CS 514: Advanced Algorithms II – Sublinear Algorithms **Rutgers: Spring 2020**

Lecture 4

February 11, 2020

Instructor: Sepehr Assadi *Scribes: Daniel Bittner and Jakob Degen*

1 Property Testing

We introduce the notion of property testing in this section, starting with a motivating example.

1.1 A Motivating Example

Let us consider the following problem.

Problem 1 (**Sortedness Problem**)**.** You are given an array $A[1:n]$ of n numbers and the goal is to determine whether or not A is *sorted* in increasing order, i.e., $A[1] \leq A[2] \leq \cdots \leq A[n]$.

This problem is easily solvable in $O(n)$ time: simply iterate over the indices and for each $i \in [n]-\{1\}$, check whether or not $A[i-1] \leq A[i]$. However, can we solve this problem in $o(n)$ time, namely, by a sublinear time algorithm? As in previous lectures, before even thinking about a sublinear time algorithm, we should first clarify our *query model* (or how the input is given to the algorithm): for this problem, we use the standard approach by assuming that we can check value of $A[i]$ for any $i \in [n]$ in $O(1)$ time. Such a query model is straightforward to implement by simply storing the array A in a random-access memory (RAM).

Let us now examine the problem of designing a sublinear time algorithm for deciding whether or not a given array A is sorted. At this point already, this problem *seems* to be "hard" for sublinear time algorithms. For instance, consider the case when the input array is sorted everywhere except possibly for two adjacent indices as in the example below:

1	2	...	$i+1$	i	...	n

Intuitively, in this scenario, an algorithm could only determine that A is not sorted if it queries at least one of indices i or $i+1$; considering i can be any index of the array, this should force the algorithm to take $\Omega(n)$ time.

Of course, the above part is merely intuition and not a formal proof. Let us now formalize this using a reduction from the query complexity of the OR_n problem. Recall $OR_n : \{0,1\}^n \to \{0,1\}$ is defined as $OR_n(x_1, \ldots, x_n) = x_1 \vee \cdots \vee x_n$. In the previous lecture we proved that any randomized algorithm for OR_n (with probability of success at least $2/3$) requires $\Omega(n)$ queries to the input x of the function in the worst-case. We use this to prove a lower bound for the sortedness problem.

Proposition 1. *Any algorithm (deterministic or randomized with probability of success $\geq 2/3$) for the sortedness problem requires $\Omega(n)$ queries to the input array A and hence $\Omega(n)$ time.*

Proof. The proof is by reduction from the query complexity of OR_n function on inputs that have *at most* one bit set to 1. In the previous lecture, we proved that query complexity of OR_n remains $\Omega(n)$ even with this additional promise on the input.

Consider an input $x = (x_1, \ldots, x_n)$ of the OR_n problem (with the given promise). Define the array $A_x[1:n]$ where $A_x[i] = i + 2 \cdot x_i$ for every $i \in [n]$. We now prove the correctness of reduction.

Recovering the correct answer: We claim that A_x is sorted if and only if $OR_n(x) = 0$ (again, we emphasize that x has *at most* one bit set to 1).

Suppose first that $OR_n(x) = 0$; in this case, $A_x[i] = i$ for all $i \in [n]$ and hence A_x is clearly sorted. Now suppose $OR_n(x) = 1$ and there exists a unique index i where $x_i = 1$. We have $A_x[i] = i + 2x_i = i + 2$ while $A_x[i+1] = i + 1$ and so $A_x[i] > A_x[i+1]$, proving that A_x is not sorted.

Simulating queries and finalizing reduction: Suppose we have an algorithm ALG for the sortedness problem and we want to use it for solving the OR_n problem. We can create the array A_x defined above from the input x of OR_n *implicitly* and *simulate* any query of ALG to input $A_x[i]$ by (actually) querying x_i and then returning $i + 2 \cdot x_i = A_x[i]$. Finally, we return $OR = 1$ if and only if ALG outputs A is *not* sorted (based on the above part). This way, ALG can be used to solve the OR_n problem also with the same query complexity and probability of success.

By the query complexity lower bound of OR_n, we obtain that query complexity of sortedness problem is $\Omega(n)$ and hence this problem requires $\Omega(n)$ time in general. □

1.2 Definition of Property Testing

Proposition 1 implies that we cannot hope for a sublinear time algorithm for sortedness problem. This is altogether not that surprising as we stated several times before that sublinear algorithms are typically only able to provide "approximate" answers not exact ones that we required in this problem. But what does it mean to approximately answer a *decision* problem (like sortedness problem)? *Property testing* provide one good answer to this question.

Roughly speaking, for any property $\mathcal{P}$ (say, sortedness property), we are interested in distinguishing between the case when our input has the property (say, is sorted), or it is "far" from having the property in some fixed distance measure. While one can also consider many problem-specific distance measures (and it does happen quite often), a general way of defining the distance is to simply measure the number of positions in the input that needs to be changed (arbitrarily) to make the input satisfy the property. We formalize this definition in the context of the sortedness problem (although the same exact definition works for any other property as well).

Definition 2 (**Property testing for sortedness**)**.** For any $\varepsilon \in (0, 1)$, we say that an algorithm ALG is an ***ε-tester*** for the sortedness problem, if given any input array A of n numbers:

(i) if A is sorted, ALG outputs *Yes* with probability at least $2/3$;

(ii) if A is ε-far from being sorted in that at least $\varepsilon \cdot n$ numbers in A needs to be changed to make A sorted, ALG outputs *No* with probability at least $2/3$.

We shall note that the constant $2/3$ in probability of success is not sacrosanct and can be switched to any constant bounded away from half (as by majority rule, we can always amplify the success probability of an ε-tester anyway with a constant overhead).

In the next section, we design a property testing algorithm for the sortedness problem.

2 Property Testing of Sortedness

We now design a sublinear time ε-tester for the sortedness property. In the following, we start with a warm-up case of Boolean entries and then switch to the general case.

2.1 Warm-Up: Boolean Entries

As a warm-up, let us consider the case when entries in A are Boolean. In this case, being sorted simply means that all 0's in the array appear before all 1's.

To design a property tester for this problem, we have to first understand something about the *structure* of arrays that are at least ε-far from being sorted. We can then try to exploit this structure algorithmically and design our tester. One such structure is intuitively as follows: An array A which is ε-far from being sorted should have $\Theta(\varepsilon \cdot n)$ 0's *after* $\Theta(\varepsilon \cdot n)$ 1's. Otherwise, by changing right-most 0's and left-most 1's we should end up getting a sorted array with $o(\varepsilon n)$ changes in A, contradicting that A was ε-far from being sorted.

This intuition is quite vague and informal at this point so let us formalize it in the following. Define:

- Left_1: as the set of $\varepsilon \cdot n/2$ *smallest* indices with value 1 in A (this is well-defined as A should have at least $\varepsilon \cdot n$ 1's; otherwise by changing all the 1's to 0 we can make A the all-zero array, a contradiction with A being ε-far from sorted).
- Right_0: as the set of $\varepsilon \cdot n/2$ *largest* indices with value 0 in A (by the above argument, this is also well-defined for the same reason).

We have the following lemma.

Lemma 3. *Suppose A is an array of size n which is ε-far from being sorted. Let i be the maximum index in $\text{Left}_1(A)$ and j be the minimum index in $\text{Right}_0(A)$ in A. Then $i < j$.*

Proof. Suppose towards a contradiction that $i > j$. Then, there are $\leq \varepsilon \cdot n/2$ 1's before i and $< \varepsilon \cdot n/2$ 0's after i (as $i > j$). As such, we can change all the 1's before i to 0 and all the 0's after i to 1, and making A sorted with $< \varepsilon n$ changes. This contradicts the assumption that A was ε-far from being sorted. □

We can now use Lemma 3 to design a tester for sortedness of Boolean arrays. The idea is that if we can sample at least one index from Left_1 and one index from Right_0, then we will encounter a 1 appearing at an index before a 0; this is something that can never happen in a sorted array and thus we can use this to distinguish the two cases. The algorithm is formally as follows:

Algorithm: An ε-tester for sortedness of a Boolean array with probability of success $1 - \delta$.

1. Sample $t := \frac{2}{\varepsilon} \ln \frac{2}{\delta}$ indices $i_1, \ldots, i_t$ uniformly at random and independently from $[n]$.
2. Query $A[i_1], \ldots, A[i_t]$.
3. Output *Yes* if there exists two sampled indices j and k where $j > k$ but $A[j] < A[k]$ and otherwise output *No*.

We prove the correctness of this algorithm first and then analyze its runtime.

Proof of correctness: The algorithm outputs the correct answer with probability 1 whenever A is sorted as any subarray of a sorted array, namely, $A[i_1], \ldots, A[i_t]$, is also sorted.

Now consider the case when A is ε-far from being sorted. We define the following two events:

- $\mathcal{E}_L$: at least one index from Left_1 is sampled.
- $\mathcal{E}_R$: at least one index from Right_0 is sampled.

We claim that if both $\mathcal{E}_L$ and $\mathcal{E}_R$ happen, then the algorithm outputs the correct answer. This is simply true by Lemma 3 as the all 1's in Left_1 appear before all 0's in Right_0 and hence the algorithm will see a 1 appearing before a 0 in its sampled indices if both events $\mathcal{E}_L$ and $\mathcal{E}_R$ happen. As such, we have,

$$\Pr\left(\text{algorithm outputs } \textit{Yes}\right) \leq \Pr\left(\overline{\mathcal{E}_L} \vee \overline{\mathcal{E}_R}\right) \leq \Pr\left(\overline{\mathcal{E}_L}\right) + \Pr\left(\overline{\mathcal{E}_R}\right). \qquad \text{(by union bound)}$$

We can bound the probability of each of these events as follows:

$$\begin{aligned}
\Pr\left(\overline{\mathcal{E}_L}\right) &= \left(1 - \frac{|\text{Left}_1|}{n}\right)^t && \text{(as each index is chosen uniformly at random and independently)} \\
&= \left(1 - \frac{\varepsilon \cdot n}{2n}\right)^{\frac{2}{\varepsilon}\ln\frac{2}{\delta}} && \text{(by the choice of parameters)} \\
&\leq \exp\left(-\frac{\varepsilon}{2} \cdot \frac{2}{\varepsilon}\ln\frac{2}{\delta}\right) && \text{(as } 1 - x \leq e^{-x} \text{ for all } x\text{)} \\
&= \frac{\delta}{2},
\end{aligned}$$

A very similar calculation would also imply $\Pr\left(\overline{\mathcal{E}_R}\right) \leq \frac{\delta}{2}$. By plugging in these bounds in the equation above, we obtain that the algorithm outputs the wrong answer with probability at most δ, finalizing the proof.

Runtime analysis: The first two steps of the algorithm require $O(t) = O(\frac{1}{\varepsilon} \cdot \ln(1/\delta))$ time clearly. The last step can also be implemented in $O(t)$ time by computing the maximum index j among all sampled indices which are 0 and minimum index k among all sampled indices which are 1.

Remark. For constant ε, δ independent of n, the runtime of this algorithm is only a constant, completely independent size of the input array.

2.2 The General Case

We now consider the general case when the numbers in the array A are arbitrary. To see the difficulty of this task, let us see why the previous algorithm fails to solve the problem on general arrays. Consider the following type of inputs:

3	2	1	6	5	4	9	8	7	$\cdots$	n	$n-1$	$n-2$

One one hand, this array is $\frac{2}{3}$-far from being sorted as two indices from each block of size three should be changed to make the array sorted. On the other hand, the only way the algorithm in the warm-up gives the correct answer on this array is if it samples two (or three) indices from the *same* block of size three. One can show that for this to happen, size of the sampled set should increase to $\Theta(\sqrt{n})$ using *birthday paradox*.

Remark (**Birthday Paradox**)**.** How many people are needed to be present in one room before we could say that with probability 50% two people will have the same birthday (assuming each day of the year is equally probable for a birthday, and people's birthdays in the room are independent)? The number is a lot less than you might think and it is only 23.

The more general question is that how many times we need to sample numbers uniformly at random and independently from $[m]$ before we see the same number twice with constant probability? And the answer is $\Theta(\sqrt{m})$. We leave the proof of this result as an exercise to the reader (although we will prove a generalization of this in the second part of this lecture).

To address this question, we are going to design a new tester based on *binary search*: we will pick an index $i \in [n]$ uniformly at random and check $A[i]$. We then do binary search on the array A to find the index of $A[i]$ (even though we already know it is i!). Of course, when A is sorted, we will definitely find i again. What about when A is not sorted? The "hope" is that in this case, the binary search should *not* be able to find i correctly. After all A is not sorted and indeed far from it and binary search is designed for sorted array (just to make sure this is in no way a proof; indeed, the main part of the proof of the algorithm is to formalize this intuitive statement).

Assumption: In designing an algorithm for this problem, we are going to assume that the numbers in A are *distinct*. This is without loss of generality because we can break the ties between equal numbers consistently by their index in the array (formally, we say $A[i]$ is smaller than $A[j]$ if $A[i] < A[j]$ or $A[i] = A[j]$ and $i < j$).

In the following, we are going to design an algorithm that always output the correct answer on sorted arrays and has a *small but non-negligible* probability of success on arrays that are ε-far from sorted. We then show how to use this algorithm as a subroutine in our final algorithm to increase its probability of success even on ε-far from being sorted arrays.

The First Algorithm: Binary-Search Tester

Before getting to the algorithm, we need the following definition.

Definition 4 (Consistent Binary Search). We say that the binary search algorithm for number a on array A (not necessarily sorted) is ***consistent*** if the following happens:

(i) the binary search correctly locates the number a in A;

(ii) when considering the sub-array $A[i:j]$ and picking pivot $p = \frac{i+j}{2}$, we have $A[i] < A[p] < A[j]$.

We can now present the algorithm.

Algorithm 1 (binary-search tester):

1. Sample $i \in [n]$ and query for $A[i]$;
2. Perform a binary search to locate $A[i]$ in A;
3. If the binary search is consistent (Definition 4), output *Yes*, otherwise *No*.

It is easy to see that the query complexity of this algorithm is $O(\log n)$ as binary search takes $O(\log n)$ iterations. We now show that the runtime of the algorithm is also $O(\log n)$. The runtime is also $O(\log n)$ because we can check whether the binary search is consistent in each step in $O(1)$ time.

We are now going to prove the correctness of the algorithm. The key idea of the proof is the following claim.

Claim 5. *Suppose binary search on $A[i]$ and $A[j]$ for indices $i < j$ are consistent. Then, $A[i] < A[j]$.*

Proof. Since both $A[i]$ and $A[j]$ were consistent and $i < j$, there should be a pivot p between i and j (the lowest common ancestor of i and j in the binary search tree), such that $A[i] < A[p] < A[j]$ (technically speaking, one of the two inequalities can be an equality but not both when $i = p$ or $j = p$). This implies that $A[i] < A[j]$, proving the result. □

We can use Claim 5 to finalize the proof of correctness.

Lemma 6. *Algorithm 1 (binary-search tester) outputs* Yes *with probability one whenever A is sorted. Moreover, the algorithm outputs* No *with probability at least ε whenever A is ε-far from being sorted.*

Proof. The first part is immediate by the correctness of binary search on sorted inputs. We now prove the second part. Define:

- $C := C(A)$: the set of indices $i \in [n]$ for which the binary search is consistent.

Let $i_1 < i_2 < \ldots < i_t$ for $t = |C|$ denote the indices in C. Then, by Claim 5, $A[i_1] < A[i_2] < \ldots < A[i_t]$, or in other words, C forms an *increasing subsequence* in A. Since A is ε-far from being sorted, we should have that $t \leq (1-\varepsilon) \cdot n$, as otherwise we could simply change the content of all indices *not* in C in the array A and make it sorted.

To finalize the proof, note that whenever the sampled index $i \in [n]$ (in the first line of the algorithm) is *not* in C, the algorithm outputs *No*. Since size of C is at most $(1-\varepsilon) \cdot n$, the probability the sampled index is not in C is at least ε, proving the lemma. □

By Lemma 6, our algorithm always outputs the correct answer on sorted arrays and "sometimes", namely, with probability ε, outputs the correct answer on ε-far from sorted arrays as well.

The Final Algorithm

We now show how to boost the probability of success of our tester from the previous part and obtain the final algorithm.

Algorithm 2:

1. Run Algorithm 1 (binary-search tester) for $k = \frac{1}{\varepsilon} \ln \frac{1}{\delta}$ times *independently*;
2. Output *Yes* if *all* runs of the algorithm output *Yes* and otherwise output *No*.

The query complexity and time complexity of the new algorithm is $O(k \cdot \log n) = O(\frac{1}{\epsilon} \ln \frac{1}{\delta} \cdot \log n)$ by the bounds for Algorithm 1. We now prove the correctness of the algorithm.

Lemma 7. *Algorithm 2 outputs* Yes *with probability one whenever A is sorted. Moreover, the algorithm outputs* No *with probability at least $1-\delta$ whenever A is ε-far from being sorted.*

Proof. The first part is immediate by the fact that of binary search tester always output *Yes* on sorted inputs (first part of Lemma 6). For the second part, on any array which is ε-far from being sorted, we have,

$$\begin{aligned}
\Pr(\text{Algorithm 2 outputs } \textit{Yes}) &= \Pr(\text{All copies of Algorithm 1 output } \textit{Yes}) \\
&\leq \prod_{i=1}^{k} \Pr(\text{copy } i \text{ of Algorithm 1 output } \textit{Yes}) \\
&\qquad\qquad \text{(by independence between the copies)} \\
&\leq (1-\varepsilon)^k \qquad \text{(by the second part of Lemma 6)} \\
&\leq \exp\left(-\varepsilon \cdot \frac{1}{\varepsilon} \ln \frac{1}{\delta}\right) \qquad \text{(as } 1-x \leq e^{-x} \text{ for all } x\text{)} \\
&= \delta.
\end{aligned}$$

This concludes the proof. □

We can conclude the following theorem now.

Theorem 8. *There is a property testing algorithm for the sortedness property that outputs the correct answer with probability at least $1-\delta$ in $O(\frac{1}{\varepsilon} \cdot \log(1/\delta) \cdot \log n)$ time.*

This theorem was first proven by Ergün, Kannan, Kumar, Rubinfeld, and Viswanathan [6] in one of the earliest property testing papers (property testing itself was introduced by Goldreich, Goldwasser, and Ron [7]). For more historical backgrounds on testing sortedness, see this excellent article by Raskhodnikova [10].

3 Distribution Testing

The field of distribution testing asks questions about distributions over $[n]$, or in particular functions of the form $\mu : [n] \to \mathbb{R}_{\geq 0}$, with $\sum_{i=1}^{n} \mu(i) = 1$. Some examples of questions that might be asked about the distribution are whether it is uniform, what its mean is, and what its entropy is. Note that for the decision version of these questions, we can again fall back to the paradigm of property testing when needed by asking question of the type: is this distribution uniform or "far" from uniform? (for some reasonable distance measure between distributions – more on this later in the section.)

Distribution testing tend to deviate from previous models we saw for sublinear time algorithms. For a sublinear time algorithm working on distribution μ, we would typically assume query access of the form what is $\mu(i)$ for some given $i \in [n]$. Instead, in distribution testing, the only type of "query" that an algorithm can make is to *sample* an element from the distribution. In this sense, the computational model of distribution testing is much less flexible than that of sublinear time algorithms. A sublinear time algorithm typically has the freedom to choose the type and parameters of the queries it makes. However, the only freedom an algorithm has in the distribution testing model is to choose the number of samples it wants to make from the distribution and how to use them to infer the final answer.

3.1 Uniformity Testing

To show case the notion of distribution testing, we are going to focus on one of the first problems studied in this area, the *uniformity testing* problem.

The particular question we will address is whether a distribution μ is uniform or not. We will use U_n to denote the uniform distribution over $[n]$, that is $U_n(i) = \frac{1}{n}$ for all $i \in [n]$. Distribution testing is similar to property testing in the sense that it does not lend itself well to asking extremely precise questions. So instead we will attempt to find an ε-tester for this question; in particular we wish to describe an algorithm that outputs *Yes* if the distribution it is sampling from is U_n, and *No* if the distribution is ε-far from U_n. Both of these conditions are of course only required with high probability.

Total Variation Distance

In the previous problem statement, we are still missing a definition of what it means for one distribution to be"ε-far" from another. While there are a variety of possible choices here, we will use the notion of **Total Variation Distance** between two distributions μ and ν defined as

$$\Delta_{\text{tvd}}(\mu, \nu) := \frac{1}{2} \cdot ||\mu - \nu||_1 = \frac{1}{2} \cdot \sum_{i=1}^{n} |\mu(i) - \nu(i)|.$$

Remark. Total variation distance is a widely used distance metric between two distributions. One particular appealing property of this distance is that:

$$\Delta_{\text{tvd}}(\mu, \nu) = \max_{\Omega \subseteq [n]} \mu(\Omega) - \nu(\Omega).$$

That is, this distance bounds the maximum difference between probability of any event in the two distributions. A corollary of this is that for any random variable X with domain $[n]$ and range $[0, 1]$,

$$|\mathop{\mathbb{E}}_{\mu}[X] - \mathop{\mathbb{E}}_{\nu}[X]| \leq \Delta_{\text{tvd}}(\mu, \nu).$$

Formal Definition of the Problem

We can now present the formal definition of the problem.

Problem 2 (**Uniformity Testing**). Given sample access to a distribution μ on domain $[n]$:

- if $\mu = U_n$, output *Yes*;
- if $\Delta_{\text{tvd}}(\mu, U_n) \geq \varepsilon$, output *No*.

The answer can be arbitrary in other case.

3.2 A Uniformity Testing Algorithm

The intuition behind the tester is the following. Consider picking two samples from distribution μ. If $\mu = U_n$, then the probability that these samples are equal, i.e., we have a *collision*, is $\frac{1}{n}$. It is also easy to see that in any other distribution, this probability can only be larger (we will prove this formally below). In fact, we are going to show that if $\Delta_{\text{tvd}}(\mu, U_n) \geq \varepsilon$, then this probability is considerably larger than $\frac{1}{n}$ by a factor of $(1 + O(\varepsilon^2))$. We can then design an algorithm to estimate the collision probability and use it to distinguish between the two cases for distribution μ.

While the Total Variation Distance is the metric we use in the problem statement, we will also repeatedly make use of the ℓ_2 or **Euclidean norm**. For some distribution μ, we define the square of the ℓ_2 norm to be

$$||\mu||_2^2 = \sum_{i=1}^{n} \mu(i)^2.$$

This is the same as calculating the ℓ_2 norm of the vector in $\mathbb{R}^n$ that represents this distribution. Note that this definition has an intuitive interpretation: It is precisely the probability that two independent samples from μ have the same value, i.e.,

$$\Pr_{x,y \sim \mu}(x = y) = ||\mu||_2^2. \tag{1}$$

Lemma 9. *If μ is a distribution with $\Delta_{tvd}(\mu, U_n) \geq \varepsilon$, then $||\mu||_2^2 \geq (1 + \varepsilon^2) \cdot \frac{1}{n}$.*

Proof. We will begin by considering $||\mu - U_n||_2^2$. The subtraction here is performed on the distributions interpreted as vectors. We have

$$\begin{aligned}
||\mu - U_n||_2^2 =& \sum_{i=1}^{n} (\mu(i) - \frac{1}{n})^2 \\
=& \sum_{i=1}^{n} \mu(i)^2 - 2\sum_{i=1}^{n} \frac{\mu(i)}{n} + \sum_{i=1}^{n} \frac{1}{n^2} \\
=& ||\mu||_2^2 - \frac{2}{n} + \frac{1}{n} \\
=& ||\mu||_2^2 - \frac{1}{n}
\end{aligned}$$

The key observation now is that knowledge about $\Delta_{\text{tvd}}(\mu, U_n) = ||\mu - U_n||_1$ allows us to argue about $||\mu - U_n||_2^2$. In particular, we will use the following standard connection between ℓ_2 and ℓ_1 norms.

Fact 10. *For any vector $x \in \mathbb{R}^n$, $||x||_1 \leq \sqrt{n} \cdot ||x||_2$.*

Based on this, we have,

$$\begin{aligned}
||\mu||_2^2 = ||\mu - U_n||_2^2 + \frac{1}{n} &\geq \frac{1}{n} \cdot ||\mu - U_n||_1^2 + \frac{1}{n} && \text{(by Fact 10)} \\
&\geq \frac{\varepsilon^2}{n} + \frac{1}{n},
\end{aligned}$$

proving the result. □

Estimating $||\mu||_2^2$ as a way to test uniformity. By Lemma 9, when $\mu = U_n$ in the *Yes* case, we have $||\mu||_2^2 = \frac{1}{n}$, while when $\Delta_{\text{tvd}}(\mu, U_n) \geq \varepsilon$ in the *No* case, we have $||\mu||_2^2 \geq (1+\varepsilon^2)/n$. As such, if we can get a $(1 \pm \varepsilon^2/3)$ approximation to value of μ we can distinguish between these two cases. In particular,

- In the *Yes* case, even if we completely overestimate the value of $||\mu||_2^2$ within the approximation range, the answer would be at most $(1+\varepsilon^2/3) \cdot \frac{1}{n}$.
- On the other hand, in the *No* case, even if we completely underestimate the value of $||\mu||_2^2$ within the approximation range, the answer would be at least $(1-\varepsilon^2/3) \cdot (1+\varepsilon^2) \cdot \frac{1}{n}$.
- Considering,

  $$(1-\varepsilon^2/3) \cdot (1+\varepsilon^2) = (1+\varepsilon^2-\varepsilon^2/3-\varepsilon^4/3) > (1+\varepsilon^2-2\varepsilon^2/3) = (1+\varepsilon^2/3), \qquad (\text{for } \varepsilon \in (0,1))$$

  we have that the largest overestimation in the *Yes* case is still smaller than the smallest underestimation in the *No* case.
- Finally, we can distinguish between the two cases as follows: if the estimate of $||\mu||_2^2 \leq (1+\varepsilon^2/3) \cdot \frac{1}{n}$, we out the distribution as uniform, namely, *Yes* case, and otherwise output the answer is *No*.

Hence, our goal now is to design a $(1 \pm \varepsilon^2/3)$-approximation algorithm for estimating the value of $||\mu||_2^2$. By the above discussion, this suffices to obtain our tester for this distribution testing algorithm.

An Algorithm for Estimating $||\mu||_2^2$

We propose the following algorithm for estimating $||\mu||_2^2$.

An algorithm for estimating $||\mu||_2^2$:

1. Sample $x_1, \ldots, x_k$ from μ for $k := \frac{24 \cdot 9}{\varepsilon^4} \cdot \sqrt{n}$.
2. For $i < j$, let $Y_{ij} \in \{0,1\}$ be the indicator random variable that is 1 if $x_i = x_j$ and 0 otherwise.
3. Return $X := \frac{\sum_{i<j} Y_{ij}}{\binom{k}{2}}$.

The sample complexity of this algorithm is $O(\sqrt{n}/\varepsilon^4)$ by the choice of k. It can also be shown that this algorithm can be implemented in $O(\sqrt{n}/\varepsilon^4)$ by sorting the samples using radix-sort and then counting the collisions (as the main focus in distribution testing is the sample complexity, we do not get into the details of the runtime further). We now prove the correctness of the algorithm in the following lemma.

Lemma 11. *Let $\lambda := \varepsilon^2/3$; then,* $\Pr\left(|X - ||\mu||_2^2| \geq \lambda \cdot ||\mu||_2^2\right) \leq \frac{1}{3}$.

Similar to some of our other sublinear algorithms, the proof of this lemma is by first showing that X is in expectation equal to the desired number, namely, $||\mu||_2^2$, and then bound the variance of X. This allows us to apply Chebyshev's inequality to obtain that X with sufficiently large probability is close to its expected value and thus $||\mu||_2^2$, and prove the lemma.

Claim 12. $\mathbb{E}[X] = ||\mu||_2^2$.

Proof. We have,

$$\begin{aligned}
\mathbb{E}[X] =& \frac{\mathbb{E}[\sum_{i<j} Y_{ij}]}{\binom{k}{2}} \\
=& \frac{\sum_{i<j} \mathbb{E}[Y_{i,j}]}{\binom{k}{2}} && \text{(linearity of expectation)} \\
=& \frac{\sum_{i<j} \Pr_{x_i,x_j\sim\mu}(x_i = x_j)}{\binom{k}{2}} && \text{(by definition of } Y_{ij}\text{)} \\
=& \frac{\sum_{i<j} ||\mu||_2^2}{\binom{k}{2}} && \text{(as shown in Eq (1))} \\
=& ||\mu||_2^2. && \text{(as there are } \tbinom{k}{2} \text{ choices of } i<j\in[k]\text{)}
\end{aligned}$$

□

Claim 13. $\mathrm{Var}[X] \le 4\cdot||\mu||_2^2/k^2 + 4\cdot||\mu||_3^3/k$.

Proof. As $\mathrm{Var}[\alpha\cdot A] = \alpha^2\cdot\mathrm{Var}[A]$ for any constant α and random variable A, we have,

$$\mathrm{Var}[X] = \mathrm{Var}\left[\frac{\sum_{i<j} Y_{ij}}{\binom{k}{2}}\right] = \frac{1}{\binom{k}{2}^2}\cdot\mathrm{Var}\left[\sum_{i<j} Y_{ij}\right]. \tag{2}$$

Thus, we only need to focus on variance of $\sum_{i<j} Y_{ij}$. We are going to use the following fact in the proof. For any two random variables A, B, *covariance* of A, B is $\mathrm{Cov}[A,B] := \mathbb{E}[AB] - \mathbb{E}[A]\,\mathbb{E}[B]$.

Fact 14. *For random variables* $A_1,\ldots,A_n$, $\mathrm{Var}[\sum_{i=1}^n A_i] = \sum_{i=1}^n\sum_{j=1}^n \mathrm{Cov}[A_i,A_j]$.

By Fact 14, we have,

$$\begin{aligned}
\mathrm{Var}\left[\sum_{i<j} Y_{ij}\right] &= \sum_{i<j,s<t} \mathrm{Cov}[Y_{ij},Y_{st}] \\
&= \sum_{\substack{i<j,s<t\\ |\{i,j,s,t\}|=2}} \mathrm{Cov}[Y_{ij},Y_{st}] + \sum_{\substack{i<j,s<t\\ |\{i,j,s,t\}|=3}} \mathrm{Cov}[Y_{ij},Y_{st}] + \sum_{\substack{i<j,s<t\\ |\{i,j,s,t\}|=4}} \mathrm{Cov}[Y_{ij},Y_{st}]. \qquad (3)
\end{aligned}$$

We bound each of the terms in Eq (3) in the following.

1. For the first term,

$$\begin{aligned}
\sum_{\substack{i<j,s<t\\ |\{i,j,s,t\}|=2}} \mathrm{Cov}[Y_{ij},Y_{st}] &= \sum_{i<j}\mathrm{Cov}[Y_{ij},Y_{ij}] && \text{(because } i=s \text{ and } j=t \text{ when } |\{i,j,s,t\}|=2\text{)} \\
&= \sum_{i<j}\mathrm{Var}[Y_{ij}] \\
&\text{(because for any random variable } A,\ \mathrm{Cov}[A,A] = \mathbb{E}[A^2] - \mathbb{E}[A]^2 = \mathrm{Var}[A]\text{)} \\
&\le \sum_{i<j}\mathbb{E}[Y_{ij}^2] = \sum_{i<j}\mathbb{E}[Y_{ij}] \\
&\qquad \text{(as } Y_{ij} \text{ is an indicator random variable and thus } Y_{ij}^2 = Y_{ij}\text{)} \\
&= \binom{k}{2}\cdot||\mu||_2^2. && \text{(by Claim 12)}
\end{aligned}$$

2. For the second term,

$$\sum_{\substack{i<j,s<t \\ |\{i,j,s,t\}|=3}} \mathrm{Cov}\left[Y_{ij}, Y_{st}\right] \leq \sum_{\substack{i<j,s<t \\ |\{i,j,s,t\}|=3}} \mathbb{E}\left[Y_{ij}Y_{st}\right]$$

$$= \sum_{\substack{i<j,s<t \\ |\{i,j,s,t\}|=3}} \Pr_{x,y,z\sim\mu}(x=y=z)$$

(when $|\{i,j,s,t\}| = 3$, $Y_{ij}Y_{st} = 1$ if the three corresponding samples are all equal and is zero otherwise)

$$= \sum_{\substack{i<j,s<t \\ |\{i,j,s,t\}|=3}} \sum_{x=1}^{n} \mu(x)^3$$

$$= \sum_{\substack{i<j,s<t \\ |\{i,j,s,t\}|=3}} ||\mu||_3^3$$

$$= 6 \cdot \binom{k}{3} \cdot ||\mu||_3^3,$$

where the last equality follows from a simple combinatorics exercise in counting number of ways of choosing $i < j, s < t$ from $[k]$ so that $|\{i,j,s,t\}| = 3$ (proof: we first pick distinct x, y, z from $[k]$ with $\binom{k}{3}$ choices; for each such choice, we pick one of x, y, z to be repeated twice among i, j, s, t with 3 choices, and finally, we can assign the remaining indices i, j, s, t in two ways, resulting in $6 \cdot \binom{k}{3}$ choices).

3. For the third term,

$$\sum_{\substack{i<j,s<t \\ |\{i,j,s,t\}|=4}} \mathrm{Cov}\left[Y_{ij}, Y_{st}\right] = 0.$$

(since $i \neq s$ and $j \neq t$ and thus $Y_{ij} \perp Y_{st}$; also $\mathrm{Cov}[A, B] = 0$ for *independent* A, B)

Putting all these together in Eq (3), we have that,

$$\mathrm{Var}\left[\sum_{i<j} Y_{ij}\right] \leq \binom{k}{2} \cdot ||\mu||_2^2 + 6 \cdot \binom{k}{3} \cdot ||\mu||_3^3 + 0.$$

Finally, we plug in this bound in Eq (2) and obtain that,

$$\mathrm{Var}[X] \leq \frac{||\mu||_2^2}{\binom{k}{2}} + \frac{k \cdot (k-1) \cdot (k-2) \cdot ||\mu||_3^3}{k^2 \cdot (k-1)^2/4}$$

$$\leq \frac{4 \cdot ||\mu||_2^2}{k^2} + \frac{4 \cdot ||\mu||_3^3}{k}. \qquad \text{(as } \tbinom{x}{y} \geq (x/y)^y\text{)}$$

This concludes the proof. □

Remark. Prior to the proof of Lemma 11, we have only worked with bounding variance of sum of random variables in the case they were *independent*. Under this assumption, Fact 14 simplifies to variance of the sum is equal to sum of individual variances. In the proof of Lemma 11 however, we no longer had the independence assumption but similar to what we saw in the proof, in many scenarios bounding variance of the sum is still manageable by considering covariance terms explicitly.

We are now ready to finalize the proof of Lemma 11, but first let us recall a simple fact about norms.

Fact 15. *For any vector $x \in \mathbb{R}^n$ and integers $p \leq q$, $||x||_p \leq ||x||_q$.*

Finally, we present the proof of Lemma 11.

Proof of Lemma 11. By Chebyshev's inequality,

$$\begin{aligned}
\Pr\left(|X - \mathbb{E}[X]| \geq \lambda \cdot ||\mu||_2^2\right) &\leq \frac{\mathrm{Var}[X]}{\lambda^2 \cdot ||\mu||_2^4} \\
&\leq \frac{4 \cdot ||\mu||_2^2}{k^2 \cdot \lambda^2 \cdot ||\mu||_2^4} + \frac{4 \cdot ||\mu||_3^3}{k \cdot \lambda^2 \cdot ||\mu||_2^4} && \text{(by Claim 13)} \\
&\leq \frac{4}{k^2 \cdot \lambda^2 \cdot ||\mu||_2^2} + \frac{4}{k \cdot \lambda^2 \cdot ||\mu||_2} && \text{(by Fact 15, } ||\mu||_3 \leq ||\mu||_2\text{)} \\
&\leq \frac{4n}{k^2 \cdot \lambda^2} + \frac{4\sqrt{n}}{k \cdot \lambda^2} && \text{(as } ||\mu||_2^2 \geq \tfrac{1}{n} \text{ for any distribution } \mu\text{)} \\
&\leq \frac{4\varepsilon^4 \cdot n}{24^2 \cdot 9^2 \cdot (\varepsilon^2/3)^2 \cdot n} + \frac{4\varepsilon^4 \cdot \sqrt{n}}{24 \cdot 9 \cdot (\varepsilon^2/3)^2 \cdot \sqrt{n}} && \text{(by the choices of } k \text{ and } \lambda\text{)} \\
&= \frac{1}{144} + \frac{1}{6} < \frac{1}{3}.
\end{aligned}$$

As by Claim 12, $\mathbb{E}[X] = ||\mu||_2^2$, we get the final result. □

By Lemma 11, we obtain an algorithm for uniformity testing with probability of success at least 2/3. We can now use our standard trick of running the algorithm $O(\log(1/\delta))$ times in parallel and return the majority answer for uniformity testing (or median answer for estimating $||\mu||_2^2$), which gives us an algorithm with probability of success $1 - \delta$. We can thus conclude the following theorem.

Theorem 16. *There is a distribution testing algorithm for testing uniformity (under total variation distance) that outputs the correct answer with probability at least $1 - \delta$ using $O(\frac{1}{\varepsilon^4} \cdot \sqrt{n} \cdot \log(1/\delta))$ samples.*

This theorem was first proved by Goldreich and Ron [8] (implicitly and in a different context), and then was reproved in the above form by Batu *et.al.* in [3] and [2]. These bounds were later improved to $O(\sqrt{n}/\varepsilon^2)$ samples by Paninski [9] (for non-too-small ε) and then by Chan *et.al.* [4], Diakonikolas *et.al.* [5], and Acharya *et.al.* [1] (for all range of $\varepsilon > 0$). The $\Omega(\sqrt{n}/\varepsilon^2)$ number of samples was also shown to be necessary by Paninski [9].

References

[1] Jayadev Acharya, Constantinos Daskalakis, and Gautam Kamath. Optimal testing for properties of distributions. In *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 3591–3599, 2015. 12

[2] Tugkan Batu, Lance Fortnow, Eldar Fischer, Ravi Kumar, Ronitt Rubinfeld, and Patrick White. Testing random variables for independence and identity. In *42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14-17 October 2001, Las Vegas, Nevada, USA*, pages 442–451, 2001. 12

[3] Tugkan Batu, Lance Fortnow, Ronitt Rubinfeld, Warren D. Smith, and Patrick White. Testing that distributions are close. In *41st Annual Symposium on Foundations of Computer Science, FOCS 2000, 12-14 November 2000, Redondo Beach, California, USA*, pages 259–269, 2000. 12

[4] Siu-on Chan, Ilias Diakonikolas, Paul Valiant, and Gregory Valiant. Optimal algorithms for testing closeness of discrete distributions. In *Proceedings of the Twenty-Fifth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2014, Portland, Oregon, USA, January 5-7, 2014*, pages 1193–1203, 2014. 12

[5] Ilias Diakonikolas, Daniel M. Kane, and Vladimir Nikishkin. Optimal algorithms and lower bounds for testing closeness of structured distributions. In *IEEE 56th Annual Symposium on Foundations of Computer Science, FOCS 2015, Berkeley, CA, USA, 17-20 October, 2015*, pages 1183–1202, 2015. 12

[6] Funda Ergün, Sampath Kannan, Ravi Kumar, Ronitt Rubinfeld, and Mahesh Viswanathan. Spot-checkers. In *Proceedings of the Thirtieth Annual ACM Symposium on the Theory of Computing, Dallas, Texas, USA, May 23-26, 1998*, pages 259–268, 1998. 6

[7] Oded Goldreich, Shafi Goldwasser, and Dana Ron. Property testing and its connection to learning and approximation. *J. ACM*, 45(4):653–750, 1998. 6

[8] Oded Goldreich and Dana Ron. On testing expansion in bounded-degree graphs. *Electronic Colloquium on Computational Complexity (ECCC)*, 7(20), 2000. 12

[9] Liam Paninski. A coincidence-based test for uniformity given very sparsely sampled discrete data. *IEEE Trans. Information Theory*, 54(10):4750–4755, 2008. 12

[10] Sofya Raskhodnikova. Testing if an array is sorted. In *Encyclopedia of Algorithms*, pages 2219–2222. 2016. 6