Sublinear Time Algorithms

• Sublinear time
  - Algorithm runs in $o(n)$ time, where $n =$ length of input
    - Assume direct access to $i^{th}$ bit of the input
  - Algorithm cannot even read the entire input!
  - (With a few exceptions) the algorithm must
    - Use randomization
    - Provide an approximately accurate answer

• Also interesting: sublinear space
  - Algorithm uses $o(n)$ additional space
Motivation

• Huge datasets
  ➢ World-wide web, social networks, genome project, sales logs, census data, high-resolution images, fine-grained scientific measurements, ...

• Need fast algorithms for subroutines that will be called frequently

• Many sublinear algorithms turn out to be streaming algorithms, which only need to access incoming data once
Exact vs Inexact Algorithms

• Exact: Always provides the right answer
• Inexact: Provides an approximately optimal answer
  ➢ $ANS = \text{right answer, } ALG = \text{output of algorithm}$
  ➢ For numerical answers (e.g., counting problems)
    o $(1 - \varepsilon) ANS \leq ALG \leq (1 + \varepsilon) ANS$
  ➢ For binary answers (e.g., yes/no problems)
    o 1-sided error:
      • $ANS = YES \Rightarrow ALG = YES$ with probability 1
      • $ANS = NO \Rightarrow ALG = NO$ with probability $\geq \frac{2}{3}$
    o 2-sided error:
      • $ALG = ANS$ with probability $\geq \frac{2}{3}$
Exact vs Inexact Algorithms

- Exact: Always provides the right answer
- Inexact: Provides an approximately optimal answer
  - $ANS = \text{right answer}, ALG = \text{output of algorithm}$
  - For “property testing”
    - Property satisfied $\Rightarrow ALG = YES$ with probability 1
    - Property at least $\epsilon$-far from being satisfied $\Rightarrow ALG = NO$ with probability $\geq \frac{2}{3}$
    - Property not satisfied, but $\epsilon$-close to being satisfied $\Rightarrow$ don’t care what $ALG$ is
    - Will see this later...
Deterministic + Exact

• Always give the right answer using a deterministic algorithm that does not read the entire input!
  ➢ Seems impossible

• You already know one such algorithm
  ➢ Binary search! $O(\log n)$ time, deterministic, exact.
  ➢ Needs to assume that input is already sorted.

• In general, deterministic + exact is impossible unless input is structured.
Deterministic + Inexact

• Approximating the diameter in a metric space
  ➢ Points $x_1, \ldots, x_n$, distance metric $d$
  ➢ Input: $n^2$ numbers $d_{ij} = d(x_i, x_j)$
  ➢ Goal: estimate diameter $D = \max_{i,j} d(x_i, x_j)$

• Algorithm: Pick arbitrary $x_i$, return $D_i = \max_{j \neq i} d_{ij}$

• Analysis:
  ➢ $O(n)$ running time, “sublinear” in the input length $O(n^2)$
  ➢ $D = d_{a,b} \leq d_{a,i} + d_{i,b} \leq D_i + D_i = 2D_i$
  ➢ Also, clearly $D \geq D_i$. Thus, we have a 2-approximation!
Deterministic + Inexact

• This is also somewhat an exception

• **Note:** If you were given $n$ explicit points in a Euclidean space instead of the $O(n^2)$ distances, then $O(n)$ would no longer be sublinear.

• For most sublinear time algorithms, randomization is a must!
Randomized + Exact

• Known as Las Vegas algorithms
  ➢ Distribution over deterministic algorithms
  ➢ Each algorithm is exact, i.e., produces the correct answer.
  ➢ The algorithms have varying costs on different instances.
  ➢ Hope is that a randomization over them will have low expected cost on every instance.

• Example: two algsos \{Alg1,Alg2\}, two instances \{I1,I2\}
  ➢ Alg1 takes 1000 steps on I1, 10 on I2.
  ➢ Alg2 takes 10 steps on I2, 1000 on I2.
  ➢ \((\frac{1}{2})\text{Alg1}+(\frac{1}{2})\text{Alg2}\) takes 505 expected steps on any instance
Searching in Sorted List

• **Input:** A sorted doubly linked list with \( n \) elements.
  ➢ Imagine you have an array \( A \) with \( O(1) \) access to \( A[i] \)
  ➢ \( A[i] \) is a tuple \((x_i, p_i, n_i)\)
    o Value,
    o index of previous element in list,
    o index of next element in list.
  ➢ Sorted: \( x_{p_i} \leq x_i \leq x_{n_i} \)

• **Task:** Given \( x \), check if there exists \( i \) s.t. \( x = x_i \)

• **Goal:** We will give a randomized + exact algorithm with expected running time \( O(\sqrt{n}) \)!
• **Motivation:**

- Often we deal with large datasets that are stored in a large file on disk, or possibly broken into multiple files.
- Creating a new, sorted version of the dataset is expensive.
- It is often preferred to “implicitly sort” the data by simply adding previous-next pointers along with each element.

- Would like algorithms that can operate on such implicitly sorted versions and yet achieve sublinear running time.
  - Just like binary search achieves for an explicitly sorted array.
Searching in Sorted List

Algorithm:

- Select $\sqrt{n}$ random indices $R$
- Access $x_j$ for each $j \in R$
- Find the nearest $x_j: j \in R$ on each side of $x$
  - $p \in R$ such that $x_p = \max\{x_j: x_j \leq x, j \in R\}$
  - $q \in R$ such that $x_q = \min\{x_j: x_j > x, j \in R\}$
  - One of the two must exist (WHY?).
- If $p$ exists, start at $A[p]$, and keep going next until you discover $x$, or you reach $A[q]$ or end of list.
- If $q$ exists, start at $A[q]$, and keep going back until you discover $x$, or you reach $A[p]$ or beginning of list.
Searching in Sorted List

• Analysis:
  ➢ Take arbitrary value $x$. Take the minimum value $x_i$ in the list that is at least $x$. The algorithm is searching for $x_i$.
  ➢ The algorithm throws $\sqrt{n}$ random “darts” on the list.
  ➢ **Chernoff bounds:** the probability that there is no dart in $c\sqrt{n}$ elements to the left (resp. right) of $x_i$ is $2^{-\Omega(c)}$.
  ➢ So, the expected distance of $x_i$ to the dart on its left (and its right) is $O(\sqrt{n})$.
  ➢ The algorithm finds these two darts in $O(\sqrt{n})$ time, and uses $O(\sqrt{n})$ search to locate $x_i$. 
Searching in Sorted List

• **Theorem:** There exists an exact randomized algorithm for searching in a sorted doubly linked list in \( O(\sqrt{n}) \) expected running time.

  ➢ **Note:** We don’t *really* require the list to be doubly linked. Just “next” pointer suffices if we have a pointer to the first element of the list (a.k.a. “anchored list”).

• We can also prove optimality of this algorithm!

• **Theorem:** No exact randomized algorithm can do this in \( o(\sqrt{n}) \) expected running time.
Yao’s Principle

• Proves a lower bound on the expected running time of the best exact randomized algorithm
  ➢ “The expected time of a randomized algorithm $R$ on the worst input $I$ is no better than the expected time taken, under the worst probability distribution $\mathbb{I}$ over inputs, by the best deterministic algorithm $A$ for that distribution.”
  ➢ $\max_I E_R[C(R, I)] \geq \max_D \min_A E_{I \sim D}[C(A, I)]$

• Special case of von Neumann’s minimax theorem for two-player zero-sum games
  ➢ Can see a randomized algorithm as a distribution over all deterministic algorithms
Searching in Sorted List

• **Adversarial distribution**: uniform random ordering of $n$ distinct values

• **Goal**: Search the max value (last element of the list)

• We want to show that any deterministic algorithm takes $\Omega(\sqrt{n})$ steps in expectation.

• Deterministic algorithms have only two operations:
  - **Op A**: Access next/previous of an already accessed element.
  - **Op B**: Compute an index $k$, and access the $k^{th}$ element.
Searching in Sorted List

• **Note:** In a B op, the algorithm can compute index $k$ using any information it has until that point.

• Let $T = \text{the set of last } \sqrt{n} \text{ elements on the list.}$

  ➢ If the algorithm doesn’t access any element of $T$ using a B op, it must take $\Omega(\sqrt{n})$ A ops to locate the last element starting from any accessed element $\Rightarrow$ done!

• We show that the algorithm requires $\Omega(\sqrt{n})$ steps in expectation to access an element of $T$ using B op

  ➢ **Note:** Each $B$ op returns a random element from the yet unexplored list. (WHY?)
Searching in Sorted List

• To show: $\Omega(\sqrt{n})$ steps required to access $T$

  ➢ After $a$ many A ops and $b$ many B ops, probability of accessing an element of $T$ in the next B op is at most

  $$\frac{|T|}{|\text{Unexplored List}|} = \frac{\sqrt{n}}{n - a - b} \leq \frac{\sqrt{n} + a + b}{n}$$

  ➢ Probability that no element of $T$ is accessed after $a$ many A ops and $b$ many B ops is at least $\left(1 - \frac{\sqrt{n} + a + b}{n}\right)^b$

  ➢ This implies that the expected number of steps until an element of $T$ is accessed is $\Omega(\sqrt{n})$. (WHY?) QED!
Sublinear Geometric Algorithms

• Chazelle, Liu, and Magen [2003] proved the $\Theta(\sqrt{n})$ bound for searching in a sorted linked list

  ➢ Their main focus was to generalize these ideas to come up with sublinear algorithms for geometric problems

  ➢ **Polygon intersection**: Given two convex polyhedra, check if they intersect.

  ➢ **Point location**: Given a Delaunay triangulation (or Voronoi diagram) and a point, find the cell in which the point lies.

  ➢ They provided optimal $O(\sqrt{n})$ algorithms for both these problems.
Randomized + Inexact

• We will now move to inexact algorithms that return approximately accurate answers.

• Let us start with a counting problem where the answer is numerical.
Estimating Avg Degree in Graph

• **Input:** Graph $G$ with $n$ vertices, and access to an oracle that returns the degree of a queried vertex in $O(1)$ time.

• **Output:** $\alpha$-approximation of the average degree $d$ of the vertices of $G$.
  ➢ $\alpha$-approximation $\Rightarrow$ answer lies in $[d/\alpha, \alpha \cdot d]$  

• **Goal:** $(2 + \epsilon)$-approximation in expected time $O\left(\epsilon^{-O(1)} \sqrt{n}\right)$
  ➢ $\epsilon$ is constant $\Rightarrow$ sublinear in input size $n$
Estimating Avg Degree in Graph

• Wait!
  ➢ Isn’t this equivalent to “given an array of \( n \) numbers between 1 and \( n - 1 \), estimate their average”?
  ➢ No! That requires \( \Omega(n) \) time for constant approximation!
    o Consider an instance with constantly many \( n - 1 \)’s, and all other 1’s: you may not discover any \( n - 1 \) until you query \( \Omega(n) \) numbers

• Why are degree sequences more special?
  o Erdős–Gallai theorem: \( d_1 \geq \cdots \geq d_n \) is a degree sequence iff their sum is even and \( \sum_{i=1}^{k} d_i \leq k(k - 1) + \sum_{i=k+1}^{n} d_i \).
  o Intuitively, we will sample \( O(\sqrt{n}) \) vertices
    • We may not discover the few high degree vertices, but we’ll find their neighbors, and thus account for their edges anyway!
Estimating Avg Degree in Graph

• Algorithm:
  ➢ Take \(8/\epsilon\) random subsets \(S_i \subseteq V\) with \(|S_i| = s\)
  ➢ Compute the average degree \(d_{S_i}\) in each \(S_i\).
  ➢ Return \(\hat{d} = \min_i d_{S_i}\)

• Analysis:
  ➢ We will show that with \(s = \Theta(\sqrt{n}/\epsilon^{O(1)})\), we can ensure \(\hat{d} \in [(1/2 - \epsilon) d, (1 + \epsilon) d]\) with probability at least \(3/4\).
    o Note: #queries (and running time) = \(O(\sqrt{n}/\epsilon^{o(1)})\)
    o Feige [2006] improved this to \(O(\epsilon^{-1}\sqrt{n}/d_0)\) if we know \(d \geq d_0\)
      • In particular, even with \(d_0 = 1\), we have \(O(\sqrt{n}/\epsilon)\) queries.
Estimating Avg Degree in Graph

• **Claim 1:** We can choose \( s = \Theta(\sqrt{n}/\epsilon^{O(1)}) \) such that \( \Pr[d_S < (1/2 - \epsilon) d] \leq \epsilon/64. \)

• **Proof:**
  - Let \( H \) be the set of \( \sqrt{\epsilon'n} \) highest degree vertices in \( G \), and \( L = V \setminus H \).
  - Sub-claim: \( \sum_{i \in L} d_i \geq (1/2 - \epsilon') \sum_{i \in V} d_i \)
    - Note that \( \sum_{i \in V} d_i \) counts each edge in the graph twice.
    - \( \sum_{i \in L} d_i \) might omit at most \( \epsilon' n \) edges within \( H \), and might only count edges between \( H \) and \( L \) once.
      - Thus, \( \sum_{i \in L} d_i \geq 1/2 \left( \sum_{i \in V} d_i - \epsilon' n \right) \)
    - The sub-claim now follows when you substitute \( n \leq \sum_{i \in V} d_i \) in the above equation (which is true because \( G \) is connected).
Estimating Avg Degree in Graph

• Proof:
  - We proved: $\sum_{i \in L} d_i \geq \left(\frac{1}{2} - \epsilon'\right) \sum_{i \in V} d_i$
    - Thus, average degree in $L \geq \left(\frac{1}{2} - \epsilon'\right) d$.
  - A lower bound on $d_S$: assume all its vertices come from $L$
    - Let $d_H = \text{minimum degree of any vertex in } H$.
    - Let $X_i = \text{degree of } i^{th} \text{ vertex in } S \Rightarrow X_i \in [1, d_H]$.
    - $E[X_i] \geq \left(\frac{1}{2} - \epsilon'\right) d \geq \left(\frac{1}{2} - \epsilon'\right)d_H |H|/n$
    - $t = E[\sum_{i=1}^{s} X_i] = \Omega(d_H)$ due to our choice of $s$
  - Hoeffding’s bound:
    - $\Pr[\sum_{i=1}^{s} X_i < (1 - \epsilon') \cdot t] \leq e^{-\frac{t(\epsilon')^2}{d_H}} \leq \frac{\epsilon}{64}$
    - Set $\epsilon'$ such that $(1 - \epsilon') \cdot \left(\frac{1}{2} - \epsilon'\right) = \frac{1}{2} - \epsilon$
Estimating Avg Degree in Graph

• **Claim 2:** \( \Pr[d_S > (1 + \epsilon)d] \leq 1 - \epsilon/2. \)

• **Proof:**
  - Markov’s inequality
  - \( \Pr[d_S > \ell] \leq \frac{E[d_S]}{\ell} = \frac{d}{(1+\epsilon)d} = \frac{1}{1+\epsilon} \leq 1 - \frac{\epsilon}{2} \)

• **Finishing the proof:**
  - \( \Pr[d_S < (1/2 - \epsilon) d] \leq \epsilon/64 \) -- low probability!
  - \( \Pr[d_S > (1 + \epsilon)d] \leq 1 - \epsilon/2 \) --- high probability!
  - Thus, we repeat \( 8/\epsilon \) times, and take the minimum.
    - With \( 3/4 \) probability, no trial goes below \( (1/2 - \epsilon) d \), but at least one comes below \( (1 + \epsilon)d \).

QED!
Effect of Input Query Model

• “Degree Queries”
  ➢ Here, we assumed that we have $O(1)$ time access to degree of a node.
  ➢ Feige’s algorithm achieves $(2 + \epsilon)$-approximation using $O(\sqrt{n}/\epsilon)$ queries.
  ➢ Feige also proved optimality of this algorithm: any algorithm that gives $(2 - \epsilon)$-approximation must use $\Omega(n)$ queries.

• What if the query model was different?
Effect of Input Query Model

• “Neighbor Queries”
  ➢ Query: \((v, j)\)
  ➢ Obtain: \(j^{th}\) neighbor of \(v\) (in some order), or “FALSE” (if \(v\) has degree less than \(j\))
  ➢ We can mimic degree query using \(O(\log n)\) queries
    o Feige’s algorithm can run using \(O(\sqrt{n} \log n \ \epsilon^{-1})\) queries
  ➢ Goldreich and Ron show that this model is actually very powerful
    o We can do \((1 + \epsilon)\)-approximation with \(O\left(\sqrt{n} \ poly(\log n, \epsilon^{-1})\right)\) queries
    o They also show a \(\Omega(\sqrt{n}/\epsilon)\) lower bound.
Estimating Maximal Matching

• Problem
  ➢ **Input:** Graph $G = (V, E)$
  ➢ **Output:** $\tilde{m}$ such that $m \leq \tilde{m} \leq m + \epsilon n$ with prob at least $2/3$, where $m$ is the size of some maximal matching
  ➢ **Goal:** $2^{O(D)}/\epsilon^2$ running time, where $D$ is max degree
    o Sublinear time when $D = o(\log n)$

• Motivation
  ➢ Size of maximum matching and maximum vertex cover both lie in $[m, 2m]$
  ➢ Gives a sublinear 2-approximation algorithms for these problems
Estimating Maximal Matching

• We will estimate the size of maximal matching (MM) produced by the greedy algorithm parametrized by an ordering $\sigma$ of the edges

**Greedy MM($\sigma$):**

- Start with empty matching.
- For $e \in E$ (in the order of $\sigma$)
  - If $e$ does not “conflict” with already created matching, add it.

• Fix an arbitrary $\sigma$

  - We can’t explicitly do this in sublinear time.
  - We’ll handle this later.
Estimating Maximal Matching

• Suppose we have access to an oracle that tests whether an edge $e$ belongs to greedy matching $M$.

• Algorithm:
  - $S \leftarrow \frac{8}{\epsilon^2}$ vertices of $V$ sampled i.i.d.
  - $X_v = 1$ if there exists an edge $e$ incident on $v \in S$ that is in $M$, and 0 otherwise
  - Return $\tilde{m} = \frac{1}{2} \cdot \left( n \cdot \frac{\sum_{v \in S} X_v}{|S|} \right) + \frac{1}{2} \cdot (n \cdot \epsilon)$
Estimating Maximal Matching

• Recall: \( \tilde{m} = \frac{1}{2} \cdot \left( n \cdot \frac{\sum_{v \in S} X_v}{|S|} \right) + \frac{1}{2} \cdot (n \cdot \epsilon) \)

• Claim: \( E[\tilde{m}] = |M| + \frac{\epsilon n}{2} \)

• Proof:
  - \( E \left[ \frac{\sum_{v \in S} X_v}{|S|} \right] = \text{prob of a random vertex being matched in } M \)
  - \( E \left[ n \cdot \frac{\sum_{v \in S} X_v}{|S|} \right] = 2 \cdot |M| \quad \text{(\#matched vertices = 2 \cdot |M|)} \)

• To prove \( |M| \leq \tilde{m} \leq |M| + \epsilon n \) with prob \( \geq \frac{2}{3} \)
  - Apply Hoeffding’s inequality
Estimating Maximal Matching

• What’s left:
  1. Design an oracle for whether $e$ is included in $M$
  2. Handle the issue of not being able to fix $\sigma$ beforehand
  3. Analyze running time

• Oracle: Does $e$ belong to greedy matching $M$?
  ➢ Observation: $e$ belongs to $M$ iff no edge $e'$ adjacent to $e$ with $\sigma(e') < \sigma(e)$ belongs to $M$.
  ➢ Recursive call on all adjacent edges with lower priority. If all return NO, return YES, else return NO.
Estimating Maximal Matching

• What’s left:
  1. Design an oracle for whether $e$ is included in $M$
  2. Handle the issue of not being able to fix $\sigma$ beforehand
  3. Analyze running time

• Generating permutation $\sigma$
  ➢ We will store a random number $r_e \sim U[0,1]$ for each $e$.
  ➢ We will store them in a binary search tree.
  ➢ Start with an empty tree.
  ➢ When we need to check the priority of $e$, see if it’s already generated. If not, generate it.
Estimating Maximal Matching

• Running time : Oracle
  ➢ Consider the adjacency tree for edge $e$.
    o Root = $e$
    o For every node, its children are all its adjacent edges.
  ➢ Consider a node $t$ at depth $k$
    o For the oracle to be called on $t$, the $k + 1$ priorities from root to $t$
      must be monotonically decreasing
    o This happens with probability $1/(k + 1)!$
  ➢ #nodes at depth $k = (2D)^k$
    o Max degree $D \Rightarrow$ fanout is at most $2D$
  ➢ Expected recursive calls $\leq \sum_{k=0}^{\infty} \frac{(2D)^k}{(k+1)!} \leq \frac{e^{2D}}{2D}$
Estimating Maximal Matching

- Running time: Algorithm
  - For $8/\epsilon^2$ nodes, call the oracle on all their incident edges (at most $D$ per node)
  - Total queries to the graph = $(8/\epsilon^2) \cdot D \cdot \frac{e^{2D}}{2D} = \frac{2^{O(D)}}{\epsilon^2}$
  - QED!
Estimating Maximal Matching

• Note
  ➢ Let $m^*$ be the size of a maximum matching
  ➢ This only ensures $\frac{m^*}{2} \leq \tilde{m} \leq 2m^* + \epsilon n$ (w.p. 2/3)
  ➢ Suppose we want to achieve $m^* \leq \tilde{m} \leq (1 + \delta)m^* + \epsilon n$ (w.p 2/3)
  ➢ Let $k = 1/\delta$
    o Nguyen and Onak show $\frac{2^{O(D^k)}}{\epsilon^2 2^{k+1}}$ query complexity
    o Yoshida, Yamamoto, and Ito improve it to $D^{O(k^2)} k^{O(k)} \epsilon^{-2}$
      • For a constant $\delta$ (thus a constant $k$), this is polynomial in $D$
Estimating Maximal Matching

• Note

➢ In all the previous algorithms...
  o We ensured sublinear running time.
  o Randomization was only used to ensure that the output is approximately accurate with high probability.

➢ In this algorithm...
  o We make sublinear calls to the oracle only in expectation. In some realizations, we might make $\Omega(n)$ oracle calls.
  o We can avoid this by “cutting off” each call to the oracle after more than $c2^{O(D)}$ recursive calls are made, for a large constant $c$.
  o Using Markov’s inequality, this has a low chance of happening.
Property Testing

• The *inexact* algorithms we saw until now were about estimating numerical values.
  ➢ I say inexact because we saw two exact algorithms for yes/no problems: binary search (deterministic) and searching in sorted list (randomized).

• We will now see inexact algorithms for yes/no problems.
  ➢ One such area is “property testing”.
  ➢ It’s one of the most prevalent applications of sublinear time algorithms, and a research area of its own.
Property Testing

• Problem:
  ➢ Given input $I$, test if it satisfies property $P$.

• Inexact goal:
  ➢ If $I$ satisfies $P$, must return “yes”.
  ➢ If $I$ is at least $\epsilon$-far from satisfying $P$, must return “no” with probability at least $2/3$.
  ➢ If $I$ violates $P$, but is $\epsilon$-close to satisfying $P$, free to return anything (we don’t care!).

• Notes
  ➢ For 2-sided error, we also require “yes” w.p. at least $2/3$.
  ➢ What’s “$\epsilon$-far”? We’ll see.
Testing Linearity of Function

• Consider a Boolean function $f : \{0,1\}^n \rightarrow \{0,1\}$

• We want to test if $f$ is linear:
  - $\exists a_1, \ldots, a_n \in \{0,1\}$ s.t. $f(x_1, \ldots, x_n) = a_1 x_1 + \cdots + a_n x_n$?
  - All computations are in $\mathbb{F}_2$ (modulo 2).
  - Equivalently: $f(x + y) = f(x) + f(y)$, $\forall x, y \in \{0,1\}^n$?

• We say that $f$ is $\epsilon$-close to being linear if $\exists g$ such that
  - $|\{x : f(x) \neq g(x)\}| \leq \epsilon 2^n$.
  - Only need to change $\epsilon$ fraction of values to make it linear.
Testing Linearity of Function

• **Input:** Oracle for accessing \( f \)

• **Goal:** 1-sided algorithm for testing linearity of \( f \) that makes \( O\left(\frac{1}{\epsilon}\right) \) queries.
  - Note: This is independent of \( n \). This is actually achievable for testing many properties.

• **Motivation**
  - Subroutine for many other property testing algorithms
  - Applications in cryptography, coding theory, program checking, PCPs (inapproximability), and Fourier analysis
Testing Linearity of Function

• Algorithm:
  - Sample $\frac{2}{\varepsilon}$ random pairs $(x, y)$
  - If $f(x + y) \neq f(x) + f(y)$ for any pair, output “no”.
  - Else, output “yes”.

• Note
  - Algorithm always outputs “yes” if $f$ is linear.
  - We want to prove that if $f$ is $\varepsilon$-far from being linear, then it outputs “no”, i.e., finds a “violating pair” with probability at least $\frac{2}{3}$.
Testing Linearity of Function

• [Bellare, Coppersmith, Hastad, Kiwi, Sudan ‘95] If $f$ is $\epsilon$-far from linear, then the test fails on a random $(x, y)$ pair with probability at least $\epsilon$.
  ➢ Deep result that uses results from Fourier analysis.

• Assuming this result...
  ➢ Probability that algorithm fails on 1 sample $\leq 1 - \epsilon$
  ➢ Probability that algorithm fails on $2/\epsilon$ samples $\leq (1 - \epsilon)^{2/\epsilon} \leq \left(\frac{1}{e}\right)^2 < \frac{1}{3}$