COMP 585 **Spring 2026**

Probabilistic Toolkit for Learning and Computing *Instructor: Maryam Aliakbarpour*

January 15, 2026 *Scriber: Maryam Aliakbarpour*

Lecture 2

Introduction to Property Testing: Testing Sortedness

Randomness is a powerful tool for designing fast, efficient algorithms. However, randomized approaches introduce unique challenges: unlikely events can occur (e.g., observing an unusual streak of heads in coin tosses), and very small deviations from a desired property can be difficult to detect reliably.

Property testing provides a formal framework to address these issues. It does so by relaxing two traditional algorithmic requirements:

- **Probability of Error:** We allow the algorithm a small, bounded probability of failing to provide the correct answer.
- **Distance to Property:** Instead of detecting any deviation, we only require the algorithm to reject inputs that are "far" from satisfying the property.

Unlike traditional deterministic algorithms that must scan every element to find even a single violation, a property tester uses a small number of random queries to seek substantial evidence of non-conformance. By accepting these relaxations, we can design testers with **sublinear complexity**, often significantly outperforming their deterministic counterparts.

In this session, we explore the specific problem of **testing the sortedness of an array** and examine how to design algorithms with rigorous error guarantees.

Defining the Problem

Let $A = (A[1], A[2], \dots, A[n])$ be an array of n elements. We say that A is *sorted* if:

$$A[1] \leq A[2] \leq \cdots \leq A[n].$$

We aim to design an algorithm that can distinguish whether A is sorted or it is "far" from being sorted. A deterministic algorithm can check if A is sorted via a linear scan, requiring $\Theta(n)$ queries. Our goal is to determine if we can achieve $o(n)$ queries under a relaxed notion of correctness.

Remark 1 (The Query Access Model)**.** *The feasibility of sublinear time complexity depends entirely on the access model. In many settings, we cannot even "read" the input in $o(n)$ time. In this lecture, we assume a **query model** (or oracle model): The input is already stored (e.g., in memory or the cloud). We do not read the entire array; instead, we query an oracle which, given an index $i \in [n]$, returns the value $A[i]$ in $O(1)$ time. Our complexity is measured by the total number of such queries made to the oracle.*

One possible randomized approach is to sample random locations and check for local consistency. However, a naive sampling approach may fail for two primary reasons:

1. **Missing the "Bad" Region:** A significantly unsorted array may appear sorted if we happen to sample a "wrong set of indices" that coincidentally follow an increasing order.
2. **Missing Tiny Deviations:** We may fail to detect a tiny deviation from sortedness (such as a single swapped pair) by not hitting the specific indices where the violation occurs.

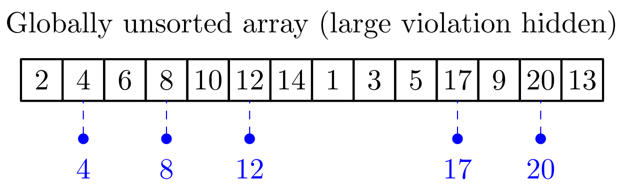

Figure 1: Random sampling may fail to detect global unsortedness if the sample is coincidentally ordered.

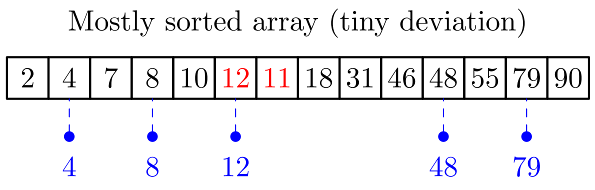

Figure 2: Random sampling may fail to detect a tiny deviation (local violation) by not hitting the specific "bad" indices.

If we insist on perfect testers, we cannot hope to have an algorithm with $o(n)$ time/query complexity.

The Property Testing Framework

Property testers are designed to determine, with a high success probability (typically $1 - \delta$), whether a large object—such as an array, a probability distribution, or a graph—possesses a specific property P or is ε-far from satisfying it. Mathematically, we define a property P

as the set of all objects that satisfy the condition; for instance, in the context of sortedness, P is the set of all sorted arrays of length n.

This definition introduces two specific notions that require formal quantification:

1. **Proximity parameter (ε):** We must define a metric to determine what it means for an object to be "far" from a property. Usually, this is determined based on the context. A good candidate may be the minimum fraction of the object that must be modified to satisfy P.

   For two arrays A, B of length n, we define their distance as the fraction of indices where they differ:
$$\mathrm{dist}(A, B) := \frac{|\{i \in [n] : A[i] \neq B[i]\}|}{n}.$$
   An array A is said to be ε-far from a property P if it requires changing more than εn elements to satisfy the property:
$$\mathrm{dist}(A, P) := \min_{B \in P} \mathrm{dist}(A, B) > \varepsilon.$$

2. **Confidence parameter(δ):** Since the algorithm is randomized, we must bound the probability of error. We require the tester to succeed with probability at least $1 - \delta$.

These two parameters, ε and δ, allow us to study the trade-off between the "strictness" of our test and the number of queries required.

$(\boldsymbol{\varepsilon}, \boldsymbol{\delta})$-Tester. An algorithm $\mathcal{A}$ is an (ε, δ)-tester for property P for input object A iff it satisfies the following with probability at least $1 - \delta$:

- **Completeness:** If $A \in P$, then $\mathcal{A}$ outputs accept.
- **Soundness:** If $\mathrm{dist}(A, P) > \varepsilon$, then $\mathcal{A}$ outputs reject.

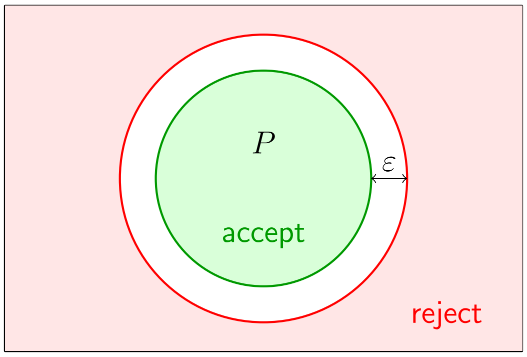

Figure 3: Property testing viewpoint. In the white region both answers are considered valid.

The Binary Case

Consider the simplified setting where $A[i] \in \{0, 1\}$. A sorted binary array must be of the form $0^k 1^{n-k}$ (a sequence of 0s followed by 1s). We propose the following algorithm to test sortedness.

Algorithm 1 Sampling-Based Sortedness Tester ($\{0, 1\}$ Case)

```
1: Input: $n$, $\varepsilon$, $\delta$, Query access to an array $A$ of length $n$
2: Parameter: $m \leftarrow \lceil \frac{2}{\varepsilon} \ln \frac{2}{\delta} \rceil$
3: Pick $m$ indices $S = \{i_1, i_2, \dots, i_m\}$ independently and uniformly at random from $[n]$
4: Sort the sampled indices such that $i_1 < i_2 < \cdots < i_m$
5: Query $A[i_j]$ for all $j \in [m]$
6: if $A[i_1] \le A[i_2] \le \cdots \le A[i_m]$ then
7:     return accept
8: else
9:     return reject
```

Proof of Correctness

Completeness: If A is sorted, any sampled subsequence is also sorted. The algorithm will always find $A[i_j] \le A[i_{j+1}]$, thus it outputs accept with probability 1.

Soundness: Suppose A is ε-far from sorted. This means we must change at least εn elements to make A sorted. We define two critical sets of indices:

- Q_1: The set of indices of the *first* $\varepsilon n/2$ occurrences of 1.
- Q_0: The set of indices of the *last* $\varepsilon n/2$ occurrences of 0.

Let $i_{\max} = \max(Q_1)$ be the index of the $(\varepsilon n/2)$-th 1, and $i_{\min} = \min(Q_0)$ be the index of the $(\varepsilon n/2)$-th 0 from the end. If $i_{\min} < i_{\max}$, any sample that picks one index from Q_1 and one from Q_0 will find a violation (a 1 appearing before a 0).

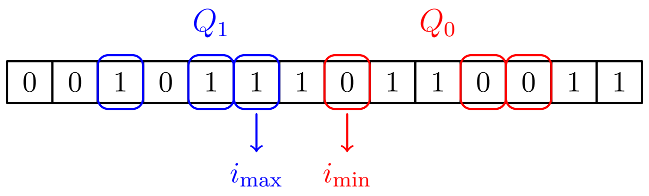

Figure 4: An illustration of the sets Q_1 and Q_0 with $\varepsilon n/2 = 3$.

Lemma 1. *If the binary array A is ε-far from sorted, then $i_{\min} > i_{\max}$.*

Proof. To understand this, it is easiest to prove the contrapositive: *If $i_{\min} < i_{\max}$, then A is not ε-far from sorted.*

Assume $i_{\max} < i_{\min}$. We can demonstrate that A can be made sorted by changing at most εn elements. Consider a "split point" k chosen such that $i_{\min} \le k < i_{\max}$. We can transform A into a sorted array of the form $0^k 1^{n-k}$ using the following two steps:

1. **Fix the left side:** For all indices $i \le k$, change any 1 to a 0. By definition, $i_{\max}$ is the index of the $(\varepsilon n/2)$-th occurrence of 1 in the array. Since $k < i_{\max}$, there are at most $\varepsilon n/2$ such ones to change.

2. **Fix the right side:** For all indices $i > k$, change any 0 to a 1. By definition, $i_{\min}$ is the index of the $(\varepsilon n/2)$-th occurrence of 0 *from the end* of the array. Since $k \ge i_{\min}$, there are at most $\varepsilon n/2$ such zeros to change.

The total number of modifications required to sort the array is:

$$\text{Total Changes} \le \underbrace{\frac{\varepsilon n}{2}}_{\text{from Step 1}} + \underbrace{\frac{\varepsilon n}{2}}_{\text{from Step 2}} = \varepsilon n.$$

By definition, an array is ε-far if it requires **more than** εn changes to become sorted. Since we have shown that A can be sorted with $\le \varepsilon n$ changes, A is not ε-far. Therefore, if A is truly ε-far, the indices must overlap such that $i_{\min} > i_{\max}$. □

Analysis of the Failure Probability

The algorithm fails to reject an ε-far array only if the set of sampled indices S misses at least one of the critical sets, Q_1 or Q_0. If the sample S contains at least one index $i \in Q_1$ and one index $j \in Q_0$, then by the previous Lemma, $j < i$. Because $A[j] = 0$ and $A[i] = 1$, the algorithm will detect the violation and correctly reject.

Let E_1 be the event that $S \cap Q_1 = \emptyset$ and E_0 be the event that $S \cap Q_0 = \emptyset$. Since we sample m indices independently and uniformly at random, and $|Q_1| = |Q_0| = \varepsilon n/2$, the probability of missing a set in a single draw is $(1 - \varepsilon/2)$. For m draws, we have:

$$\mathbf{Pr}[E_1] = \left(1 - \frac{\varepsilon}{2}\right)^m \le e^{-\varepsilon m/2}, \quad \mathbf{Pr}[E_0] = \left(1 - \frac{\varepsilon}{2}\right)^m \le e^{-\varepsilon m/2}.$$

By the **Union Bound**, the total probability of error is bounded by:

$$\mathbf{Pr}[\text{Failure}] = \mathbf{Pr}[E_1 \cup E_0] \le \mathbf{Pr}[E_1] + \mathbf{Pr}[E_0] \le 2e^{-\varepsilon m/2}.$$

To satisfy the (ε, δ) requirement, we set $2e^{-\varepsilon m/2} \le \delta$. Solving for m:

$$e^{-\varepsilon m/2} \le \frac{\delta}{2} \implies -\frac{\varepsilon m}{2} \le \ln\left(\frac{\delta}{2}\right) \implies m \ge \frac{2}{\varepsilon} \ln\left(\frac{2}{\delta}\right).$$

Consequently, a query complexity of $m = O\left(\frac{1}{\varepsilon}\log\frac{1}{\delta}\right)$ is sufficient to achieve the desired confidence.