes of every q^{th} smallest sum between the two strips. We fix

$$q := w \log w$$

(this choice isn't too important so long as q is on the order of poly(w) and is sufficiently large). Note that Select-Intervals also partitions the strip pairs into sections (block buckets) corresponding to the local indexes of the sums between every q^{th} smallest value. Combine the two sorted sections corresponding to each block bucket by simply copying them into a new array and marking the beginning and end of the 2 constituent sorted sections. Also copy the corresponding (i_b, j_b) global block index and append it to the list in each block bucket. Store each in data structure C in a manner consistent with the given definitions. We add the following specifications:

- All block buckets from the same pair of strips are stored directly adjacent in memory.
- Later block buckets dominate previous block buckets from the same pair of strips.
- Different strip pairs from the same block have their block buckets stored in memory such that all block buckets from the strip pair with the larger Y position come after all block buckets from the strip pair with the smaller Y position.

Across all strip pairs in all blocks, the Select-Intervals algorithm takes $O((n^2 \log q)/q)$ time. Each block bucket requires $O(\lceil (q \log d)/w \rceil)$ to copy and there are $O(n^2/q)$ block buckets, meaning that the total time complexity is $O((n^2 \log d)/w)$ because $q \log d \geq \Omega(w)$ (this is implied because $q := w \log w$).

All block buckets are then sorted by Sort-Block-Buckets. This process replaces each block bucket's local index pairs with a permutation that corresponds to the sorted sums. Because the block buckets for each strip pair were already grouped in dominating order (according to the specifications mentioned), sorting the block buckets ends up sorting the entirety of each strip pair.

The above procedure is now applied again with the newly sorted strip pairs as the input strips (each now of height 2); after this, we repeat with strips of height 4, then 8, and so on. After a total of $O(\log d)$ rounds, the entirety of each block is sorted. If d is not a power of 2, simply skip some of the strip merges in each round as needed (the time complexity won't increase and the algorithm can easily operate on strips of unequal length). The overall time complexity is $O(\log d((n^2 \log d)/w + T_{SBB}(q, d, n)))$. □

5.2 Sorting Block Buckets Using Truth Tables and Sorting Networks

We now describe how to permute block buckets into sorted order. Data structure C, which contains all input lists for Sort-Block-Buckets, is inherited from the previous section. Recall that all lists are bit packed.

Let **Eval-BBQ** be the problem:

Given n^2/t bit packed quadruplet lists (***query lists***) $\langle (x_0, y_0, x'_0, y'_0), (x_1, y_1, x'_1, y'_1), \ldots (x_{t/2-1}, y_{t/2-1}, x'_{t/2-1}, y'_{t/2-1})\rangle$ of local index quadruples in $[d] \times [d] \times [d] \times [d]$, each list with an appended global block index (i_b, j_b), determine whether $(X[x_k + i_b d] + Y[y_k + j_b d] > X[x'_k + i_b d] + Y[y'_k + j_b d])$ for all $k \in [t/2]$ and store the answers as ***Boolean result lists*** of $t/2$ values in $\{TRUE, FALSE\}$ directly adjacent to their corresponding query lists.

Query lists are expected to be grouped together by block, and each block is presented in lexicographical order according to (X global block index, Y global block index). In our implementation, all coordinate combinations from a block will be assigned to exactly one query list, so there will be d^2/t query lists for each block. The time complexity to solve an instance of this problem is $T_{EBBQ}(t, d, n)$.

We store query lists and the output Boolean lists in ***data structure*** **D** as follows:

$D^{(i_b, j_b)}$ (for $k \in [d^2/t]$ and $i_b, j_b \in [n/d]$) stores the k^{th} query list of $t/2$ quadruples in the $(i_b, j_b)^{\text{th}}$ block along with the appended pair (i_b, j_b), and D is the set of all query lists. In memory, query lists are grouped in lexicographical order according to (i_b value, j_b value, k value).

The Bitonic Sorting Network. Sort-Block-Buckets is evaluated by using the bitonic sorting network [Bat68]. A bitonic sequence is either:

- The concatenation of a monotone nondecreasing and a monotone nonincreasing sequence, or
- A rotation of another bitonic sequence.

One round of the sorting network is as follows. Given an input bitonic sequence of length q, the bitonic sorting network compares all pairs of values separated by a distance of $q/2$, swapping pairs as needed so that the greater value comes later in the sequence. All comparisons and swaps can be executed in parallel. The resulting list contains two contiguous bitonic sequences of length $q/2$ values each. Every value in the last bitonic subsequence dominates every value in the first bitonic subsequence. This procedure is repeated recursively on the smaller bitonic sequences until every bitonic sequence is of length one, yielding a sorted list. When applied to block buckets, this procedure is modified. Instead of executing the comparisons directly, it creates query lists encoding the comparisons which are passed to Eval-BBQ for evaluation and then uses the Boolean result lists to determine which values to swap.

This method does require significantly more comparisons than would be required by a fast comparison sort, seemingly making it inefficient. However, we can execute comparisons in a bit-packed manner, unlike with a normal comparison sort. It turns out that only $\log^{O(1)} \log n$ times more comparisons will be required, and we can execute $\log^{\Omega(1)} n$ times more comparisons at once via bit packing, leading to an overall improvement in time complexity.

The Reduction. We now reduce Sort-Block-Buckets to multiple instances of Eval-BBQ using the bitonic sorting network.

Lemma 5.3. *$T_{SBB}(q, d, n) = O(\log q(n^2/t + T_{EBBQ}(t, d, n)) + 2^{\epsilon w})$ for $t \log d \leq O(w)$ and $t \leq q$ where t is a function of q, d, n, and w*

Proof. We give a witness algorithm supporting the lemma. Recall that each $C_k^{(i_b, j_b)}$ is a block bucket, and each block bucket consists of a sequence of pointers to two nondecreasing sequences of sums. Reverse the order of the pointers in one sorted subsequence in each $C_k^{(i_b, j_b)}$ in total

time $O((n^2 \log d)/w + 2^{\epsilon w})$ (because $q \geq w/\log d$) by constructing and using truth tables on small constant fractions of a word at a time. This converts every $C_k^{(i_b,j_b)}$ into a list of local indexes whose corresponding sums form a bitonic sequence with length q.

We use the bitonic sorting network to begin sorting the block buckets. Since each block bucket initially consists of a single large bitonic sequence, elements of each block bucket separated by a distance (***gap size***) of $g \leftarrow q/2$ index pairs in memory will be compared first. Empty data structure D and apply Algorithm 4 to C and D using the current value of g and parameters q, t, d, and n.

Algorithm 4 Creating the queries for all block buckets during a single round.

Given: parameters g, q, t, d, n and data structures C and D
for $i_b \leftarrow 0$ to $n/d - 1$ **do**
  for $j_b \leftarrow 0$ to $n/d - 1$ **do**
    for $k \leftarrow 0$ to d^2/q **do**
      for IDX $\leftarrow 0$ to $q/(2g) - 1$ **do**
        Conceptually divide the lower half of the IDX$^{\text{th}}$ bitonic sequence in $C_k^{(i_b,j_b)}$ into contiguous segments of size $t/2$ local index pairs and label the segments $S_0, S_1, \ldots S_{2g/t-1}$.
        Conceptually divide the upper half of the IDX$^{\text{th}}$ bitonic sequence in $C_k^{(i_b,j_b)}$ into contiguous segments in the same way to produce segments $T_0, T_1, T_{2g/t-1}$.
        for $l \leftarrow 0$ to $2g/t - 1$ **do**
          Create the query list $\langle (S_l[0], T_l[0]), (S_l[1], T_l[1]), \ldots (S_l[t/2], T_l[t/2]) \rangle$ and concatenate it to D (along with the (i_b, j_b) block identifier) using truth tables. This entry will be indexed as $D_{kq/t+2\text{IDX}g/t+l}^{(i_b,j_b)}$.
        end for
      end for
    end for
  end for
end for

A description of the Algorithm 4 is as follows. The outer 3 for loops just iterate over the block buckets in lexicographical order. The IDX for loop iterates over every bitonic sequence that is a member of the given block bucket. These bitonic sequences are divided into segments of size $t/2$ local index pairs such that, for all $l \in [2g/t]$, each sum pointed to by a local index pair in segment S_l will be compared against the sum pointed to by the local index pair in the same position in segment T_l. Dividing the block buckets is conceptual and requires no time. The innermost for loop just handles creating the query lists. This is done by using truth tables to operate on small constant fractions of each T_l and S_l at a time, building each query list in $O(1)$ time, or $O(n^2/t)$ over all query lists in all blocks. The truth tables take only $O(2^{\epsilon w})$ time to make and $O(1)$ to use because $t \log d \leq O(w)$. Note that the truth tables only need to be created once, not at every invocation of this algorithm. Each query list, when formed as described, is technically of the form

$$\langle ((x_0, y_0), (x'_0, y'_0)), ((x_1, y_1), (x'_1, y'_1)), \ldots ((x_{t/2-1}, y_{t/2-1}), (x'_{t/2-1}, y'_{t/2-1})) \rangle$$

(along with the global block identifier). The extra parentheses are removed conceptually, requiring no time and creating a list of the form

$$\langle (x_0, y_0, x'_0, y'_0), (x_1, y_1, x'_1, y'_1), \ldots (x_{t/2-1}, y_{t/2-1}, x'_{t/2-1}, y'_{t/2-1}) \rangle.$$

The indexing scheme

$$D^{(i_b,j_b)}_{kq/t+2\text{IDX}g/t+l}$$

ensures that the query lists are stored densely and in the order that they are created. Note, however, that each query list is assigned to its own set of word(s); in other words, we don't worry about the small portion of unutilized space in the last word of every query list.

After this, invoke EVAL-BBQ on data structure D, which stores the Boolean result lists back in data structure D such that the results are directly adjacent to their corresponding query lists. Index every Boolean result list in D and use these to execute the desired swaps in every block bucket of C. Every TRUE value means that the corresponding two local index pairs need to be swapped. Swaps can be executed quickly (in $O(1)$ time per query list because $t \log d \leq O(w)$, or in $O(q/t)$ time per block bucket) by again using truth tables on small constant fractions of a word. When this is done, every block bucket in C now contains twice as many bitonic sequences, and every bitonic sequence in a block bucket dominates the previous. We now assign the gap size to $g \leftarrow g/2$.

We repeatedly invoke Algorithm 4 as described, with the gap sizes constantly deacreasing by a factor of 2 each time. Stop when g becomes less than $t/2$. There are $O(\log(q/t))$ rounds, and each round takes $O(n^2/t + T_{EBBQ}(t, d, n))$ time assuming that the truth tables were already created.

A slightly different method is applied to finish sorting. Because of the constraints on $t \log d$, every bitonic sequence now fits within a constant number of words. This means that the previous two paragraphs can be applied again, but, when it comes time to make query lists, multiple bitonic sequences might contribute to a single query list. The partitioning process might be messier than before as well; this occurs in the case when the bitonic sequences are larger than the number of index pairs a truth table can read but smaller than t index pairs. Handling these involves adjusting the segmentation process and truth tables a bit, but these modifications do not affect the big-O complexity. After $O(\log t)$ rounds, the bitonic sequences reach a size of one, meaning that all the block buckets are fully sorted. Each round requires the same amount of time as before. The total time is $O(\log(q/t)(n^2/t + T_{EBBQ}(t, d, n)) + \log t(n^2/t + T_{EBBQ}(t, d, n))) = O(\log q(n^2/t + T_{EBBQ}(t, d, n)))$. □

5.3 Creating Families of Queries

Inherit the set of query lists D from the previous sections. In this section, all query lists from D will be pigeonhole sorted [Bel58] into ***families***, and the query lists inside each family will be evaluated efficiently during later reductions.

Families and Provinces. Recall that each query list consists of $t/2$ local coordinate quadruples of the form (x_i, y_i, x'_i, y'_i) for $x_i, y_i, x'_i, y'_i \in [d]$ along with the global block index (i_b, j_b) for $i_b, j_b \in [n/d]$. Two query lists are sorted into the same family if and only if:

- They have identical $t/2$ local coordinate quadruples in identical order (but different global block indexes).
- Their corresponding blocks are from the same ***province***.

Provinces are defined as follows. Conceptually, take the sorted lists X and Y of n real numbers each and divide Y into $n/(pd)$ contiguous segments of size pd values each for some new parameter p. The Cartesian sum of the entirety of X with an arbitrary segment from Y defines a rectangle p blocks high and n/d blocks wide. All of the blocks within this rectangle are part of the same

province, and all blocks outside of this rectangle are not part of this province. There are $n/(pd)$ total provinces and d^{2t} combinations for the entries in a query list of size $t/2$, so there are

$$f := nd^{2t-1}/p$$

total families.

New Definitions and the Reduction. We define ***data structure* E** as follows:

> E stores all of the query lists grouped together into families. $E_e^{(l,a)}$ encodes the e^{th} query list in the family that comes from the l^{th} province with a as the list of $t/2$ local index quadruples shared by all query lists in the family. $E^{(l,a)}$ is the set of all query lists for the family indicated by (l, a). In memory, every member $E_e^{(l,a)}$ of $E^{(l,a)}$ simply consists of the ordered triple (i_b, j_p, k) such that $D_k^{(i_b, j_p+lp)}$ addresses the original query list in D for $i_b \in [n/d]$, $j_p \in [p]$, and $k \in [d^2/t]$.

In other words, every $E_e^{(l,a)}$ only *indicates* a query list and does not actually store a list of queries. Given the correct province, each triple in E contains all the information needed to copy the query result lists back to their corresponding location in D in $O(1)$ time, even if each neighboring query result list came from very different locations. Note that triples in E are stored in lexicographical order according to (family identifier, i_b value, j_p value). The ordering of different families relative to each other can be arbitrary so long as it is consistent throughout each reduction step and easy to work with. For example, we could use lexicographical order according to (province l identifier, province a identifier). The ordering of k's can be disregarded because no two entries in the same family can come from the same block. Note that the length of each family $E^{(l,a)}$ may vary significantly. At the very beginning of each family in memory, the family's (l, a) pair is stored.

Let **Family-Queries** be the problem:

> Resolve all query lists indicated by the triples in data structure E (parameters t, d, p, and n should define the size and characteristics of E). Afterwards, store the triples back into E such that each has appended a Boolean result list of length $t/2$ bits corresponding to the results of the $t/2$ queries in the query list indicated by the respective triple and family. Families should still be present in the same order as before, but the order of triples within each family can be shuffled arbitrarily (or not shuffled).

$T_{FQ}(t, d, p, n)$ is the time complexity of Family-Queries when using parameters t, d, p, and n (we assume that the input is in data structure E).

We can now prove the following lemma.

Lemma 5.4. *$T_{EBBQ}(t, d, n) = O(n^2/t + f + T_{FQ}(t, d, p, n))$ for $p \leq n/d$ where $f := nd^{2t-1}/p$ and p is a function of t, d, n, and w*

Proof. We give a witness reduction. Initialize linked lists for each pigeonhole corresponding to each possible family that a query list from D could be sorted into. At the beginning of each linked list, store the (l, a) pair associated with the family. Scan through D and visit every $D_k^{(i_b, j_b)}$ in lexicographical order according to (i_b value, j_b value, k value). When each query list is visited, generate the corresponding triple $(i_b, j_p, k) := (i_b, j_b \bmod p, k)$ and copy this triple into the correct pigeonhole. By scanning D in this way, each pigeonhole of E is ordered lexicographically as desired. Once finished, the linked lists are scanned and concatenated with each other to produce E. D has n^2/t entries, there are f pigeonholes, and producing/transferring the triples for each pigeonhole requires $O(1)$. The time complexity is $O(n^2/t + f)$.

Resolve the queries in E by invoking FAMILY-QUERIES. Afterwards, scan E (which might now be in a slightly different order) and copy each list of Boolean results back to its corresponding location in D using the (i_b, j_p, k) triple and family (l, a) identifier in $O(1)$ time per query result list. Note that k is used to determine exactly which word in D the Boolean results should be copied to. This requires $O(n^2/t + T_{FQ}(t, d, p, n))$. □

5.4 Processing Families into Sparse Squares

Inherit the structure of data structure E from the previous section. We will now describe a reduction that groups family members into a regular pattern, making later steps faster.

A New Grouping Strategy. To produce ***Sparse squares***, we make groupings of v^2 triples from E (for a new parameter v) from the same family with a specific property. For every triple (i_b, j_p, k) in a sparse square, there are exactly $v-1$ other triples in the sparse square with the same i_b value and exactly $v-1$ other triples in the sparse square with the same j_p value. If the triples are arranged graphically based on the locations of their corresponding blocks and horizontally/vertically adjacent triples are connected with lines, the resulting figure will look like a rectangle divided into $(v-1)^2$ smaller rectangles (possibly of width/height zero) by $v-2$ vertical lines and $v-2$ horizontal lines. The v triples located along the bottom edge of the sparse square are referred to as the ***bottom edge***. The v triples located along the left edge of the sparse square are referred to as the ***left edge***. Sparse squares are stored using $O(v^2)$ words each. All query lists in a family are either sorted into a sparse square or processed directly.

Each sparse square is represented and stored as a contiguous ordered list

$$\langle i_b^{(0)}, i_b^{(1)}, \ldots i_b^{(v-1)}, j_p^{(0)}, j_p^{(1)}, \ldots j_p^{(v-1)}, k^{(0,0)}, \ldots k^{(0,v-1)}, k^{(1,0)}, \ldots k^{(v-1,v-1)} \rangle.$$

There is no need to bit pack these values as long as each $i_b^{(g)}$, $j_p^{(g)}$, and $k^{(g,g')}$ is stored in a constant number of words. The i_b values are the i_b values from the triples along the bottom edge in increasing order. The j_p values are the j_p values from the left edge in increasing order. Each $k^{(g,g')}$ value is simply the k value corresponding to the sparse square's triple whose $i_b = i_b^{(g)}$ and $j_p = j_p^{(g')}$. Together, these values encode all of the information needed to represent the original v^2 triples that are part of the sparse square.

New Definitions and the Algorithm. Define ***data structure*** **F** as follows:

> All sparse squares (not query lists themselves) are stored in data structure F. All sparse squares from the same family are stored contiguously in memory in arbitrary order, and different families are stored in the same order as in data structure E. At the beginning of each family's memory block, the corresponding (l, a) identifier is stored.

Recall that, in the (l, a) identifier, l indicates the family's province and a indicates the family's exact list of $t/2$ local index quadruples.

Let **SQUARE-QUERIES** be the problem:

> Resolve all query lists indicated by the information in each sparse in data structure F and the associated family (l, a) pair. Specifically, append each ordered Boolean result list to its corresponding $k^{(g,g')}$ value in each sparse square without changing the position of sparse squares relative to each other. F's size and attributes are defined by the parameters t, d, p, v, n, and a new parameter u (which is the total number of sparse squares in F).

Let $T_{SQ}(t, d, p, u, v, n)$ be the time complexity of this problem when using parameters t, d, p, v, u, and n (we assume that the input is in data structure F). This time complexity is nondecreasing as the number of input sparse squares increases because, when evaluating SQUARE-QUERIES, additional sparse squares don't give us additional information about the other sparse squares already present. This is since, according to the problem definition, sparse squares can contain arbitrary information.

We can now give the following lemma.

Lemma 5.5. *$T_{FQ}(t, d, p, n) = O(n^2/t + fp^v tv^2 + fntv/d + T_{SQ}(t, d, p, n^2/(tv^2), v, n))$ for $v \leq p$ where $f := nd^{2t-1}/p$ and where v is a function of t, d, p, n, and w*

Proof. The proof consists of a witness reduction. Initialize new data structures E' and F to empty. As before, $f := nd^{2t-1}/p$ is the number of possible families. We will assume that the input to this reduction algorithm is stored in data structure E.

Step 1: Pigeonhole Sorting. For a single family, create a set of ***left edge pigeonholes*** corresponding to all possible combinations of v consecutive j_p values such that the j_p values are in increasing order (repeat j_p values are not considered). Left edge pigeonholes are linked lists with an associated identifier that looks like

$$\langle j_p^{(0)}, j_p^{(1)}, \ldots j_p^{(v-1)} \rangle.$$

Since every j_p is a value in $[p]$, there are

$$p!/((p-v)!v!) \leq p^v$$

left edge pigeonholes for this single family.

Looking at data structure E, we note that all triples from this family sharing the same i_b value will be grouped contiguously in memory (as a result of the lexicographical ordering). For each i_b value in order, divide its associated triples into contiguous sections of size v triples each (and at most one section of less than v triples if necessary). This is accomplished by just scanning E. For all sections of exactly v triples, create the ordered list

$$\langle i_b, k^{(0)}, k^{(1)}, \ldots k^{(v-1)} \rangle.$$

i_b is the same between all v triples, and $k^{(g)}$ should come from the source triple (out of the v chosen triples) with the g^{th} smallest j_p value (the values are implicitly sorted already). For each newly created ordered list into the left edge pigeonhole corresponding to the j_p values from its triples. This takes $O(v)$ per ordered list because of the size assumptions on d and p.

For all ***underfilled*** sections of less than v triples, attach the (l, a) family identifier to each triple to produce a pentuple (i_b, j_p, k, l, a) and then append all pentuples to the holder data structure E' in arbitrary order. There is at most one underfilled section per i_b value per family, so there are at most $n(v-1)/d$ pentuples per family. Pentuples require $O(1)$ time each to create and process.

Repeat the pigeonhole sorting procedure from the previous three paragraphs across all families, which processes every triple in E. Make sure to scan the families in the same order that they appear in E. There are at most $n^2/(tv)$ ordered lists $\langle i_b, k^{(0)}, k^{(1)}, \ldots k^{(v-1)} \rangle$ and less than fnv/d pentuples. The total time complexity to pigeonhole sort ordered lists and generate/store pentuples is thus $O(n^2/t + fp^v + fnv/d)$ across all families.

Step 2: More Pigeonhole Sorting. For a single family, append the associated (l, a) identifier to F. Then, scan through that family's left edge pigeonholes one at a time. Divide each left edge pigeonhole into contiguous sections of v ordered lists each (recall that a single list looks like $\langle i_b, k^{(0)}, k^{(1)}, \ldots k^{(v-1)} \rangle$) and at most one shorter section (if necessary) by scanning the left

edge pigeonhole. Each full size section, in conjunction with the left edge pigeonhole identifier $\langle j_p^{(0)}, \ldots j_p^{(v-1)} \rangle$, is processed in $O(v^2)$ time to produce a sparse square data structure. Append each of these to F. Each sparse square occupies $O(v^2)$ words and takes $O(v^2)$ time to produce because of the constraints on parameters. Note that this procedure naturally groups sparse squares into families because the original query list representations were grouped into families in E.

For all underfilled sections of less than v ordered lists, use each ordered list $\langle i_b, k^{(0)}, k^{(1)}, \ldots k^{(v-1)} \rangle$ from that section in conjunction with the associated left edge pigeonhole identifier $\langle j_p^{(0)}, \ldots j_p^{(v-1)} \rangle$ and family identifier (l, a) to reproduce the corresponding triples from E. Then, produce pentuples from these triples as before and append them to E' in arbitrary order. This whole process requires $O(1)$ time per pentuple. There is at most one underfilled section per left edge pigeonhole per family.

Apply the procedure from the previous two paragraphs to all families, which processes all data in every left edge pigeonhole. There are $O(fp^v)$ left edge pigeonholes across all families, so there are $O(fp^v v^2)$ possible pentuples for this step. The time to process all sparse squares and pentuples from this step is $O(n^2/t + fp^v v^2)$.

Step 3: Evaluating Sparse Squares and Pentuples. Evaluate the Boolean results for the sparse squares in F using Square-Queries (there are at most $n^2/(tv^2)$ sparse squares). Evaluate all Boolean results for the pentuples in E' directly by reproducing each pentuple's query list in $O(1)$ time, manually comparing each query quadruple in each query list in $O(1)$ time per query quadruple, and associating the each Boolean result list with its corresponding pentuple. This takes $O(t)$ time per pentuple, and there are at most $O(fp^v v^2 + fnv/d)$ pentuples, so the total time is $O(fp^v tv^2 + fntv/d)$. Now, E' and F contain the Boolean result list associated with each triple in each family (although E' still represents them as Boolean result lists connected to pentuples). Create pigeonholes for every family. Copy every (triple, Boolean result list) pair represented in F into the corresponding pigeonhole (they are already grouped into families, so this is easy). Then, scan E' and convert all (pentuple, Boolean result list) pairs into (triple, Boolean result list) pairs and copy each into its respective pigeonhole based on the family indicated by the original pentuple. Concatenate all pigeonholes by scanning them and store the result in E such that families appear in the same order as before. The time complexity is $O(n^2/t + fp^v tv^2 + fntv/d + T_{SQ}(t, d, p, n^2/(tv^2), v, n))$. □

Remark. Both upper bounds for the number of pentuples become unnecessarily large if they exceed n^2/t, because this is the total number of query lists. However, the algorithm as a whole becomes slower than the naive algorithms if this occurs, so we don't account for this.

5.5 Processing Sparse Square Queries

We now describe a procedure to get the results for each query list encoded by the sparse squares in data structure F. One of our tools is Fredman's trick [Fre76], which is the following observation:

$$X[x] + Y[y] > X[x'] + Y[y'] \leftrightarrow X[x] - X[x'] > Y[y'] - Y[y].$$

Applying Fredman's Trick and Taking Advantage of Province Shape. Recall that all individual queries are quadruples of local indexes (x_k, y_k, x'_k, y'_k) from inside the same block. Also recall that each query is asking whether

$$X[x_k + i_b d] + Y[y_k + lpd + j_p d] > X[x'_k + i_b d] + Y[y'_k + lpd + j_p d]$$

is true for block identifier (i_b, j_p), province identifier (l, a), and fixed $k \in [t/2]$. If these query quadruples (x_k, y_k, x'_k, y'_k) are converted into an X index pair (x_k, x'_k) and a Y index pair (y_k, y'_k) and evaluated using Fredman's trick, there are only nd possible combinations for differences of the

form $X[x_k + i_b d] - X[x'_k + i_b d]$ across all queries in a single province. Interestingly, there are only pd^2 possible combinations for differences of the form $Y[y'_k + lpd + j_p d] - Y[y_k + lpd + j_p d]$ across all queries in a single province because l is fixed for all differences in the province and j_p is only a value in $[p]$. In the algorithm, the actual real value of each of these differences is evaluated directly and stored (this process is repeated across all provinces, not across all families). Then, the real differences for each province are sorted into nondecreasing order (which takes negligible time across all provinces) and assigned nondecreasing rankings. These are the ***province difference rankings***. Comparing the values of an X difference ranking and a Y difference ranking amounts to evaluating the comparison $X[x] + Y[y] > X[x'] + Y[y']$. Because comparisons only ever occur between X differences and Y differences (and not X vs X), it is possible to encode the necessary information in all province difference rankings using an integer in $[O(pd^2)]$. Assuming $p > d$, every ranking only occupies $O(\log p)$ bits.

Shortening Rankings Within Sparse Squares. Conceptually, if a sparse square is populated with its corresponding query lists, and every list is adjusted as above, an arbitrary query list from the sparse square will have the exact same $t/2$ real X differences as $v - 1$ other member query lists and the exact same $t/2$ real Y differences as $v - 1$ other member query lists (which are also different from the lists with the same real X differences). This means that only v lists of $t/2$ X province difference rankings and v lists of $t/2$ Y province difference rankings must be copied to each sparse square to resolve all queries. In other words, all v^2 query lists in the sparse square are just comparing different combinations of one of the v X difference ranking lists and one of the v Y difference ranking lists. Importantly, we also note that the *order* of differences in each list is the same, so we won't have to do any shuffling.

Assume that we have copied the $2v$ lists of $t/2$ province difference rankings each into a given sparse square. Each list may occupy more than a constant number of words; as such, the difference rankings need to be shortened before use. We rank them once again, this time only against the other province difference rankings from the given specific sparse square, to produce functionally equivalent difference rankings (***sparse square difference rankings***) in $[vt]$ using Packed-Integer-Sort. It is okay to use packed integer sorting here because the number of difference rankings for a sparse square is *significantly* smaller than the number of rankings we would originally have to sort using our baseline methods (from Section 3.1), which is $d^{\Theta(1)}$. These shorter rankings are compressed so that the ranking lists each fit in a constant number of words. We then populate all the query lists inside the given sparse square by copying in the appropriate shortened X and Y difference ranking lists, which takes $O(1)$ time per list. The query lists in the given sparse square are then evaluated using truth tables comparing the sparse square difference rankings, which finally produces the required Boolean result lists. We repeat this procedure across all sparse squares.

A More Rigorous Version. We now formalize the above procedure.

Lemma 5.6. $T_{SQ}(t, d, p, u, v, n) = O((n^2 \log n)/p + fnt/d + uv\lceil(t \log p + \log n)/w\rceil + u\lceil(vt \log p)/w\rceil \log(vt) + uv^2 + 2^{\epsilon w})$ *where* $f := nd^{2t-1}/p$ *for* $\log p \geq \Omega(\log(tv))$ *assuming* $p \geq d$

Proof. Our witness algorithm is as follows.

Step 1: Making Difference Rankings. In a single province (this is completely disregarding families), create the set of pentuples

$$\{(l, i, i', i_b, X[i + i_b d] - X[i' + i_b d]) : i_b \in [n/d] \text{ and } i, i' \in [d]\} \cup$$

$$\{(l, j', j, j_p, Y[j' + j_p + lp] - Y[j + j_p + lp]) : j', j \in [d] \text{ and } j_b \in [p]\}$$

for the given province's l value (each pentuple should also implicitly indicate whether it corresponds to an X difference or a Y difference). Directly compute the real differences by accessing and subtracting the real numbers at the appropriate locations in X and Y. Sort the pentuples into nondecreasing order based on their real differences using a fast comparison sort. This takes $O(nd \log(nd))$ for the single province. Initialize a counter r_p to zero. Scan the pentuples, starting from the smallest, and replace the real value in the pentuple with the integer value of r_p. Increment r_p if and only if transitioning from reading a quadruple of X values to a quadruple of Y values or vice versa. The maximum value of the counter is $\min\{O(pd^2), O(nd)\}$, meaning it need only occupy $O(\log(pd))$ bits. The r_p value stored in each pentuple is the province difference ranking. Don't bother bit packing right now as long as each pentuple occupies a constant number of words. Pigeonhole sort every X difference ranking into array $X\_Province\_Rankings$ indexed by its quadruple (l, i, i', i_b). Pigeonhole sort every Y difference ranking into $Y\_Province\_Rankings$ indexed by the its quadruple (l, j, j', j_p). Pigeonholes work here because it is predetermined exactly which index combinations will be computed, and the indexing schemes (l, i, i', i_b) and (l, j, j', j_p) are dense enough that the arrays won't have gaps that exceed a constant factor of the amount of information stored. The total time to pigeonhole sort is $O(nd)$. Repeat this sorting and pigeonholing process on all provinces and store all difference rankings in either $X\_Province\_Rankings$ or $Y\_Province\_Rankings$. The total time per province is $O(nd \log n)$, and there are $O(n/(pd))$ provinces, so the overall time is $O((n^2 \log n)/p)$.

Step 2: Assigning Rankings to Representatives in Each Sparse Square. Consider a single family in F. Initialize pigeonholes for all possible i_b values. For each pigeonhole, produce a single ordered X province difference ranking list of $t/2$ rankings based on the family's (l, a) identifier and the pigeonhole's i_b value. These province difference rankings should align with the X index pairs in each quadruple in list a. This requires $O(t)$ per pigeonhole or $O(nt/d)$ per family by simply indexing $X\_Province\_Rankings$. An analogous process is applied to make and initialize pigeonholes for the j_p values, but it is faster because there are fewer pigeonholes. When we produce these ranking lists, they should be bit packed in memory; recall that each ranking is only $O(\log p)$ bits long, so $\Omega(w/\log p)$ may be packed into a single word. Each ranking list thus occupies $O(\lceil (t \log p)/w \rceil)$ words (the ceiling comes from the fact that each is stored in its own set of words).

For every sparse square in this single family, append to each $i_b^{(g)} : g \in [v]$ and each $j_p^{(g)} : g \in [v]$ the absolute RAM address h where that value is located in F (this is an integer $O(\log n)$ bits long) so that each sparse square is now stored as

$$\langle (i_b^{(0)}, h^{(0)}), \ldots (i_b^{(v-1)}, h^{(v-1)}), (j_p^{(0)}, h'^{(0)}), \ldots (j_p^{(v-1)}, h'^{(v-1)}), k^{(0,0)}, \ldots k^{(0,v-1)}, k^{(1,0)}, \ldots k^{(v-1,v-1)} \rangle$$

if read sequentially from memory. Then, consider just the ordered pairs corresponding to X global block indexes (they are of the form $(i_b^{(g)}, h^{(g)})$). Pigeonhole sort every such ordered pair from all sparse squares in the given family based on the i_b value (each takes $O(1)$ to copy) using the pigeonholes initialized earlier. Append the ordered province difference rankings for each pigeonhole to all ordered pairs $(i_b^{(g)}, h^{(g)})$ in the pigeonhole to produce ordered triplets. These ordered triplets are copied back to their respective locations in F using their h address. Because each ordered list of $t/2$ province difference rankings occupies $O(\lceil (t \log p)/w \rceil)$ words, each copy operation will require $O(\lceil (t \log p)/w + (\log n)/w \rceil)$ time and space. After copying back into the sparse square, delete $h^{(g)}$ from each triplet. An analogous procedure is applied to the ordered pairs corresponding to the Y differences, only with fewer pigeonholes, so the time taken is not greater.

Repeat the above two paragraphs for all families in F. The total time is $O(fnt/d + uv\lceil (t \log p +$

$\log n)/w\rceil$). All sparse squares are now stored as

$$\langle (i_b^{(0)}, a_r^{(0)}), (i_b^{(1)}, a_r^{(1)}), \ldots (i_b^{(v-1)}, a_r^{(v-1)}), (j_p^{(0)}, a_r'^{(0)}), (j_p^{(1)}, a_r'^{(1)}), \ldots (j_p^{(v-1)}, a_r'^{(v-1)}),$$

$$k^{(0,0)}, \ldots k^{(0,v-1)}, k^{(1,0)}, \ldots k^{(v-1,v-1)} \rangle$$

where each $a_r^{(g)}$ and $a_r'^{(g)}$ is an ordered list of $t/2$ province difference rankings, each list $O(t \log p)$ bits long.

Step 3: Making the Difference Rankings Shorter and Evaluating Query Lists. Apply the following paragraphs to all sparse squares individually. For each province difference ranking in each list of province difference rankings in the sparse square, append an identifier $h \in [tv]$ recording its position (relative to the other individual province difference rankings in its sparse square) to create a pair of the form $(a_r^{(g)}[g_i], h^{(g,g_i)})$ or $(a_r'^{(g)}[g_i'], h^{(g',g_i')})$ for $g, g' \in [v]$ and $g_i, g_i' \in [t/2]$. Note that $a_r^{(g)}[g_i]$ is the ${g_i}^{\text{th}}$ difference ranking in the r^{th} X difference ranking list, and $a_r'^{(g)}[g_i']$ is the ${g_i'}^{\text{th}}$ difference ranking in the ${r'}^{\text{th}}$ Y difference ranking list. Currently, the *identifiers* should be arranged in ascending order. Apply Packed-Integer-Sort from Lemma 4.2 to sort the ordered pairs into nondecreasing order based on the value of each province difference ranking. Note that we only need to construct the truth tables for Packed-Integer-Sort once, which just requires $O(2^{\epsilon w})$ time (for a slightly larger ϵ than the one in Section 4.2).

Next, replace the province difference rankings with ***sparse square difference rankings*** in $[vt]$. We do this by initializing a counter r_l to zero and then scanning over the sorted pairs and replacing each province difference ranking with r_l. r_l is incremented after every replacement. Apply Lemma 4.2 to sort the ordered pairs back into their original position based on each $h^{(g,g_i)}$ or $h^{(g',g_i')}$ value (these values also automatically encoded whether the ranking is for an X difference or a Y difference). Compress the lists of difference rankings, which now contain sparse square rankings instead of the province rankings, so that each list of $t/2$ rankings occupies $O(1)$ words. This can be done using truth tables applied to small constant fractions of each word. It is possible to fit all $t/2$ rankings inside $O(1)$ words because $t \log d \leq O(w)$ (and other restrictions) imply $t \log(vt) \leq O(w)$. This sorting process takes $O(\lceil (vt \log p)/w \rceil \log((vt \log p)/w + 2)) \leq O(\lceil (vt \log p)/w \rceil \log(vt))$ per sparse square. $O(2^{\epsilon w})$ is required to produce the truth tables, but this only occurs once for the entire 3Sum-Indexing preprocessing algorithm (they can be inherited over multiple calls to this step/algorithm). Copy the X and Y sparse square difference ranking lists and place them next to the respective $k^{(g,g')}$ values inside the sparse square. Finally, apply truth tables operating on small fractions of a word to evaluate the Boolean results of the queries associated with each $k^{(g,g')}$. This takes $O(1)$ per k value or $O(v^2)$ for the sparse square assuming the truth tables were already created. The time complexity for one sparse square is thus $O(\lceil (vt \log p)/w \rceil \log(vt) + v^2)$.

Multiplied over all sparse squares, the time complexity for this last step is $O(u\lceil (vt \log p)/w \rceil \log(vt) + uv^2 + 2^{\epsilon w})$. The sparse squares are in the same order as before applying the algorithms from this entire proof, so we don't need to bother rearranging them. □

5.6 Optimizing Parameters

Reminder. ***Theorem 1:*** 3Sum-Indexing *has an algorithm with* $\min\{O((n^2 \log^{5/3}(m \log n)/\log^{2/3} n) \log^{4/3} \log n), O((n^2 \log^3(m \log n)/\log n) \log \log n)\}$ *preprocessing time and* $O(n/m)$ *query time for any desired* $m \leq n^{O(1/\log^{1/2} \log n)}$.

We will prove the first time bound.

Proof. To summarize, we have the following due to Lemmas 5.1, 5.2, 5.3, 5.4, 5.5, and 5.6:

$$T_{Q3SUM}(d,n) = O((n\log n)/d)$$
$$T_{BS}(d,n) = O(\log d((n^2\log d)/w + T_{SBB}(q,d,n)))$$
$$T_{SBB}(q,d,n) = O(\log q(n^2/t + T_{EBBQ}(t,d,n)) + 2^{\epsilon w})$$
$$T_{EBBQ}(t,d,n) = O(n^2/t + f + T_{FQ}(t,d,p,n))$$
$$T_{FQ}(t,d,p,n) = O(n^2/t + fp^v tv^2 + fntv/d + T_{SQ}(t,d,p,n^2/(tv^2),v,n))$$
$$T_{SQ}(t,d,p,u,v,n) = O((n^2\log n)/p + fnt/d + uv\lceil(t\log p + \log n)/w\rceil + u\lceil(vt\log p)/w\rceil\log(vt) + uv^2 + 2^{\epsilon w})$$

for $q \le d \le p \le n/d$; $t\log d \le O(w)$; $v \le p$; $\log p \ge \Omega(\log(tv))$ and where $q := w\log w$; $f := nd^{2t-1}/p$. Assume d, t, v, and p are functions of n and w satisfying the restrictions from the above lemmas, and assume m is a function of n. Backwards substitution yields:

$$T_{Q3SUM}(d,n) = O((n\log n)/d)$$

$$T_{BS}(d,n) = O(n^2\log^2 d/w + (n^2\log d\log w)/t + nd^{2t-1}p^{v-1}tv^2\log d\log w +$$
$$(n^2 d^{2t-2}tv\log d\log w)/p + (n^2\log n\log d\log w)/p + (n^2\lceil(t\log p + \log n)/w\rceil$$
$$\log d\log w)/(tv) + (n^2\lceil(vt\log p)/w\rceil\log(vt)\log d\log w)/(tv^2) + 2^{\epsilon w})$$

Assume a conservative word size of $w = \Theta(\log n)$. Substitute

$$d := \frac{m\log^3 n}{\log^2(m\log n)}$$
$$t := \frac{\log^{2/3} n}{\log^{2/3} d\log^{1/3}\log n}$$
$$v := \frac{\log n}{2t\log d}$$
$$p := d^{2t-1}$$

into the above equations. From here, standard simplification techniques will yield:

$$T_{Q3SUM}(d,n) \le O(n/m)$$
$$T_{BS}(d,n) \le O((n^2\log^{5/3}(m\log n)\log^{4/3}\log n)/\log^{2/3} n)$$

This proves the first time bound. □

Remark. For t to be in $\Omega(1)$ and for p to be at least d, m must be in $n^{O(1/\log^{1/2}\log n)}$ (the constant factors for parameters can be adjusted if $m = n^{\Theta(1/\log^{1/2}\log n)}$ exactly). This logic applies to every other section with the "$m \le n^{O(1/\log^{1/2}\log n)}$" bound. t could be adjusted slightly to accommodate m values almost equal to n, but this would worsen the running time by a fraction of a $\log\log n$ factor.

Remark. Solving for the optimal values of d, t, v, and p is somewhat nontrivial. d is set to be sufficiently larger than m so that the binary searches from the beginning of Section 5 are efficient. v is set to the maximum possible value as a function of t and p (which will then be converted to a function of n and d). t and p are optimized by first adjusting them to make the exponential terms subquadratic in n and then graphing every remaining term of the time complexity as a function of t, p, n and m, identifying the intersections of the hypersurfaces, and finding the minimum point above all hypersurfaces for different values of m as a function of n.

6 Another Query List Evaluation Scheme

To prove the second bound in Theorem 1, we will evaluate query lists using a method different from the one in Lemma 5.4 (but will follow all of the reduction steps up to that point). This gives us a new algorithm for Block-Sort that has the desired running time. The steps in this section are very similar to the steps for evaluating queries in Chan's paper [Cha18]. We save a whole w factor in time complexity in exchange for extra $\log d$ factors, giving us an overall preprocessing time complexity of $O((n^2 \log^3(m \log n) \log\log n)/\log n)$ when $w = \Theta(\log n)$ and a time complexity of $O(n/m)$ to evaluate queries. The idea for this section is to apply Fredman's trick as in Section 5.5; however, we will create and manipulate difference rankings for much smaller groupings of blocks (as opposed to operating on whole provinces) and handle queries from each query list individually.

Lemma 6.1. *$T_{EBBQ}(t, d, n) = O(n^2/t + (n^2 \log^2 d)/w + 2^{\epsilon w})$ for $dt \log d \leq n$ and $\log d \geq \Omega(\log t)$*

Proof. We now describe a witness algorithm.

Step 1: Making Difference Rankings. Take sorted lists X and Y of n real numbers each and divide them into contiguous segments $X_0, X_1, \ldots X_{n/d-1}$ and $Y_0, Y_1, \ldots Y_{n/d-1}$ of size d values each. As before, the Cartesian sum of an X segment and a Y segment is a block. Conceptually, make contiguous square ***counties*** of $t \log d$ by $t \log d$ blocks each. For each county, create the set of quadruples

$$\{(i, i', i_b \bmod t \log d, X[i + i_b d] - X[i' + i_b d]) : i, i' \in [d]\} \cup$$
$$\{(j', j, j_b \bmod t \log d, Y[j' + j_b d] - Y[j + j_b d]) : j', j \in [d]\}$$

where i_b and j_b assume all possible values that could correspond to the (i_b, j_b) identifier of a block in the county (each quadruple should also implicitly indicate whether it corresponds to an X difference or a Y difference). Sort each county's set of quadruples into nondecreasing order based on the real differences using a fast comparison sort (we don't need to bit pack the quadruples at all). Initialize a counter to zero. Scan the quadruples in order and replace each quadruple's real difference with the value of the counter, incrementing the counter after every write. Repeat this across all counties. The number of bits required to represent the value of this counter (which writes the ***county difference rankings***) is only $O(\log d)$. Create a unique $X\_Difference\_Rankings$ and $Y\_Difference\_Rankings$ array for each county and pigeonhole sort each county's difference rankings into the corresponding array as indexed by (i, i', i_c) or (j, j', j_c) where $i_c := i_b \bmod (t \log d)$ is an index in $[t \log d]$ and j_c is analogous. Note that each of these indexes only occupies $O(\log d)$ bits. Also note that, unlike in Section 5.5, each county has its own separate difference ranking array (instead of the difference rankings being pooled into two big arrays). This difference is purely semantic. There are $n^2/(d^2 t^2 \log^2 d)$ counties and each one requires $O(d^2 t \log^2 d)$ time to sort/process the $O(d^2 t \log d)$ quadruples. The total time complexity is $O(n^2/t)$.

Step 2: Organizing into Counties and Prepping Difference Queries. The local index query lists in D (the data structure from Section 5.2) are already sorted into blocks in lexicographical order. We use this regular ordering to copy the query lists into another data structure D' in $O(1)$ time per query list such that they are grouped into lexicographical order according to (X position of associated county, Y position of associated county, X position of associated block relative to county, Y position of associated block relative to county). Afterwards, for each query list in D', split each of its $t/2$ quadruples (i, j, i', j') to be of the form $(i, i'), (j', j)$. Using the (i_b, j_b) block address, append to each (i, i') pair the index i_c ($i_c := i_b \bmod (t \log d)$) and append to each (j', j) pair the index j_c ($j_c := j_b \bmod (t \log d)$). Append to each resulting triple its absolute RAM address h (for the (i, i', i_c) triples) or h' (for the (j, j', j_c) triples) relative to the beginning of the county in RAM. This address also encodes each quadruple's position within the word because it will be used

to copy the quadruples back into their original locations later so that the correct pairs of quadruples can be compared. These three operations convert each of the query list's quadruples into the two adjacent quadruples $(i, i', i_c, h), (j, j', j_c, h')$, each of which occupies only $O(\log d)$ bits. By using truth tables on small constant fractions of a word, only $O(1)$ time is required per query list, which amounts to $O(n^2/t + 2^{\epsilon w})$ total time.

Step 3: Evaluating Queries. The process of converting query lists into quadruples creates $O(d^2t^2 \log^2 d)$ quadruples per county, or $O(n^2)$ quadruples across all counties. Sort all quadruples within the same county based on their (i, i', i_c) triple or (j', j, j_c) triple (all (i, i', i_c) triples are defined as less than all (j', j, j_c) triples) by applying Packed-Integer-Sort from Lemma 4.2. As before, we only need to construct the truth tables for Packed-Integer-Sort once, which takes time $O(2^{\epsilon w})$ for a slightly larger ϵ than the one in Section 4.2. The time required across all counties is $O((n^2 \log^2 d)/w)$. In each county, there are $O(d^2t \log d)$ total combinations for (i, i', i_c) or (j', j, j_c) triples, but there are $O(d^2t^2 \log^2 d)$ total quadruples, meaning that, amortized, there are $\Omega(t \log d)$ quadruples grouped next to each other that share the same triple.

We now replace each quadruple (i, i', i_c, h) or (j', j, j_c, h') with its respective county difference ranking (retain the h or h' value but destroy the (i, i', i_c) or (j', j, j_c) portion) to produce pairs (r_c, h) or (r_c, h') using the county difference ranking arrays created earlier. Note that we can evaluate a superconstant number of quadruples in constant time. This is because we have grouped the quadruples based on their (i, i', i_c) or (j', j, j_c) triple. Every contiguous section with the same (i, i', i_c) or (j', j, j_c) triple only requires time proportional to the number of words the section occupies, which is $O(\log d)$ amortized, because the rankings can be copied into an entire word at once. Across all possible pairs for a single county, this requires time $O(d^2t \log^2 d)$, so it requires $O(n^2/t)$ time total.

Sort the new pairs (r_c, h) or (r_c, h') based on their h or h' values, which puts them in the original order, and discard the h and h' values. By Lemma 4.2, this takes $O((n^2 \log^2 d)/w)$. Note that this process naturally orders the pairs within a word, too. Essentially, every query list's $t/2$ quadruples (i, j, i', j') have been replaced with $t/2$ pairs of county difference rankings in the same order. By applying truth tables to small constant fractions of a word, these difference rankings may be compared in time $O(n^2/t + 2^{\epsilon w})$ to produce an array of Boolean results. Because D' is ordered lexicographically, all Boolean result lists from D' may be copied back to their respective locations in D in $O(n^2/t)$ time. □

6.1 Alternative Time Complexity Evaluation

Reminder. ***Theorem 1:*** 3Sum-Indexing *has an algorithm with* $\min\{O((n^2 \log^{5/3}(m \log n)/\log^{2/3} n) \log^{4/3} \log n), O((n^2 \log^3(m \log n)/\log n) \log \log n)\}$ *preprocessing time and* $O(n/m)$ *query time for any desired* $m \leq n^{O(1/\log^{1/2} \log n)}$.

We prove the second bound.

Proof. We will use the following equations, which are due to Lemmas 5.1, 5.2, 5.3, and 6.1:

$$T_{Q3SUM}(d, n) = O((n \log n)/d)$$
$$T_{BS}(d, n) = O(\log d((n^2 \log d)/w + T_{SBB}(q, d, n)))$$
$$T_{SBB}(q, d, n) = O(\log q(n^2/t + T_{EBBQ}(t, d, n)) + 2^{\epsilon w})$$
$$T_{EBBQ}(t, d, n) = O(n^2/t + (n^2 \log^2 d)/w + 2^{\epsilon w})$$

for $q \leq d \leq n$; $t \log d \leq O(w)$; $dt \log d \leq n$ and where $q := w \log w$. Assume that d and t are functions of n and w satisfying the restrictions for these lemmas, and assume m is a function of n. The combined time complexity is:

$$T_{Q3SUM}(d, n) = O((n \log n)/d)$$
$$T_{BS}(d, n) = O((n^2 \log d \log w)/t + (n^2 \log^3 d \log w)/w + 2^{\epsilon w})$$

Assume a conservative word size of $w = \Theta(\log n)$. We prove the theorem by substituting

$$d := \frac{m \log^2 n}{\log(m \log n)}$$

$$t := \frac{\log n}{\log d}$$

into the above equation and using standard simplification techniques. For all $(m \log^3 n)/\log(m \log n) \leq O(n)$, we are left with:

$$T_{Q3SUM}(d, n) \leq O(m/n)$$
$$T_{BS}(d, n) \leq O((n^2 \log^3(m \log n) \log \log n)/\log n)$$

for any $(m \log^3 n)/\log(m \log n) \leq o(n)$, which implies the same result for $m \leq n^{O(1/\log^{1/2} \log n)}$. □

These bounds are achieved without constant factors in the thousands due to expander graphs (as opposed to the method extrapolated from Chan's paper [Cha18] with a similar time complexity in Section 3.1). However, this method does cost an extra $\log \log n$ factor.

7 Matrix Multiplication and Graph Cycles

In this section, we prove Theorem 2, which pertains to detecting target weight cycles in graphs. Our algorithm for detecting cycles in graphs will use a specific variant of tropical matrix multiplication that we call **Online-(target,+)-Matrix-Multiplication**. This is defined as follows:

> Given are two n by n matrices of values R and S along with a certain amount of preprocessing time. After preprocessing is finished, one or more $n \times n$ matrices of values T are given. The goal is to identify, for each $T_{i,j}$, the largest value from the multiset $\{R_{i,k} + S_{k,j} : k \in [n]\}$ less than or equal to $T_{i,j}$. Store the results in matrix U so that the result for $T_{i,j}$ is stored in $U_{i,j}$.

Let us assume the following lemma bounding the running time of an algorithm for Online-(target,+)-Matrix-Multiplication. We will prove this lemma in the following sections.

Lemma 7.1. Online-(target,+)-Matrix-Multiplication *has an algorithm with preprocessing time* $\min\{O((n^3 \log^{5/3}(m \log n) \log^{4/3} \log n)/\log^{2/3} n), O((n^3 \log^3(m \log n) \log \log n)/\log n)\}$ *that can evaluate each query matrix* T *in time* $O(n^3/m)$ *for any desired* $m \leq n^{O(1/\log^{1/2} \log n)}$.

We now describe how to use Online-(target,+)-Matrix-Multiplication to find small target weight cycles. Note that weighted digraphs are represented as a weighted adjacency matrix (entries corresponding to ordered pairs of vertices not connected by an arc have weight ∞). Let us call this adjacency matrix H. The order of vertices should be the same for the columns and for

the rows of the adjacency matrix. We limit ourselves to simple digraphs, so there is at most one arc between every pair of vertices. Note that the entry $H_{i,j}$ indicates the weight of the edge from vertex i to vertex j.

Reminder. ***Theorem 2:*** *Given an n vertex digraph with real edge weights and $\min\{O((n^3 \log^{5/3}(m \log n)/\log^{2/3} n) \log^{4/3} \log n), O((n^3 \log^3(m \log n)/\log n) \log \log n)\}$ preprocessing time (for any desired $m \leq n^{O(1/\log^{1/2} \log n)}$), we can answer the following queries deterministically in $O(n^3/m)$ time:*

- *Determine whether there exists a triangle with edge weights summing to a query target value.*
- *Determine whether there exists a zero-sum 4-cycle containing a specific query vertex.*
- *Determine whether there exists a zero-sum 5-cycle containing a specific query edge.*

Proof. We describe a procedure for the problem that meets the required time bounds. Copy input adjacency matrix H to new adjacency matrices R and S. Use Lemma 7.1 to preprocess R and S in time $\min\{O((n^3 \log^{5/3}(m \log n) \log^{4/3} \log n)/\log^{2/3} n), O((n^3 \log^3(m \log n) \log \log n)/\log n)\}$. For an arbitrary target matrix T, we can now perform Online-(target,+)-Matrix-Multiplication to get matrix U in time $O(n^3/m)$. Below, we describe how to produce, in negligible time, a target matrix that will allow us to evaluate each of the possible graph queries when compared with U.

Target Weight Triangles. Populate each $T_{(i,j)}$ with the value $r - H_{(j,i)}$, where r is the desired target cycle weight. Make sure to use the indexing scheme $H_{(j,i)}$ and not $H_{(i,j)}$. Calculate matrix U by applying Online-(target,+)-Matrix-Multiplication to matrices R, S, and T. Scan U and T for any instances where $U_{i,j} = T_{i,j}$. If such an instance is found, this indicates a triangle of the desired weight; if no instances are found, there are no triangles of the desired weight. Excluding the time spent calculating U, the entire time is $O(n^2) < O(n^3/m)$ because of the bounds on m. Correctness is straightforward; performing Online-(target,+) multiplication between the arrays R and S computes all possible distances for paths of length 2 between all vertex pairs. The target matrix checks, for every path of length two, whether there exists an edge connecting the path's end to its start such that the weight of the resulting triangle is the target weight. All ∞ entries will never result in a cycle because R and S only ever have $+\infty$'s and T only ever has $-\infty$'s.

Zero-Sum 4-Cycles. Initialize T so that all of its entries are $-\infty$'s. Enumerate all paths of length two with the query vertex as the intermediate vertex by simply computing all combinations of a vertex on an arc *to* the query vertex and a vertex on an arc *from* the query vertex. For every such path, replace $T_{i,j}$ with the *negative* of the total weight of the path where i is the *ending* vertex and j is the *starting* vertex. The time to populate T in this way is at most $O(n^2)$. Calculate matrix U by applying Online-(target,+)-Matrix-Multiplication to matrices R, S, and T. Scan U and T for any instances where $U_{i,j} = T_{i,j}$. If such an instance exists, this corresponds to a zero-weight cycle of length 4 containing the query vertex; if no instance exists, the desired 4-cycle does not exist. The time to scan is at most $O(n^2)$. Correctness is straightforward and results from an analogous argument to the argument for target weight triangles.

Zero-Sum 5-Cycles. Initialize T so that all of its entries are $-\infty$'s. Enumerate all paths of length three with the query edge as the intermediate edge by simply computing all combinations of a vertex on an arc to the query edge's starting vertex and a vertex on an arc from the query edge's ending vertex. For every such path, replace $T_{i,j}$ with the negative of the total weight of the path where i is the ending vertex and j is the starting vertex. Calculate matrix U by applying Online-(target,+)-Matrix-Multiplication to matrices R, S, and T. Scan U and T for any instances where $U_{i,j} = T_{i,j}$. If such an instance exists, this corresponds to a zero-weight cycle of length 5 through the query edge; if no instance exists, the desired 5-cycle does not exist. The time to scan

is at most $O(n^2)$. Again, correctness is straightforward and results from an analogous argument to the argument for target weight triangles. □

7.1 (multi-target,+)-Matrix-Multiplication

In the next few subsections, we will prove Lemma 7.1. Matrices in this section are indexed in a slightly unusual way. For an n by n matrix, $M_{0,0}$ is the bottom left entry, $M_{n-1,0}$ is the top left entry, and $M_{n-1,n-1}$ is the top right entry. This scheme means that indexing a matrix looks the same as indexing $X+Y$. Let $A\hat{+}B$ be the list of sums $\langle A[0]+B[0], A[1]+B[1], \ldots A[n-1]+B[n-1]\rangle$ for two vectors A and B of length n each. This is referred to as the ***convolution sum***.

As with our algorithm for 3Sum-Indexing, we will first present a bound on the time required to preprocess the input matrices R and S, then present a bound on the time required to evaluate queries assuming that R and S have been preprocessed. Note that we can assume $m = \Omega(1)$; if not, the folklore tropical matrix multiplication techniques give an algorithm with the desired preprocessing and query time. We assume $\Omega(\log n) \leq w < n^{o(1)}$ and only assign w to the intended value at the end of the analysis. Restrictions on parameters are inherited between subsections.

Preprocessing. The idea for Online-(target,+)-Matrix-Multiplication is to partition every vector $\langle R_{i,0}, R_{i,1}, \ldots R_{i,n-1}\rangle$ into contiguous sections $R_i^{(0)}, R_i^{(1)}, \ldots R_i^{(n/d-1)}$ of size d each such that $R_i^{(g)}$ is the vector

$$\langle R_{i,gd}, R_{i,gd+1}, \ldots R_{i,gd+d-1}\rangle.$$

Every vector $\langle S_{0,j}, S_{1,j}, \ldots S_{n-1,j}\rangle$ is partitioned analogously. For our preprocessing step, we now perform the following procedure:

Sort $R_i^{(g)}\hat{+}S_j^{(g)}$ for all $i, j \in [n]$ and $g \in [n/d]$.

Let us call this problem Matrix-Block-Sort, with time complexity $T_{MBS}(d, n)$.

Evaluating Queries. We now describe how to evaluate the results for a given query matrix T. Assuming Matrix-Block-Sort has already been applied to matrices R and S, let us define the time complexity of evaluating Online-(target,+)-Matrix-Multiplication for R, S, and T to be $T_{QTMult}(d, n)$.

Lemma 7.2. *$T_{QTMult}(d, n) = O((n^3 \log n)/d)$ for $d \leq n$ where d is a function of n and w*

Proof. An algorithm for the problem with the desired running time is as follows. Initialize the $n \times n$ matrix U to be empty. For a fixed $i, j \in [n]$, do the following: For each $g \in [n/d]$, binary search for the largest value less than or equal to $T_{i,j}$ in $R_i^{(g)}\hat{+}S_j^{(g)}$. Populate $U_{i,j}$ with the maximum resulting value from all of the binary searches. The total time is $O((n \log n)/d)$.

Repeat the above for all different choices of i and j. The total time increases by a factor of $O(n^2)$, leaving us with a time complexity of $O((n^3 \log n)/d)$. Note that this procedure takes the same amount of time regardless of whether we are operating on bit packed lists of indices or actual values as long as the time to access an entry is $O(1)$. □

Reducing to an Intermediate Problem. To evaluate instances of Matirx-Block-Sort, we will reduce to a problem that is more similar to the Block-Sort problem from Section 5. Let us define **Sparse-Convolution-Sort** as follows, and let us call the time complexity of this problem $T_{SCS}(d, n)$.

Given arrays of lists of values P and Q such that $|P| = |Q| = n/d$ and such that $|P^{(0)}| = |P^{(1)}| = \cdots = |P^{(n/d-1)}| = |Q^{(0)}| = |Q^{(1)}| = \cdots = |Q^{(n/d-1)}| = d$, sort $P^{(g)} \hat{+} Q^{(g')}$ for all $g, g' \in [n/d]$.

In this section, the values in P and Q will always be real numbers. As before, sorting means producing a data structure that will report the k^{th} smallest value in $O(1)$ time. This procedure is unusual because it takes all combinations of g and g' instead of having $g = g'$ as one might expect.

We now describe a reduction converting instances of Matrix-Block-Sort into many instances of Sparse-Convolution-Sort.

Lemma 7.3. *$T_{MBS}(d, n) = O(n^2 d + ndT_{SCS}(d, n))$ for $d \leq n$ where d is a function of n and w.*

Proof. Our witness algorithm is as follows. Partition every $\langle R_{i,0}, R_{i,1}, \ldots R_{i,n-1} \rangle$ into contiguous sections of size d and label them $R_i^{(g)}$ for the appropriate $i \in [n]$ and $g \in [n/d]$. Partition every $\langle S_{0,j}, S_{1,j}, \ldots S_{n-1,j} \rangle$ into contiguous sections of size d and label them $S_j^{(g)}$ for the appropriate $j \in [n]$ and $g \in [n/d]$. Choose fixed indexes $g \in [n/d]$, $i_g \in [d]$, and $j_g \in [d]$. Invoke Sparse-Convolution-Sort with

$$P := \{R^{(g)}_{i+i_g n/d} : i \in [n/d]\} \text{ and}$$

$$Q := \{S^{(g)}_{j+j_g n/d} : j \in [n/d]\}.$$

Repeat this procedure for all possible g's, i_g's, and j_g's. This sorts all possible multisets of the form $R_i^{(g)} \hat{+} S_j^{(g)}$ for all $i, j \in [n]$ and $g \in [n/d]$. The time to pass all values/indexes for a single invocation is $O(n)$ without bit packing. There are $O((n/d)d^2) = O(nd)$ total invocations of Sparse-Convolution-Sort for a total time complexity of $O(n^2 d + ndT_{SCS}(d, n))$. □

7.2 Sparse-Convolution-Sort

Inherit the structure of the input lists P and Q from the previous subsections. Sparse-Convolution-Sort requires us to sort $P^{(g)} \hat{+} Q^{(g')}$ for all $g, g' \in [n/d]$, which is similar to the problem Block-Sort from Section 5.1. To see this, concatenate all lists in P and all lists in Q and then use the two lists to produce a Cartesian sum matrix just like we did for X and Y in previous sections. When this matrix is divided into square blocks of side length d, the d sums along the largest diagonal from bottom left to top right in each of the n^2/d^2 blocks are the convolution sums that we want to permute into sorted order. As such, Sparse-Convolution-Sort is only different in that it operates on n^2/d total local index pairs (instead of n^2) and its input lists aren't in sorted order. The decrease in local index pairs means that most of the steps from Block-Sort and the problems it reduces to will execute faster by a factor of d. The lack of initial order within each block is a problem because Sort-Block-Buckets assumes that every block bucket to be sorted is a concatenation of a constant number of sorted subsequences, and its use of the bitonic sorting network won't work if this isn't the case. This is addressed by applying the bitonic sorting network differently to generate sorted ***kernels*** of size q and then proceeding by combining kernels together normally.

New Definitions for Blocks and Buckets. Define a ***diagonal block*** to be the multiset $P^{(g)} \hat{+} Q^{(g')}$ for a fixed g and g'. The sorted ordering of a diagonal block will eventually be represented as a bit packed list of local index pairs in $[d] \times [d]$ (along with the diagonal block identifier (i_b, j_b) where $i_b = g$ and $j_b = g'$). Note that one local index would suffice for each point because there is only one X index for every Y index along the diagonal, but writing out the index pairs explicitly makes later calculations easier. Divide each of the n^2/d^2 diagonal blocks of d sums each into contiguous sections of q sums each. Each contiguous section (***kernel block bucket***) should be

represented as a global block index pair (i_b, j_b) followed by the bit packed list of local index pairs in $[d] \times [d]$ as for 3SUM-INDEXING. These are stored in ***data structure*** **G** as described below:

> $G_i^{(i_b,j_b)}$ is the i^{th} kernel block bucket in the diagonal block corresponding to $g = i_b$ and $g' = j_b$. Ensure that these kernel block buckets are stored with the same lexicographical properties as the data structure C from previous sections.

Let **SORT-KERNELS** be the problem described below:

> Permute the local index pairs in every entry of G such that their corresponding sums are in nondecreasing order.

$T_{SK}(g, d, n)$ is the time complexity of this problem.

The Reductions. Let **SORT-DIAGONAL-BLOCK-BUCKETS** be the problem analogous to SORT-BLOCK-BUCKETS, except that it only receives d/q query lists per block instead of d^2/q query lists, and it operates on data structure G (which we just defined). It still expects that every input bucket should be of length q and contain two smaller sub-lists of local indexes, each of whose corresponding sums are in nondecreasing order. Let $T_{SDBB}(q, d, n)$ be the time complexity of this problem.

Lemma 7.4. *$T_{SCS}(d, n) = O(T_{SK}(q, d, n) + (n^2 \log d)/(dw) + \log(d/q)((n^2 \log d)/(dw) + T_{SDBB}(q, d, n)))$ where $q := w \log w$ for $q \leq d$*

Proof. Our witness reduction is very similar to the one in Section 5.1. Set parameter

$$q := w \log w$$

for the same reasons as in Section 5.1. Enumerating the bit packed lists of local index pairs for all kernel block buckets requires $O((n^2 \log d)/(dw))$. Apply SORT-KERNELS to order each kernel block bucket to permute the index pairs such that their corresponding sums are in nondecreasing order. Now, combine pairs of sorted kernel block buckets as in Section 5.1 to produce multiple new diagonal block buckets for each pair of sorted kernel block buckets. Sort the new diagonal block buckets using SORT-DIAGONAL-BLOCK-BUCKETS, which will imply sorted lists of double the original length (as explained in Section 5.1). The only difference is that there will be less buckets by a factor of d. Recurse to produce larger and larger sorted lists. After $\log(d/q)$ rounds, the entire diagonal block is sorted. □

We now discuss sorting the kernels. In Section 5.2, we used the bitonic sorting network to sort a list of length q representing a bitonic sequence into nondecreasing order using $\log q$ rounds, and each round required $O(q)$ comparison queries and swaps. Each round divided q into smaller and smaller bitonic sequences such that each bitonic sequence dominated the previous. Once the bitonic sequences reached a size of one, the entire list of q values was sorted into nondecreasing order. Applying this method to a completely unordered list of indexes may seem problematic at first. However, there is a relatively simple solution that involves reversing the order of the sorting steps from Section 5.2.

Lemma 7.5. *$T_{SK}(q, d, n) = O(\log q((n^2 \log d)/(dw) + T_{SDBB}(q, d, n)) + 2^{\epsilon w})$ for $t \log d \leq O(w)$ and $t \leq q$ where t is a function of q, d, n, and w*

Proof. This reduction is as follows. Apply only the last step from the bitonic sorting network from the proof in Section 5.2 to the completely unordered kernels (which are represented as bit packed lists with an appended identifying pair). This creates $q/2$ sorted sequences of size 2 each. Reverse the order of every other sequence using truth tables on a small constant fraction of a word at once.

Now apply only the penultimate and last step from the proof in Section 5.2. This produces $q/4$ sorted sequences of size 4 each. Again, reverse the order of every other sequence using truth tables on a small constant fraction of a word at once. Repeat this process $O(\log q)$ times until the entire sequence is sorted. Note that, every time the length of sorted lists is doubled, one more step is required. Also note that Sort-Diagonal-Block-Buckets is equivalent to the last set of steps where two lists of length $q/2$ each are combined to make a sorted list of length q. Thus, every time the sorted sequence length is doubled, the time taken is less than or equal to $T_{SDBB}(q, d, n)$. Each of the $O(\log q)$ rounds requires $O((n^2 \log d)/(dw) + T_{SDBB}(q, d, n))$ time assuming the truth tables were already constructed. □

Let **Eval-DBBQ** be the exact same as Eval-BBQ, except that it only receives d/t quadruplet lists per block instead of d^2/t. Its time complexity is $T_{EDBBQ}(t, d, n)$. Let **Diagonal-Family-Queries** be Family-Queries but with $n^2/(dt)$ total quadruplet lists instead of n^2/t. Its time complexity is $T_{DFQ}(t, d, p, n)$.

Lemma 7.6. *$T_{SDBB}(q, d, n) = O(\log q(n^2/(dt) + T_{EDBBQ}(t, d, n)) + 2^{\epsilon w})$ for $t \log d \leq O(w)$*

Proof. As in the proof of Lemma 5.3. Just reduce the number of query lists and block buckets by a factor of d compared to Sort-Block-Buckets. □

Lemma 7.7. *$T_{EDBBQ}(t, d, n) = O(n^2/(dt) + f + T_{DFQ}(t, d, p, n))$ for $p \leq n/d$ where $f := nd^{2t-1}/p$ and p is a function of d, t, n, and w. Also, $T_{EDBBQ}(t, d, n) = O(n^2/(dt) + (n^2 \log^2 d)/(dw) + 2^{\epsilon w})$ for $d^2 t \log d \leq n$ and $\log d \geq \Omega(\log t)$.*

Proof. As in the proofs for Lemma 5.4 and Lemma 6.1. Reduce the total number of query lists from n^2/t to $n^2/(dt)$. Note that the number of families actually stays the same. To achieve the second bound, repeat the methods for Lemma 6.1 but make counties of size $dt \log d$ by $dt \log d$ blocks. □

Lemma 7.8. *$T_{DFQ}(t, d, p, n) = O(n^2/(dt) + fp^v tv^2 + fntv/d + T_{SQ}(t, d, p, n^2/(dtv^2), v, n))$ where $f := nd^{2t-1}/p$ for $v \leq p$ where v is a function of t, d, p, n, and w*

Proof. As in the proof of Lemma 5.5. Reduce the total number of query lists from n^2/t to $n^2/(dt)$. The number of families and total number of pigeonholes each list is sorted into remains the same. Note that Square-Queries may actually be called here without adapting the proof for Square-Queries at all. □

7.3 Optimizing Parameters

Reminder. ***Lemma 7.1:*** Online-(target,+)-Matrix-Multiplication *has an algorithm with preprocessing time* $\min\{O((n^3 \log^{5/3}(m \log n) \log^{4/3} \log n)/\log^{2/3} n), O((n^3 \log^3(m \log n) \log \log n)/\log n)\}$ *that can evaluate each query matrix T in time $O(n^3/m)$ for any desired $m \leq n^{O(1/\log^{1/2} \log n)}$.*

We will prove each bound independently by providing a witness algorithm for each. By executing these algorithms in parallel during preprocessing and ending preprocessing once either of them terminates, we can achieve the claimed running time.

We prove the first bound below.

Proof. Recall the following bounds from Lemmas 7.2, 7.3, 7.4, 7.5, 7.6, 7.7, 7.8, and 5.6:

$$T_{QTMult}(d, n) = O((n^3 \log n)/d)$$
$$T_{MBS}(d, n) = O(n^2 d + ndT_{SCS}(d, n))$$

$$T_{SCS}(d,n) = O(T_{SK}(q,d,n) + (n^2 \log d)/(dw) + \log(d/q)((n^2 \log d)/(dw) + T_{SDBB}(q,d,n)))$$
$$T_{SK}(q,d,n) = O(\log q((n^2 \log d)/(dw) + T_{SDBB}(q,d,n)) + 2^{\epsilon w})$$
$$T_{SDBB}(q,d,n) = O(\log q(n^2/(dt) + T_{EDBBQ}(t,d,n)) + 2^{\epsilon w})$$
$$T_{EDBBQ}(t,d,n) = O(n^2/(dt) + f + T_{DFQ}(t,d,p,n))$$
$$T_{DFQ}(t,d,p,n) = O(n^2/(dt) + fp^v tv^2 + fntv/d + T_{SQ}(t,d,p,n^2/(dtv^2),v,n))$$
$$T_{SQ}(t,d,p,u,v,n) = O((n^2 \log n)/p + fnt/d + uv\lceil(t \log p + \log n)/w\rceil + u\lceil(vt \log p)/w\rceil \log(vt) + uv^2 + 2^{\epsilon w})$$

for $q \leq d \leq p \leq n/d$; $t \log d \leq O(w)$; $v \leq p$; $\log p \geq \Omega(\log(tv))$ and where $q := w \log w$; $f := nd^{2t-1}/p$. Assume d, t, p, and v are functions of n and w satisfying the constraints from the above lemmas, and assume m is a function of n. Backwards substitution gives:

$$T_{QTMult}(d,n) = O((n^3 \log n)/d)$$

$$T_{MBS}(m,n) = O(n^2 d + (n^3 \log^2 d)/w + (n^3 \log d \log w)/t + n^2 d^{2t} p^{v-1} tv^2 \log d \log w +$$
$$(n^3 d^{2t-1} tv \log d \log w)/p + (n^3 d \log n \log d \log w)/p + (n^3 \lceil(t \log p + \log n)/w\rceil \log d \log w)/(tv) +$$
$$(n^3 \lceil(vt \log p)/w\rceil \log(vt) \log d \log w)/(tv^2) + 2^{\epsilon w})$$

Assume a conservative word size of $w = \Theta(\log n)$ bits. Substitute

$$d := \frac{m \log^3 n}{\log^2(m \log n)}$$
$$t := \frac{\log^{2/3} n}{\log^{2/3} d \log^{1/3} \log n}$$
$$v := \frac{\log n}{(2t+1) \log d}$$
$$p := d^{2t}$$

into the above equations and then simplify. This gives:

$$T_{QTMult}(d,n) \leq O(n^3/m)$$
$$T_{MBS}(d,n) \leq O((n^3 \log^{5/3}(m \log n) \log^{4/3} \log n)/\log^{2/3} n)$$

for $m \leq n^{O(1/\log^{1/2} \log n)}$. □

We now prove the second bound from Lemma 7.1.

Proof. We use Lemmas 7.2, 7.3, 7.4, 7.5, 7.6, and 7.7:

$$T_{QTMult}(d,n) = O((n^3 \log n)/d)$$
$$T_{MBS}(d,n) = O(n^2 d + ndT_{SCS}(d,n))$$
$$T_{SCS}(d,n) = O(T_{SK}(q,d,n) + (n^2 \log d)/(dw) + \log(d/q)((n^2 \log d)/(dw) + T_{SDBB}(q,d,n)))$$
$$T_{SK}(q,d,n) = O(\log q((n^2 \log d)/(dw) + T_{SDBB}(q,d,n)) + 2^{\epsilon w})$$

$$T_{SDBB}(q,d,n) = O(\log q(n^2/(dt) + T_{EDBBQ}(t,d,n)) + 2^{\epsilon w})$$
$$T_{EDBBQ}(t,d,n) = O(n^2/(dt) + (n^2 \log^2 d)/(dw) + 2^{\epsilon w})$$

for $q \leq d \leq n$; $t \log d \leq O(w)$; $d^2 t \log d \leq n$ and where $q := w \log w$. Assume d and t are functions of n and w satisfying the restrictions, and assume m is a function of n. Backwards substitution gives:

$$T_{QTMult}(d,n) = O((n^3 \log n)/d)$$

$$T_{MBS}(d,n) = O(n^2 d + (n^3 \log d \log w)/t + (n^3 \log^3 d \log w)/w + 2^{\epsilon w})$$

Use a conservative word size of $w = \Theta(\log n)$ bits and substitute:

$$d := \frac{m \log^2 n}{\log(m \log n)}$$

$$t := \frac{\log n}{\log d}$$

This yields:

$$T_{QTMult}(d,n) \leq O(n^3/m)$$
$$T_{MBS}(d,n) \leq O((n^3 \log^3(m \log n) \log\log n)/\log n)$$

for any $(m \log^3 n)/\log(m \log n) \leq o(n)$, which implies the same result for $m \leq n^{O(1/\log^{1/2}\log n)}$. □

8 Faster Cartesian Selection

In this section, we prove Theorem 3, which pertains to selection in Cartesian sums. The main focus will be on preprocessing $X + Y$ into a rectangular matrix with sorted rows and columns; the actual queries can be handled by applying Frederickson and Johnson's selection algorithm [FJ84] to this matrix.

Let **Long-Matrix** be the the following problem, with time complexity $T_{LM}(d,n)$:

> Given lists X and Y of n real numbers each, permute the entries from $X + Y$ into an $O(nd) \times O(n/d)$ matrix K with sorted rows and columns. The matrix K may be represented using any data structure that reports entries in $O(1)$ time.

We reduce Long-Matrix to the problem Block-Sort from Section 5.

Lemma 8.1. *$T_{LM}(d,n) = O(n^2/d^2 + n \log n + T_{BS}(d,n))$ where d is a function of n and w*

Proof. The following is a witness algorithm proving the lemma. Start by sorting X and Y and applying Block-Sort to their Cartesian sum (use the optimized parameters from Section 5.6). Create $O(n/d)$ long sorted diagonals as per Lemma 2.1 (each long sorted diagonal consists of a chain of blocks stacked diagonally up and to the right). List the long sorted diagonal from the upper left as the first column of matrix K. This list is too short; pad the end with ∞ tokens to the proper length. Take the next long sorted diagonal (second from the upper left) and pad the beginning with $2d^2$ $-\infty$ tokens. Make this list the second column of K and pad the end with ∞ tokens as necessary. Every entry in the second column is guaranteed to come from a block that is

dominated by the block of the corresponding entry in the first column. The third column of K is the next long sorted diagonal with $4d^2$ $-\infty$ tokens appended to the beginning and ∞ tokens appended to the end as needed. The fourth column is the same but with $6d^2$ leading $-\infty$'s. Repeating this process for all $O(n/d)$ long sorted diagonals produces the desired array K. Note that the time to construct this array can be just $O(n^2/d^2 + T_{BS}(d, n))$ because mapping the bottom left corner of every block to the correct location in K implies where the rest of the block should go, and we can indicate where $-\infty$ and ∞ tokens should be placed without writing them all out explicitly. This means that, instead of producing array K explicitly, we create a data structure that will report any requested entry in $O(1)$ time. □

Recall the definition of **Square-Select**:

> Given are lists of real numbers X and Y (each of size n) and a certain amount of preprocessing time. After preprocessing is finished, one or more input query rankings r are given sequentially; for each, select the r^{th} smallest sum from $X + Y$ and report this value.

Assuming that our algorithm for Long-Matrix has been applied to X and Y, let $T_{QSqr}(d, n)$ be the time to evaluate a single ranking query.

Lemma 8.2. *$T_{QSqr}(d, n) = O((n \log d)/d)$ where $d \leq n$ and d is a function of n and w*

Proof. We use one of Frederickson and Johnson's algorithms as a witness proving the lemma. They give an algorithm to select the k^{th} smallest value in an $a \times b$ matrix of values with sorted rows and columns (where $a \leq b$) in time $O(a \log(b/a))$. For $T_{QSqr}(d, n)$, we have an $n/d \times nd$ matrix, so the time taken is $O((n \log d)/d)$. Note that their algorithm works on any data structure representing a matrix with sorted rows and columns that has $O(1)$ access time per entry. □

We are now ready to prove Theorem 3. Note that we can assume $m = \Omega(1)$; if not, the folklore algorithm from Section 1.1 can be easily adapted to give the desired preprocessing and query time.

Reminder. ***Theorem 3:*** Square-Select *has a deterministic algorithm with* $\min\{O((n^2 \log^{5/3}(m \log n)/\log^{2/3} n) \log^{4/3} \log n), O((n^2 \log^3(m \log n)/\log n) \log \log n)\}$ *preprocessing time and $O(n/m)$ query time for any desired $m \leq n^{O(1/\log^{1/2} \log n)}$.*

Proof. By Lemmas 8.1 and 8.2, we have the following (assuming our parameters are in the appropriate ranges):

$$T_{LM}(m, n) = O(n^2/d^2 + n \log n + T_{BS}(d, n))$$

$$T_{QSqr}(d, n) = O((n \log d)/d)$$

Note that the first equation bounds the preprocessing time and the second equation bounds the query time. To get the first bound in the theorem, we substitute

$$d := \frac{m \log^3 n}{\log^2(m \log n)}$$

and repeat the evaluation of $T_{BS}(d, n)$ that appeared in Section 5.6. To get the second bound in the theorem, we substitute

$$d := \frac{m \log^2 n}{\log(m \log n)}$$

and repeat the evaluation for $T_{BS}(d, n)$ that appeared in Section 6.1. The min function in the time bound for preprocessing is achieved by performing both possible algorithms for Block-Sort simultaneously and terminating when one of them completes. □

9 Discussion

Our algorithms for 3Sum-Indexing take advantage of the ability to produce difference rankings and then compress them in multiple stages. They utilize bit tricks reminiscent of fast integer sorting techniques. This technique gives us new results for several "linear" problems, but it appears to struggle when applied to nonlinear domains, and we are unsure if our methods can yield an algorithm with subquadratic preprocessing time and truly sublinear query time for 3Sum-Indexing.

Open Problem 1. *Find faster* 3Sum-Indexing *algorithms for the algebraic setting, where the goal is to identify instances of the form* $f(x, y, z) = 0$ *for some constant degree polynomial function* f*.*

Open Problem 2. *Find an algorithm for* 3Sum-Indexing *with truly sublinear query time and subquadratic preprocessing time.*

References

[AC22] Boris Aronov and Jean Cardinal. Geometric pattern matching reduces to k-sum. *Discrete & Computational Geometry*, 68(3):850–859, 2022.

[ACLL14] Amihood Amir, Timothy M. Chan, Moshe Lewenstein, and Noa Lewenstein. On hardness of jumbled indexing. In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt, and Elias Koutsoupias, editors, *Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II*, volume 8573, pages 114–125. Springer, 2014.

[ADKF70] Vladimir L'vovich Arlazarov, Yefim A Dinitz, MA Kronrod, and IgorAleksandrovich Faradzhev. On economical construction of the transitive closure of an oriented graph. In *Doklady Akademii Nauk*, volume 194, pages 487–488. Russian Academy of Sciences, 1970.

[AH97] Susanne Albers and Torben Hagerup. Improved parallel integer sorting without concurrent writing. *Information and Computation*, 136(1):25–51, 1997.

[AKS83] Miklós Ajtai, János Komlós, and Endre Szemerédi. An 0 (n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, pages 1–9, 1983.

[AR18] Udit Agarwal and Vijaya Ramachandran. Fine-grained complexity for sparse graphs. In *Proceedings of the 50th Annual ACM SIGACT Symposium on Theory of Computing*, pages 239–252, 2018.

[AWW14] Amir Abboud, Virginia Vassilevska Williams, and Oren Weimann. Consequences of faster alignment of sequences. In *Automata, Languages, and Programming: 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part I 41*, pages 39–51. Springer, 2014.

[Bat68] Kenneth E Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, pages 307–314, 1968.

[BCD+06] David Bremner, Timothy M Chan, Erik D Demaine, Jeff Erickson, Ferran Hurtado, John Iacono, Stefan Langerman, and Perouz Taslakian. Necklaces, convolutions, and x+ y. In *Algorithms–ESA 2006: 14th Annual European Symposium, Zurich, Switzerland, September 11-13, 2006. Proceedings 14*, pages 160–171. Springer, 2006.

[BDP05] Ilya Baran, Erik D Demaine, and Mihai Pătraşcu. Subquadratic algorithms for 3sum. In *Algorithms and Data Structures: 9th International Workshop, WADS 2005, Waterloo, Canada, August 15-17, 2005. Proceedings 9*, pages 409–421. Springer, 2005.

[Bel58] David A Bell. The principles of sorting. *The Computer Journal*, 1(2):71–77, 1958.

[BGL+22] Philip Bille, Inge Li Gørtz, Moshe Lewenstein, Solon P Pissis, Eva Rotenberg, and Teresa Anna Steiner. Gapped string indexing in subquadratic space and sublinear query time. *arXiv preprint arXiv:2211.16860*, 2022.

[BHP01] Gill Barequet and Sariel Har-Peled. Polygon containment and translational in-hausdorff-distance between segment sets are 3sum-hard. *International Journal of Computational Geometry & Applications*, 11(04):465–474, 2001.

[CGL23] Eldon Chung and Kasper Green Larsen. Stronger 3sum-indexing lower bounds. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 444–455. SIAM, 2023.

[Cha18] Timothy M Chan. More logarithmic-factor speedups for 3sum,(median,+)-convolution, and some geometric 3sum-hard problems. In *29th Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2018*, pages 881–897. Association for Computing Machinery, 2018.

[CHC09] Kuan-Yu Chen, Ping-Hui Hsu, and Kun-Mao Chao. Approximate matching for run-length encoded strings is 3sum-hard. In *Combinatorial Pattern Matching: 20th Annual Symposium, CPM 2009 Lille, France, June 22-24, 2009 Proceedings 20*, pages 168–179. Springer, 2009.

[CL15] Timothy M Chan and Moshe Lewenstein. Clustered integer 3sum via additive combinatorics. In *Proceedings of the forty-seventh annual ACM symposium on Theory of computing*, pages 31–40, 2015.

[Cop82] Don Coppersmith. Rapid multiplication of rectangular matrices. *SIAM Journal on Computing*, 11(3):467–471, 1982.

[CWX22] Timothy M Chan, Virginia Vassilevska Williams, and Yinzhan Xu. Hardness for triangle problems under even more believable hypotheses: reductions from real apsp, real 3sum, and ov. In *Proceedings of the 54th Annual ACM SIGACT Symposium on Theory of Computing*, pages 1501–1514, 2022.

[DJWW22] Mina Dalirrooyfard, Ce Jin, Virginia Vassilevska Williams, and Nicole Wein. Approximation algorithms and hardness for n-pairs shortest paths and all-nodes shortest cycles. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 290–300. IEEE, 2022.

[FJ82] Greg N Frederickson and Donald B Johnson. The complexity of selection and ranking in x+ y and matrices with sorted columns. *Journal of Computer and System Sciences*, 24(2):197–208, 1982.

[FJ84] Greg N Frederickson and Donald B Johnson. Generalized selection and ranking: sorted matrices. *SIAM Journal on computing*, 13(1):14–30, 1984.

[Fre76] Michael L Fredman. New bounds on the complexity of the shortest path problem. *SIAM Journal on Computing*, 5(1):83–89, 1976.

[Fre15] Ari Freund. Improved subquadratic 3sum. *Algorithmica*, 2(77):440–458, 2015.

[GGH+20] Alexander Golovnev, Siyao Guo, Thibaut Horel, Sunoo Park, and Vinod Vaikuntanathan. Data structures meet cryptography: 3sum with preprocessing. In *Proceedings of the 52nd annual ACM SIGACT symposium on theory of computing*, pages 294–307, 2020.

[GKLP17] Isaac Goldstein, Tsvi Kopelowitz, Moshe Lewenstein, and Ely Porat. Conditional lower bounds for space/time tradeoffs. In *Algorithms and Data Structures: 15th International Symposium, WADS 2017, St. John's, NL, Canada, July 31–August 2, 2017, Proceedings 15*, pages 421–436. Springer, 2017.

[GLP19] Isaac Goldstein, Moshe Lewenstein, and Ely Porat. On the hardness of set disjointness and set intersection with bounded universe. *arXiv preprint arXiv:1910.00831*, 2019.

[GO95] Anka Gajentaan and Mark H Overmars. On a class of o (n2) problems in computational geometry. *Computational geometry*, 5(3):165–185, 1995.

[GP14] Allan Grønlund and Seth Pettie. Threesomes, degenerates, and love triangles. In *2014 IEEE 55th Annual Symposium on Foundations of Computer Science*, pages 621–630. IEEE, 2014.

[GS15] Omer Gold and Micha Sharir. Improved bounds for 3sum, k-sum, and linear degeneracy. *arXiv preprint arXiv:1512.05279*, 2015.

[GU18] François Le Gall and Florent Urrutia. Improved rectangular matrix multiplication using powers of the coppersmith-winograd tensor. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1029–1046. SIAM, 2018.

[KLM+09] Telikepalli Kavitha, Christian Liebchen, Kurt Mehlhorn, Dimitrios Michail, Romeo Rizzi, Torsten Ueckerdt, and Katharina A Zweig. Cycle bases in graphs characterization, algorithms, complexity, and applications. *Computer Science Review*, 3(4):199–243, 2009.

[KLM19] Daniel M Kane, Shachar Lovett, and Shay Moran. Near-optimal linear decision trees for k-sum and related problems. *Journal of the ACM (JACM)*, 66(3):1–18, 2019.

[KP19] Tsvi Kopelowitz and Ely Porat. The strong 3sum-indexing conjecture is false. *arXiv preprint arXiv:1907.11206*, 2019.

[KPP16] Tsvi Kopelowitz, Seth Pettie, and Ely Porat. Higher lower bounds from the 3sum conjecture. In *Proceedings of the twenty-seventh annual ACM-SIAM symposium on Discrete algorithms*, pages 1272–1287. SIAM, 2016.

[LWW18] Andrea Lincoln, Virginia Vassilevska Williams, and Ryan Williams. Tight hardness for shortest cycles and paths in sparse graphs. In *Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 1236–1252. SIAM, 2018.

[Pat10] Mihai Patrascu. Towards polynomial lower bounds for dynamic problems. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 603–610, 2010.

[Wil14] Ryan Williams. Faster all-pairs shortest paths via circuit complexity. In *Proceedings of the forty-sixth annual ACM symposium on Theory of computing*, pages 664–673, 2014.

[YZ04] Raphael Yuster and Uri Zwick. Detecting short directed cycles using rectangular matrix multiplication and dynamic programming. In *SODA*, volume 4, pages 254–260, 2004.