Data Structures and Algorithms
Lecture Outline

Testing Sortedness

Given an array A of n elements, we wish to determine if A is sorted in non-decreasing order, i.e. $A[1] \leq A[2] \leq \cdots \leq A[n]$

An $O(n)$ solution to this problem is clear: simply iterate once through the array and check if adjacent elements are in increasing order.

Now we ask if we can solve this problem in $o(n)$ time, or in other words, in *sublinear* time? As it turns out, this is not possible. Indeed, consider the scenario in which the input array is sorted everywhere except for two adjacent indices, i and $i+1$. In this scenario, an algorithm could only determine that A is not sorted if it queries either index i or $i+1$. Since i can be any index of the array, this forces any correct algorithm to take $\Omega(n)$ time.

But can we get an *approximate* answer in sublinear time? What does it mean for an array to be *approximately* sorted?

Definition. An array A is ε-far if changing $\geq \varepsilon \cdot n$ elements in A results in a sorted array, where $\varepsilon \in (0, 1)$

Example.

13	28	3	46	79	91	14	66	100
1	2	3	4	5	6	7	8	9

This array is $\frac{1}{3}$-far as we can change a third of the elements, the elements at positions $3, 7$, and 8, to get a sorted array.

With this definition in hand, we can try to provide an approximate solution in sublinear time. In particular, given an array A of n integers that is either sorted or ε-far from being sorted, we would like to

A. Output YES with probability $\geq \frac{2}{3}$ if A is sorted

B. Output NO with probability $\geq \frac{2}{3}$ if A is ε-far from being sorted

Boolean Array

To motivate intuition, first consider the case where A consists of only 0's and 1's. Thus, being sorted would mean all 0's appear before all the 1's.

First, suppose A is ε-far. Observe the following:

- A must have $\geq \varepsilon \cdot n$ 0's
- A must have $\geq \varepsilon \cdot n$ 1's

Why must this hold? Otherwise, we could change these $< \varepsilon \cdot n$ elements to get a sorted array which in turns means that the array is not ε-far.

We use these observations to define two sets.

- $Left_1$: Set of the smallest $\varepsilon \cdot \frac{n}{2}$ indices in A that contain a 1 (this is well-defined as A should have at least $\varepsilon \cdot n$ 1's, otherwise we could change all the 1's to 0 we can make A the all-zero array, a contradiction with A being ε-far from sorted).
- $Right_0$: Set of the largest $\varepsilon \cdot \frac{n}{2}$ indices in A that contain a 0 (this is well-defined for the same argument as $Left_1$).

Example.

0	1	1	1	0	1	0	0	1	1
1	2	3	4	5	6	7	8	9	10

Given this array and $\varepsilon = \frac{1}{2}$, we have $\frac{\varepsilon \cdot n}{2} = 2$. Thus: $Left_1 = \{2, 3\}$ and $Right_0 = \{7, 8\}$

Lemma 1. Let i be the largest index in $Left_1$ and j be the smallest index in $Right_0$. Then $i < j$.

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

Algorithm.

- Sample t indices $i_1, i_2, \ldots, i_t$
- Query $A[i_1], A[i_2], \ldots, A[i_t]$
- If any pair of number is inverted ($i < j$ but $A[i] > A[j]$) then output No, else output Yes

Lemma 2. If A is sorted then the probability that our algorithm outputs an incorrect answer is 0.
Proof. This is immediately clear, since a sorted array would have no inversions

Lemma 3. If A is ε-far from being sorted, and $t \geq \frac{2}{\varepsilon} \ln \frac{2}{\delta}$, then Pr[Wrong Answer] $\leq \delta$.

Proof. Define the following two events

- E_L = event that we choose one sample from $Left_1$.
- E_R = event that we choose one sample from $Right_0$.

If E_L and E_R both happen, then by Lemma 1 we will find an inversion and output the correct answer – A is ε-far. Thus we have

$$\begin{aligned}
\Pr[\text{Wrong Answer}] &\leq \Pr[\bar{E_L} \cup \bar{E_R}] \\
&\leq \Pr[\bar{E_L}] + \Pr[\bar{E_R}] \\
&= (1 - \frac{\frac{\varepsilon \cdot n}{2}}{n})^t + (1 - \frac{\frac{\varepsilon \cdot n}{2}}{n})^t \\
&\leq 2e^{-\frac{\varepsilon t}{2}} \qquad\qquad (1 + x \leq e^x)
\end{aligned}$$

Now, suppose we are allowed to be wrong with probability at most δ. We then require

$$\begin{aligned}
2e^{\frac{\varepsilon t}{2}} &\leq \delta \\
e^{\frac{\varepsilon t}{2}} &\geq \frac{2}{\delta} \\
\frac{\varepsilon t}{2} &\geq \ln \frac{2}{\delta} \\
t &\geq \frac{2}{\varepsilon} \ln \frac{2}{\delta}
\end{aligned}$$

Hence, if we let $t \geq \frac{2}{\varepsilon} \ln \frac{2}{\delta}$, then $\Pr[\text{Wrong Answer}] \leq \delta$, as claimed!

Runtime

The first two steps of the algorithm require $O(t)$ time. The last step can also be implemented in $O(t)$ time by computing the maximum index i among all sampled indices which are 0 and minimum index j among all sampled indices which are 1 and determining if $i > j$. Thus the runtime of this algorithm is $O(t) = O(\frac{2}{\varepsilon} \ln \frac{2}{\delta})$.

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!

General Case

We will now address the general case where the numbers in the array A are arbitrary. First, notice that the previous algorithm fails to solve the problem on general arrays in constant time with. Consider the following input array:

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

This array is $\frac{2}{3}$-far from being sorted as 2 indices from each block of 3 must be changed to make the array sorted. Then, the only way the above algorithm gives the correct answer on this array is if it samples two (or three) indices from the same block of size three. One can readily show that for this to happen, the

size of the sampled set must be increased to $\Theta(\sqrt{n})$. Demonstrating this is an exercise left to the reader. Hint: the birthday paradox!

Our new "tester" will be based on binary search. For the remainder of this, we also assume that the numbers in A are distinct. This loses no generality because we can break the ties between equal numbers consistently by their index in the array. That is, we can say $A[i]$ is "smaller" than $A[j]$ if either $A[i] < A[j]$ or $A[i] = A[j]$ and $i < j$).

We now require the following definition:

Definition. A binary search of a number a on array A (which may or may not be sorted) is termed *consistent* if it returns a.

Example.

25	11	39	45	61	29	80
1	2	3	4	5	6	7

In this example, a binary search on 11 is consistent as we would query the element at index 4, then it would query the element at position 2 which is 11.

However, a binary search on 29 is not consistent as it would query the element at index 4, then it would query the element at position 2, then it would query the element at position 3 and return that 29 is not in the array.

Lemma 4. Suppose binary search on $A[i]$ and $A[j]$ is consistent. If $i < j$ then $A[i] < A[j]$.

Proof. Consider the first time when the pivot p is chosen between i and j. Then, since the binary search on $A[i]$ is consistent we have:

$$A[i] \leq A[p]$$

Similarly, since the binary search on $A[j]$ is consistent we have:

$$A[j] \geq A[p]$$

Since we assume the elements are distinct, equality can only hold in *one* of the above. Thus we have

$$A[i] < A[j]$$

With these results, we are ready present our tester, based on binary search.

Binary Search Tester

- Sample $i \in \{1, 2, \ldots, n\}$
- Do *binary search* on $A[i]$
- Output YES if the *binary search* is consistent and NO otherwise

Lemma 5. The binary search tester outputs the wrong answer, NO, with probability 0 whenever A is sorted. Moreover, the algorithm outputs the wrong answer, YES, with probability at most $1 - \varepsilon$ whenever A is ε-far from being sorted.

Proof. When A is sorted the algorithm always outputs YES, as we only outputs NO if the binary search is not consistent. Clearly, binary search is always consistent for every element of a sorted array.

Now, suppose A is ε-far. To prove this part of the lemma, define a new set $C(A)$: the set of all indices on which binary search is consistent. Explicitly: $C(A) = \{i_1, i_2, \ldots, i_t\}$. Observe that $t \leq (1 - \varepsilon)n$, as we can change $\varepsilon \cdot n$ of the elements and the (1 - ε) $\cdot\, n$ unchanged elements would now be consistent – A

Now the only way that our algorithm will output the wrong answer is if we pick an element from the set $C(A)$

$$\begin{aligned} \Pr[\text{Wrong Answer}] &\leq \frac{(1-\varepsilon)n}{n} \\ &= 1 - \varepsilon \end{aligned}$$

Final Algorithm

Now we will decrease the probability of getting a wrong answer by running the algorithm multiple times.

Algorithm.

- Run *Binary Search Tester* k times independently
- Output YES if all runs output YES and output NO otherwise

Analysis. First, note that this algorithm will always output the correct answer for a sorted array – *every* binary search will be consistent.

Now consider the case when A is ε-far. Since this algorithm runs the tester k times *independently*, we can apply Lemma 5 repeatedly to bound the probability of a wrong answer. Namely, since each run of the tester is wrong on an ε-far array with probability $1 - \varepsilon$, we have:

$$\begin{aligned} \Pr[\text{Wrong Answer}] &\leq (1-\varepsilon)^k \\ &\leq e^{-\varepsilon k} \qquad\qquad (1 + x \leq e^x) \end{aligned}$$

We want this probability to be at most δ.

$$\begin{aligned} e^{-\varepsilon k} &\leq \delta \\ e^{\varepsilon k} &\geq \frac{1}{\delta} \\ \varepsilon k &\geq \ln \frac{1}{\delta} \\ k &\geq \frac{1}{\varepsilon} \ln \frac{1}{\delta} \end{aligned}$$

Runtime. Since we run binary search k times on an array of size n, the algorithm's runtime is $O(k \log n)$. Thus, our algorithm outputs the correct answer with probability $1 - \delta$ in $O(k \log n)$ time.