CS 590 Sublinear algorithms Feb 2, 2015 – **Monotonicity Testing Begins**

Lecture 1

Lecturer: Akash Kumar *Scribe: Akash Kumar*

1 Introduction to Property Testing

Let us start the story of Property Testing – a venerable field of Theoretical Computer Science. The goal in this field is to get super fast algorithms for a whole bunch of problems. In fact, we require so fast algorithms that do not even examine the entire dataset (i.e., examine only $o(n)$ many elements of the dataset where the dataset is assumed to have n elements). Let us motivate this better through a few examples. In what follows, we first discuss for what kinds of problems can we expect such a super fast algorithm

1.1 The forbidden question

You might have heard of the curious small world property. It's a phenomenon observed in social networks like Facebook which says that the distance between any pair of people on the Facebook graph is no bigger than 6. Suppose, I ask you – is this true for all pairs of people in the Facebook graph? You are tempted to say no as an answer to this decision question because you feel that there is some pair of people who are more than 6 hops away (as in consider someone who just joined Facebook and has no friends yet). A more methodical approach to this question would force you to look at everyone in the Facebook graph and that sadly is not a sublinear time solution. The obstacle here is that there is a $\forall$ quantifier built-in the problem statement that forces the algorithms necessarily have $\Omega(n)$ runtime.

Let us consider another question. Are there any pair of students at Purdue University who have the same fingerprints? Again, you are tempted to say no. But a methodical answer would force you to consider the fingerprint of every Purdue student – which is a very boring task. Note that this time around, the question had a $\exists$ quantifier. So, it seems we cannot have *sublinear time* algorithms for questions that have a universal quantifier attached. Thus, its natural to wonder

1.2 What kinds of questions can we get a super fast algorithm for

The previous subsection rules out sublinear time algorithms for "exact" decision questions. What if we were a bit sloppy with the problem statement? For example, we can ask – is it safe to hypothesize that Facebook graph is "close to" having the small world property ? Or is it safe to hypothesize that Purdue University is "close to" having no pair of people with the same fingerprints? Depending on some quantification (implicit in close to/ or far-off that we will mostly use later), this might become an easy question to answer. You can randomly sample enough pairs of people from Facebook graph and see what fraction violates the small world property. If that fraction is negligible, we can trust the small world hypothesis. Similarly, if you randomly sample enough pair of people at Purdue and most of them have different fingerprints, you can say that at most a negligible fraction of Purdue students have same fingerprints with a high degree of confidence (that depends on the number of sampled pairs). This narrative

suggests that we can ask these "relaxed" questions. This suggests some characteristic features of the kinds of questions we will ask in the property testing model.

- The problems asked will necessarily have a "gap". The questions will check whether an input has a property or whether its blatantly far from having the property.
- The algorithms will be necessarily randomized. You cannot demand perfection from an algorithm that does not check the entire input. If the algorithm does not check the entire input, then an unlucky run can fail in the simplest tasks. For example, you might fail to notice that an array is not sorted even if you miss checking a single entry in the array. This also suggests that we better have a failsafe suggested below.
- The probability of wrong answer should be small (say at most $1/3$ – or any constant less than half). Very often, we are interested in developing testers that have *one-sided error*. These testers always accept the object which has the property. For these testers, we just need to have a tester which returns an incorrect answer with probability bounded above from 1. Note that this does not affect the asymptotic query complexity of the tester as by just repeating the test a constant number of times we can bring the probability of error down to a desired constant.
- Finally, we assume that we can query whatever part of the input we like in constant time which is also often expressed by saying that we assume oracle access to the input. In fact, this is one necessary simplification to build a useful theory. There is some justification to this – if accessing data takes too long, then perhaps getting a sublinear time algorithm for that problem is already hopeless.

2 A more abstract treatment

In property testing, as motivated above we do not ask an exact decision question. Instead we ask one with a "gap". Here is the setup. You are given a mathematical universe $\mathcal{U}$, a mathematical object $\mathcal{O} \in \mathcal{U}$, and a property $\mathbb{P}$ (where $\mathbb{P} \subseteq \mathcal{U}$ is a collection of objects from the universe that have a similar behavior). You are asked to determine using a probabilistic algorithm with error probability no more than $1/3$ whether

- $\mathcal{O} \in \mathbb{P}$
- $d(\mathcal{O}, \mathbb{P}) \geq \varepsilon$ (See below for the meaning of this expression – this communicates the intent that $\mathcal{O}$ blatantly fails to have $\mathbb{P}$)

d:$\mathcal{U} \times \mathcal{U} \to \mathbb{R}^{\geq 0}$ is the distance function which measures distance between a pair of objects in the universe (which is a non-negative real number). This distance function is also required to satisfy a few desirable properties. These are

- $d(\mathcal{O}, \mathcal{O}) = 0 \quad \forall \mathcal{O} \in \mathcal{U}$
- $d(\mathcal{O}, \mathcal{O}') = d(\mathcal{O}', \mathcal{O})$ i.e., the distance function is symmetric
- $d(\mathcal{O}_1\mathcal{O}_3) \leq d(\mathcal{O}_1, \mathcal{O}_2) + d(\mathcal{O}_2, \mathcal{O}_3) \quad \forall \mathcal{O}_1, \mathcal{O}_2, \mathcal{O}_3 \in \mathcal{U}$ i.e., $d()$ satisfies triangle inequality

This is sometimes also captured by saying that the distance function $d()$ is a *metric*. We also define distance of an object from a set which is defined as

$$d(\mathcal{O}, \mathbb{P}) := \inf_{\mathcal{O}' \in \mathbb{P}} d(\mathcal{O}, \mathcal{O}')$$

that is, distance of an object $\mathcal{O}$ from a set is defined to be the distance of $\mathcal{O}$ from the "closest object" which has the property. Typically, we deal with discrete problems for which the infimum can in fact be replaced by minimum in the definition above. The distance function is usually problem dependent – that is, depending on the problem you want to solve you want to appropriately define some distance function. Again, we emphasize that this is unlike exact decision questions (which necessarily involve $\forall$ or $\exists$ quantifiers) where your goal is to determine whether

- $\mathcal{O} \in \mathbb{P}$
- $\mathcal{O} \notin \mathbb{P}$

Now, that we have defined the setup abstractly, let us try to see a few concrete examples.

3 Monotonicity testing

The basic setup is fairly simple to describe. You are given an array A containing n natural numbers. Your goal is to determine whether the array is sorted vs whether its far from being sorted. One natural way to define what "far" might mean is to use the fractional hamming distance between a pair of arrays. Thus, for 2 arrays A and B we define their distance as

$$d(A, B) := \Pr_{i \in [n]} (A[i] \neq B[i]).$$

Then note that the distance of an array from the monotone arrays (or the set of arrays sorted in ascending order on n elements) can be defined as the *normalized* minimum number of modifications that need to be made to get a sorted array. Let $\mathcal{M}$ denote the set of all monotone n-element arrays

Now, we can formalize the question as – given an array A determine whether

- $A \in \mathcal{M}$
- $d(A, \mathcal{M}) \geq \varepsilon$

We first make a crucial definition.

Definition 1 *(Violation) A pair of indices $i, j \in [n]$ is a violation to monotonicity if $i < j$ but $A[i] > A[j]$*

So, what can a natural test for this question look like? Perhaps a first approach might do the following to catch a violation to monotonicity.

This tester picks $t = t(n, \varepsilon)$ many samples. We are interested in understanding if the number of these samples can be made sublinear in n. Unfortunately, it can be shown that such a simple tester does not work. Consider the following input

Algorithm 1 Adjacent pair tester

$i = 0$
for ($i < t(n, \varepsilon)$) **do** // $t(n, \varepsilon)$ is the threshold defined later
  Generate a random $r \in [n-1]$
  If the adjacent pair (A_r, A_{r+1}) violates monotonicity, reject
  $i \leftarrow i + 1$
end for

Accept and exit

$$A = [1, 1, 1, 1, 1, 0, 0, 0, 0, 0]$$

which has $n/2$ ones followed by $n/2$ zeroes. This tester can catch a violation only if it picks the middle pair of indices. The probability that this happens in any given trial is $1/n$. So, the expected number of samples this tester needs is $\Omega(n)$. Which means that this is sadly not a sublinear query complexity tester.

Perhaps, we need a better idea. How about querying a bunch of random indices and comparing them all? Let us see how well we do with this tester.

Algorithm 2 Random Subset tester

Pick randomly (with replacement) $t(n, \varepsilon)$ samples
if (any violated pair found) **then**
  Reject and Exit
else
  Accept and Exit
end if

We make the following claim which shows that this is indeed a sublinear tester.

Claim 2 *The above tester with $t = \Omega\left(\sqrt{\frac{n}{\varepsilon}}\right)$ queries rejects an array A that is ε-far from $\mathcal{M}$ with probability $\geq \frac{2}{3}$*

Proof Let us define an indicator variable

$$X_{ij} = \begin{cases} 1 & \text{if the pair } (i, j) \text{ is a violation} \\ 0 & \text{otherwise} \end{cases}$$

Let $X = \sum_{i,j} X_{ij}$ count the total number of violations observed in the sample. We will show that $X \geq 1$ with high probability. First, we see that since the array is ε-far from monotone, then it must follow that there are at least εn violated pairs inside the array. This is because even the best fix which recovers the monotone array needs to change at least εn entries and each of those entries contributes one new violated pair – i.e., no pair of indices $u < v$ are such that they both need to be changed in the best fix, (u, v) is a violation and this is the only

violation that u or v participate in. (This contradicts the minimality of the best fix. Can you see why?) Next, observe that just by linearity of expectations,

$$E[X] = \sum_{i,j} E[X_{ij}] \geq \binom{t}{2} \cdot \frac{\varepsilon n}{n^2} = \binom{t}{2} \cdot \frac{\varepsilon}{n}.$$

Thus, $E[X] = 1$ if $t = \Omega\left(\sqrt{\frac{n}{\varepsilon}}\right)$. Finally, by using the second moment method, it can be shown that the

$$\Pr(X = 0) \leq 1/3$$

which finishes the proof. ■

Well, now that we know this is a sublinear tester, its time to ask for more. Did we analyze the above algorithm perfectly? Is there any hope that by some clever trick one can prove an even better upper bound on the number t of samples needed? Turns out, asymptotically, the above bound is the best possible.

Claim 3 *The Random Subset Tester needs $t = \Omega(\sqrt{n})$ many queries to catch an array that is $\Omega(1)$-far from $\mathcal{M}$.*

Proof To show this claim, we will try to construct an adversarial input. This input has the property that it is ε-far from being sorted and if you make fewer than $\Omega(\sqrt{n})$ queries, then whp you cannot reject it. Let us consider the following input

$$A = [10, 9, 8, 7, 6|20, 19, 18, 17, 16|30, 29, 28, 27, 26 \cdots].$$

The input is an array of length n made up of many segments of length 5 such that the violations are all inside the same segment. There are overall $n/5$ such segments. In order to get a violation, the tester must hit the same segment more than once. By birthday paradox, in order to see a violation with constant probability of success, this requires $\Omega(\sqrt{n})$ many queries. The proof also needs an additional argument to show that the input is in fact ε-far. You are requested to provide the details for this argument. ■

Finally, its time to get even more greedy. Can we get an even better tester for testing monotonicity? Turns out the answer is yes. Below, we provide an $O(\frac{1}{\varepsilon} \log n)$ tester for monotonicity. This is also the best possible by a theorem of Fergun et al. Below we present the tester

Algorithm 3 Binary Search Based Tester

Pick randomly $S \subseteq [n]$ with $|S| = 100/\varepsilon$ many indices from A
for ($i \in S$) **do**
  Do binary search for A_i
  If the search does not end at location i, reject.
end for

Accept and Exit

So, the tester picks a bunch of uniformly random indices and does binary search for the elements stored at those indices. Why does this test work? To understand this, let us begin

by calling an index i *good* if the binary search is successful for A_i. Thus, the above algorithm accepts an array if all the queried indices are good. We have the following claim which is crucial to the correctness of this algorithm.

Claim 4 *If an array A is ε-far then it has at most $(1-\varepsilon)n$ many good elements.*

Proof Exercise ■

With the above claim, finishing the rest of the proof is easy. Let p denote the probability that all of the queried indices are good. Then

$$p \leq (1-\varepsilon)^{100/\varepsilon} \leq 1/e^{100}$$

Thus the algorithm fails with a very small probability as desired. And this shows that the algorithm is correct.

3.1 The Range Effects

Now, let us try to see what happens if we do monotonicity testing over a smaller range. To begin with let us say we are given a boolean array – that is all elements inside the array are either 0 or 1. In this case, can we determine whether the array is sorted or ε-far any quicker than $O(1/\varepsilon \log n)$? Turns out, the answer is again yes. This time in fact we can get an $O(1/\varepsilon)$ query complexity tester. Note that this tester has a constant query complexity – independent of n. We will just sketch the tester and the proof of its correctness super briefly. The tester just picks a random set of $100/\varepsilon$ many indices and compares every pair of elements at these indices. Thus, it is in fact the random subset tester that we saw earlier.

Why does this work? Well, it clearly accepts a monotone array (as it does not find any violations which of course do not exist in a monotone input). Now let us consider an input that is ε-far from monotone. Let $D \subseteq [n]$ denote the smallest set of indices flipping which gives a sorted boolean array. Write $D = D_0 \cup D_1$ where D_0 is the set of 0's that need to be converted to a 1 and D_1 is the set of 1's that need to be converted to 0's. W.log, let $|D_1|$ be the bigger of the two. Let M denote the middle element of D_1. Let M_{follow} denote the set of indices where 0's appear after M. We see that $|M_{follow}| \geq |D_1|/2$ (else flipping those 0's gives a better fix than D). The number of elements in D_1 before $M \geq \varepsilon n/4$ and also the $|M_{follow}| \geq \varepsilon n/4$. With constant probability, in $100/\varepsilon$ many samples we hit both D_1 (in a location before M) and the set M_{follow}. This gives us a violation.