6.5240 Sublinear Time Algorithms December 9, 2022

Lecture 24 (Makeup)

Lecturer: Ronitt Rubinfeld *Scribe: Thomas Guo*

1 Outline

The following topics were covered in class:

- Monotonicity testing on a sequence or list (function with domain $\{1, 2, \ldots, n\}$)
- Monotonicity testing on a function $f : \{0, 1\}^n \to \{0, 1\}$ (binary labeling of a hypercube)

2 Monotonicity testing on a sequence

2.1 Introduction

We are now interested in property testing of functions rather than distributions, where each query returns the value of the function at a given point or index. We will start with monotonicity testing on sequences, or functions with domain $\{1, 2, \ldots, n\}$. This problem has many real-world applications such as checking whether the books in a library are sorted alphabetically.

Goal: $c \log n$ time algorithm to determine whether a list of size n is sorted or ϵ-far from sorted.

Definition 1 A sequence of length n is *ϵ-close* to sorted if at most ϵn values can be deleted to make it sorted. A sequence is *ϵ-far* from sorted otherwise.

Here, we are talking about being sorted in increasing order only. Note that this problem is equivalent to determining whether the longest increasing subsequence of a list has length n or length less than $(1-\epsilon)n$. (This is because if k is the minimum number of values needed to be deleted to make the list sorted, the remaining $n - k$ values must form an increasing subsequence of the original list. Also, to minimize the number of deleted values, we maximize the length of the increasing subsequence, so $n - k$ is simply the length of the longest increasing subsequence.)

We have the standard requirements for our algorithm:

- **Pass** sorted lists.
- **Fail** lists that are ϵ-far from sorted with probability at least 3/4.

This problem can be solved in $O(\epsilon^{-1} \log n)$ time, which is provably tight.

2.2 First attempt

Algorithm: we repeatedly pick a random entry and test that the entry and its right neighbour are in the correct order (i.e. pick random i and test that $y_i \leq y_{i+1}$).

However, here is one "bad" input type:

$$1, 2, 3, \ldots, n/4, 1, 2, 3, \ldots, n/4, 1, 2, 3, \ldots, n/4, 1, 2, 3, \ldots, n/4.$$

It is difficult for our algorithm to find the "breakpoint" since there are only $O(1)$ of them (exactly 3 in this case), so we require $O(n)$ queries to fail with constant probability. Note that this input is very far from sorted, as at least $\frac{3n}{4} - 3$ elements must be removed.

2.3 Second attempt

Algorithm: we pick many random entries and pass if all of them are in the correct order.

However, we again have a "bad" input type: swap adjacent neighbours, so we get

$$2, 1, 4, 3, 6, 5, \ldots, n, n-1.$$

Now, we will only fail if we pick two neighbours, so we require $O(\sqrt{n})$ queries to fail with constant probability by the birthday problem. Once again, note that this input is very far from sorted, as at least $\frac{n}{2}$ elements must be removed.

2.4 A minor simplification

Claim: we can assume the list elements are distinct, by treating the element x_i as the ordered pair $y_i = (x_i, i)$.

When comparing two ordered pairs, we compare the first element and tiebreak by the second element. Now, any increasing subsequence remains increasing, and any inversion (i.e. $i > j$ with $x_i < x_j$) remains an inversion, so the longest increasing subsequence does not change. Hence, solving the problem on the y_i's yields the same answer as solving the problem on the x_i's, so we can work with the distinct y_i's moving forward.

2.5 A test that works

Algorithm: repeat the following $O(1/\epsilon)$ times

1. Pick a random index i.
2. Query the value of y_i.
3. Perform a binary search for y_i.
4. If the binary search does not find y_i at location i (note that if we find y_i, it must be at location i by our assumption of distinctness), FAIL.
5. If the binary search finds any inconsistencies (i.e. elements in decreasing order), FAIL.
6. Otherwise, PASS.

The runtime is clearly $O(\epsilon^{-1} \log n)$, since each binary search takes $O(\log n)$ time. Now, we will prove correctness.

Definition 2 An index i is *good* if a binary search for y_i is successful (i.e. returns y_i at location i without finding any inconsistencies).

Our algorithm can be restated as picking $O(1/\epsilon)$ random i's and passing if and only if they are all good. If the list is sorted, then clearly all i's are good (using distinctness), so the test always passes. Now, we will show that if a list is ϵ-far from sorted, it fails the test with high probability.

Key observation: good elements form an increasing subsequence.

Proof: suppose $i < j$ and i, j are both good. We will show $y_i < y_j$. Consider k, the least common ancestor of i, j in the binary search. At this location, we must have had y_i go left and y_j go right, since $i < j$ and by assumption, they take different paths at this point. Thus, $y_i < y_k < y_j$, as desired.

Thus, for any list that is ϵ-far from sorted, the longest increasing subsequence has length at most $(1-\epsilon)n$, so there are at most $(1-\epsilon)n$ good indices. This means that at least ϵn indices are not good, so each iteration of the test fails with probability at least ϵ. Thus, after $2/\epsilon = O(1/\epsilon)$ iterations, the algorithm fails with probability at least

$$1-(1-\epsilon)^{2/\epsilon} \approx 1-\frac{1}{e^2} > 0.75$$

2.5.1 Note on adaptivity

Though we perform binary search during our algorithm which seems to be adaptive, it is actually very straightforward to implement it non-adaptively. For each randomly chosen y_i, we instead query the indices for binary search *assuming* that we end up at index i (this is equivalent to binary searching for i in the list $[1, 2, \ldots, n]$), and fail if these values are not in increasing order. Since these queried indices are independent of the values y_i, this algorithm is thus non-adaptive.

3 Monotonicity testing on functions

Definition 3 A function $f : \{0,1\}^n \to \{0,1\}$ is *monotone* if and only if increasing a bit of x does not decrease $f(x)$.

Note that the domain of f is a partially ordered set, not totally ordered like we were working with before.

Goal: determine whether f is monotone or ϵ-far from monotone (i.e. f has to change on more than ϵ-fraction of its points to become monotone).

Time complexity:

1. Today: $O(n/\epsilon)$, logarithmic in the size of the input, 2^n.
2. Newer: $\Theta(\sqrt{n}/\epsilon^2)$ for nonadaptive tests, $\Omega(n^{1/3})$.

Definition 4 When looking at the n-dimensional hypercube, we say an increasing edge $x \to y$ is *violated* if $f(x) > f(y)$.

Idea: show that functions that are ϵ-far from monotone violate many edges.

Algorithm: $EdgeTest(f, \epsilon)$: pick $2n/\epsilon$ edges (x, y) uniformly at random from the hypercube, and reject if any (x, y) is violated. Otherwise, accept.

If f is monotone, $EdgeTest$ always accepts. If f is ϵ-far from monotone, we will show that at least $\frac{\epsilon}{n}$-fraction of the edges (i.e. $\epsilon/n \cdot 2^{n-1}n = \epsilon 2^{n-1}$) are violated by f. Note that we have $n2^{n-1}$ total edges since each of the 2^n nodes have n adjacent edges (one for each coordinate flip), and we divide by two since each edge is counted twice.

Now, let $V(f)$ denote the number of edges violated by f. We will show the contrapositive, that if $V(f) < \epsilon 2^{n-1}$, then f can be made monotone by changing fewer than $\epsilon 2^n$ values.

Lemma 1 Repair lemma: f can be made monotone by changing at most $2 \cdot V(f)$ values.

Proof idea: transform f into a monotone function by repairing edges in one dimension at a time. We repair a $1 \to 0$ edge by swapping it to become $0 \to 1$.

Claim: repairing all edges in dimension i does not increase V_j for all dimensions $j \neq i$.

Since we only look at two dimensions at a time (i and j), it is enough to prove this claim for squares, since the edges in dimensions i and j form 2^{n-2} disjoint squares (by varying the other $n-2$ coordinates).

Now, this is just casework. If both dimension i edges are swapped or neither are swapped, then the resulting (unordered) pair of edges in dimension j is the same, so the number of violating edges in dimension j does not increase. Otherwise, if exactly one edge in dimension i is swapped, consider the other edge. If it is $v \to v$ for $v \in \{0, 1\}$, then the resulting pair of edges in dimension j is the same again. If it is $0 \to 1$, then we actually fix the violating edge in dimension j and are left with one fewer violating edge.

Thus, repairing all edges in dimension i indeed does not increase V_j for all $j \neq i$, so we can iterate through the dimensions $1 \leq i \leq n$ and repair all edges in dimension i each time, and we must be left with no violating edges at the end. After repairing dimensions 1 through $i - 1$, there are still at most V_i violating edges in dimension i, so we repair at most V_i edges on iteration i. Hence, the total number of repaired edges is at most $V_1 + V_2 + \cdots + V_n = V(f)$, so we can indeed transform f into a monotone function by changing at most $2 \cdot V(f)$ values (as each repaired edge changes two values). This concludes the proof of the lemma.

Finally, the repair lemma implies that if $V(f) < \epsilon 2^{n-1}$, then f can indeed be made monotone by changing fewer than $\epsilon 2^n$ values. Taking the contrapositive, this shows that if f is ϵ-far from monotone, then at least $\epsilon 2^{n-1}$ edges are violated by f, which is $\frac{\epsilon}{n}$-fraction of the total edges. Hence, if f is ϵ-far from monotone, each random edge in $EdgeTest$ is violating with probability at least ϵ/n, so the probability that $EdgeTest$ fails is at least

$$1 - (1 - \epsilon/n)^{2n/\epsilon} \approx 1 - \frac{1}{e^2} > 0.75$$