Tolerant Algorithms

Rolf Klein[1], Rainer Penninger[1], Christian Sohler[2], and David P. Woodruff[3]

[1] University of Bonn
[2] TU Dortmund
[3] IBM Research-Almaden

Abstract. Assume we are interested in solving a computational task, e.g., sorting n numbers, and we only have access to an unreliable primitive operation, for example, comparison between two numbers. Suppose that each primitive operation fails with probability at most p and that repeating it is not helpful, as it will result in the same outcome. Can we still approximately solve our task with probability $1-f(p)$ for a function f that goes to 0 as p goes to 0? While previous work studied sorting in this model, we believe this model is also relevant for other problems. We

- find the maximum of n numbers in $O(n)$ time,
- solve 2D linear programming in $O(n \log n)$ time,
- approximately sort n numbers in $O(n^2)$ time such that each number's position deviates from its true rank by at most $O(\log n)$ positions,
- find an element in a sorted array in $O(\log n \log \log n)$ time.

Our sorting result can be seen as an alternative to a previous result of Braverman and Mossel (SODA, 2008) who employed the same model. While we do not construct the maximum likelihood permutation, we achieve similar accuracy with a substantially faster running time.

1 Introduction

Many algorithms can be designed in such a way that the input data is accessed only by means of certain primitive queries. For example, in comparison-based sorting an algorithm might specify two key indices, i and j, and check whether the relation $q_i < q_j$ holds. Except for such yes/no replies, no other type of information on the key set $\{q_1, \ldots, q_n\}$ is necessary, or this information might not even be available. Similarly, in solving a linear program, given a point and a line, the primitive may return which side of the line the point is on.

Given an oracle that can answer all possible primitive queries on the input data, one can develop an algorithm without worrying about the input data type or numerical issues. This allows for more modular and platform-independent algorithm design. A natural question is what happens if the oracle errs?

There are several ways to model such errors. One natural model is that each time the oracle is queried it returns the wrong answer with a small error probability bounded by some $p > 0$, independent of past queries. In this case, repeating a query can be used to boost the success probability at the cost of additional work. In this paper we shall employ a different model introduced by Braverman and Mossel [3] in the context of sorting. The oracle errs on each query independently with probability at most p, but now each possible primitive query on the input is answered by the oracle *only once*.

There are good reasons to study this model. First, it may not be possible to repeat a query, as observed in [3]. For example, in ranking soccer clubs, or even just determining the best soccer club, the outcome of an individual game may differ from the "true" ordering of the two teams and it is impossible to repeat this game. A different example is ranking items by experts [3], which provides an inherently noisy view of the "true" ranking. Another reason to study this model is that the oracle may err on a particular query due to a technical problem whose consequences are deterministic, so that we would obtain the same wrong answer in any repetition. This is common in computational geometry algorithms due to floating point errors. Geometric algorithms based on primitive queries that additionally work with faulty primitives are more modular and tolerant to errors.

In this model, two natural questions arise: (1) Given the answers to all possible primitive queries, what is the best possible solution one can find if the computational time is unlimited? (2) How good of a solution can be found efficiently, i.e., by using only a subset of the set of oracle answers?

Previous work. Braverman and Mossel [3] consider the sorting problem and provide the following answers to the questions above. With high probability, they construct the maximum likelihood permutation σ with respect to the set of oracle answers. They prove that permutation σ does not place any key more than $O(\log n)$ positions away from its true position. This fact allows the computation of σ using only $O(n \log n)$ key comparisons. The algorithm employs dynamic programming and has running time in $O(n^{3+24c_3})$ for some $c_3 > 0$ that depends on the error probability p.

Feige et al. [5] and Karp and Kleinberg [8] study the model where repeated queries are always independent. Searching games between questioner and a lying responder have been extensively studied in the past; see Pelc [11]. In his terminology, our model allows the responder random lies in response to nonrepetitive comparison questions in an adaptive game. Another related model is studied by Blum, Luby and Rubinfeld [2]. They consider self-testing and self-correction under the assumption that one is given a program P which computes a function f very quickly but possibly not very reliably. The difference with our approach is that we need to work with unreliable *primitives*.

In a series of papers culminating in work by Finocchi *et al* [6], a thorough study of resilient sorting, searching, and dictionaries in a faulty memory RAM model was performed. Here, at most δ memory words can be corrupted, while there is a small set of $O(1)$ memory words that are guaranteed not to be corrupted. The main difference is that only non-corrupted keys need to be sorted correctly. In contrast to this approach, we need to ensure that with high probability every key appears near its true position.

Our results. In this paper we assume the same model as in [3]. That is, primitive queries may fail independently with some probability at most p and noisy answers to all possible queries are pre-computed and available to the algorithm. While [3] only considers the problem of sorting, we initiate the study of a much wider class of problems in this model.

In Section 2 we show how to compute, with probability $1-f(p)$, the maximum of n elements in $O(n)$ time. Here $f(p)$ is a function independent of n that tends to 0 as p does. There is only a constant factor overhead in the running time for this problem in this model. In Section 3 a sorting algorithm is discussed. Like the noisy sorting algorithm presented by Braverman and Mossel [3], ours guarantees that each key is placed within distance $O(\log n)$ of its true position, even though we do not necessarily obtain the maximum likelihood permutation. Also, we need $O(n^2)$ rather than $O(n \log n)$ key comparisons. However, our algorithm is faster than the algorithm of [3], as the running time is $O(n^2)$, providing a considerable improvement over the $O(n^{3+24c_3})$ time of [3]. Finally, in Section 4, we briefly discuss the noisy search problem. By a random walk argument we show that the true position of a search key can be determined in a correctly sorted list with probability $(1-2p)^2$ if all n comparison queries are made. With $O(\log n \log \log n)$ comparisons, we can find the correct position with probability $1 - f(p)$. As an application of our max-finding result, we present in Section 5 an $O(n \log n)$ time algorithm for linear programming in two dimensions. Linear programming is a fundamental geometric problem which could suffer from errors in primitives due, e.g., to floating point errors. Our algorithm is modular, being built from simple point/line side comparisons, and robust, given that it can tolerate faulty primitives. It is based on an old technique of Megiddo [9]. We do not know how to modify more "modern" algorithms for linear programming to fit our error model. The correctness and running time hold with probability $1 - f(p)$.

2 Finding the Maximum of *n* numbers

We are given an unordered set $a_1, \ldots, a_n$ of n distinct numbers and we would like to output the maximum using only comparison queries. Each comparison between input elements fails independently with probability p. If the same comparison is made twice, the same answer is given.

Our algorithm LinearMax consists of two phases and each phase consists of several stages. In each stage the set of input numbers is pruned by a constant fraction such that with sufficiently high probability the maximum remains in the set. The pruning is done by sampling a set S and comparing each number outside of S with each number inside of S. During the first phase the size of the sample set increases with each stage. The second phase begins once the size of the set of numbers is successfully reduced to below some critical value. Then, we sample fewer elements in each stage so that the probability to accidently sample the maximum remains small. We will say that a stage has an error if either the maximum is removed during the stage or the input is not pruned by a constant fraction. If during a stage the input set is not sufficiently pruned, the algorithm stops and outputs "error". We remark that the purpose of this stopping rule is to simplify the analysis (this way, we make sure that any stage is executed only once and allows us directly to sum up error probabilities of different stages).

If the maximum is removed this will not be detected by the algorithm. In our analysis we derive a bound for the error probability of each stage and then use a union bound to bound the overall error probability.

In the proof we show that for any constant $1/2 > \lambda > 0$ there exists a constant $C = C(\lambda)$ such that for any $p \leq 1/64$, algorithm LinearMax succeeds with probability at least $1 - C \cdot p^{\frac{1}{2}-\lambda}$.

```
LinearMax($M, p$)
1.  $n = |M|$
2.  while $|M| \geq \max\{n^{1-\lambda}, \frac{100}{\sqrt{p}}\}$ do
3.      $j = 1$
4.      Select $i$ such that $n \cdot \left(\frac{15}{16}\right)^{i-1} \geq |M| > n \cdot \left(\frac{15}{16}\right)^{i}$
5.      while $j < 100 \cdot i \cdot \log(1/p)$ and $|M| > n \cdot \left(\frac{15}{16}\right)^{i+1}$ do
6.          Select a set $S$ of $s_i = 4i$ elements from $M$ uniformly at random
7.          Remove all elements in $S$ from $M$
8.          Compare all elements from $S$ with $M$
9.          Let $M$ be the list of elements larger than at least $\frac{3}{4} \cdot s_i$ elements of $S$
10.         $j = j + 1$
11.     if $|M| > n \cdot \left(\frac{15}{16}\right)^{i+1}$ then output "error" and exit
12. while $|M| \geq \frac{100}{\sqrt{p}}$ do
13.     $j = 1$
14.     Select $i$ such that $\left(\frac{16}{15}\right)^{i} \geq |M| > \left(\frac{16}{15}\right)^{i-1}$
15.     $i = i - \lceil \log_{16/15} \frac{100}{\sqrt{p}} \rceil + 1$
16.     while $j < 100 \cdot i \cdot \log(1/p)$ and $|M| > \left(\frac{16}{15}\right)^{i-1}$ do
17.         Select a set $S$ of $s_i = 4i$ elements from $M$ uniformly at random
18.         Remove all elements in $S$ from $M$
19.         Compare all elements from $S$ with $M$
20.         Let $M$ be the list of elements larger than at least $\frac{3}{4} \cdot s_i$ elements of $S$
21.         $j = j + 1$
22.     if $|M| > \left(\frac{16}{15}\right)^{i-1}$ then output "error" and exit
23. Run any maximum-finding algorithm to determine the maximum $m$ in $M$
```

Phase 1 of the algorithm begins at line 2 and Phase 2 begins at line 12. The stages of the phases correspond to the value of i (in Phase 2 this corresponds to the value of i after the subtraction in line 15).

We first analyze the expected running time of the algorithm. We observe that the running time is dominated by the number of comparisons the algorithm performs. In the first phase in stage i in each iteration of the loop the algorithm samples $4i$ elements uniformly at random and compares them to at most $n \cdot \left(\frac{15}{16}\right)^{i}$ elements of M. We will show that the expected number of loop iterations in each stage is $O(1)$. We need the following lemma.

Lemma 2.1. *The probability that in a fixed stage of Phase 1 or 2 the algorithm performs more than $100k$ iterations is at most $(\frac{1}{2})^k$.*

The proof of the lemma can be found in the full version. From the lemma, it follows immediately that the expected number of loops in a fixed stage is $O(1)$. By linearity of expectation, the overall expected running time for Phase 1 is $O\left(n \cdot \sum_{i=1}^{\infty} \mathbf{E}[\text{iterations in Stage } i] \cdot i \cdot \left(\frac{15}{16}\right)^{i-1}\right) = O(n)$. In order to analyze the second stage, let $i_0 = O(\log n)$ be the maximal value of i, i.e. the first value of i computed in line 15 when the algorithm enters Phase 2. We observe that for any stage $i \le i_0$ we have $\left(\frac{16}{15}\right)^i \le n^{1-\lambda}$. It follows that the expected running time of the second stage is $O\left(\sum_{i=1}^{i_0} \mathbf{E}[\text{iterations in Stage } i] \cdot i \cdot \left(\frac{16}{15}\right)^i\right) = O(n)$. The running time of the standard maximum search is also $O(n)$. Hence, the overall expected running time is $O(n)$. We continue by analyzing the error probability of the algorithm. An error happens at any stage if

(a) the maximum is contained in the sample set S,
(b) the maximum is reported to be smaller than $\frac{1}{4} \cdot s_i$ of the elements of S, or
(c) the while loop makes more than $100 i \log(1/p)$ iterations.

We start by analyzing (a) during the first phase. The error probability at each loop iteration is $|S|/|M|$. The number of items in M in stage i of phase 1 is at least $n \cdot \left(\frac{15}{16}\right)^i \ge n^{1-\lambda}$. We also have $i = O(\log n)$ and $n \ge \frac{100}{\sqrt{p}}$, which implies $|S|/|M| = O(\log n)/n^{1-\lambda}$. Summing up over the at most $O(i \log(1/p)) \le O(\log^2 n)$ loop iterations, we obtain that the overall error probability of item (a) in Phase 1 is at most $O(\log^3 n)/n^{1-\lambda} \le O(1)/n^{1-2\lambda}$. Using that $n \ge 100/\sqrt{p}$ implies $n^{1-2\lambda} \ge (100/p^{1/2})^{1-2\lambda} \ge 1/(p^{1/2-\lambda})$, we obtain that the overall error probability of item (a) in Phase 1 is at most $\frac{C}{5} \cdot p^{1/2-\lambda}$ for a sufficiently large constant $C > 0$. In the second phase, the error probabiliy is at most

$$\sum_{i=1}^{\infty} 4 \cdot i \cdot \left(\frac{15}{16}\right)^{i + \lceil \log_{16/15} \frac{100}{\sqrt{p}} \rceil - 1} \le \frac{4\sqrt{p}}{100} \cdot \sum_{i=1}^{\infty} i \cdot \left(\frac{15}{16}\right)^{i-1} \le \frac{C}{5} \cdot \sqrt{p},$$

for sufficently large constant $C > 0$.

We continue by analyzing (b). Here, an error occurs in stage i if at least $\frac{1}{4} \cdot s_i$ comparisons fail. By the following lemma, this probability is small.

Lemma 2.2. *Let $1 \ge p \ge 0$ be the failure probability of a comparison. Let $k > 0$ be a multiple of 4. The probability that at least $k/4$ out of k comparisons fail is at most $(4ep)^{k/4}$.*

We also prove this lemma in the full version. Since, $p \le 1/64$, it follows that for $C > 0$ sufficiently large, the probability of failure in each phase is $\sum_{i=1}^{O(\log n)} 100 i \cdot \log(1/p) \cdot (4ep)^i \le \frac{C}{5} \cdot \sqrt{p}$. Next, we analyze (c). By Lemma 2.1 and since $p \le 1/64$, we have that the error probability for this item is bounded by $\sum_{i=0}^{\infty} \left(\frac{1}{2}\right)^{i \cdot \log \frac{1}{p}} = \sum_{i=0}^{\infty} p^i \le \frac{C}{5} \cdot \sqrt{p}$, for $C > 0$ a sufficiently large constant. Finally, we consider the error probability of the standard maximum search. A maximum search over a set of n items uses $n - 1$ comparisons. If $n \le 100/\sqrt{p}$ then the expected number of errors is at most $100p/\sqrt{p} = 100\sqrt{p}$. Let X be the random variable for the number of errors. Since the number of errors is integral, by Markov's inequality we get that the probability of error is

$$\mathbf{Pr}[\text{at least one error occurs}] \le \mathbf{Pr}[X \ge \frac{1}{100\sqrt{p}} \cdot \mathbf{E}[X]] \le 100\sqrt{p} \le \frac{C}{5} \cdot \sqrt{p}$$

for C a sufficiently large constant. Summing up all errors yields an overall error probability of at most $C \cdot \sqrt{p}$.

3 Sorting

In the following section we are given a set S of n distinct keys and the comparison relation $<_E$ with errors. We present an algorihm *SortWithBuckets* that computes an output sequence such that the position of every key in this sequence differs from its rank by at most an additive error of $O(\log n)$. In the following, we use $\text{rank}(x, R)$ to refer to the (true) rank of input key x within $R \subseteq S$, i.e., $1 + |\{y \in R : y < x\}|$. We also use $\text{rank}_E(x, R)$ to refer to the *virtual* rank of x with respect to a set R, i.e., $1 + |\{y \in R : y <_E x\}|$. Note that there may be more than one key having the same virtual rank.

The algorithm. In the following, we will assume that n is a power of 2. If this is not the case, we can add additional special items which will be assumed to be larger than any input key and run our algorithm on the modified input. Here we may assume no errors in the comparisons as the algorithm can keep track of these items. The algorithm will partition the set $\{1, \dots, n\}$ into buckets each corresponding to a set of 2^i consecutive numbers for certain i. We call this set the *associated range* of the bucket. At the beginning we will assume that there is a single bucket with associated range $\{1, .., n\}$. In the next step, we will subdivide this bucket into two bucket, with associated ranges $\{1, \dots, n/2\}$ and $\{n/2 + 1, \dots, n\}$, respectively. Then the two resulting buckets are further subdivided into four buckets, and so on. The algorithm stops, if the associated ranges contain $O(\log n)$ numbers. Ideally, we would like our algorithm to maintain the invariant that each bucket contains all input numbers whose ranks are in its associated range, e.g., the bucket corresponding to numbers $\{1, \dots, 2^i\}$ is supposed to contain the 2^i smallest input numbers. Due to the comparison errors, the algorithm cannot exactly maintain this invariant. However, we can almost maintain it in the sense that an item with rank k is either in the bucket whose associated range contains k or it is in one of the neighboring buckets.

This is done as follows. Let us consider a set of buckets B_j with associated ranges of 2^i numbers such that bucket B_j has associated range$\{(j-1)2^i + 1, \dots, j2^i\}$. Now assume that the input numbers have been inserted into these buckets in such a way that our relaxed invariant is satisfied. For each bucket B_j let us use S_j to denote the set of input keys inserted into B_j. Now we would like to refine our buckets, i.e., insert the input numbers in buckets B'_j with associated ranges $\{(j-1)2^{i-1} + 1, j2^{i-1}\}$. This is simply done by inserting an item that is previously in bucket B_j into bucket B'_r, where

$$r = \lfloor \frac{\mathrm{rank}_E(x, \bigcup_{j-2 \le k \le j+2} S_j) + \left|\bigcup_{k<j-2} S_j\right|)}{2^{i-1}} \rfloor .$$

Thus, the algorithm computes the cardinality of the buckets up to B_{j-3} and adds to it the virtual rank of x with respect to buckets $B_{j-2}, \ldots, B_{j+2}$. The idea behind this approach is that $\mathrm{rank}(x, S)$ can be written as $\left|\bigcup_{k<j-3} S_j\right| + rank(x, \bigcup_{j-2 \le k \le j+2} S_j)$ since, by our relaxed invariant, the keys in buckets $1, \ldots, j-3$ are smaller than x and the keys in $j+3, \ldots$ are larger than x. Now, since we do not have access to $\mathrm{rank}(x, \bigcup_{j-2 \le k \le j+2} S_j)$ we approximate it by $\mathrm{rank}_E(x, \bigcup_{j-2 \le k \le j+2} S_j)$. Since the rank involves fewer elements when the ranges of the buckets decrease, this estimate becomes more and more accurate.

Analysis. Thus, it remains to prove that the relaxed invariant is maintained. The difficulty in analyzing this (and other) algorithms is that there are many dependencies between different stages of the algorithm. In our case, the bucket of an element x is highly dependent on the randomness of earlier iterations. Since analyzing such algorithmic processes is often close to impossible, we use a different approach. We first show that certain properties of the comparison relation $<_E$ hold for certain sets of elements with high probability. Then we show that these properties already suffice to prove that the algorithm sorts with additive error $O(\log n)$. The proof follows using Chernoff bounds and can be found in the full version.

Lemma 3.1. *Let $p \le 1/20$. Let $R \subseteq S$ be a set of $k = 9 \cdot 2^i \ge 100000 \log n$ keys and let $x \in R$. Let $X = |\{y \in R : x < y \text{ and } y <_E x\}| + |\{y \in R : x > y \text{ and } y >_E x\}|$ be the number of false comparisons of x with elements from R. Then $\mathbf{Pr}[X \ge 2^{i-1}] \le n^{-10}$.*

Corollary 3.1. *For $\log(100000 \log n) \le i \le \log n$ and $1 \le j \le n/2^i - 8$ let $S_{ij} = \{x \in S : (j-1) \cdot 2^i + 1 \le \mathrm{rank}(x) \le (j+8) \cdot 2^i\}$. With probability at least $1 - 1/n^8$ we have that every $x \in S_{ij}$ has less than 2^{i-1} comparison errors with elements from S_{ij}.*

Proof: We apply Lemma 3.1 for each S_{ij}. The number of choices for indices i, j is bounded by n^2. Hence by the union bound, the probability that there is an error in any of the set is at most n^{-8}. ⊔⊓

We now claim that if the comparison relation $<_E$ satisfies Corollary 3.1 then our algorithm computes a sequence such that any element deviates from its true rank by at most $O(\log n)$. In order to prove this claim, let us assume that our relaxed invariant is maintained for a set of buckets B_j with associated ranges of size 2^i. Let x be an element and j^* be the bucket whose associated range contains x. By our relaxed invariant, x is either in bucket B_{j^*-1}, B_{j^*} or B_{j^*+1}. Hence, to sort x into the next finer bucket, the algorithm inspects (a subset of) buckets $B_{j^*-3}, \ldots, B_{j^*+3}$. By our relaxed invariant, these buckets only contain elements x with $\mathrm{rank}(x) \in S_{i\ell}$ with $\ell = j^* - 4$. Now, in order to sort x into a finer bucket, we compare x with a subset of elements from $S_{i\ell}$. Therefore, the number of errors in this comparison is certainly bounded by the number of errors within $S_{i\ell}$, which is less than 2^{i-1}. Since the next finer buckets have associated ranges of size 2^{i-1}, this implies that our relaxed invariant will be maintained. We summarize our results in the following theorem.

Theorem 3.1. *Let $p \leq 1/20$. There is a tolerant algorithm that given an input set S of n numbers computes in $O(n^2)$ time and with probability at least $1 - 1/n^8$ an output sequence such that the position of every element in this sequence deviates from its rank in S by at most $O(\log n)$.*

4 Searching

In this section we assume that we are given a sorted sequence of keys $a_1 < a_2 < \ldots < a_n < a_{n+1} = \infty$, and a query key q. We know that the sorting is accurate, and that q is different from all a_i. Our task is to determine rank(q), the smallest index i such that $q < a_i$ holds. A noisy oracle provides us with a table containing answers $q <_E a_i$ or $q >_E a_i$ to all possible key comparisons between q and the numbers a_i. Each of them is wrong independently with probability $p_i \leq p < 1/2$.

Let conf(j) denote the number of table entries that would be in conflict with rank$(q) = j$. Our first algorithm, Search, takes $O(n)$ time to read the whole table and to output a position j that minimizes conf(j); ties are broken arbitrarily. By the same argument as in Braverman and Mossel [3], Search reports a maximum likelihood position for q, given the answer table. As opposed to the sorting problem, we can prove a lower bound to the probability that Search correctly computes the rank of q.

Theorem 4.1. Search *reports* rank(q) *with probability at least* $(1 - 2p)^2$.

Proof: We want to argue that positions j to the left or to the right of rank(q) are less likely to get reported because they have higher conflict numbers. By definition,

$$\mathrm{conf}(j) = \begin{cases} \mathrm{conf}(j-1) + 1, \text{ if } & q <_E a_{j-1} \\ \mathrm{conf}(j-1) - 1, \text{ if } & q >_E a_{j-1} \end{cases}$$

holds, as can be quickly verified. Let us first consider the indices $j = \mathrm{rank}(q), \ldots, n$ to the right of rank(q). It makes our task only harder to assume that each oracle answer is wrong with maximum probability p. Thus, the oracle's process of producing these answers, for indices j increasing from rank(q) to n, corresponds to a *random walk* of the value of conf(j) through the integers, starting from conf(rank(q)). With probability $1 - p$, the value of conf(j) will increase by 1 (since $q < a_j$ holds, by assumption), and with probability p decrease. With probability $\geq (1-p) \cdot (1 - \frac{p}{1-p}) = 1 - 2p$, conf$(j)$ will increase in the first step *and* never sink below this value again; see the full version for an easy proof of this fact. Thus, with probability $\geq 1 - 2p$, all j to the right of rank(q) will have values conf(j) higher than conf(rank(q)) and, therefore, not get reported. A symmetric claim holds for the indices j to the left of rank(q). Consequently, Search does with probability $\geq (1-2p)^2$ report rank(q). ⊔⊓

Theorem 4.1 casts some light on what can be achived utilizing all information available. If sequence $a_1, \ldots, a_n$ is given as a linear list, the $O(n)$ time algorithm SEARCH is of practical interest, too. For a sorted sequence stored in an array, we have a more efficient tolerant binary search algorithm based on SEARCH. The proof of Theorem 4.2 can be found in the full version.

Theorem 4.2. *Given any constant error bound $p \in (0, 1/4)$ we can in time $O(\log n \cdot \log\log n)$ compute the rank of an element in a sorted list of length n with probability at least $1 - f(p)$. Here function $f(p)$ goes to 0 as p goes to 0.*

5 Linear Programming in 2 Dimensions

As an application of our LINEARMAX algorithm, we consider the linear programming problem in two dimensions, namely, the problem of minimizing a linear objective function subject to a family $\mathcal{F}$ of half-plane constraints. We assume our problem is in standard form [4], namely that the problem is to find the lowest (finite) point in the non-empty feasible region, defined by the intersection of a non-degenerate family $\mathcal{F}$ of n half-planes. Define a *floor* to be a half-plane including all points in the plane above a line, whereas a *ceiling* is a half-plane including all points in the plane below a line. We say a half-plane is *vertical* if the line defining it is vertical. In the standard setting, we can assume that vertical lines have been preprocessed and replaced with the constraint $L \leq x \leq R$ for reals L and R, and that no two ceilings and no two floors are parallel. The half-planes are given in a sorted list according to their slope. Our algorithm is based on several basic geometric primitives which can make mistakes.

Error Model: We are given a few black boxes that perform side comparisons.

The first box is SIDECOMPARATOR, which is given four lines ℓ_A, ℓ_B, ℓ_C, and ℓ_D, and decides if the the intersection of ℓ_A and ℓ_B is to the left of the intersection of ℓ_C and ℓ_D. This description can be simplified to the following: "given two points, is one to the left of the other"? (however, since the input does not contain explicit points, we have chosen to describe the test in this more abstract way).

The second box is VERTICALCOMPARATOR, which is given four lines ℓ_A, ℓ_B, ℓ_C, and ℓ_D, and returns the line in the set $\{\ell_C, \ell_D\}$ whose signed vertical distance is larger from the intersection point of ℓ_A and ℓ_B. This description can be simplified to the following: "given a point and two lines, which line is closer in vertical distance"? (again, since the input does not contain explicit points, we choose to describe the test in this more abstract way).

More precisely, if the intersection of ℓ_A and ℓ_B is the point (α, β), and we draw the line $x = \alpha$, then we look at the signed distance from (α, β) to the intersection point of ℓ_C and $x = \alpha$ as well as the intersection point of ℓ_D and $x = \alpha$. Here, by signed, we mean that if (α, β) is above one of these intersection points, then its distance to that point is negative, otherwise it is non-negative. If the signed distances are the same, i.e., the lines ℓ_C and ℓ_D meet $x = \alpha$ at the same point, VERTICALCOMPARATOR reports this.

Such primitives are basic, and play an essential role in geometric algorithms.

We assume the primitives have a small error probability p of failing. In the case of SideComparator, the box reports the opposite side with probability p. In the case of VerticalComparator, the box reports the further line (in signed vertical distance), or fails to detect if two lines have the same signed distance, with probability p. Multiple queries to the tester give the same answer.

The output of our algorithm is not given explicitly, but rather is specified as the intersection of two half-planes in $\mathcal{F}$ (since this is all that can be determined given abstract access to the input lines).

Theorem 5.1. *There is an algorithm* LP *that terminates in $O(n \log n)$ time with probability at least $1 - 1/n$. The correcntess probability is $1 - f(p)$ where $f(p)$ approaches 0 as p approaches 0.*

We now review Megiddo's algorithm and then describe our main new ideas.

Megiddo's Algorithm: Megiddo's algorithm defines: $g(x) = \max\{a_i x + b_i \mid y \geq a_i x + b_i \text{ is a floor}\}$, and $h(x) = \min\{a_i x + b_i \mid y \leq a_i x + b_i \text{ is a ceiling}\}$. A point x is feasible iff $g(x) \leq h(x)$, that is, if it is above all floors and below all ceilings. The algorithm has $O(\log n)$ stages. The number of remaining constraints in $\mathcal{F}$ in the i-th stage is at most $(7/8)^i \cdot n$. If at any time there is only a single floor g in $\mathcal{F}$, then the algorithm outputs the lowest point on g that is feasible. Otherwise it arbitrarily groups the floors into pairs and the ceilings into pairs, and computes the pair (ℓ_A, ℓ_B) whose intersection point has the median x-coordinate of all intersection points of all pairs.

The algorithm checks if the intersection point (α, β) of ℓ_A and ℓ_B is feasible, i.e., if $g(\alpha) \leq h(\alpha)$. The algorithm then attempts to determine if the optimum is to the right or the left of (α, β). It is guaranteed there is a feasible solution - an invariant maintained throughout the algorithm - and only one side of (α, β) contains a feasible point if (α, β) is infeasible. Megiddo defines the following:
$s_g = \min\{a_i \mid y \geq a_i x + b_i \text{ is a floor and } g(\alpha) = a_i\alpha + b_i\}$,
$S_g = \max\{a_i \mid y \geq a_i x + b_i \text{ is a floor and } g(\alpha) = a_i\alpha + b_i\}$,
$s_h = \min\{a_i \mid y \leq a_i x + b_i \text{ is a ceiling and } h(\alpha) = a_i\alpha + b_i\}$,
$S_h = \max\{a_i \mid y \leq a_i x + b_i \text{ is a ceiling and } h(\alpha) = a_i\alpha + b_i\}$.

Suppose first that (α, β) is infeasible. This means that $g(\alpha) > h(\alpha)$. Then if $s_g > S_h$, any feasible x satisfies $x < \alpha$. Also, if $S_g < s_h$, then any feasible x satisfies $x > \alpha$. The last case is that $s_g - S_h \leq 0 \leq S_g - s_h$, but this implies the LP is infeasible, contradicting the above invariant.

Now suppose that (α, β) is feasible. As Megiddo argues, if $g(\alpha) < h(\alpha)$ then if $s_g > 0$, then the optimal solution is to the left of α. Also, if $S_g < 0$ then the optimal solution is to the right of α. Otherwise $s_g \leq 0 \leq S_g$, and (α, β) is the optimal solution. Finally, if $g(\alpha) = h(\alpha)$, then if (1) $s_g > 0$ and $s_g \geq S_h$, then the optimum is to the left of α, or if (2) $S_g < 0$ and $S_g \leq s_h$, then the optimum is to the right of α. Otherwise (α, β) is the optimum.

Hence, in $O(n)$ time, the algorithm finds the optimum or reduces the solution to the left or right of (α, β). In this case, in each of the pairs of constraints on the other side of α, one of the two constraints can be removed since it cannot participate in defining the optimum. When there are a constant number of constraints left, the algorithm solves the resulting instance by brute force.

Intuition of Our Algorithm: Let Π be a partition of the set $\mathcal{F}$ of input floors and ceilings into pairs. Inspired by our maximum-finding algorithm, instead of computing the median of pairs of intersection points, we randomly sample a set S of $\Theta(\log n)$ pairs from Π. For each pair in Π, we find if the optimum is to the left or right of the intersection point. If (α, β) is the intersection point of a pair of constraints in Π, we useVerticalComparator to find the *lower floor* $\{y \geq a_i x + b_i$ is a floor and $g(\alpha) = a_i\alpha + b_i\}$ as well as the *upper ceiling* $\{y \leq a_i x + b_i$ is a ceiling and $h(\alpha) = a_i\alpha + b_i\}$. By non-degeneracy, each of these sets has size at most 2. We modify our earlier linearmax algorithm to return the maximum two items (i.e., constraints) instead of just the maximum, using VerticalComparator to perform the comparisons. By a union bound, we have the lower floor and upper ceiling with large probability. Using the slope ordering of s_g, S_g, s_h, and S_h we know which side of (α, β) the optimum is on.

To avoid performing the same comparison twice, in any phase each primitive invocation has as input at least one of the lines in our sample set. Since we discard the sample set after a phase, comparisons in different phases are independent. To ensure that comparisons in the same phase are independent, when computing the upper ceiling and lower floor of a sampled intersection point (α, β), we do not include the other sampled pairs of constraints in the comparisons. This does not introduce errors, with high probability, since we have only $\Theta(\log n)$ randomly sampled constraints, while the union of the upper ceiling and lower floor of an intersection point has at most four constraints, and so it is likely the upper ceilings and lower floors of all sampled pairs are disjoint.

Since the sample size is $O(\log n)$, we can show that with high probability we throw away a constant fraction of constraints in each phase, and so after $O(\log n)$ recursive calls the number of remaining constraints is bounded as a function of p alone. The total time is $O(n \log n)$. Our main algorithm is described below.

```
LP($\mathcal{F}, p$)
1.  If there are at most poly($1/p$) constraints in $\mathcal{F}$ solve the problem by brute force.
2.  If there is at most one floor constraint $f \in \mathcal{F}$, if the slope of $f$ is positive,
    output the intersection of $f$ and the line $x = L$. If the slope of $f$ is negative,
    output the intersection of $f$ and the line $x = R$.
3.  Otherwise, randomly partition the floors into pairs, as well as the ceilings
    into pairs (possibly with one unpaired floor and one unpaired ceiling).
    Let the set of pairs be denoted $\Pi$.
    Draw a set $S$ of $\Theta(\log|\mathcal{F}|)$ pairs of constraints from the pairs in $\Pi$
    uniformly at random, without replacement.
4.  Let $\Phi_F$ be the set of floors in $\mathcal{F}$, excluding those in $S$.
    Let $\Phi_C$ be the set of ceilings in $\mathcal{F}$, excluding those in $S$.
5.  For each pair $(\ell_A, \ell_B)$ of constraints in $S$,
a.      Let $U(\ell_A, \ell_B)$ =Lowest($\ell_A, \ell_B, \Phi_F, p$).
b.      Let $L(\ell_A, \ell_B)$ =Highest($\ell_A, \ell_B, \Phi_C, p$).
c.      Compute Tester($\ell_A, \ell_B, U(\ell_A, \ell_B), L(\ell_A, \ell_B)$).
```

```
d.        Let $T \subseteq S$ be the pairs for which TESTER does not output “fail”, and compute
          the majority output direction $dir$ of the result of TESTER on the pairs in $T$.
12.     For each pair $(\ell_A, \ell_B)$ of constraints in $\Pi \setminus S$,
13.         For each pair $(\ell_C, \ell_D) \in S$, compute SIDECOMPARATOR$(\ell_A, \ell_B, \ell_C, \ell_D)$.
14.             If for at least a $2/3$ fraction of pairs in $S$, the pair $(\ell_A, \ell_B)$ is to the right
                (resp. to the left), and if $dir$ is to the left (resp. to the right), then
                remove the constraint in the pair $(\ell_A, \ell_B)$ from $\mathcal{F}$ that cannot
                participate in the optimum assuming the optimum is really
                to the left (resp. to the right) of the pair $(\ell_A, \ell_B)$.
15.     Return $LP(\mathcal{F} \setminus S, p)$.
```

We defer the analysis to the full version. The subroutines LOWEST, HIGHEST, and TESTER are also described there. Intuitively, LOWEST finds the upper envelope[4] of a point (that is, the lowest ceilings), and HIGHEST finds the lower envelope (the highest floors). TESTER tests which side of the optimum the point is on based on the slope information in the union of upper and lower envelopes.

References

1. M. Ajtai,V. Feldman,A.Hassidim, and J. Nelson Sorting and Selection with Imprecise Comparisons. ICALP (1), pp. 37–48, 2009.
2. M. Blum,M. Luby, and R. Rubinfeld. Self-Testing/Correcting with Applications to Numerical Problems. JCSS, 47(3):549-595, 1993.
3. M. Braverman and E. Mossel. Noisy Sorting Without Resampling. Proc. 19th Annual ACM-SIAM Symp. on Discrete Algorithms (SODA'08), pp. 268–276, 2008.
4. K. L. Clarkson. Las Vegas Algorithms for Linear and Integer Programming when the Dimension is Small, J. ACM 42(2), 1995, pp. 488–499.
5. U. Feige, D. Peleg, P. Raghavan, and E. Upfal. Computing with Unreliable Information. Proceedings 22nd STOC, 1990, pp. 128–137.
6. I. Finocchi, F. Grandoni, and G. Italiano. Resilient Dictionaries. ACM Transactions on Algorithms 6(1), pp. 1–19, 2009.
7. R. L. Graham, D. E. Knuth, and O. Patashnik. Concrete Mathematics. Addison-Wesley, 2nd edition, 1994.
8. D. Karp and R. Kleinberg. Noisy Binary Search and Applications. 18th SODA, 2007, pp. 881–890.
9. N. Megiddo. Linear Programming in Linear Time When the Dimension Is Fixed, J. ACM 31(1), 1984, pp. 114–127.
10. M. Mitzenmacher and E. Upfal. Probability and Computing : Randomized Algorithms and Probabilistic Analysis. Cambridge University Press, January 2005.
11. A. Pelc. Searching Games with Errors - Fifty Years of Coping with Liars. Theoretical Computer Science 270(1-2), 2002, pp. 71–109.
12. S. Schirra. Robustness and Precision Issues in Geometric Computation. In J.-R. Sack and J. Urrutia (Eds.) *Handbook of Computational Geometry*, pp. 597–632, Elsevier, 2000.
13. R. Seidel. Small-Dimensional Linear Programming and Convex Hulls Made Easy, Discrete & Computational Geometry(6), 1991, pp. 423–434.

[4] Sometimes envelope refers to all boundary lines that touch the convex hull. Here we use it to simply refer to the extremal two constraints of a point.