COMP 585 **Spring 2026**
Probabilistic Toolkit for Learning and Computing *Instructor: Maryam Aliakbarpour*
January 20, 2026 *Scriber: Maryam Aliakbarpour*

Lecture 3

Testing Sortedness: General Case

In this lecture, we focus on testing sortedness of general array where the value of each entry can be an arbitrary number. Unlike the binary case, random sampling does not work for testing sortedness because the array may contain a small number of local violations while being globally well ordered. Randomly checking indices is therefore unlikely to hit one of these violations.

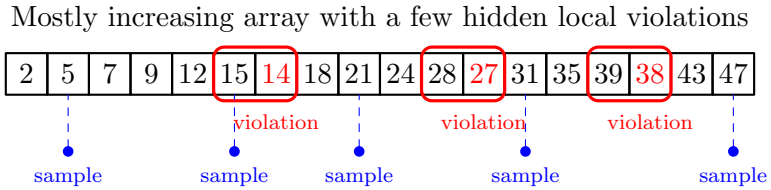

Figure 1: A few local violations can be hidden inside an otherwise increasing global trend. Randomly sampled indices may miss the bad adjacencies unless we sample many locations or search large neighborhoods.

One might try to fix this by sampling a random index and then searching for violations in its vicinity, but this raises two problems: we do not know how large the vicinity must be, and we still need to sample enough locations to ensure the global order is correct. Searching many random locations with large neighborhoods quickly leads to a suboptimal algorithm.

A different approach is to exploit other structural properties of sorted (or nearly sorted) arrays. It is well known that binary search is designed to work on sorted arrays. In particular, if an array A is sorted, then for every index i, running binary search on value $A[i]$ returns exactly i. In this lecture, we show the contrapositive idea: if binary search fails at many locations, then the array must be far from being sorted. Our goal is to make this intuition precise and turn it into a tester.

For simplicity, assume without loss of generality that all values in A are distinct (ties can be broken lexicographically by index). We use the standard recursive binary search procedure, even though the array may not be sorted.

Distance to sortedness: Similar to the binary case, for arrays A, A' of length n, define the normalized Hamming distance

$$\text{dist}(A, A') \;=\; \frac{1}{n}\left|\{k \in [n] : A[k] \neq A'[k]\}\right|.$$

Let $\mathcal{P}$ denote the set of sorted arrays of length n. The distance of A from being sorted is

$$\text{dist}(A, \mathcal{P}) \;=\; \min_{A' \in \mathcal{P}} \text{dist}(A, A').$$

We say that A is *ϵ-far from sortedness* if $\text{dist}(A, \mathcal{P}) \geq \epsilon$.

Background: Binary Search

We begin by recalling the standard binary search algorithm. Binary search is typically used to locate a target value, namely x, in a *sorted* array by repeatedly comparing against the middle element and discarding half of the search interval.

Algorithm 1 BinarySearch(A, x, ℓ, r)

```
Input: Array A, target value x, interval [ℓ, r)
while ℓ < r do
    m ← ⌊(ℓ + r)/2⌋
    if A[m] = x then
        return m
    else if A[m] > x then
        r ← m
    else
        ℓ ← m + 1
return "not found"
```

If A is sorted and contains x, binary search always returns the unique index i such that $A[i] = x$. If the array is not sorted, however, the algorithm may return an incorrect index or fail to find x altogether.

Binary-Search-Based Tester

The tester uses binary search as a *consistency check*: it verifies whether binary search behaves as if the array were sorted at randomly chosen locations. At a high-level, each iteration selects a random index i and checks whether binary search can correctly locate the value $A[i]$ by returning i. If the array is sorted, this always succeeds. If the array is far from sorted, binary search fails on a noticeable fraction of indices, causing the tester to reject with high probability.

Algorithm 2 Binary-Search-Based Sortedness Tester

Input: n, ϵ, δ, query access to array A
$s \leftarrow \left\lceil \frac{1}{\epsilon} \log \frac{1}{\delta} \right\rceil$
for $t = 1$ to s **do**
    Pick $i \sim \text{Unif}([n])$
    Run BinarySearch$(A, A[i], 1, n+1)$
    if the returned index $\neq i$ **then**
        return reject
return accept

Proof of Correctness

In this part, we show that Algorithm 2 is an (ϵ, δ)-tester for sortedness.

Completeness. First, we argue about the case that A is sorted. Clearly, if A is sorted, binary search behaves correctly at every index i upon search for $A[i]$, the procedure returns i. Thus, the tester always accepts.

Soundness. Next, we show that if the array is ϵ-far from being sorted, then our tester rejects with probability at least $1 - \delta$. The argument proceeds in two steps: We show that correctness at many indices enforces a global ordering. Hence, binary search should fail for a sufficiently far-from-sorted array. Next, we pick s in a way that guarantees that with high probability we will observe a failure.

We define a *nice index* for which the binary search works correctly. In particular, an index $i \in [n]$ is called *nice* if BinarySearch$(A, A[i], 1, n+1)$ returns i. The key observation is that binary search correctness is not merely local: if it holds at many indices, it enforces a strong global ordering.

Lemma 1 (Nice indices preserve order)**.** *If $i < j$ are both nice indices, then $A[i] < A[j]$.*

Proof. Let S_i and S_j be the sequences of pivot indices m visited by BinarySearch when searching for $A[i]$ and $A[j]$, respectively. Both searches begin with the same interval $[1, n+1)$ and follow the same path of pivots until they diverge. Let $m^\star$ be the *last mutual pivot* common to both sequences.

Since i and j are nice, we know that $i \in S_i$ and $j \in S_j$. We consider the relative position of i, j, and $m^\star$:

- **Case $i = m^\star$:** Since $i < j$, the search for $A[j]$ must have branched to the right at index $m^\star$ to eventually reach j. According to Algorithm 1, the search moves to the right subinterval only if $A[m^\star] < A[j]$. Thus, $A[i] < A[j]$.
- **Case $j = m^\star$:** Since $i < j$, the search for $A[i]$ must have branched to the left at index $m^\star$ to eventually reach i. The algorithm moves to the left subinterval only if $A[m^\star] > A[i]$. Thus, $A[i] < A[j]$.

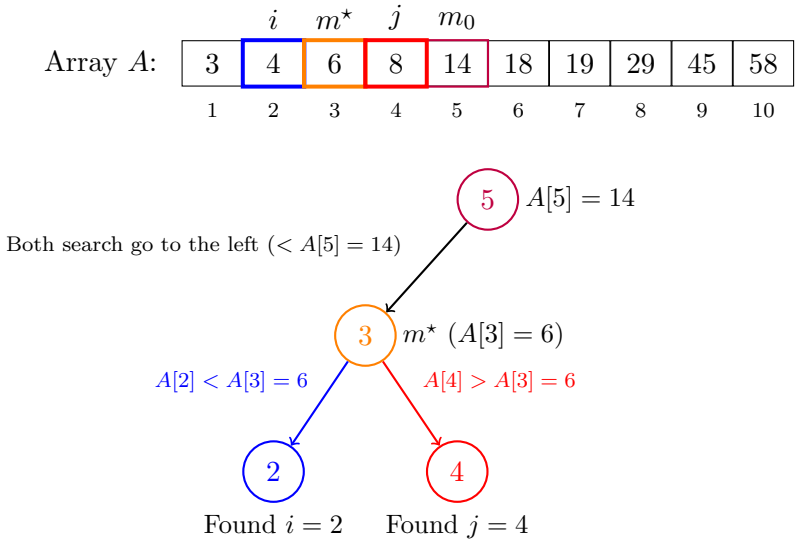

Figure 2: Binary search paths for nice indices $i = 2$ and $j = 4$. They diverge at $m^\star = 3$ because $A[2] < A[3] < A[4]$.

- **Case $i \neq m^\star$ and $j \neq m^\star$:** Since $m^\star$ is the last mutual pivot and the searches must eventually reach i and j, the two paths must diverge at this point. Given $i < j$, the search for $A[i]$ must branch left ($A[i] < A[m^\star]$) and the search for $A[j]$ must branch right ($A[m^\star] < A[j]$). By transitivity, $A[i] < A[j]$.

In all cases, the condition $i < j$ for nice indices implies $A[i] < A[j]$. An illustration of this proof is shown in Figure 2.

□

Next, we show that in an array that is ϵ-far from being sorted, there are many indices that are **not** nice. Let

$$p \coloneqq \mathbf{Pr}_{i \sim \mathrm{Unif}([n])}[i \text{ is not nice}].$$

Note that p is exactly the probability that a single iteration of the tester rejects. Let $N \subseteq [n]$ denote the set of nice indices. By Lemma 1, the subsequence $\{A[i] : i \in N\}$ is strictly increasing. We construct a sorted array A' as follows:

- For all $i \in N$, set $A'[i] = A[i]$.
- Modify the remaining entries so that A' is globally sorted.

Only indices outside N are modified, so at most $n - |N|$ entries change. Hence,

$$\mathrm{dist}(A, \mathcal{P}) \leq \frac{n - |N|}{n} = \mathbf{Pr}_{i \sim \mathrm{Unif}([n])}[i \text{ is not nice}] = p.$$

Therefore, if A is ϵ-far from sorted, we must have $p \geq \epsilon$. That is, for more than ϵ-fraction of the indices in $[n]$, the algorithm will reject.

We aim to set s in a way that the tester reject with probability at least $1 - \delta$. Note that the tester accepts only if all s trials succeed. In a single round, the probability that the algorithm does not reject is $1 - p$. Thus, for s rounds, we have:

$$\mathbf{Pr}[\text{outputting } \mathsf{accept}] \;=\; (1-p)^s \;\leq\; (1-\epsilon)^s \;\leq\; e^{-\epsilon s}\,.$$

In the second inequality we use $1 - x < e^{-x}$ for all real x. Choosing s to be at least $\left\lceil \left(\frac{1}{\epsilon}\log\frac{1}{\delta}\right)\right\rceil$ ensures that the probability of outputting accept is at most δ, implying rejection with probability at least $1 - \delta$ as desired.

Bibliographic Note

The binary-search-based tester for sortedness was introduced by Ergün, Kannan, Kumar, Rubinfeld, and Viswanathan in their seminal paper "Spot-checking phenomena" (STOC 1998) [EKK+98]. This work was among the first to demonstrate that global structural properties, like monotonicity, can be tested in sublinear time by checking for internal consistency of classical algorithms.

References

[EKK+98] 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 (STOC)*, pages 259–268, 1998.