Electronic Colloquium on Computational Complexity, Report No. 103 (2017)

Parameterized Property Testing of Functions[*][†]

Ramesh Krishnan S. Pallavoor[‡] Sofya Raskhodnikova[‡] Nithin Varma[‡]

June 12, 2017

Abstract

We investigate the parameters in terms of which the complexity of sublinear-time algorithms should be expressed. Our goal is to find input parameters that are tailored to the combinatorics of the specific problem being studied and design algorithms that run faster when these parameters are small. This direction enables us to surpass the (worst-case) lower bounds, expressed in terms of the input size, for several problems. Our aim is to develop a similar level of understanding of the complexity of sublinear-time algorithms to the one that was enabled by research in parameterized complexity for classical algorithms.

Specifically, we focus on testing properties of functions. By parameterizing the query complexity in terms of the size r of the image of the input function, we obtain testers for monotonicity and convexity of functions of the form $f : [n] \to \mathbb{R}$ with query complexity $O(\log r)$, with no dependence on n. The result for monotonicity circumvents the $\Omega(\log n)$ lower bound by Fischer (Inf. Comput., 2004) for this problem. We present several other parameterized testers, providing compelling evidence that expressing the query complexity of property testers in terms of the input size is not always the best choice.

1 Introduction

In this paper, we set out to investigate the parameters in terms of which the complexity of sublinear-time algorithms should be expressed. Our goal is to find input parameters that are tailored to the combinatorics of the specific problem being studied and design algorithms that run faster when these parameters are small. This direction could enable one to surpass the (worst-case) lower bounds on the problem complexity that are usually expressed in terms of the input size. The spirit of our study is similar to that in the field of parameterized complexity. In parameterized complexity, the focus is on expressing the complexity of problems as a function of one or more input parameters in order to obtain a fine-grained complexity classification, for example, of NP-hard problems. Our aim is to develop a similar level of understanding of the complexity of sublinear-time algorithms to the one that was enabled by research in parameterized complexity for classical algorithms.

We focus our study on the framework of property testing, introduced by Goldreich et al. [29] and Rubinfeld and Sudan [42]. In property testing, an algorithm (an ε-tester) for property $\mathcal{P}$, where $\mathcal{P}$ is viewed as a class of functions, is given a parameter $\varepsilon \in (0, 1)$ as input and has oracle

[*]This work was supported by NSF grant CCF-1422975; the third author was also supported by Pennsylvania State University College of Engineering Fellowship and Pennsylvania State University Graduate Fellowship.

[†]A preliminary version of this work appeared in the proceedings of ITCS 2017 [38].

[‡]Pennsylvania State University, rxp271@cse.psu.edu, sofya@cse.psu.edu, nithvarma@psu.edu.

ISSN 1433-8092

access to a function f. The tester has to accept with probability at least 2/3 if f belongs to the class $\mathcal{P}$, and reject with probability at least 2/3 if f is ε*-far* from $\mathcal{P}$, that is, differs from every function in $\mathcal{P}$ on at least an ε fraction of function values. In the context of property testing of functions, the query complexity of a tester is usually expressed in terms of ε and the size of the domain of the input function. This works well for properties whose query complexity depends only on the proximity parameter ε. However, for other properties, it is not clear whether the domain size is the right parameter to express their testing complexity.

Consider, for example, the widely studied problem of testing monotonicity of real-valued functions (see, e.g., [28, 23, 24, 37, 27, 25, 32, 1, 33, 2, 11, 10, 13, 16, 12, 9, 17, 18, 15, 20, 19, 36, 4, 5, 22], and recent surveys [40, 14]). For functions over a discrete domain $[n]$ (also called the line), monotonicity testing is equivalent to testing sortedness of arrays. Algorithms for sortedness testing have found use, for instance, in determining the "state of sortedness" of relational databases [6], where the testing step is performed to decide on the sorting algorithms to be run on the database. The complexity of sortedness testing (for constant ε) is $\Theta(\sqrt{n})$ if the tester is only allowed to make independent and uniformly random queries [27]; it is $\Theta(\log n)$ if the tester is allowed to make arbitrary queries [24, 25].

From the above discussion, it might appear that one cannot make any more improvements to the complexity of monotonicity testing over $[n]$. However, we argue that this is the case only when the complexity of the problem is parameterized in terms of n, the domain size.

In this work, we ask whether better monotonicity testers can be designed by parameterizing the query complexity in terms of the size of the image of the input function. The starting point for our investigation is the folklore result that, for ε-testing monotonicity of Boolean functions over $[n]$, only $O(1/\varepsilon)$ queries suffice. A slightly more general corollary of this result is that monotonicity of functions over $[n]$ with image size at most 2 can be ε-tested with only $O(1/\varepsilon)$ queries. The only bound for monotonicity testing (over $[n]$) that is expressed in terms of the image size r of the input function is the bound of $\Omega(\log r)$ for nonadaptive[1] testers due to Blais et al. [12]. We design an ε-tester for monotonicity of functions over $[n]$ with query complexity $O((\log r)/\varepsilon)$, where r is an upper bound on the size of the image of the input function. This result circumvents Fischer's lower bound of $\Omega(\log n)$ for this problem by focusing on a different parameter for measuring query complexity.

The size of the image is one of the natural parameters in terms of which one can express the complexity of property testing algorithms. In this work, we show that there are several testing problems for which parameterizing the complexity in terms of the image size works well. Another example where parameterization has helped in the design of efficient testers is the work of Jha and Raskhodnikova [35] on Lipschitz testing, even though they do not view their results from this angle. The complexity of their testers is expressed in terms of the *image diameter*. The *image diameter* of a function $f : \mathcal{D} \mapsto \mathbb{R}$ is $\max_{x,y\in\mathcal{D}} |f(x) - f(y)|$. In many situations, the image diameter is much smaller than the domain size. We believe that all this evidence is compelling enough to make one rethink the way in which the complexity of sublinear-time algorithms is expressed. Our paper is a first step towards formalizing this notion and finding what we think are the right parameters to express the complexity of some central problems in sublinear-time algorithms.

[1]Testers whose queries do not depend on the answers to previous queries are called *nonadaptive*; general testers that do not satisfy this requirement are *adaptive*.

1.1 Parameters and Properties Studied in this Work

We study the dependence of complexity of monotonicity and convexity testers on the image size of the input functions. The image of a function is defined as follows.

Definition 1.1 (Image of a function)**.** *Let f be a function defined over a finite, discrete domain $\mathcal{D}$. The image of f, denoted $Im(f)$, is the set $\{f(x) : x \in \mathcal{D}\}$ or, in other words, the set of all values taken by f on points in $\mathcal{D}$.*

For the special case, when $\mathcal{D}$ is $[n]$, a function $f : [n] \mapsto \mathbb{R}$ can also be viewed as a real-valued array of length n. Here, $Im(f)$ is equal to the set of distinct values in the array.

We restrict our attention to real-valued functions defined over the following domains. These are domains for which testing monotonicity and convexity have been studied extensively.

Definition 1.2 (Hypergrid, Hypercube, Line)**.** *For $x \in [n]^d$, let x_i denote the i^{th} coordinate of x. A* hypergrid *is a partial order $([n]^d, \preceq)$ where $x \preceq y$ means that $x_i \leq y_i$ for all $x, y \in [n]^d$ and $i \in [d]$. The partial order $([2]^d, \preceq)$ is called a* hypercube *and the total order $([n], \preceq)$ is called a* line*.*

Next, we summarize some of the previous work on testing monotonicity and convexity of real-valued functions.

Monotonicity. A function $f : \mathcal{D} \mapsto \mathbb{R}$ defined over a partial order $(\mathcal{D}, \preceq)$ is *monotone* if $f(x) \leq f(y)$ for all $x, y \in \mathcal{D}$ satisfying $x \preceq y$. Monotonicity is one of the most widely studied properties in the field of property testing [28, 23, 24, 37, 27, 25, 32, 1, 33, 2, 11, 10, 13, 16, 12, 9, 17, 18, 15, 20, 19, 36, 4, 5, 22]. The complexity of ε-testing monotonicity of functions of the form $f : [n]^d \mapsto \mathbb{R}$ is $\Theta\left(\frac{d \log n}{\varepsilon}\right)$ [16, 17]. For the special case of the line, the testing complexity is $\Theta\left(\frac{\log n}{\varepsilon}\right)$ [24, 25]. For functions defined over general poset domains $\mathcal{D}$, the complexity of monotonicity testing is $O\left(\sqrt{|\mathcal{D}|/\varepsilon}\right)$ [27].

Convexity. For a convex set $\mathcal{D}$, a function $f : \mathcal{D} \mapsto \mathbb{R}$ is convex if $\forall x, y \in \mathcal{D}$ and $t \in [0, 1]$, $f(tx + (1-t)y) \leq tf(x) + (1-t)f(y)$. For real-valued functions over $[n]$, convexity can be ε-tested using $O(\frac{\log n}{\varepsilon})$ queries [39]. This bound is tight (for constant ε) for nonadaptive testers [12].

1.2 Our Results

In this section, we describe the key technical contributions of our work. We design efficient testers for monotonicity over various hypergrid domains and convexity over the line. For monotonicity of functions over the line $[n]$, which is equivalent to the property of sortedness of arrays of length n, we design efficient testers under two different models of input access: (i) query access and (ii) uniform samples. Our testers are given an upper bound r on the image size of the input function, and their complexity is parameterized in terms of r. In addition to analyzing query (sample) complexity of our algorithms, we also analyze their running times in the model where each oracle query takes unit time.

Sortedness testing. We present our tester for sortedness of n-element arrays (monotonicity over the line $[n]$) in Section 3. The complexity of our tester is independent of n. Our tester has 1-sided error, that is, it always accepts a function with the property. (In contrast, the general tester is said to have 2-sided error.) We prove the following theorem.

Theorem 1.3. *There exists a 1-sided error ε-tester making $O\left(\frac{\log r}{\varepsilon}\right)$ queries to test sortedness of arrays with at most r distinct values. The tester runs in time $O\left(\frac{\log r}{\varepsilon}\right)$.*

An important ingredient in our sortedness tester is a nearly optimal nonadaptive tester for this task, presented in Section 2. Its performance is summarized in the next theorem.

Theorem 1.4. *There exists a nonadaptive, 1-sided error ε-tester making $O\left(\frac{1}{\varepsilon}\log\frac{r}{\varepsilon}\right)$ queries to test sortedness of arrays with at most r distinct values. The tester runs in time $O\left(\frac{1}{\varepsilon}\log\frac{r}{\varepsilon}\right)$.*

The query complexity of our nonadaptive tester matches (for constant ε) the $\Omega(\log r)$ lower bound for nonadaptive sortedness testers in [12]. Note that for $r \geq 1/\varepsilon$, the complexity of the nonadaptive tester is $O\left(\frac{\log r}{\varepsilon}\right)$. The tester that we design to prove Theorem 1.3 runs the nonadaptive tester for $r \geq 1/\varepsilon$ and a different (adaptive) tester, presented in Section 3, for $r < 1/\varepsilon$.

Uniform sortedness testing. The work that defined property testing [29], in addition to the model with oracle access to the input, also considered testers that are allowed access to function values only at points sampled uniformly and independently at random from the domain. This model of property testing, known as *uniform* or *sample-based* testing, was further studied by Goldreich and Ron [31], Fischer et al. [26], Berman et al. [8] and Berman et al. [7]. The query complexity of ε-testing sortedness of n-element arrays (for constant ε) using only uniformly and independently drawn samples is $\Theta(\sqrt{n})$ [27]. We design optimal (up to the dependence on ε) uniform testers whose query complexity is parameterized in terms of the number or distinct elements in the input arrays. These results can be found in Sections 5 and 6.

Theorem 1.5. *There exists a 1-sided error ε-tester that makes $O(\sqrt{r}/\varepsilon)$ uniform and independent queries to test sortedness of arrays with at most r distinct values. The tester runs in time $O(\sqrt{r}/\varepsilon)$.*

Theorem 1.6. *Testing sortedness of arrays with values in $[r]$ requires $\Omega(\sqrt{r})$ uniform queries, even with 2-sided error.*

Monotonicity testing over hypergrids. We present our tester for monotonicity of real-valued functions over hypergrid domains in Section 4 and prove the following theorem.

Theorem 1.7. *There exists a 1-sided error ε-tester that makes $O\left(\frac{d}{\varepsilon}\log\frac{d}{\varepsilon}\log r\right)$ queries to test monotonicity of real-valued functions $f : [n]^d \mapsto \mathbb{R}$ over the hypergrid domain, where $|Im(f)| \leq r$. The tester runs in time $O\left(\frac{d}{\varepsilon}\log\frac{d}{\varepsilon}\log r\right)$.*

Note that our tester has a better complexity (up to log factors) than the optimal tester for monotonicity of real-valued functions over the hypergrid domains that makes $O\left(\frac{d\log n}{\varepsilon}\right)$ queries [16] for small r. Parameterizing the complexity of testing in terms of the image size of the functions being tested is what enables us to bypass the $\Omega\left(\frac{d\log n}{\varepsilon}\right)$ lower bound for monotonicity testing of functions over hypergrid domains in [17].

Convexity testing over the line. Finally, in Section 7, we give a nonadaptive convexity tester for real-valued functions over the line and prove the following theorem.

Theorem 1.8. *There exists a nonadaptive, 1-sided error ε-tester for convexity of functions $f : [n] \mapsto \mathbb{R}$ that takes an integer $r \geq |Im(f)|$ as input and makes $O(1/\varepsilon)$ queries when $r < \varepsilon n/3$ and $O\left(\frac{\log(r/\varepsilon)}{\varepsilon}\right)$ queries otherwise. The tester runs in time $O(1/\varepsilon)$ when $r < \varepsilon n/3$ and $O\left(\frac{\log(r/\varepsilon)}{\varepsilon}\right)$ otherwise.*

Recall that for real-valued functions over $[n]$, the complexity of (nonadaptively) ε-testing convexity (for constant ε) is $\Theta(\log n)$. Contrary to this, our tester makes only a constant number of queries when the image size of the function is small.

1.3 Related Work

A related concept of parameterized testability of graph properties was studied by Iwama and Yoshida [34]. The focus of their work was to design efficient algorithms for the property testing variants of several NP-hard decision problems on graphs, by expressing their complexity in terms of parameters that have been successfully used in the literature on parameterized algorithms. In most of the cases, the parameters that they used are NP-hard to compute. In contrast, our goal is to determine the right input parameters in terms of which to express the complexity of property testers and, more generally, sublinear-time algorithms. The parameters we use are often easy to compute or estimate and, in many situations, can be assumed to be given to the algorithm. We also believe that the parameters that we use are tied to the intrinsic combinatorial structure of the properties and give insights into complexity of testing them.

2 The Nonadaptive Sortedness Tester

In this section, we describe a nonadaptive, 1-sided error ε-tester for sortedness of arrays containing at most r distinct values and prove Theorem 1.4. Our tester (Algorithm 1) uses a proximity oblivious tester (POT) for sortedness as a subroutine.

Definition 2.1 (POT, Goldreich and Ron [30])**.** *A* proximity oblivious tester *for a property $\mathcal{P}$ is an algorithm that has oracle access to a function f and*

1. *always accepts if $f \in \mathcal{P}$;*
2. *rejects with probability at least $dist(f, \mathcal{P})$ if $f \notin \mathcal{P}$, where $dist(f, \mathcal{P})$ is the minimum fraction of values in f that needs to be changed, so that $f \in \mathcal{P}$.*

Observe that a POT for $\mathcal{P}$ can be repeated $O(1/\varepsilon)$ times to obtain a 1-sided error ε-tester for $\mathcal{P}$. We note that Definition 2.1 is a special case of the definition of POT in [30]. Specifically, Goldreich and Ron [30] allow the rejection probability of a POT to be a non-decreasing function of $\text{dist}(f, \mathcal{P})$. However, the special case in Definition 2.1 is sufficient for our purposes.

We now give an overview of Algorithm 1. It runs for $O(1/\varepsilon)$ iterations. In each iteration, it first runs a POT for sortedness on a subarray B of the input array A consisting of $1 + 2r/\varepsilon$ (nearly) equally spaced indices. Next, it picks an index $i \in [n]$ uniformly at random. It compares $A[i]$ with the array values of the indices closest to i that were included in B. Algorithm 1 rejects if either of these steps finds elements out of order.

At least three distinct POTs for sortedness of arrays with $O(\log n)$ query complexity are known [24, 10, 16]. We can use any of them in Algorithm 1. Note that Algorithm 1 is not proximity oblivious itself, as it uses the proximity parameter ε to determine its queries. For simplicity, we assume throughout that $2r/\varepsilon$ is an integer that divides n.

```
Algorithm 1: The Nonadaptive Sortedness Tester
  input : query access to an array $A$ of size $n$, an upper bound $r$ on the number of distinct
          values in $A$, and a distance parameter $\varepsilon \in (0,1)$.
1 Let $B$ be the subarray of $A$ consisting of the indices $1, \frac{\varepsilon n}{2r}, \frac{2\varepsilon n}{2r}, \dots, \left(\frac{2r}{\varepsilon}-1\right)\frac{\varepsilon n}{2r}, n$. // No
    need to explicitly construct $B$.
2 repeat $\lceil \frac{8}{\varepsilon} \rceil$ times
3     Run a POT for sortedness of arrays (e.g., from [24], [10] or [16]) on $B$ and reject if it
        rejects.
4     Query an index $i$ from $A$ uniformly at random.
5     Set $k = \lfloor \frac{2ri}{\varepsilon n} \rfloor + 1$. // Note that $B[k] = A\left[\frac{(k-1)\varepsilon n}{2r}\right]$ and $B[k+1] = A\left[\frac{k\varepsilon n}{2r}\right]$.
6     Query $B[k]$ and $B[k+1]$.
7     Reject if $(B[k], A[i], B[k+1])$ is not in sorted order.
8 Accept.
```

Proof of Theorem 1.4. We prove that Algorithm 1 is a nonadaptive, 1-sided error ε-tester making $O\left(\frac{1}{\varepsilon}\log\frac{r}{\varepsilon}\right)$ queries to test sortedness of arrays with at most r distinct values. Algorithm 1 is nonadaptive, since its queries can be chosen in advance. It has 1-sided error as it always accepts sorted arrays. Lemmas 2.2, 2.3 and 2.5 complete the proof of Theorem 1.4. $\square$

Lemma 2.2. *Algorithm 1 makes $O\left(\frac{1}{\varepsilon}\log\frac{r}{\varepsilon}\right)$ queries.*

Proof. The query complexity of Step 3 is $O(\log|B|) = O(\log(r/\varepsilon))$. Steps 4-7 make a constant number of queries. Steps 3-7 are executed $O(1/\varepsilon)$ times. Hence, the overall query complexity of the tester is $O\left(\frac{1}{\varepsilon}\log\frac{r}{\varepsilon}\right)$. $\square$

Recall that an array is ε*-far* from sorted if at least an ε fraction of elements need to be modified to make it sorted; otherwise, it is ε*-close* to sorted.

Lemma 2.3. *Algorithm 1, with probability at least $2/3$, rejects every array that has at most r distinct values and is ε-far from sorted.*

Proof. Consider an array A that has at most r distinct values and is ε-far from sorted. Let B be the subarray of A as defined in Step 1 of Algorithm 1. If B is $\frac{\varepsilon}{7}$-far from sorted, then by the definition of POT for sortedness, Step 3 of our tester rejects with probability at least $\frac{\varepsilon}{7}$ in each iteration. In the rest of the proof, we consider the case when B is $\frac{\varepsilon}{7}$-close to sorted.

Claim 2.4. *If B is $\frac{\varepsilon}{7}$-close to sorted, then Steps 4-7 reject with probability at least $\frac{\varepsilon}{7}$ in each iteration.*

Proof. The subarray B consists of $1 + 2r/\varepsilon$ (nearly) equally spaced indices, which partition A into $2r/\varepsilon$ intervals of nearly the same size. Let $\mathcal{I} = \{I_1, I_2, \dots, I_{2r/\varepsilon}\}$ denote the set of these intervals. Here, I_1 denotes the interval[2] $\left[2..\frac{\varepsilon n}{2r}-1\right]$ and, for $k > 1$, the interval $\left[\frac{(k-1)\varepsilon n}{2r}+1..\frac{k\varepsilon n}{2r}-1\right]$ is

[2]We use $[a..b]$ to denote $\{a, a+1, \dots, b-1, b\}$ for $a, b \in \mathbb{Z}, a < b$.

denoted by I_k. Note that, by definition, $B[k]$ and $B[k+1]$ denote the values of the elements in A present immediately to the left and right of I_k, respectively.

An interval I_k is *nearly-constant* if $B[k] = B[k+1]$. Let $\mathcal{C}_t$ be the set of arrays with all their values equal to t. Let $A[I_k]$ denote the subarray of A on the indices in I_k. Let $d(I_k)$ and $D(I_k)$ denote the fractional and absolute Hamming distance of $A[I_k]$ from the property $\mathcal{C}_{B[k]}$. Note that $d(I_k) = D(I_k)/|I_k|$.

We now prove Claim 2.4 as follows in two steps. First, we show that $\sum_{I_k \in \mathcal{I}'} D(I_k) > \varepsilon n/7$, where $\mathcal{I}' = \{I_k \in \mathcal{I} : I_k \text{ is nearly-constant}\}$. Second, we show that Steps 4-7 of Algorithm 1 reject with probability at least $\sum_{k \in \mathcal{I}'} D(I_k)/n$ in each iteration.

Since B is $\frac{\varepsilon}{7}$-close to sorted, there exists a set S of at most $\varepsilon|B|/7$ indices in B whose values can be changed to make B sorted. Note that, for $r \geq 3$, we have $|S| < r/3$ since $|B| = 1 + 2r/\varepsilon$. Consider the set of intervals E_1 in $\mathcal{I}$ adjacent to at least one index from S. As each index in S is adjacent to at most two intervals, $|E_1| < 2r/3$.

Let E_2 denote the set of intervals in $\mathcal{I} \setminus E_1$ that are not nearly-constant. For all k such that $I_k \in E_2$, we have $B[k] < B[k+1]$. This is so because, if $B[k] > B[k+1]$, then $I_k \in E_1$ and if $B[k] = B[k+1]$, then I_k is nearly-constant. The total number of distinct values taken by the elements belonging to intervals in E_2 is at least $|E_2|$. But A has at most r distinct values, and hence, $|E_2| \leq r$. Consequently, $|E_1 \cup E_2| < \frac{2r}{3} + r = \frac{5r}{3}$.

Consider the subarray A'' of A induced by the indices in the intervals in $\mathcal{I} \setminus (E_1 \cup E_2)$. Let $D_{\mathcal{S}}(A)$ denote the absolute Hamming distance of the array A to the sortedness property. As $D_{\mathcal{S}}(A) \geq \varepsilon n$, we get $D_{\mathcal{S}}(A'') > \varepsilon n - \frac{5r}{3} \cdot \frac{\varepsilon n}{2r} = \frac{\varepsilon n}{6}$. Note that all the intervals in A'' are nearly-constant. Hence, $(\mathcal{I} \setminus (E_1 \cup E_2)) \subseteq \mathcal{I}'$ and, consequently,

$$\sum_{I_k \in \mathcal{I}'} D(I_k) \geq D_{\mathcal{S}}(A'') > \frac{\varepsilon n}{6} > \frac{\varepsilon n}{7}.$$

This completes the first step of the proof.

Consider a nearly-constant interval $I_k \in \mathcal{I}'$ such that $D(I_k) > 0$. As $B[k] = B[k+1]$, there exists $D(I_k)$ elements in I_k whose values are not $B[k]$, i.e.,

$$|\{x \in I_k : A[x] \neq B[k]\}| = D(I_k).$$

Algorithm 1 rejects if it samples an index $x \in I_k$ in Step 4 such that $B[k] = B[k+1]$ (i.e., $I_k \in \mathcal{I}'$) and $A[x] \neq B[k]$. As there are $\sum_{I_k \in \mathcal{I}'} D(I_k)$ such indices in A, Steps 4-7 of Algorithm 1 reject A with probability at least $\sum_{I_k \in \mathcal{I}'} D(I_k)/n$. Since $\sum_{I_k \in \mathcal{I}'} D(I_k) > \varepsilon n/7$, the proof of Claim 2.4 is complete. □

Hence, the probability that Steps 3-7 reject in each iteration is at least $\varepsilon/7$. The probability that Algorithm 1 accepts after $\lceil 8/\varepsilon \rceil$ iterations is at most $(1 - \varepsilon/7)^{8/\varepsilon} \leq e^{-8/7} < 1/3$. This completes the proof of Lemma 2.3. □

Lemma 2.5. *The time complexity of Algorithm 1 is* $O\left(\frac{1}{\varepsilon} \log \frac{r}{\varepsilon}\right)$.

Proof. Step 1 introduces notation and is not a step of the algorithm. The time complexity of Step 3 is $O(\log |B|) = O\left(\log \frac{r}{\varepsilon}\right)$. Steps 4-7 run in constant time. Hence, the running time of each iteration of Steps 3-7 is $O\left(\log \frac{r}{\varepsilon}\right)$. As these steps are executed $O(1/\varepsilon)$ times, the time complexity of Steps 2-7 is $O\left(\frac{1}{\varepsilon} \log \frac{r}{\varepsilon}\right)$, which is also the overall time complexity of Algorithm 1. □

3 The Sortedness Tester with $O\left(\frac{\log r}{\varepsilon}\right)$ Query Complexity

In this section, we describe a 1-sided error ε-tester for sortedness of arrays containing at most r distinct values and prove Theorem 1.3. The tester, described in Algorithm 2, runs the nonadaptive tester (Algorithm 1) described in Section 2 when $r \geq 1/\varepsilon$, and a different procedure, which is described in Algorithm 2, otherwise.

Algorithm 2: The Sortedness Tester

```
input : query access to an array $A$ of size $n$, an upper bound $r$ on the number of distinct
        values in $A$, and distance parameter $\varepsilon \in (0,1)$.
1  If $r \geq 1/\varepsilon$, run **Algorithm 1** and **reject** if it rejects.
2  If $A[1] > A[n]$, **reject**.
3  Initialize a doubly linked list $L$ to contain keys 1 and $n$.
   // Define $\mathsf{successor}(i) = \min\{j \in L : j > i\}$;  $\mathsf{predecessor}(i) = \max\{j \in L : j < i\}$.
4  **while** $\exists i, j \in L$ *such that* $j = \mathsf{successor}(i)$ *and* $|i-j| > \frac{\varepsilon n}{2r}$ *and* $A[i] < A[j]$ **do**
5      Set $m = \left\lfloor \frac{i+j}{2} \right\rfloor$ and query $A[m]$.
6      **if** $A[i] \leq A[m] \leq A[j]$ **then** insert $m$ into $L$ **else reject**.
7      **if** $i > 1$ *and* $A[\mathsf{predecessor}(i)] = A[i] = A[m]$ **then**
8          Delete $i$ from $L$.
9      **if** $j < n$ *and* $A[m] = A[j] = A[\mathsf{successor}(j)]$ **then**
10         Delete $j$ from $L$.
11 **repeat** $\left\lceil \frac{2\ln 3}{\varepsilon} \right\rceil$ **times**
12     Sample an index $x$ from $[n]$ uniformly at random and query $A[x]$.
13     **if** $(A[\mathsf{predecessor}(x)], A[x], A[\mathsf{successor}(x)])$ is not in sorted order **then reject**.
14 **Accept**.
```

Proof of Theorem 1.3. We prove that Algorithm 2 is a 1-sided error ε-tester for sortedness of arrays with at most r distinct values and that its query and time complexity are both $O\left(\frac{\log r}{\varepsilon}\right)$. When $r \geq 1/\varepsilon$, Algorithm 2 runs Algorithm 1 and outputs its answer. By Theorem 1.4, Algorithm 1 is a 1-sided error ε-tester with query and time complexity $O\left(\frac{1}{\varepsilon}\log\frac{r}{\varepsilon}\right)$ which is equal to $O\left(\frac{\log r}{\varepsilon}\right)$ as $r \geq 1/\varepsilon$. When $r < 1/\varepsilon$, Algorithm 2 only rejects if it finds array elements out of order, and so, it has 1-sided error. Lemmas 3.1, 3.2 and 3.3 complete the proof of Theorem 1.3. $\square$

Lemma 3.1. *For $r < 1/\varepsilon$, Algorithm 2 makes $O\left(\frac{\log r}{\varepsilon}\right)$ queries.*

Proof. We first bound the query complexity of Steps 4-10. Let w be the number of times Steps 4-10 are run by Algorithm 2. For $k \in [0..w]$, let L_k be the snapshot of the doubly linked list L of array indices (initialized in Step 3) at the end of iteration k. Note that $L_0 = \{1, n\}$ and L_w is the list at the end of the algorithm. Let $V_k = \{v : v = A[i] \text{ for some } i \in L_k\}$ be the set of all array values of indices in L_k. Observe that once a value v is in V_k, it remains in $V_{k'}$ for all $k' > k$. For $v \in V_k$, define $\mathsf{successor\text{-}distance}(v, L_k) = |i - \mathsf{successor}(i)|$ such that $A[i] = v$ and $A[\mathsf{successor}(i)] \neq v$, where $\mathsf{successor}$ is defined with respect to the list L_k (for $v = A[n]$, define $\mathsf{successor\text{-}distance}(v, L_k) = 0$

for all k). Consider the k^{th} iteration of Steps 4-10 where $k \in [w]$. In Step 4 of k^{th} iteration, an index $i \in L$ is chosen such that $\textsf{successor-distance}(A[i], L_{k-1}) > \varepsilon n/2r$. At the end of the iteration, $\textsf{successor-distance}(A[i], L_k) = \textsf{successor-distance}(A[i], L_{k-1})/2$ ignoring the errors due to rounding. Generalizing this argument, for each iteration $k \in [w]$, there exists some $v_k \in V_{k-1} \setminus \{A[n]\}$, such that $\textsf{successor-distance}(v_k, L_k) = \textsf{successor-distance}(v_k, L_{k-1})/2$.

Fix $v^* \in V_w \setminus \{A[n]\}$. Let $k_1, k_2, \ldots, k_q \in [w]$, where $k_1 < k_2 < \ldots < k_q$, be the iterations where the choice of i in Step 4 satisfies $A[i] = v^*$. From the description of the tester, for any $i \in [2..q]$, we have $\textsf{successor-distance}(v^*, L_{k_i}) = \textsf{successor-distance}(v^*, L_{k_{i-1}})/2$. By extending this relation, we get $\textsf{successor-distance}(v^*, L_{k_q}) = \textsf{successor-distance}(v^*, L_{k_1})/2^{q-1}$. But $\textsf{successor-distance}(v^*, L_{k_1}) < n$ and $\frac{\varepsilon n}{4r} < \textsf{successor-distance}(v^*, L_{k_q}) \le \frac{\varepsilon n}{2r}$. Solving for q, we get

$$2^{q-1} = \frac{\textsf{successor-distance}(v^*, L_{k_1})}{\textsf{successor-distance}(v^*, L_{k_q})} < \frac{n}{\varepsilon n/4r} = \frac{4r}{\varepsilon};$$
$$q < \log \frac{8r}{\varepsilon}.$$

Hence, the tester runs at most $\log(8r/\varepsilon)$ iterations where $\textsf{successor-distance}(v^*, \cdot)$ is halved. Accounting for all the iterations for each value in $V_w \setminus \{A[n]\}$, we get

$$w < |V_w| \cdot \log(8r/\varepsilon) \le r \log(8r/\varepsilon),$$

since $|V_w| \le r$. In each iteration, the tester makes a constant number of queries. So, the overall query complexity of Steps 4-10 is $O\left(r \log \frac{r}{\varepsilon}\right)$. The query complexity of Steps 11-13 is $O(1/\varepsilon)$. Hence, the overall query complexity of the tester is $O\left(\frac{1}{\varepsilon} + r \log \frac{r}{\varepsilon}\right)$.

Now, we prove that $O\left(r \log \frac{r}{\varepsilon}\right) = O\left(\frac{\log r}{\varepsilon}\right)$ for $r < 1/\varepsilon$. We have $O\left(r \log \frac{r}{\varepsilon}\right) = O\left(r \log \frac{1}{\varepsilon}\right)$ as $r < 1/\varepsilon$. Note that the function $g(x) = \frac{x}{\log x}$ is increasing for $x \ge 3$. Hence, for $r < 1/\varepsilon$, we have $\frac{r}{\log r} < \frac{1/\varepsilon}{\log(1/\varepsilon)}$, and hence $r \log \frac{1}{\varepsilon} < \frac{\log r}{\varepsilon}$. Therefore, the query complexity of Algorithm 2 is $O\left(\frac{\log r}{\varepsilon}\right)$. □

Lemma 3.2. *Steps 2-14 of Algorithm 2, with probability at least* $2/3$*, reject every array that has at most* r *distinct values and is* ε*-far from sorted, when* $r < 1/\varepsilon$.

Proof. Consider an array A that has at most r distinct values and is ε-far from sorted, where $r < 1/\varepsilon$. Algorithm 2 rejects whenever it finds elements out of order. We show that Steps 11-13 reject with probability at least $2/3$, if Steps 2-10 do not find array elements out of order.

Consider the indices in the list L at the end of the while loop. Let $E = \{j \in L : A[j] < A[\textsf{successor}(j)]\}$ be the indices in L whose array values differ from that of their respective successor in L. As A has at most r distinct values, by Pigeonhole principle, $|E| < r$. Each $i \in E$ satisfies $|i - \textsf{successor}(i)| \le \varepsilon n/2r$. Define $E' = \{k \in [n] : i < k < \textsf{successor}(i), i \in E\}$. Clearly, $|E'| \le \frac{\varepsilon n}{2r} \cdot |E| < \frac{\varepsilon n}{2}$. Consider the subarray of A indexed by $[n] \setminus E'$. This subarray is $\frac{\varepsilon}{2}$-far from sorted as A is ε-far from sorted. Also, all $k \in [n] \setminus E'$ satisfy $\textsf{predecessor}(k) < k < \textsf{successor}(k)$ and $A[\textsf{predecessor}(k)] = A[\textsf{successor}(k)]$ (note that the definitions of $\textsf{predecessor}$ and $\textsf{successor}$ are applicable to all elements in $[n]$). That is, for all such indices k, we know what the element $A[k]$ should be if A is sorted. Recall that if $A[i] = A[j]$, then $[i..j]$ constitutes a *nearly-constant* interval, as defined in Section 2. By the proof method used in Lemma 2.3, there exists at least $\varepsilon n/2$ indices of the form $k \in [n]$ such that $A[\textsf{predecessor}(k)] = A[\textsf{successor}(k)]$ and $A[k] \ne A[\textsf{successor}(k)]$. The probability that Steps 12 and 13 fail to capture such an index in any of its $\lceil \frac{2\ln 3}{\varepsilon} \rceil$ iterations is at most

$$(1-\varepsilon/2)^{\frac{2\ln 3}{\varepsilon}} \leq 1/3.$$

□

Lemma 3.3. *Algorithm 2 runs in time* $O\left(\frac{\log r}{\varepsilon}\right)$.

Proof. If $r \geq 1/\varepsilon$, then Algorithm 1 is run. The time complexity of Algorithm 1 is $O\left(\frac{1}{\varepsilon}\log\frac{r}{\varepsilon}\right)$, which is $O\left(\frac{\log r}{\varepsilon}\right)$ for $r \geq 1/\varepsilon$.

For $r < 1/\varepsilon$, Steps 2-14 are executed. The main idea of implementing Steps 4-10 efficiently is to maintain a pointer p to the smallest index in the list L that satisfies the condition. The pointer p is initialized to point to 1, which is present in L in the first iteration of Step 4. We repeat Steps 4-10 until the index that p points to, either no longer satisfies the while condition, or is deleted from L. In both cases, we update p to point to the successor of the current index. In order to efficiently implement Steps 11-13, we first copy the indices in L into an array D of size $|L|$. Note that D is a sorted array. Step 13, which involves finding the successor and predecessor of a sampled index, can then be performed by doing a binary search in D, which is a sorted array.

Steps 2 and 3 run in constant time. Since p points to the smallest index in L satisfying the conditions of Step 4, the running time of Step 4 over all the iterations is $O\left(\frac{\log r}{\varepsilon}\right)$, the query complexity of Steps 4-10. Steps 5-10 run in time $O(1)$, since the operations of insertion, deletion and finding the successor and predecessor can all be done in $O(1)$ time for the list L. Steps 4-10 are executed $O\left(\frac{\log r}{\varepsilon}\right)$ times (Lemma 3.1), and hence, the running time over all the iterations of Steps 4-10 is $O\left(\frac{\log r}{\varepsilon}\right)$.

We now analyze the running time of Steps 11-13. The size of L is at most the query complexity of Steps 4-10, which is $O\left(\frac{\log r}{\varepsilon}\right)$. Hence, copying the indices in L to the array D takes only $O\left(\frac{\log r}{\varepsilon}\right)$ time. Since D has at most two indices per each of the (at most r) distinct values in A, a binary search on D can be done in time $O(\log r)$. Thus, the overall running time of Steps 11-13 is $O\left(\frac{\log r}{\varepsilon}\right)$. Hence, the overall time complexity of Algorithm 2 is $O\left(\frac{\log r}{\varepsilon}\right)$. □

4 The Monotonicity Tester over Hypergrids

In this section, we describe a monotonicity tester for functions over hypergrid domains and prove Theorem 1.7. We prove the correctness of this tester using the correctness of the sortedness tester described in Section 3, a dimension reduction theorem by Chakrabarty et al. [15] and the work investment strategy by Berman et al. [9].

An axis-parallel line ℓ of the hypergrid $[n]^d$ is a set of n points that agree on all but one coordinate. Let $f|_\ell$ denote the restriction of a function f to ℓ. Note that $f|_\ell$ can be thought of as a real-valued function over $[n]$.
The tester iteratively samples uniformly random axis-parallel lines, runs Algorithm 2 on each of them, and rejects if any run of Algorithm 2 rejects. We now analyze the tester and prove Theorem 1.7.

```
Algorithm 3: The Monotonicity Tester over Hypergrids
  input : query access to $f : [n]^d \mapsto \mathbb{R}$, an upper bound $r$ on $|\mathrm{Im}(f)|$, and a distance
          parameter $\varepsilon \in (0,1)$.
1 for $i = 1$ to $\lceil 3 \log \frac{4d}{\varepsilon} \rceil$ do
2     repeat $\lceil \frac{16 d \ln 4}{2^i \varepsilon} \rceil$ times
3         Sample a uniformly random axis-parallel line $\ell$.
4         Repeat twice: run Algorithm 2 on the array induced by $f|_\ell$, with the distance
            parameter set to $2^{-i}$ and the upper bound on the number of distinct elements set to
            $r$; reject if it rejects at least once.
5 Accept.
```

Proof of Theorem 1.7. We prove that Algorithm 3 is a 1-sided error ε-tester that makes $O\left(\frac{d}{\varepsilon} \log \frac{d}{\varepsilon} \log r\right)$ queries to test monotonicity of real-valued functions $f : [n]^d \mapsto \mathbb{R}$ over the hypergrid domain, where $|\mathrm{Im}(f)| \leq r$. Algorithm 3 has 1-sided error because Algorithm 2, which it runs as a subroutine, has 1-sided error. Lemmas 4.1 and 4.2 complete the proof of Theorem 1.7. □

Lemma 4.1. *The query and time complexity of Algorithm 3 are both* $O\left(\frac{d}{\varepsilon} \log \frac{d}{\varepsilon} \log r\right)$.

Proof. The query complexity of a single execution of Step 4 during the i^{th} iteration of the outermost loop (Step 1) is $O(2^i \log r)$. As Step 4 is repeated $O\left(\frac{d}{2^i \varepsilon}\right)$ times in the i^{th} iteration, the overall query complexity of the i^{th} iteration of the tester is $O\left(\frac{d}{\varepsilon} \log r\right)$. The outermost loop is executed $O(\log \frac{d}{\varepsilon})$ times, and hence the query complexity of the tester is $O\left(\frac{d}{\varepsilon} \log \frac{d}{\varepsilon} \log r\right)$. The analysis of the time complexity is the same as that of the query complexity. □

Lemma 4.2. *Algorithm 3, with probability at least* $2/3$*, rejects every function over the hypergrid domain which is* ε*-far from sorted and has image size is at most* r.

Proof. Let $f : [n]^d \mapsto \mathbb{R}$ be ε-far from monotone, with $|Im(f)| \leq r$. Let $\mathcal{L}_{n,d}$ denote the set of all axis-parallel lines in $[n]^d$ and $d_{\mathcal{M}}(f)$ denote the relative distance of f to monotonicity. We also use $d_{\mathcal{M}}(f|_\ell)$ to denote the relative distance to monotonicity of the function $f|_\ell$. We have $|\mathrm{Im}(f|_\ell)| \leq r$ since $|\mathrm{Im}(f)| \leq r$. We use the following dimension reduction theorem proved by Chakrabarty et al. [15].

Theorem 4.3 (Chakrabarty et al. [15])**.**

$$\mathbb{E}_{\ell \leftarrow \mathcal{L}_{n,d}}[d_{\mathcal{M}}(f|_\ell)] \geq \frac{d_{\mathcal{M}}(f)}{4d}.$$

We note that Theorem 4.3 is a special case of the dimension reduction theorem proved in [15]. Clearly, if $d_{\mathcal{M}}(f) \geq \varepsilon$, then, $\mathbb{E}_{\ell \leftarrow \mathcal{L}_{n,d}}[d_{\mathcal{M}}(f|_\ell)] \geq \varepsilon/4d$. We use the work investment strategy due to Berman et al. [9] to extend the monotonicity tester on the line domain to the hypergrid domain.

Theorem 4.4 (Berman et al. [9])**.** *For a random variable* $X \in [0,1]$ *with* $\mathbb{E}[X] \geq \mu$ *for* $\mu < 1/2$*, let* $p_i = \Pr[X \geq \frac{1}{2^i}]$ *and* $\delta \in (0,1)$ *be the desired probability of error. Let* $k_i = \frac{4 \ln 1/\delta}{2^i \mu}$. *Then,*

$$\prod_{i=1}^{\lceil 3 \log(1/\mu) \rceil} (1 - p_i)^{k_i} \leq \delta.$$

Consider running Algorithm 3 on f. Let $X = d_{\mathcal{M}}(f|_\ell)$, where ℓ is sampled uniformly at random from $\mathcal{L}_{n,d}$. We apply the work investment strategy (Theorem 4.4) on X with error probability $\delta = 1/4$. By Theorem 4.3, $\mathbb{E}[X] \geq \varepsilon/4d$. Thus, in Theorem 4.4, we set $\mu = \varepsilon/4d$ and $k_i = \frac{16d \ln 4}{2^i \varepsilon}$ for all $i \in \left[\left\lceil 3 \log \frac{4d}{\varepsilon} \right\rceil\right]$. By Theorem 4.4, with probability at least 3/4, for some $i \in \left[\left\lceil 3 \log \frac{4d}{\varepsilon} \right\rceil\right]$, we sample a line ℓ such that $d_{\mathcal{M}}(f|_\ell) \geq 2^{-i}$ in Step 3. Conditioned on sampling such a line, Step 4 rejects ℓ with probability at least 8/9. Thus, given a function f that is ε-far from sorted, Algorithm 3 rejects f with probability at least $\frac{3}{4} \cdot \frac{8}{9} = \frac{2}{3}$, as required. This completes the proof of Lemma 4.2. □

Note on a nonadaptive tester for hypergrids. We can get a nonadaptive, 1-sided error ε-tester for monotonicity over hypergrids by using Algorithm 1 instead of Algorithm 2 in Step 4 of Algorithm 3. The same analysis goes through for this case and the overall query complexity of the tester is $O\left(\frac{d}{\varepsilon} \log \frac{d}{\varepsilon} \log \frac{rd}{\varepsilon}\right)$.

5 The Uniform Tester for Sortedness

In this section, we first describe a nonadaptive ε-tester that makes $O(\sqrt{r}/\varepsilon)$ uniform and independent queries to test sortedness of arrays containing at most r distinct values. The expected running time of the tester is $O(\sqrt{r}/\varepsilon)$. We then show how to use this tester to obtain another tester that meets the requirements of Theorem 1.5.

Recall that a pair of indices (x, y), where $x, y \in [n]$ and $x < y$, is *violated* in an array A if $A(x) > A(y)$. Two indices x and y are *adjacent* in a sample S if there is no index $z \in S$ such that $x < z < y$. Algorithm 4, uses the fact that there is a violated pair in a sample of indices if and only if there is a violated pair consisting of adjacent indices in that sample.

The bound on the query complexity of the tester follows directly from its description. The tester has 1-sided error as it always accepts sorted arrays. In the rest of the section, we show that the time complexity of the tester is $O(\sqrt{r}/\varepsilon)$ and that, with high probability, the tester rejects arrays that are ε-far from sorted.

Algorithm 4: The Uniform Sortedness Tester

```
input : query access to an array $A$ of size $n$, an upper bound $r$ on the number of distinct
        values in $A$, and a distance parameter $\varepsilon \in (0, 1)$.
1 Sample and query $\left\lceil \frac{32\sqrt{r}}{\epsilon} \right\rceil$ indices from $A$ uniformly and independently at random.
2 Sort the sampled indices in increasing order using Bucket Sort [21] with $\left\lceil \frac{32\sqrt{r}}{\epsilon} \right\rceil$ buckets.
3 Reject if the sample contains a violated pair consisting of adjacent indices; else, accept.
```

Lemma 5.1. *Algorithm 4, with probability at least* 3/4*, rejects every array that has at most* r *distinct elements and is* ε*-far from sorted.*

Proof. Consider an array A that has at most r distinct values and is ε-far from sorted. Consider the undirected *violation graph* $G = ([n], E)$ of A, where an edge $\{u, v\} \in E$ if (u, v) is a violated pair. Dodis et al. [23, Lemma 7] show that if A is ε-far from sorted then G has a matching M of size at least $\varepsilon n/2$.

For a pair $(x, y) \in [n] \times [n]$ such that $x < y$, we refer to x as its lower endpoint and y as its higher endpoint. We first partition the pairs in M into r classes as follows. Let $v_1 < v_2 < \cdots < v_r$ be the values in the range. A pair $(x, y) \in M$ such that $x < y$ belongs to the i^{th} class C_i, if $A(x) = v_i$. Note that C_1 is empty. For each $i \in [r]$, let C_i^{L} and C_i^{H} denote the set of lower and higher endpoints of pairs in C_i, respectively. Note that $|C_i| = |C_i^{\text{L}}| = |C_i^{\text{H}}|$. For each $i \in [r]$, define the i^{th} lower bucket B_i^{L} to consist of the smallest $\lceil |C_i|/2 \rceil$ indices in C_i^{L} and the i^{th} higher bucket B_i^{H} to consist of the largest $\lceil |C_i|/2 \rceil$ indices in C_i^{H}. Note that $\left|\bigcup_{i\in[r]} B_i^{\text{L}}\right| = \left|\bigcup_{i\in[r]} B_i^{\text{H}}\right| \geq \varepsilon n/4$. It is easy to see that for each $i \in [r]$, every pair in $B_i^{\text{L}} \times B_i^{\text{H}}$ is a violated pair. Therefore, if for some $i \in [r]$, there exist indices from both B_i^{L} and B_i^{H} in the algorithm's sample, there also exists a violated pair in the sample consisting of adjacent indices, and the algorithm rejects. To bound the probability of the former event from below, we use the following generalization of the Birthday Paradox proved by Goldreich et al. [28, Lemma 19].

Claim 5.2 ([28, Lemma 19])**.** *Let $S_1, S_2 \ldots, S_r, T_1, T_2 \ldots, T_r$ be disjoint subsets of a universe U. For each $i \in [r]$, let $p_i = |S_i|/|U|$ and $q_i = |T_i|/|U|$. Let $\rho = \sum_i \min\{p_i, q_i\}$. Then, if we uniformly sample $8\sqrt{r}/\rho$ elements from U, with probability at least $3/4$, for some $i \in [r]$, the sample will contain at least one element from both S_i and T_i.*

If we set $S_i = B_i^{\text{L}}$ and $T_i = B_i^{\text{H}}$ for each $i \in [r]$ in Claim 5.2, we have $\rho \geq \varepsilon/4$. Therefore, a uniform sample of $32\sqrt{r}/\varepsilon$ points from $[n]$, with probability at least 3/4, will have, for some $i \in [r]$, an index from B_i^{L} and B_i^{H}, and the algorithm will reject. This completes the proof of the lemma. □

Lemma 5.3. *The expected running time of Algorithm 4 is $O(\sqrt{r}/\varepsilon)$.*

Proof. Step 1 takes only $O(\sqrt{r}/\varepsilon)$ time to run. Since the indices in the sample are drawn uniformly and independently at random from $[n]$, Bucket Sort sorts the sampled indices in expected time linear in the size of the sample ([21], Chapter 8.4). Hence, Step 2 runs in expected time $O(\sqrt{r}/\varepsilon)$. To check whether any pair of adjacent indices in the sample forms a violated pair, we just need to make a single pass over the sorted list of indices. Thus, Step 3 also takes time $O(\sqrt{r}/\varepsilon)$. This completes the proof of the lemma. □

Proof of Theorem 1.5. Let c be a constant such that the expected running time of Algorithm 4 is at most $c \cdot \sqrt{r}/\varepsilon$. The tester as described in the statement of Theorem 1.5, say T, can be obtained by running Algorithm 4 for exactly $12c \cdot \sqrt{r}/\varepsilon$ steps and rejecting if and only if Algorithm 4 rejects. The query complexity of T is $O(\sqrt{r}/\varepsilon)$. It is easy to see that T accepts if the array is sorted. If the array is ε-far from sorted, the tester T accepts if either the execution of Algorithm 4 accepts, or its running time exceeds $12c\sqrt{r}/\varepsilon$. Using both Markov's inequality and the union bound, we can see that the probability of T accepting in this case is at most 1/3. This completes the proof of Theorem 1.5.

6 A Lower Bound for the Uniform Sortedness Tester

In this section, we prove that $\Omega(\sqrt{r})$ uniform queries are required to test sortedness of an array with at most r distinct values, even when one allows for 2-sided error, and prove Theorem 1.6. The proof uses Yao's principle [43], the version with two distributions (see, e.g., Raskhodnikova and Smith [41]). We first define two hard distributions $\mathcal{P}$ and $\mathcal{N}$ on arrays with r distinct values such that every array drawn from $\mathcal{P}$ is in sorted order and every array drawn from $\mathcal{N}$ is $\frac{1}{8}$-far from sorted. We then show that, for any tester that uses $o(\sqrt{r})$ uniform queries, the statistical difference between tester's views of the two distributions is small, and hence, with high probability, it cannot distinguish between the distributions.

The statistical distance between two distributions $\mathcal{D}_1$ and $\mathcal{D}_2$, denoted by $\mathrm{SD}(\mathcal{D}_1, \mathcal{D}_2)$, is defined as

$$\mathrm{SD}(\mathcal{D}_1, \mathcal{D}_2) = \max_{S \subseteq (\text{support}(\mathcal{D}_1) \cup \text{support}(\mathcal{D}_2))} \left(\left| \Pr_{x \leftarrow \mathcal{D}_1}[x \in S] - \Pr_{x \leftarrow \mathcal{D}_2}[x \in S] \right| \right).$$

We write $\mathcal{D}_1 \approx_\delta \mathcal{D}_2$ to denote $\mathrm{SD}(\mathcal{D}_1, \mathcal{D}_2) \leq \delta$.

Proof of Theorem 1.6. First, we define two distributions $\mathcal{P}$ and $\widehat{\mathcal{N}}$ on arrays of size n taking values in the set $[r]$, where $n \geq 16r \ln 6r$. Without loss of generality, we assume that r is an even number that divides n.

The distribution $\mathcal{P}$ is constructed as follows. Partition an n-element array into $r/2$ blocks, each of length $2n/r$. For $i \in [r/2]$, set all elements in the i^{th} block to the same value; choose this value to be $2i$ with probability $\frac{1}{2}$ and $2i-1$ with probability $\frac{1}{2}$.

The distribution $\widehat{\mathcal{N}}$ is constructed as follows. As before, partition an n-element array into $r/2$ blocks, each of length $2n/r$. For $i \in [r/2]$, the value at each index in the i^{th} block is set to either $(2i-1)$ or $2i$ uniformly and independently at random.

Note that every array drawn from $\mathcal{P}$ is in sorted order. We will show that, with high probability, an array drawn from $\widehat{\mathcal{N}}$ is $\frac{1}{8}$-far from sorted.

Lemma 6.1. *Let E denote the event that an array chosen according to $\widehat{\mathcal{N}}$ is $\frac{1}{8}$-far from sorted. Then,*

$$\Pr[E] > \frac{5}{6}.$$

Proof. Consider an array A chosen according to $\widehat{\mathcal{N}}$. Consider the i^{th} block of A for some $i \in [r/2]$. Let Y_{2i} denote the number of elements with value $2i$ in the first half of this block and Y_{2i-1} denote the number of elements with value $(2i-1)$ in the second half of the block. As the size of each half of the block is n/r, and the value at each index is assigned either $(2i-1)$ or $2i$ uniformly and independently at random,

$$\mathbb{E}[Y_{2i}] = \mathbb{E}[Y_{2i-1}] = \frac{n}{2r}.$$

By a Chernoff bound, for all $i \in \left[\frac{r}{2}\right]$,

$$\begin{aligned}\Pr\left[Y_{2i} \leq \frac{n}{4r}\right] = \Pr\left[Y_{2i-1} \leq \frac{n}{4r}\right] &= \Pr\left[Y_{2i-1} \leq \left(1 - \frac{1}{2}\right)\left(\frac{n}{2r}\right)\right] \\ &\leq \exp\left(-\frac{n}{16r}\right) \\ &< \frac{1}{6r}.\end{aligned}$$

If $Y_{2i} > n/4r$ and $Y_{2i-1} > n/4r$, then at least $n/4r$ elements in i^{th} block need to be changed to make it sorted, as all the indices with value $2i$ in the first half or all the indices with value $2i-1$

in the last half need to be changed. By the union bound,

$$\Pr\left[\bigvee_{j=1}^{r}\left(Y_j \le \frac{n}{4r}\right)\right] \le r \cdot \Pr\left[Y_1 \le \frac{n}{4r}\right] < \frac{1}{6}.$$

With probability at least $5/6$, we have $Y_{2i} > n/4r$ and $Y_{2i-1} > n/4r$ for all $i \in [r/2]$. This implies that at least $n/4r$ elements need to be changed in each of the $r/2$ blocks to make them all sorted. Hence, with probability at least $5/6$, the array A is $\frac{1}{8}$-far from sorted. □

Denote the conditional distribution $\widehat{\mathcal{N}}|_E$ by $\mathcal{N}$, where E denotes the event that an array chosen according to $\widehat{\mathcal{N}}$ is $\frac{1}{8}$-far from sorted. Any instance sampled according to $\mathcal{N}$ is $\frac{1}{8}$-far from sorted. The statistical distance $\mathrm{SD}(\widehat{\mathcal{N}}, \mathcal{N})$ can be bounded using the following lemma proven by Raskhodnikova and Smith [41].

Lemma 6.2 ([41, Claim 4])**.** *Let E be an event that happens with probability at least $1-\delta$ under the distribution $\mathcal{D}$. Then, $\mathcal{D} \approx_{\delta'} \mathcal{D}|_E$, where $\delta' = \frac{1}{1-\delta} - 1$.*

Applying Lemma 6.2 to $\mathcal{N}$ and $\widehat{\mathcal{N}}$, we get $\mathcal{N} \approx_{1/5} \widehat{\mathcal{N}}$.

Consider any $\frac{1}{8}$-tester for sortedness that makes q queries where $q \le \sqrt{r}/5$. Define $\mathcal{P}$-view to be the distribution of values at the q locations queried by the tester in an array sampled according to $\mathcal{P}$. Similarly, define $\widehat{\mathcal{N}}$-view and $\mathcal{N}$-view. Next, we show that it is hard to distinguish $\mathcal{P}$-view from $\mathcal{N}$-view.

Lemma 6.3.

$$\mathit{SD}(\mathcal{P}\text{-}\mathit{view}, \mathcal{N}\text{-}\mathit{view}) < \frac{1}{3}.$$

Proof. Let F denote the event that at least 2 out of the tester's q uniform samples from an array A are from the same block. An upper bound on the probability of this event can be obtained using the following lemma.

Lemma 6.4 (Bellare and Rogaway [3])**.** *Consider q balls and N bins, where each ball is assigned uniformly and independently at random to one of the bins. The probability that there exists a pair of balls assigned to the same bin is at most $\frac{q(q-1)}{2N}$.*

By Lemma 6.4, we get $\Pr[F] \le \frac{q(q-1)}{2 \cdot r/2} < \frac{q^2}{r} = \frac{1}{25}$. Then, by Lemma 6.2,

$$\mathcal{P}\text{-view} \approx_{1/24} \mathcal{P}\text{-view}|_{\overline{F}}; \tag{1}$$

$$\widehat{\mathcal{N}}\text{-view} \approx_{1/24} \widehat{\mathcal{N}}\text{-view}|_{\overline{F}}. \tag{2}$$

Since $\mathcal{N} \approx_{1/5} \widehat{\mathcal{N}}$, the definition of statistical difference implies that

$$\mathcal{N}\text{-view} \approx_{1/5} \widehat{\mathcal{N}}\text{-view}. \tag{3}$$

It remains to show that $\mathcal{P}\text{-view}|_{\overline{F}} = \widehat{\mathcal{N}}\text{-view}|_{\overline{F}}$. Let x be an index in the i^{th} block, for some $i \in [r/2]$.Then $\Pr[A[x] = (2i-1)] = \Pr[A[x] = 2i] = 1/2$ irrespective of whether $A \leftarrow \mathcal{P}$ or $A \leftarrow \widehat{\mathcal{N}}$. If $\overline{F}$ holds, then at most 1 index from each block is sampled by the tester. By the definition of $\mathcal{P}$

and $\widehat{\mathcal{N}}$, for any two indices from different blocks, the values assigned to them are independent of each other. Hence, $\mathcal{P}$-view$|_{\overline{F}} = \widehat{\mathcal{N}}$-view$|_{\overline{F}}$. By (1)-(3),

$$\mathrm{SD}(\mathcal{P}\text{-view}, \mathcal{N}\text{-view}) \leq \frac{1}{24} + \frac{1}{24} + \frac{1}{5} < \frac{1}{3}.$$

This completes the proof of Lemma 6.3. □

By Yao's principle [43], as stated in [41, Claim 5], for $q \leq \sqrt{r}/5$, it is hard for any $\frac{1}{8}$-tester using q uniform queries to distinguish $\mathcal{P}$ from $\mathcal{N}$. Thus, uniform testers for sortedness of arrays with values in $[r]$ require $\Omega(\sqrt{r})$ queries. This completes the proof of Theorem 1.6. □

7 Testing Convexity

In this section, we describe a nonadaptive tester for convexity of functions $f : [n] \mapsto \mathbb{R}$ and prove Theorem 1.8. Recall that a function $f : [n] \mapsto \mathbb{R}$ is convex if $f(i) - f(i-1) \leq f(i+1) - f(i)$ for $1 < i \leq n$. Our convexity tester is Algorithm 5. It uses the nonadaptive convexity tester of Parnas et al. [39] as a black box.

Algorithm 5: The Convexity Tester

```
  input : query access to $f : [n] \mapsto \mathbb{R}$, an upper bound $r$ on $|\mathrm{Im}(f)|$, and a distance
          parameter $\varepsilon \in (0, 1)$.
  if $r \geq \frac{\varepsilon n}{3}$ then
1     Run the $\varepsilon$-tester for convexity by Parnas et al. [39] on $f$ and reject if it rejects.
  else
2     Let $M \leftarrow [r+1, \dots, n-r]$.
3     Sample $\lceil \frac{4}{\varepsilon} \rceil$ indices from $M$ uniformly and independently at random.
4     Reject if $f$ restricted to those indices is not constant.
5 Accept.
```

The query complexity of our tester is $O(1/\varepsilon)$ when $r < \varepsilon n/3$, as is evident from its description. In the other case, $n \leq 3r/\varepsilon$, our tester runs the tester of [39], which makes $O(\log n/\varepsilon)$ queries. Substituting the upper bound on n, we get the query complexity bound claimed in Theorem 1.8. The arguments for the bounds on the time complexity are the same as that for the query complexity.

Given a function $f : [n] \mapsto \mathbb{R}$ and a set $S \subseteq [n]$, let f_S denote the restriction of f to the indices in S whenever $S \neq \emptyset$. To prove the correctness of our tester, we first prove the following characterization of convex functions with image size at most r.

Claim 7.1. *If $f : [n] \mapsto \mathbb{R}$ is convex and $|Im(f)| \leq r$, then f_M for $M = [r+1..n-r]$ is a constant function.*

Proof. We can assume that $r < n/2$, for otherwise, $M = \emptyset$. Assume for the sake of contradiction that there exists points $x, x+1 \in M$ such that $f_M(x) \neq f_M(x+1)$. If $f_M(x) < f_M(x+1)$, then f has to be monotonically increasing on the domain restricted to $[x+1, \dots, n]$, which has more than r elements in it as $x < n-r+1$. By the pigeonhole principle, this results in a contradiction, as $|\mathrm{Im}(f)| \leq r$. If $f_M(x) > f_M(x+1)$, then f has to be monotonically decreasing on the set

$[1, \ldots, x+1]$, which has more than r elements in it since $x \geq r$. By the pigeonhole principle, this also leads to a contradiction, as $|\mathrm{Im}(f)| \leq r$. Hence, f can take only one value on M and therefore, f_M is a constant function. $\square$

We will now show that the tester accepts every function that is convex and rejects with probability at least $2/3$, every function that is ε-far from convex.

Lemma 7.2. *Consider a function $f : [n] \mapsto \mathbb{R}$. Algorithm 5, on input $r \geq |Im(f)|$ and ε, accepts if f is convex and rejects, with probability at least $2/3$, if f is ε-far from convex.*

Proof. If $r \geq \frac{\varepsilon n}{3}$, Algorithm 5 runs the tester for convexity by [39], and so the correctness follows from their analysis.

Consider the case where $r < \varepsilon n/3$. It follows from Claim 7.1 that Algorithm 5 accepts f if it is convex. Now assume that f is ε-far from convex. It remains to prove that f_M is $\varepsilon/3$-far from being a constant function, where $M = [r+1, \ldots, n-r]$. Assume for the sake of contradiction that f_M is $\varepsilon/3$-close to constant. We will construct a convex function $g : [n] \mapsto \mathbb{R}$ such that g is ε-close to f and satisfies $|\mathrm{Im}(g)| \leq r$. This will give us the required contradiction. Let the constant function closest to f_M be h, where $h(x) = c$ for every $x \in M$. The function g is then defined as a constant function taking the value c on all points in $[n]$. Since the Hamming distance of f_M from h is at most $\varepsilon n/3$, the total Hamming distance of f from g is at most $\varepsilon n/3 + 2r < \varepsilon n$. This contradicts the fact that f is ε-far from convex. Hence, f_M is $\frac{\varepsilon}{3}$-far from being a constant function. The probability that $4/\varepsilon$ samples fail to detect that f_M is $\varepsilon/3$-far from constant is at most $(1-\varepsilon/3)^{4/\varepsilon} \leq \exp(-4/3) < 1/3$. $\square$

References

[1] Nir Ailon and Bernard Chazelle. Information theory in property testing and monotonicity testing in higher dimension. *Inf. Comput.*, 204(11):1704–1717, 2006.

[2] Tugkan Batu, Ronitt Rubinfeld, and Patrick White. Fast approximate PCPs for multidimensional bin-packing problems. *Inf. Comput.*, 196(1):42–56, 2005.

[3] Mihir Bellare and Phillip Rogaway. Lecture notes on modern cryptography, 2005. URL: `https://cseweb.ucsd.edu/~mihir/cse207/w-birthday.pdf`.

[4] Aleksandrs Belovs and Eric Blais. Quantum algorithm for monotonicity testing on the hypercube. *Theory of Computing*, 11:403–412, 2015.

[5] Aleksandrs Belovs and Eric Blais. A polynomial lower bound for testing monotonicity. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing, STOC 2016, Cambridge, MA, USA, June 18-21, 2016*, pages 1021–1032, 2016.

[6] Sagi Ben-Moshe, Yaron Kanza, Eldar Fischer, Arie Matsliah, Mani Fischer, and Carl Staelin. Detecting and exploiting near-sortedness for efficient relational query evaluation. In *Database Theory - ICDT 2011, 14th International Conference, Uppsala, Sweden, March 21-24, 2011, Proceedings*, pages 256–267, 2011.

[7] Piotr Berman, Meiram Murzabulatov, and Sofya Raskhodnikova. The power and limitations of uniform samples in testing properties of figures. In *36th IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2016, December 13-15, 2016, Chennai, India*, pages 45:1–45:14, 2016.

[8] Piotr Berman, Meiram Murzabulatov, and Sofya Raskhodnikova. Testing convexity of figures under the uniform distribution. In *32nd International Symposium on Computational Geometry, SoCG 2016, June 14-18, 2016, Boston, MA, USA*, pages 17:1–17:15, 2016.

[9] Piotr Berman, Sofya Raskhodnikova, and Grigory Yaroslavtsev. L_p-testing. In *Symposium on Theory of Computing, STOC 2014, New York, NY, USA, May 31 - June 03, 2014*, pages 164–173, 2014.

[10] Arnab Bhattacharyya, Elena Grigorescu, Kyomin Jung, Sofya Raskhodnikova, and David P. Woodruff. Transitive-closure spanners. *SIAM J. Comput.*, 41(6):1380–1425, 2012.

[11] Eric Blais, Joshua Brody, and Kevin Matulef. Property testing lower bounds via communication complexity. *Computational Complexity*, 21(2):311–358, 2012.

[12] Eric Blais, Sofya Raskhodnikova, and Grigory Yaroslavtsev. Lower bounds for testing properties of functions over hypergrid domains. In *IEEE 29th Conference on Computational Complexity, CCC 2014, Vancouver, BC, Canada, June 11-13, 2014*, pages 309–320, 2014.

[13] Jop Briët, Sourav Chakraborty, David García-Soriano, and Arie Matsliah. Monotonicity testing and shortest-path routing on the cube. *Combinatorica*, 32(1):35–53, 2012.

[14] Deeparnab Chakrabarty. Monotonicity testing. In *Encyclopedia of Algorithms*, pages 1352–1356. Springer, 2016.

[15] Deeparnab Chakrabarty, Kashyap Dixit, Madhav Jha, and C. Seshadhri. Property testing on product distributions: Optimal testers for bounded derivative properties. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2015, San Diego, CA, USA, January 4-6, 2015*, pages 1809–1828, 2015.

[16] Deeparnab Chakrabarty and C. Seshadhri. Optimal bounds for monotonicity and Lipschitz testing over hypercubes and hypergrids. In *Symposium on Theory of Computing Conference, STOC'13, Palo Alto, CA, USA, June 1-4, 2013*, pages 419–428, 2013.

[17] Deeparnab Chakrabarty and C. Seshadhri. An optimal lower bound for monotonicity testing over hypergrids. *Theory of Computing*, 10:453–464, 2014.

[18] Deeparnab Chakrabarty and C. Seshadhri. An $o(n)$ monotonicity tester for Boolean functions over the hypercube. *SIAM J. Comput.*, 45(2):461–472, 2016.

[19] Xi Chen, Anindya De, Rocco A. Servedio, and Li-Yang Tan. Boolean function monotonicity testing requires (almost) $n^{1/2}$ non-adaptive queries. In *Proceedings of the Forty-Seventh Annual ACM on Symposium on Theory of Computing*, STOC '15, pages 519–528, New York, NY, USA, 2015.

[20] Xi Chen, Rocco A. Servedio, and Li-Yang Tan. New algorithms and lower bounds for monotonicity testing. In *55th IEEE Annual Symposium on Foundations of Computer Science, FOCS 2014, Philadelphia, PA, USA, October 18-21, 2014*, pages 286–295, 2014.

[21] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Second Edition.* The MIT Press and McGraw-Hill Book Company, 2001.

[22] Kashyap Dixit, Sofya Raskhodnikova, Abhradeep Thakurta, and Nithin M. Varma. Erasure-resilient property testing. In *43rd International Colloquium on Automata, Languages, and Programming, ICALP 2016, July 11-15, 2016, Rome, Italy*, pages 91:1–91:15, 2016.

[23] Yevgeniy Dodis, Oded Goldreich, Eric Lehman, Sofya Raskhodnikova, Dana Ron, and Alex Samorodnitsky. Improved testing algorithms for monotonicity. In *RANDOM-APPROX'99, Berkeley, CA, USA, August 8-11, 1999, Proceedings*, pages 97–108, 1999.

[24] Funda Ergün, Sampath Kannan, Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. Spot-checkers. *J. Comput. Syst. Sci.*, 60(3):717–751, 2000.

[25] Eldar Fischer. On the strength of comparisons in property testing. *Inf. Comput.*, 189(1):107–116, 2004.

[26] Eldar Fischer, Oded Lachish, and Yadu Vasudev. Trading query complexity for sample-based testing and multi-testing scalability. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 1163–1182, 2015.

[27] Eldar Fischer, Eric Lehman, Ilan Newman, Sofya Raskhodnikova, Ronitt Rubinfeld, and Alex Samorodnitsky. Monotonicity testing over general poset domains. In *Proceedings on 34th Annual ACM Symposium on Theory of Computing, May 19-21, 2002, Montréal, Québec, Canada*, pages 474–483, 2002.

[28] Oded Goldreich, Shafi Goldwasser, Eric Lehman, Dana Ron, and Alex Samorodnitsky. Testing monotonicity. *Combinatorica*, 20(3):301–337, 2000.

[29] Oded Goldreich, Shafi Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *J. ACM*, 45(4):653–750, 1998.

[30] Oded Goldreich and Dana Ron. On proximity-oblivious testing. *SIAM J. Comput.*, 40(2):534–566, 2011.

[31] Oded Goldreich and Dana Ron. On sample-based testers. *TOCT*, 8(2):7, 2016.

[32] Shirley Halevy and Eyal Kushilevitz. Distribution-free property-testing. *SIAM J. Comput.*, 37(4):1107–1138, 2007.

[33] Shirley Halevy and Eyal Kushilevitz. Testing monotonicity over graph products. *Random Struct. Algorithms*, 33(1):44–67, 2008.

[34] Kazuo Iwama and Yuichi Yoshida. Parameterized testability. In *Innovations in Theoretical Computer Science, ITCS'14, Princeton, NJ, USA, January 12-14, 2014*, pages 507–516, 2014.

[35] Madhav Jha and Sofya Raskhodnikova. Testing and reconstruction of Lipschitz functions with applications to data privacy. *SIAM J. Comput.*, 42(2):700–731, 2013.

[36] Subhash Khot, Dor Minzer, and Muli Safra. On monotonicity testing and Boolean isoperimetric type theorems. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 52–58, 2015.

[37] Eric Lehman and Dana Ron. On disjoint chains of subsets. *J. Comb. Theory, Ser. A*, 94(2):399–404, 2001.

[38] Ramesh Krishnan S. Pallavoor, Sofya Raskhodnikova, and Nithin Varma. Parameterized property testing of functions. In *8th Innovations in Theoretical Computer Science, ITCS'17, Berkeley, CA, USA, January 9-11*, 2017.

[39] Michal Parnas, Dana Ron, and Ronitt Rubinfeld. On testing convexity and submodularity. *SIAM J. Comput.*, 32(5):1158–1184, 2003.

[40] Sofya Raskhodnikova. Testing if an array is sorted. In *Encyclopedia of Algorithms*, pages 2219–2222. Springer, 2016.

[41] Sofya Raskhodnikova and Adam D. Smith. A note on adaptivity in testing properties of bounded degree graphs. *Electronic Colloquium on Computational Complexity (ECCC)*, 13(089), 2006.

[42] Ronitt Rubinfeld and Madhu Sudan. Robust characterizations of polynomials with applications to program testing. *SIAM J. Comput.*, 25(2):252–271, 1996.

[43] Andrew Chi-Chih Yao. Probabilistic computations: Toward a unified measure of complexity (extended abstract). In *18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977*, pages 222–227, 1977.