Title: Testing if an Array Is Sorted
Name: Sofya Raskhodnikova
Affil./Addr. Pennsylvania State University
Keywords: Property testing, sublinear-time algorithms, monotonicity, sorted arrays
SumOriWork: 2000; Ergün, Kannan, Kumar, Rubinfeld, Viswanathan
2014; Berman, Raskhodnikova, Yaroslavtsev

Testing if an Array Is Sorted

Sofya Raskhodnikova
Pennsylvania State University

Years aud Authors of Summarized Original Work

2000; Ergün, Kannan, Kumar, Rubinfeld, Viswanathan
2014; Berman, Raskhodnikova, Yaroslavtsev

Keywords

Property testing, sublinear-time algorithms, monotonicity, sorted arrays

Problem Definition

Suppose we would like to check whether a given array of real numbers is sorted (say, in nondecreasing order). Performing this task exactly requires reading the entire array. Here we consider the approximate version of the problem: testing whether an array is sorted or "far" from sorted. We consider two natural definitions of the distance of a given array from a sorted array. Intuitively, we would like to measure how much the input array must change to become sorted. We could measure the change by

1. the number of entries changed;
2. the sum of the absolute values of changes in all entries.

It is not hard to see that looking at the number of entries that must be deleted in an array to make it sorted is equivalent to the measure in item 1.

To define the two distance measures formally, let $a = (a_1, \ldots, a_n)$ be the input array and $\mathcal{S}$ be the set of all sorted arrays of length n. We denote by $[n]$ the set $\{1, 2, \ldots, n\}$. The *Hamming distance* from a to $\mathcal{S}$, denoted $dist(a, \mathcal{S})$, is $\min_{b \in \mathcal{S}} |\{i \in [n] : a_i \neq b_i\}|$. The L_1 *distance* from a to $\mathcal{S}$, denoted $dist_1(a, \mathcal{S})$, is $\min_{b \in \mathcal{S}} \sum_{i \in [n]} |a_i - b_i|$. Given a parameter $\epsilon \in (0,1)$, an array is ϵ-far from sorted with respect to the Hamming distance or, respectively, L_1 distance, if the corresponding distance from a to $\mathcal{S}$ is at least ϵn.

A *tester* for sortedness is a randomized algorithm that is given parameters $\epsilon \in (0,1)$ and n, and direct access to an input array a. It is required to accept with probability at least $2/3$ if the array is sorted and reject with probability at least $2/3$ if the array is ϵ-far from sorted. We consider two types of testers, Hamming and L_1, corresponding to the two distance measures we defined. The query complexity of a tester is the number of array entries it reads. The goal is to design testers for sortedness with the smallest possible query complexity and running time.

There are two special cases of testers we will discuss. A tester is *nonadaptive* if it makes all queries in advance, before receiving any query answers. A tester has *1-sided error* if it always accepts all sorted arrays.

Bibliographical notes The Hamming testers for sortedness were first studied by Ergün et al. [7]. The L_1-testers (and, more generally, L_p-testers, which use the L_p distance for some $p \geq 1$) were introduced by Berman, Raskhodnikova, and Yaroslavtsev [2]. The two distance measures we discussed, *dist* and $dist_1$, are identical for arrays with 0/1 entries, which we call Boolean arrays. The L_1-tester in [2] builds on the sortedness tester for Boolean arrays by Dodis et al. [6].

Observe that an array $(a_1, a_2, \ldots, a_n)$ of real numbers can be represented by a function $f : [n] \to \mathbb{R}$ defined by $f(i) = a_i$ for all $i \in [n]$. The formulated problem is equivalent to testing if a function f over an ordered finite domain is monotone. In fact, the L_1-tester we will discuss can be easily adapted to work for functions over infinite domains (specifically, bounded intervals), because its complexity is independent of the domain size. The problem of Hamming testing monotonicity of functions over domain $[n]^d$ was first investigated by Goldreich et al. [11]; general partially ordered domains were studied by Fischer et al. [10]. These problems are discussed in the encyclopedia entry "Monotonicity Testing".

Key Results

Ergün et al. [7] designed two Hamming testers for sortedness that run in time $O\left(\frac{\log n}{\epsilon}\right)$. Later, Bhattacharyya et al. [3] and Chakrabarty and Seshadhri [5] gave different testers with the same complexity, with additional features that made them useful as subroutines in testing monotonicity of high-dimensional functions. Fischer [9] proved that the running time of these testers is optimal. Berman, Raskhodnikova, and Yaroslavtsev [2] gave an L_1-tester for sortedness with running time $O(1/\epsilon)$, which is also optimal.

Here we present two Hamming testers from [7; 3] and the L_1-tester from [2].

Hamming Testers for Sortedness

A Tester Based on Binary Search [7] We present and analyze the first tester for sortedness (Algorithm 1) with the assumption that all entries in the array a are distinct. This assumption can be removed by treating element a_i as $\langle a_i, i \rangle$ for all $i \in [n]$.

Algorithm 1: Hamming Tester for Sortedness Based on Binary Search

```
  input : parameters $n$ and $\epsilon$; direct access to array $a$.
1 repeat $\lceil \frac{\ln 3}{\epsilon} \rceil$ times:
2     pick $i \in [n]$ uniformly at random;
3     perform a binary search for the value $a_i$ in the array $a$;
4     if $a_i$ is not located by the binary search, // it leads to another position
5         reject;
6 accept
```

Analysis of the First Tester The tester always accepts all sorted arrays. Now consider an array that is ϵ-far from sorted (in Hamming distance). We say that a position $i \in [n]$ is *searchable* if a_i can be found by a binary search in Step 3, and *not searchable* otherwise. If positions i and j such that $i < j$ are both searchable then $a_i < a_j$, because both a_i and a_j are in the correct position with respect to their common ancestor in the binary search tree. Thus, all numbers in searchable positions are sorted. Since the array is ϵ-far from sorted, at least ϵn positions must be unsearchable. If the tester picks an unserachable position in Step 2, it rejects. The probability that it happens in one trial is at least ϵ. Therefore, the probability that it fails to happen in $\lceil \frac{\ln 3}{\epsilon} \rceil$ trials is at most

$$(1-\epsilon)^{\lceil \frac{\ln 3}{\epsilon} \rceil} \le \exp(-\epsilon \cdot \frac{\ln 3}{\epsilon}) = 1/3. \qquad (1)$$

Thus, the tester rejects an array that is ϵ-far from sorted with probability at least 2/3.

A Tester Based on Graph Spanners [3] The next tester we discuss is based on graph spanners. We can represent the requirement that the array is sorted as a directed graph G, where nodes are positions in $[n]$, and there is an edge (i, j) for all $i < j$. That is, an edge (i, j) represents that $a_i \le a_j$. A *2-spanner* of G is a subgraph H of G with vertex set $[n]$ such that for every edge (i, j) in G, there is a path of length at most 2 from i to j in H. It is not hard to construct a 2-spanner of G with at most $n \log n$ edges[3; 12]. (For example, it can be done using divide-and-conquer as follows: connect all nodes to the one in the middle, orienting the edges towards the nodes with larger indices, remove the middle node, and recurse on the two resulting sublists.)

The tester simply repeats the following step $\lceil \frac{(2\ln 3)\log n}{\epsilon} \rceil$ times: pick a uniformly random edge (i, j) of the 2-spanner H and reject if this edge is *violated*, namely, if $a_i > a_j$. If the tester does not find a violated edge, it accepts.

Analysis of the Second Tester If the input array is sorted, it does not have any violated edges, and the tester always accepts. Now consider an array that is ϵ-far from sorted (in Hamming distance). We call a position $i \in [n]$ *bad* if node i is an endpoint of a violated edge in the 2-spanner H; otherwise, i is *good.* Note that any two good positions i, j such that $i < j$ are connected by a path of length at most 2 of non-violated edges in H. If this path is (i, j), it implies that $a_i \le a_j$. If this path is (i, k, j) for some node k, it implies that $a_i \le a_k \le a_j$. Consequently, for any two good positions i, j such that $i < j$, the numbers a_i and a_j are in the correct order. That is, all numbers in good positions are sorted. As in the analysis of Algorithm 1, we can conclude that there are at least ϵn bad positions. But each bad position is adjacent to a violated edge. Each violated edge can contribute at most two new bad positions. Thus, there are at least $\epsilon n/2$ violated edges. By a simple calculation similar to (1), the second algorithm rejects an array that is ϵ-far from sorted with probability at least 2/3.

L_1-Tester for Sortedness

The L_1-tester for sortedness [2] requires only a uniform sample from the input (as opposed to the ability to query an arbitrary position). It picks $\lceil \frac{2\ln 6}{\epsilon} \rceil$ positions uniformly and independently at random and accepts iff the numbers in these positions are sorted.

The main ingredient in the analysis of the tester is a reduction to the case of Boolean arrays. It states that if the tester is nonadaptive and has 1-sided error, it suffices to show that it works for Boolean arrays. We omit the proof of the reduction.

Clearly, the L_1-tester is nonadaptive and always accepts sorted arrays. Now consider a Boolean array a which is ϵ-far from sorted. It remains to show that it is rejected with probability at least 2/3. Let X_0 be the set of the $\epsilon n/2$ largest indices i for which $a_i = 0$. Similarly, let X_1 be the set of the $\epsilon n/2$ smallest indices i for which $a_i = 1$. It is easy to show that $i < j$ for all $i \in X_1$ and $j \in X_0$, because a is ϵ-far from sorted. The L_1-tester samples no index from X_0 with probability at most 1/6. The same holds for X_1. Thus, by a union bound, with probability at least 2/3, it samples an index from X_0 and an index from X_1, and detects a violation.

Running time We explained why the algorithm that samples $\lceil \frac{2 \ln 6}{\epsilon} \rceil$ positions uniformly and independently at random is an L_1-tester for sortedness. Now we analyze its running time for the case of general arrays. The L_1-tester makes $O(1/\epsilon)$ queries. To determine whether the elements in these positions are sorted, the tester can use bucket sort to sort the sampled positions, and then simply check if the sequence of queried elements is nondecreasing. Since the positions are sampled uniformly at random, the bucket sort can be implemented to run in expected time $O(1/\epsilon)$, where the expectation is taken over the choice of the samples. By standard methods, the algorithm can be modified to run in $O(1/\epsilon)$ time in the worst case. Observe that the running time does not depend on the length of the input. This is impossible for Hamming testers for sortedness, which, as we mentioned, must query $\Omega(\log n)$ positions [9].

Applications

Testers for sortedness are used as subroutines in other property testers, e.g., for monotonicity of high-dimensional functions [6; 5; 2] and for the property that given points represent ordered vertices of a convex polygon [7]. They are also used to construct fast approximate probabilistically checkable proofs for different optimization problems [8]. Ben-Moshe et al. [1] employed sortedness testers (with additional features) to speed up query evaluation in databases.

Open Problems

Consider the case when all numbers in the input array lie in some specified small set such as $[r]$ for some integer r. As we discussed, for Boolean arrays, testing sortedness can be done in $O(1/\epsilon)$ time [6; 2]. It is not hard to see that for larger ranges, it can be done in $O(r/\epsilon)$ time. When $r \ll n$, can one test sortedness it time polylogarithmic in r? Is $O\left(\frac{\log r}{\epsilon}\right)$ running time achievable?

Fischer's lower bound for testing sortedness [9] applies only to $n \ll r$. The best known lower bound that takes into account both parameters is $\Omega(\min(\log r, \log n))$, due to [4], but it applies only to nonadaptive testers.

Cross-References

Monotonicity Testing.

Acknowledgements

The author was supported in part by NSF CAREER award CCF-0845701 and Boston University's Hariri Institute for Computing and Center for Reliable Information Systems and Cyber Security.

Recommended Reading

1. Ben-Moshe S, Kanza Y, Fischer E, Matsliah A, Fischer M, Staelin C (2011) Detecting and exploiting near-sortedness for efficient relational query evaluation. In: ICDT, pp 256–267
2. Berman P, Raskhodnikova S, Yaroslavtsev G (2014) L_p-testing. In: Shmoys DB (ed) STOC, ACM, pp 164–173
3. Bhattacharyya A, Grigorescu E, Jung K, Raskhodnikova S, Woodruff DP (2012) Transitive-closure spanners. SIAM J Comput 41(6):1380–1425
4. Blais E, Raskhodnikova S, Yaroslavtsev G (2014) 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, pp 309–320
5. Chakrabarty D, Seshadhri C (2013) Optimal bounds for monotonicity and Lipschitz testing over hypercubes and hypergrids. In: STOC, pp 419–428
6. Dodis Y, Goldreich O, Lehman E, Raskhodnikova S, Ron D, Samorodnitsky A (1999) Improved testing algorithms for monotonicity. In: RANDOM, pp 97–108
7. Ergün F, Kannan S, Kumar R, Rubinfeld R, Viswanathan M (2000) Spot-checkers. J Comput Syst Sci 60(3):717–751
8. Ergün F, Kumar R, Rubinfeld R (2004) Fast approximate probabilistically checkable proofs. Inf Comput 189(2):135–159
9. Fischer E (2004) On the strength of comparisons in property testing. Inf Comput 189(1):107–116
10. Fischer E, Lehman E, Newman I, Raskhodnikova S, Rubinfeld R, Samorodnitsky A (2002) Monotonicity testing over general poset domains. In: STOC, pp 474–483
11. Goldreich O, Goldwasser S, Lehman E, Ron D, Samorodnitsky A (2000) Testing monotonicity. Combinatorica 20(3):301–337
12. Raskhodnikova S (2010) Transitive-closure spanners: A survey. In: Goldreich O (ed) Property Testing, Springer, Lecture Notes in Computer Science, vol 6390, pp 167–196