Sublinear Algorithms January 12, 2012

Lecture 2

Lecturer: Sofya Raskhodnikova *Scribe(s): Madhav Jha*

1 Testing if a List is Sorted

Input: a list $x_1, ..., x_n$ of arbitrary numbers.

Goal: an ϵ-tester for sortedness of the list with running time $O\left(\frac{\log n}{\epsilon}\right)$.

Testers that do not work. First, let us look at a couple of simple testers that do not work well.

1. Proposed test: Pick a random i and reject if $x_i > x_{i+1}$.
   Counterexample: $\underbrace{111\ldots1}_{n/2}\underbrace{000\ldots0}_{n/2}$.
   Out of $n-1$ consecutive pairs, only one is out of order. Therefore, we would need to repeat our test $\Omega(n)$ times to get a constant probability of error.

2. Proposed test: Pick random $i < j$ and reject if $x_i > x_j$.
   Counterexample: 1 0 2 1 3 2 4 3 5 4 6 5 ..., composed of two interleaved sorted lists.

   The distance of this list to the property "sortedness" is $\geq \frac{1}{2}$ because for each black/blue consecutive pair ((1,0), (2,1), etc.) at least one of the two numbers has to be changed to make the list sorted. *But* any other pair is correctly ordered. This implies, by Birthday Paradox, that we need to select $s = \Omega(\sqrt{n})$ numbers to find a pair of numbers that is out of order with probability $\geq \frac{2}{3}$.

A working test. We present the tester for sortedness from [DGL+99]. The idea is to associate positions in the list with vertices of the directed line, L_n. That is we construct the line graph $L_n = ([n], \{(i, i+1) : i \in [n-1]\})$. We then construct a graph (called 2-spanner graph of the line) on the vertex set $[n]$ by adding a few "shortcut" edges (i, j) for $i < j$. The resulting graph will have the property that each pair of vertices is connected by a path of length at most 2. We show that sampling edges uniformly and independently at random from this construction yields a sublinear time tester for the sortedness property of the list. But first we show how to construct 2-spanner H with the required property and having at most $n \log n$ edges.

2-Spanner construction. Our 2-spanner, H, is a graph with vertex set $[n]$. We construct the edge set of H recursively. First, define the middle node $v_{mid} = n$. Use this node as a *hub*: namely, add edges (v, v_{mid}) for all nodes $v < v_{mid}$ and edges (v_{mid}, v) for all nodes $v > v_{mid}$. Then recurse on the two line segments resulting from removing v_{mid} from the current line. Proceed until each line segment contains exactly one node.

H is a 2-spanner on the vertex set $[n]$, since every pair of nodes $u, v \in [n]$ is connected by a path of length at most 2 via a hub. This happens in the stage of the recursion during which u and v are separated into different line segments, or one of these two nodes is removed.

To get the bound on the size of the 2-spanner, observe that there are $\log n$ stages of the recursion. In each stage, every non-hub node connects to the hub in its current line segment, adding a total of at most n edges. Therefore, the constructed spanner has at most $n \log n$ edges.

The test. Algorithm 1 is the required tester.

Algorithm 1: 2-TC Spanner based sortedness tester.

1 Let H be the 2-spanner on the vertex set $[n]$ constructed above with at most $n \log n$ edges.
2 Select $s = 4 \log n/\epsilon$ edges uniformly and independently from H and query the list on their endpoints.
3 If any selected edge (x_i, x_j) is violated, that is, $x_i > x_j$, reject; otherwise, accept.

Analysis: We say a vertex $x \in [n]$ is assigned a *bad* label if x has an incident violated edge in H; otherwise, x has a *good* label. We claim the following.

Claim 1. *All vertices with good labels are sorted.*

Proof. Consider any two good numbers (that is, numbers labelled good), x_i and x_j. They are connected by a path of (at most) two good edges (x_i, x_k), (x_k, x_j) in H by definition of 2-spanner. Therefore, we have $x_i \leq x_k$ and $x_k \leq x_j$ implying $x_i \leq x_j$, as required. □

Claim 2 (Repairing partially sorted list)**.** *The list can be changed into a sorted list by modifying it only on the vertices labelled bad.*

Proof. Follows from Claim 1 and Exercise 2.1. □

Since the list is ϵ-far from sortedness, Claim 2 implies that there are at least ϵn vertices with bad labels. Since each violated edge in H contributes at most 2 distinct endpoints to bad vertices, the number of violated edges in H is at least $\epsilon n/2$. Since H has at most $n \log n$ edges, the algorithm finds a violated edge (and therefore rejects) with probability at least 2/3, as required.

The following exercise was used in the proof above.

Exercise 2.1. *Let $i_1 < i_2 < \cdots < i_r$ be indices in $[n]$ and let $S = \{i_j : j \in [r]\}$. Given a (partially) sorted list $\ell : S \to \mathbb{R}$ satisfying $\ell(i_1) \leq \ell(i_2) \leq \cdots \leq \ell(i_r)$, one can extend it to a sorted list on the entire domain $[n]$ by only modifying points in $[n] \setminus S$.*

Generalization. The same test/analysis apply to any *edge-transitive* property of a list of numbers that *allows extension*. We define these terms for properties of functions defined on $[n]$. (Observe that a list of n numbers can equivalently be viewed as a function $f : [n] \to \mathbb{R}$ where $f(i)$ gives the i'th number in the list.)

Definition 3. *A property $\mathcal{P}$ (of a function defined on $[n]$) is* edge transitive *if the following conditions hold:*

1. *It can be expressed in terms of requirements on* ordered *pairs of numbers such as (x, y).*
2. *It is transitive: whenever (x, y) and (y, z) satisfy (1), so does (x, z).*

An edge-transitive property $\mathcal{P}$ allows extension if any function that satisfies (1) on a subset of the numbers can be extended to a function with the property.

As mentioned earlier, the same test/analysis as used for sortedness of a list of numbers (an edge-transitive property which allows extension) holds for *any* edge-transitive property of functions that allows extension. In particular, it applies to testing whether a function of the form $f : [n] \to \mathbb{R}$ is *Lipschitz*. This is the content of the next section.

2 Testing Lipschitz Property on Line [JR11]

Consider a function $f : D \to R$ mapping a metric space $(D, dist_D)$ to a metric space $(R, dist_R)$, where $dist_D$ and $dist_R$ denote the distance functions on the domain D and range R, respectively. Function f has Lipschitz constant c if $dist_R(f(x), f(y)) \leq c \cdot dist_D(x, y)$ for all $x, y \in D$. We call such a function c-Lipschitz and say a function is Lipschitz if it is 1-Lipschitz. (Note that rescaling by a factor of $1/c$ converts a c-Lipschitz function into a Lipschitz function.) In other words, Lipschitz functions are not very sensitive to small changes in the input.

Lipschitz continuity is a fundamental notion in mathematical analysis, the theory of differential equations and other areas of mathematics and computer science. A Lipschitz constant c of a given function f is used, for example, in probability theory in order to obtain tail bounds via McDiarmids inequality [McD89]; in program analysis, it is considered as a measure of robustness to noise [CGLN11]; in data privacy, it is used to scale noise added to output $f(x)$ to preserve differential privacy of a database x [DMNS06]. In these three examples, one often needs to compute a Lipschitz constant of a given function f or, at least, verify that f is c-Lipschitz for a given number c. However, in general, computing a Lipschitz constant is computationally infeasible. The decision version is undecidable when f is specified by a Turing machine that computes it, and NP-hard if f is specified by a circuit.

Note that the Lipschitz property was defined in terms of pairs of domain elements. Consider a function $f : [n] \to \mathbb{R}$, where the domain and range are equipped with distance functions $dist_D(x, y) = |x - y|$ and $dist_R(f(x), f(y)) = |f(y) - f(x)|$. We say a pair (x, y) is violated if $|f(y) - f(x)| > |y - x|$. Then if (x, y) and (y, z) are not violated, it implies that neither is (x, z). Thus, Lipschitz property for function on $[n]$ is an edge-transitive property. Exercise 2.2 shows that it also allows extension. Therefore, from observation made in the previous section, we get an $O(\frac{\log n}{\epsilon})$ query complexity (and running time) Lipschitz tester for functions of the form $f : [n] \to \mathbb{R}$ via the optimal 2-TC-spanner construction of the line.

Exercise 2.2. *Show that the Lipschitz property of function $f : [n] \to \mathbb{R}$ allows extension.*

Exercise 2.3. *Does the spanner-based test apply if the range is $\mathbb{R}^2$ with Euclidean distances? What about $\mathbb{Z}^2$ with Euclidean distances?*

We saw two properties of n numbers, sortedness and Lipschitz. The testers for both these properties had running time $O(\frac{\log n}{\epsilon})$. While it is known that the tester is optimal for the former (sortedness property), the following question about Lipschitz property remains open.

Open Problem 1 (Testing Lipschitz property of line)**.** *What is the optimal query complexity of testing Lipschitz property of function $f : [n] \to \mathbb{R}$?*

3 Testing Monotonicity on Hypercube [GGL$^+$00, DGL$^+$99]

A function $f : \{0,1\}^n \to \{0,1\}$ is monotone if increasing a bit of x does not decrease $f(x)$. In this section, we explore the problem of testing monotoncity. Towards this, we think of the domain $\{0,1\}^n$ as the directed hypercube. Edge (x, y) is violated by f if $f(x) > f(y)$. The general idea behind the algorithm is to show that functions that are far from monotone violate many edges.

The test. We present the algorithm from [GGL$^+$00, DGL$^+$99].

Algorithm 2: Edge tester for monotonicity.

1 Pick $\frac{2n}{\epsilon}$ edges (x, y) uniformly at random from the hypercube.
2 Reject if some (x, y) is violated (i.e. $f(x) > f(y)$). Otherwise, accept.

Theorem 4. *There is a nonadaptive one-sided error tester for the monotonicity property of functions $f : \{0,1\}^n \to \{0,1\}$ that runs in $O(\frac{n}{\epsilon})$ time.*

Proof. Algorithm 2 is the required tester. If f is monotone, Algorithm 2 always accepts. If f is ϵ-far from monotone, by Witness Lemma (refer Lecture 1), it suffices to show that at least $\frac{\epsilon}{n}$ fraction of edges (that is, $\frac{\epsilon}{n} \cdot n2^{n-1} = \epsilon 2^{n-1}$ edges) are violated by f. Let $V(f)$ denote the number of edges violated by f. Then, it is sufficient to prove the following contrapositive:

Lemma 5 (Repair lemma)**.** *If $V(f) < \epsilon 2^{n-1}$, then f can be made monotone by changing $< \epsilon 2^n$ values.*

To prove the lemma, we show how to transform an arbitrary function $f : \{0,1\}^n \to \{0,1\}$ into a monotone function by changing f on a set of points, whose size is related to the number of the hypercube edges violated by f. This is achieved by repairing one dimension of the hypercube at a time with the swapping operator S_i, defined below. The operator modifies values of f on the endpoints of each violated edge in dimension i by swapping the values at the endpoints.

Definition 6 (Swapping operator S_i)**.** *Let $f : \{0,1\}^n \to \{0,1\}$, $x \in \{0,1\}^n$, y be obtained from x by flipping the i'th bit, and $x \prec y$. Then*

$$S_i[f](x) = f(y) = 0 \text{ and } S_i[f](y) = f(x) = 1 \text{ if } f(x) > f(y),$$
$$S_i[f](x) = f(x) \text{ and } S_i[f](y) = f(y) \text{ otherwise.}$$

We would like to argue that while we are repairing dimension i with the swapping operator, other dimensions are not getting worse. This is the content of the next lemma. For $i \in [n]$, let V_i denote the number of violated edges along dimension i.

Lemma 7. *For all $i, j \in [n]$, where $i \neq j$, and every function $f : \{0,1\}^n \to \{0,1\}$, applying the* swapping *operator S_i does not increase the number of violated edges in dimension j, i.e., $V_j(S_i[f]) \leq V_j(f)$.*

Proof. The important observation is that the statement of the lemma is concerned with only two dimensions, i and j. Ignoring edges in all other dimensions, we can view the hypercube as 2^{n-2} disconnected squares. Each of these squares represents a two-dimensional boolean function, obtained from f by fixing all coordinates other than i and j. To prove the lemma, it is enough to show that it holds for two-dimensional functions. It will demonstrate that the swapping operator S_i does not increase the number of violated edges in dimension j.

We prove the lemma for two-dimensional functions by case analysis. We depict a two-dimensional function f as shown. A directed edge between $f(x)$ and $f(y)$ indicates that x and y differ in exactly one coordinate and $x \prec y$.

Without loss of generality, let $i = 1$ and $j = 2$. Then the swapping operator swaps edges in the horizontal dimension, and the goal is to prove that the number of violated edges in the vertical dimension does not increase.

Case 1: f does not have violated edges in the horizontal dimension. Then $f \equiv S_i[f]$, and hence $V_j(S_i[f]) = V_j(f)$.

Case 2: Both horizontal edges are violated in f. Then the vertical edges are swapped, and $V_j(S_i[f]) = V_j(f)$.

Case 3: The upper edge is violated; the lower one is not. Consider two possibilities: $f(00) = f(10)$ and $f(00) \neq f(10)$. If $f(00) = f(10)$, then the vertical edges are swapped, and $V_j(S_i[f]) = V_j(f)$. If not, then since the lower edge is not violated, $f(00) = 0$ and $f(10) = 1$. In this case, the vertical violated edge is repaired.

Case 4: The lower edge is violated; the upper one is not. This case is symmetrical to Case 3. □

The crux of the proof is showing how to make a function $f : \{0,1\}^n \to \{0,1\}$ monotone by redefining it on at most $2 \cdot V(f)$ points. We apply a sequence of swapping operators as follows: we define $f_0 = f$ and for all $i \in [n]$, let $f_i = S_i[f_{i-1}]$.

$$f = f_0 \xrightarrow{S_1} f_1 \xrightarrow{S_2} f_2 \to \cdots \to f_{n-1} \xrightarrow{S_n} f_n.$$

We claim that f_n is monotone. By definition of the swapping operator S_i, each step above makes one dimension i free of violated edges. By Lemma 7, S_i preserves the monotonicity property along dimensions fixed in the previous steps. Thus, eventually there are no violated edges, and f_n is monotone.

Now we bound the number of points on which f and f_n differ, that is, $Dist(f, f_n)$. For all $i \in [n]$,

$$Dist(f_{i-1}, f_i) \quad = \quad Dist(f_{i-1}, S_i[f_{i-1}]) \leq 2 \cdot V_i(f_{i-1}) \leq 2 \cdot V_i(f). \tag{1}$$

The first inequality holds because S_i modifies f only on the endpoints of violated edges along dimension i. The final inequality holds because, by Lemma 7, operators S_j for $j \neq i$ do not increase the number of violated edges in dimension i. The distance from f to f_n is

$$Dist(f, f_n) \leq \sum_{i \in [n]} Dist(f_{i-1}, f_i) \leq \sum_{i \in [n]} 2 \cdot V_i(f) = 2 \cdot V(f). \tag{2}$$

The first inequality above follows from the triangle inequality while the second uses Equation 1. This completes the proof. For completeness, we show the full argument.

Consider a function f which is ϵ-far from the monotonicity property. Since f_n is monotone, $Dist(f, f_n) \geq \epsilon \cdot 2^n$. Together with (2), it gives $V(f) \geq \epsilon \cdot 2^{n-1}$, as required. □

References

[CGLN11] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara NavidPour. Proving programs robust. In *SIGSOFT FSE*, pages 102–112, 2011.

[DGL+99] Yevgeniy Dodis, Oded Goldreich, Eric Lehman, Sofya Raskhodnikova, Dana Ron, and Alex Samorodnitsky. Improved testing algorithms for monotonicity. In *RANDOM*, pages 97–108, 1999.

[DMNS06] Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In *TCC*, pages 265–284, 2006.

[GGL+00] Goldreich, Goldwasser, Lehman, Ron, and Samorodnitsky. Testing monotonicity. *COMBINAT: Combinatorica*, 20, 2000.

[JR11] Madhav Jha and Sofya Raskhodnikova. Testing and reconstruction of lipschitz functions with applications to data privacy. In *FOCS*, pages 433–442, 2011.

[McD89] Colin McDiarmid. On the method of bounded differences. In *Surveys in Combinatorics, 1989, J. Siemons ed., London Mathematical Society Lecture Note Series 141, Cambridge University Press*, pages 148–188, 1989.