CS 514: Advanced Algorithms II – Sublinear Algorithms **Rutgers: Fall 2021**

Lecture 4

September 28, 2021

Instructor: Sepehr Assadi

Topics of this Lecture

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* provides 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 meaning 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 of the above definition 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 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 $\varepsilon, \delta > 0$ independent of n, the runtime of this algorithm is only a constant, completely independent of the 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 arrays (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 binary search correctly locates the number a in A.*

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. 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. Consider the process of binary searching for $A[i]$ and $A[j]$ in A. We claim that there should be a pivot p such that when binary search compared $A[i]$ with $A[p]$, it recursed on the "left" sub-problem on array $A[i : p]$, whereas for $A[j]$, it recursed on the "right" sub-problem on array $A[p : j]$. Suppose not; then:

- Either, in *all* subproblems of binary search, $A[i], A[j]$ remained on the same side of every pivot; this will imply that both searches ends up in a single index. However, since both $A[i]$ and $A[j]$ were consistent binary searches and $i < j$, this cannot happen;
- Or, there is a pivot p, where the binary search instead recursed on the "right" side of $A[p : i]$ and "left" side of $A[j : p]$, for $A[i]$ and $A[j]$, respectively. But since $i < j$, this means that at least one of the searches cannot locate the corresponding index correctly, contradicting with both search being consistent.

Finally, by definition of binary search, for the special pivot identified above, we have that $A[i] \leq A[p]$ and $A[j] \geq A[p]$. The equality cannot happen for both either (as we assumed entries of A are distinct), thus, $A[i] < A[j]$ as desired. □

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

Lemma 6. *The Binary-search tester outputs* Yes *with probability* 1 *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:

- $C := C(A)$ as 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 (the parts in C are already 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 the Binary-search tester (Algorithm 1) 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* 1 *whenever A is sorted, and 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 the 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,

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

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 [1] in one of the earliest property testing papers (property testing itself was introduced by Goldreich, Goldwasser, and Ron [2]). For more historical backgrounds on testing sortedness, see this excellent article by Raskhodnikova [3].

References

[1] F. Ergün, S. Kannan, R. Kumar, R. Rubinfeld, and M. 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. 7

[2] O. Goldreich, S. Goldwasser, and D. Ron. Property testing and its connection to learning and approximation. *J. ACM*, 45(4):653–750, 1998. 7

[3] S. Raskhodnikova. Testing if an array is sorted. In *Encyclopedia of Algorithms*, pages 2219–2222. 2016. 7