Notes for lecture #1: Introduction to sublinear algorithms

- The focus of the course is on *sublinear algorithm*. This may mean more than one thing. In particular we'll be interested in algorithms whose running time is sublinear in the size of the input, and so, in particular, they don't even read the whole input. Other families of problems assume that the algorithm does read the whole input but is allowed only sublinear space (e.g., streaming algorithms). Will talk mainly about the first kind.

- Two very simple examples:

  1. Let $w \in \{0,1\}^n$. Is w the all-0 string? If we have to answer exactly, then must examine the whole string. But suppose we are given a parameter $\epsilon > 0$, and are required to say YES if $w = \vec{0}$, and are required to say NO only if w contains more than ϵn '1's (think of '1's as "errors"). Then we can perform this task with high success probability by observing only $s = \Theta(1/\epsilon)$ uniformly and independently at random selected bits from the string. Elaborate on probability of error:

     $$\Pr[\text{ no 1s in sample} \,|\, \exists \text{ more than } \epsilon n \text{ 1s}] < (1-\epsilon)^s < e^{-\epsilon s} = e^{-c}$$

  2. Let $w \in \{0,1\}^n$. What is the fraction of 1's in w? Again, in order to answer exactly must read every bit. But if we want just an approximate answer (like in polls) then sufficed to look at $\Theta(1/\epsilon^2)$ bits in order to get an additive error of ϵ, and suffices to look at $\Theta(1/(\gamma^2 p))$ bits to get a multiplicative error of $(1 \pm \gamma)$ where p is lower bound on the actual fraction of 1's.

- The above examples were of two types: the first required for an *approximate decision*, and the second for an *approximate value*. We shall start by talking about problems of the first type, which we refer to as *Property Testing* problems. In the above example the property was "being the all-0 string" where only one string obeys the property. Another example where there is a larger set of strings that obey the property is: "The string contains the same number of 1's as 0's". Note that in this case we might err both in the case that the string has an equal number and in the case it doesn't.

- In general, a property testing problem is defined by a family of objects (e.g., strings over $\{0,1\}^n$) and a property $\mathcal{P}$ (a subset of the objects that obey the property). There is also a normalized distance measure $\text{dist}(\cdot,\cdot)$ defined on pairs of objects, and (with slightly abuse of notation) we denote $\text{dist}_{\mathcal{P}}(O) = \min_{O' \text{ has } \mathcal{P}}\{\text{dist}(O,O')\}$. We say that O is ϵ-far from having $\mathcal{P}$ if $\text{dist}_{\mathcal{P}}(O) > \epsilon$, otherwise it is ϵ-close.

  A testing algorithm (tester) is given query access to an object O (e.g., can ask what is the i'th bit for any index $1 \leq i \leq n$), and is given a distance or approximation parameter ϵ. We require that:

  - If O has property $\mathcal{P}$ then the algorithm should accept with probability at least $2/3$;
  - If O is ϵ-far from having $\mathcal{P}$ then the algorithm should reject with probability at least $2/3$.

  If the algorithm accepts objects that have the property with probability 1 then it has *one-sided error*. Can easily boost the success probability by repetitions (homework exercise).

  Our goal is to design property testing algorithms whose query and time complexities are sublinear in the size of the object (or possibly even independent of it (discussion of $\log |O|$)).

  To summarize, need to know what are objects, what is property, what is distance measure and what are the allowed queries (sometimes this is clear from the context, but not always, and there maybe be more than one reasonable choice).

- Some motivating discussion.
  - In some cases object is HUGE (e.g. very large databases), so can't allow even a linear time algorithm, and must do with approximate decision.
  - In some case, object is not huge but problem is hard (e.g., deciding 3-colorability) so some form of approximation is inevitable. This is one such form.
  - Even if object is not huge and problem is not hard, it might be worthwhile to get an approximate answer (and say, "pass" objects that are not perfect), since the algorithm is much more efficient than the exact algorithm.
  - If must get an exact answer (but can do with a probability of failure), then run testing algorithm as a preprocessing step. If the object is very far, then it will catch it quickly. Otherwise, the testing algorithm might not catch it, and then run exact algorithm.

- Types of problems studied in the context of property testing: graph properties, algebraic properties of functions (with applications to coding), string properties, clustering, properties of boolean functions and more. (Everything can of course be encoded as a string, but then sometimes loose natural structure.)

  In all cases have algorithms with query complexity that is sublinear in the size of the object. At best, they are even independent of the size of the object.

- First example (other than the simple ones shown above): Objects are functions $f : \{1, \ldots, n\} \to \mathbb{N}$ (can also be viewed as strings of length n over natural numbers). The property is that function is monotone. Namely, $f(i) \leq f(j)$ for all $1 \leq i < j \leq n$. The distance measure is Hamming distance, that is, the number of domain elements whose value should be modified. Queries are simply queries to the function (i.e., what is $f(i)$?).

  There are actually several algorithms to test this property. Their query complexity and running time are $O(\log n/\epsilon)$, and it can be proved that $\Omega(\log n)$ queries are necessary for constant ϵ. (CAN WE GET $\log n/\epsilon$?).

  We first show that the "naive" algorithm, which simply takes a uniform sample of indices in $\{1, \ldots, n\}$ and checks whether non-monotonicity is observed (i.e., whether in the sample there are $i < j$ such that $f(i) > f(j)$), requires $\Omega(\sqrt{n})$ queries for constant ϵ. To see why this is true, consider the function $f(i) = i + 1$ for odd i and $f(i) = i - 1$ for even i. That is, the values of the function are: $2, 1, 4, 3, \ldots, n, n-1$. We call each pair $i, i+1$ where $f(i) = i+1$

and $f(i+1) = i$ a *matched pair*. Note that the algorithm rejects only if it gets a matched pair.

Claim 1 *The function f is $1/2$-far from being monotone.*

Proof: For each matched pair must modify at least one of the members of the pair in order to make it monotone. ■

Claim 2 *The probability that a uniform sample of size $s \leq \sqrt{n}/2$ contains a matched pair is less than $1/3$.*

Proof: Let $i_1, \ldots, i_s$ be the indexes selected by the sample. For each $1 \leq k < \ell \leq s$ let $E_{k,\ell}$ denote the event that i_k, i_ℓ are a matched pair (that is $i_k = 2j,\ i_\ell = 2j+1$ or $i_k = 2j+1,\ i_\ell = 2j$ for $j = 1 \leq j \leq n/2$). Then $\Pr[E_{k,\ell}] = \frac{1}{n}$. Now, the probability that the sample contains a matched pair is

$$\Pr\left[\bigcup_{k<\ell} E_{k,\ell}\right] \leq \sum_{k<\ell} \Pr[E_{k,\ell}] = \binom{s}{2} \cdot \frac{1}{n} < \frac{1}{4}$$

■

This is a special case of the lower bound in what is known as the "birthday paradox". (The other direction will imply that a sample of size $c \cdot \sqrt{n}$ for some constant c will contain a matched pair with high probability.)

To describe the algorithm assume that all values are distinct. This can be assumed without loss of generality by observing that if this is not the case then we can replace each value with a pair $(f(i), i)$, where $(f(i), i) < (f(j), j))$ if and only if either $f(i) < f(j)$ or $f(i) = f(j)$ but $i < j$. The distance to monotonicity of the new function is the same as the old function.

Monotonicity Testing Algorithm

- Uniformly and independently at random select $s = 2/\epsilon$ indices $i_1, \ldots, i_s$.
- For each index i_r selected, query $f(i_r)$, and perform a binary search on f for $f(i_r)$. (Middle element is element number $\lceil \frac{n}{2} \rceil$.)
- If for any i_r, the binary search failed, then output reject. Otherwise output accept.

(Show example: 21436587109, and search for 7 and then 3).

Clearly if the function f is monotone then the algorithm always accepts (by definition of a binary search on a sorted list). Assume from this point on the f is ϵ-far from being monotone. We show that the algorithm rejects with probability at least $2/3$.

We say that an index j is a *witness* (to the non-monotonicity of f), if a binary search for $f(j)$ fails.

Lemma 3 *If f is ϵ-far from being monotone than there are least $\epsilon \cdot n$ witnesses $1 \leq j \leq n$.*

Proof: Assume, contrary to the claim, that there are less than ϵn witnesses. We shall show that f is ϵ-close to being monotone, in contradiction to the premise of the lemma. Specifically, we will show that if we consider all non-witnesses then they constitute a monotone sequence. For each pair of non-witnesses, j, j' where $j < j'$ consider the steps of the binary search for $f(j)$ and $f(j')$, respectively. Let u be the first index for which the two searches diverge. Namely, $j < u < j'$ and $f(j) < f(u)$ and $f(j') > f(u)$ (explain with drawing: j and j' both belong to an interval that u is its midpoint, where one is on one side and the other on the other side). But then $f(j) < f(j')$, as required. But then we are done since by modifying each witness to obtain the value of the nearest non-witness, we can make f into a monotone function by modifying the at most ϵn witnesses. ■

Corollary 4 *If f is ϵ-far from monotone then algorithm rejects with probability at least* $2/3$.

Proof: The probability that the algorithm doesn't reject (i.e., accepts) equals the probability that no witness is selected in the sample.

$$\Pr[\text{ not witness is selected }] < (1-\epsilon)^s < e^{-\epsilon s} = e^{-2} < 1/3$$

■