Lecture 10
Sublinear Time Algorithms (contd)
Recap

• Sublinear time algorithms
  ➢ Deterministic + exact: binary search
  ➢ Deterministic + inexact: estimating diameter in a metric space
  ➢ Randomized + exact: searching in a sorted list
    o Lower bound (thus optimality) using Yao’s principle
  ➢ Randomized + inexact:
    o Estimating average degree in a graph
    o Estimating size of maximal matching in a graph
    o Property testing
      • Testing linearity of a Boolean function
Today

• Continue sublinear time property testing
  ➢ Testing if an array is sorted
  ➢ Testing if a graph is bipartite

• Some comments about sublinear space algorithms

• Begin streaming algorithms
  ➢ Find the missing element(s)
  ➢ Finding very frequent or very rare elements
  ➢ Counting the number of distinct elements
Testing Monotonicity of Array

• **Input:** Array $A$ of length $n$ with $O(1)$ access to $A[i]$

• **Check:** $A[i] < A[i + 1]$ for every $i \in \{1, \ldots, n - 1\}$

• **Definition of “at least $\epsilon$-far”:** You need to change at least $\epsilon n$ entries to make it monotonic
  - Equivalently, there are at least $\epsilon n$ entries that are not between their adjacent values.

• **Goal:** 1-sided algorithm with $O \left( \frac{\log n}{\epsilon} \right)$ queries
Testing Monotonicity of Array

• Proposal:
  ➢ Pick $t$ random indices $i$, and return “no” even if $x_i > x_{i+1}$ for even one of them.

• No!
  ➢ For $1 1 1 \ldots 1 0 0 0 \ldots 0$ ($n/2$ each), we’ll need $t = \Omega(n)$

• Proposal:
  ➢ Pick $t$ random pairs $(i, j)$ with $i < j$, and return “no” if $x_i > x_j$ for even one of them.

• No!
  ➢ $1 0 2 1 3 2 4 3 5 4 6 5 \ldots$ (two interleaved sorted lists)
  ➢ $1/2$-far (WHY?), but need $t \geq \Omega(n)$ (by Birthday Paradox, we also must access $\Omega(\sqrt{n})$ elements) (WHY?)
Testing Monotonicity of Array

• **Algorithm:**
  - Choose $2/\epsilon$ random indices $i$.
  - For each $i$, do a binary search for $A[i]$.
  - Return “yes” if all binary searches succeed.

• Assume all elements are distinct w.l.o.g.
  - Can replace $A[i]$ by $(A[i], i)$ and use lexicographic comparison

• Important observation:
  - “searchable” elements form an increasing subsequence!
    (WHY?)
Testing Monotonicity of Array

• Algorithm:
  ➢ Choose $2/\epsilon$ random indices $i$.
  ➢ For each $i$, do a binary search for $A[i]$.
  ➢ Return “yes” if all binary searches succeed.

• Thus:
  ➢ If $\alpha \cdot n$ elements searchable $\Rightarrow$ array is at most $(1 - \alpha)$-far from monotonic
  ➢ If array is at least $\epsilon$-far from monotonic $\Rightarrow$ at least $\epsilon \cdot n$ elements must not be searchable
    • Each iteration fails to detect violation w.p. at most $1 - \epsilon$
    • All $2/\epsilon$ iterations fail to detect w.p. at most $(1 - \epsilon)^{2/\epsilon} \leq 1/3$
Graph Property Testing

• It’s an active area of research by itself.
• Let $G = (V, E)$ with $n = |V|$ and $m = |E|$.

• Input models:
  ➢ **Dense**: Represented by adjacency matrix
    o Query if $(i, j) \in E$ in $O(1)$ time
    o $\epsilon$-far from satisfying $P$ if $\epsilon n^2$ matrix entries must be changed to satisfy $P$
    o Change required = $\epsilon$-fraction of the input
Graph Property Testing

• It’s an active area of research by itself.
• Let $G = (V, E)$ with $n = |V|$ and $m = |E|$

• Input models:
  ➢ **Sparse**: Represented by adjacency lists
    • Query if $(v, i)$ to get the $i^{th}$ neighbor of $v$ in $O(1)$ time
    • We only use it for graphs with degrees bounded by $d$
    • $\epsilon$-far from satisfying $P$ if $\epsilon(dn)$ matrix entries must be changed to satisfy $P$
    • Change required = $\epsilon$-fraction of the input
  
  ➢ Generally, dense is *easier* than sparse
Testing Bipartiteness

• Dense model:
  ➢ Upper bound: $O(1/\varepsilon^2)$ (independent of $n$)
  ➢ Lower bound: $\Omega(1/\varepsilon^{1.5})$

• Sparse model (for constant $d$):
  ➢ Upper bound: $O\left(\sqrt{n} \cdot \text{poly}\left(\frac{\log n}{\varepsilon}\right)\right)$
  ➢ Lower bound: $\Omega(\sqrt{n})$
Testing Bipartiteness

• In the dense model:

• Algorithm [Goldreich, Goldwasser, Ron]
  - Pick a random subset of vertices $S$, $|S| = \Theta \left( \frac{\log^{1/2}}{\epsilon^2} \right)$
  - Output “bipartite” iff the induced subgraph is bipartite

• Analysis:
  - Easy: If the graph is bipartite, algorithm always accepts.
  - Claim: If the graph is $\epsilon$-far, it rejects w.p. at least $2/3$
  - Running time: trivially constant (i.e., independent of $n$)
Testing Bipartiteness

• Q: Why doesn’t this work for the sparse model?
  ➢ Take a line graph of \( n \) nodes. Throw \( \epsilon n \) additional edges.
  ➢ In the dense model, we don’t care about this instance because it’s not \( \epsilon \)-far (only \( \epsilon / n \)-far).
  ➢ In the sparse model, we care about it, and the previous algorithm will not work.
Testing Bipartiteness

• In the sparse model:

  • Algorithm [Goldreich, Ron]
  • Repeat $O(1/\epsilon)$ times:
    ➢ Pick a random vertex $v$
    ➢ Run $OddCycle(v)$, and if it finds an odd cycle, REJECT.
  • If no trial rejected, then ACCEPT.

• OddCycle:
  ➢ Performs $poly(\log n/\epsilon)$ random walks from $v$, each of length $poly(\log n/\epsilon)$.
  ➢ If a vertex is reachable by an even-length path and an odd-length prefix, an odd cycle is detected.
Limitations of Sublinear Time

• The problems we saw are rather exceptions.
• For most problems, there is not much you can do in sublinear time.
• For instance, these problems require $\Omega(n^2)$ time:
  ➢ Estimating $\min_{i,j} d_{i,j}$ in a metric space $d$.
    o Contrast this with the sublinear algorithm we saw for estimating $\max_{i,j} d_{i,j}$ (diameter)
  ➢ Estimating the cost of the minimum-cost matching
  ➢ Estimating the cost of $k$-median for $k = \Omega(n)$
  ➢ ...

Sublinear Space Algorithms

• An important topic in complexity theory

• Fundamental unsolved questions:
  ➢ Is $\text{NSPACE}(S) = \text{DSPACE}(S)$ for $S \geq \log n$?
  ➢ Is $P = L$? ($L = \text{DSPACE}(\log n)$, and we know $L \subseteq P$)
  ➢ What’s the relation between $P$ and $\text{polyL} = \text{DSPACE}\left((\log n)^{O(1)}\right)$?
    o We know $P \neq \text{polyL}$, but don’t know if $P \subset \text{polyL}$, $\text{polyL} \subset P$, or if neither is contained in the other.

• Savitch’s theorem:
  ➢ $\text{DSPACE}(S) \subseteq \text{NSPACE}(S) \subseteq \text{DSPACE}(S^2)$
USTCON vs STCON

• USTCON (resp. STCON) is the problem of checking if a given source node has a path to a given target node in an undirected (resp. directed) graph.
  - USTCON $\in \text{RSPACE}(\log n)$ was shown in 1979 through a random-walk based algorithm
  - After much effort, Reingold [2008] finally showed that USTCON $\in \text{DSPACE}(\log n)$

• Open questions:
  - Is STCON in RSPACE($\log n$), or maybe even in RSPACE($\log n$)?
  - What about $o(\log^2 n)$ instead of $\log n$ space?
  - Is RSPACE($S$) = DSPACE($S$)?
Streaming Algorithms

• Input data comes as a stream $a_1, \ldots, a_m$, where, say, each $a_i \in \{1, \ldots, n\}$.
  ➢ The stream is typically too large to fit in the memory.
  ➢ We want to use only $S(m, n)$ memory for sublinear $S$.
    o We can measure this in terms of the number of integers stored, or the number of actual bits stored (might be $\log n$ times).
  ➢ It is also desired that we do not take too much processing time per element of the stream.
    o $O(1)$ is idea, but $O(\log(m + n))$ might be okay!
  ➢ If we don’t know $m$ in advance, this can often act as an online algorithm.
Streaming Algorithms

- Input data comes as a stream $a_1, \ldots, a_m$, where, say, each $a_i \in \{1, \ldots, n\}$.
  - Most questions are about some statistic of the stream.
  - E.g., “how many distinct elements does it have?” or “count the #times the most frequent element appears”
  - Once again, we will often approximate the answer.
  - Most algorithms process the stream in one pass, but sometimes you can achieve more if you can do two or more passes.
Missing Element Problem

• **Problem:** Given a stream \( \{a_1, \ldots, a_{n-1}\} \), where each element is a distinct integer from \( \{1, \ldots, n\} \), find the unique missing element.

• An \( n \)-bit algorithm is obvious
  - Keep a bit for each integer.
  - At the end, spend \( O(n) \) time to search for the 0 bit.

• We can do \( O(\log n) \) bits by maintaining the sum.
  - Missing element = \( \frac{n(n+1)}{2} - SUM \)

• Deterministic + exact.
Missing Elements Problem

• **Problem:** Given a stream \( \{a_1, \ldots, a_{n-k}\} \), where each element is a distinct integer from \( \{1, \ldots, n\} \), find all \( k \) missing elements.

• The previous algorithm can be generalized:
  - Instead of just computing the sum, compute power-sums.
  - \( \{S_j\}_{1 \leq j \leq k} \) where \( S_j = \sum_{i=1}^{n-k} (a_i)^j \)
  - At the end, we have \( k \) equations, and \( k \) unknowns.
  - This uses \( O(k^2 \log n) \) space.
  - Computationally expensive to solve the equations
    - Using Newton’s identities followed by finding roots of a polynomial
Missing Elements Problem

• We can design much more efficient algorithms if we use randomization.
  ➢ There is a streaming algorithm with space and time/item that is $O(k \log k \log n)$.
  ➢ It can also be shown that $\Omega\left(k \log \left(\frac{n}{k}\right)\right)$ space is necessary.
Frequency Moments

• Another classic problem is that of computing frequency moments.

➢ Let $A = a_1, ..., a_m$ be a data stream with $a_i \in \{1, ..., n\}$.
➢ Let $m_i$ denote the number of occurrences of value $i$.
➢ Then for $k \geq 0$, the $k^{th}$ frequency moment is defined as

$$F_k = \sum_{i \in [n]} (m_i)^k$$

➢ $F_0 = \#$ distinct elements
➢ $F_1 = m$
➢ $F_2 = \text{Gini’s homogeneity index}$
   • The greater the value of $F_2$, the greater the homogeneity in $A$
Frequency Moments

• Goal: Given $\epsilon, \delta$, find $F'_k$ s.t.
  \[ \Pr[|F_k - F'_k| > \epsilon F_k] \leq \delta \]

• Seminal paper by Alon, Matias, Szegedy [AMS’99]
  ➢ $k = 0$: For every $c > 2$, $O(\log n)$ space algorithm s.t.
    \[ \Pr[(1/c)F_0 \leq F'_0 \leq cF_0] \geq 1 - 2/c \]
  ➢ $k = 2$: $O \left( (\log n + \log m) \log(1/\delta)/\epsilon \right) = \tilde{O}(1)$ space
  ➢ $k \geq 3$: $\tilde{O} \left( m^{1-1/k} \text{poly}(1/\epsilon) \text{polylog}(m,n,1/\delta) \right)$ space
  ➢ $k > 5$: Lower bound of $\Omega(m^{1-5/k})$
Frequency Moments

• Goal: Given $\epsilon, \delta$, find $F'_k$ s.t.
  \[
  \Pr[|F_k - F'_k| > \epsilon F_k] \leq \delta
  \]

• Seminal paper by Alon, Matias, Szegedy [AMS’99]
  
  ➢ $k = 0$: For every $c > 2$, $O(\log n)$ space algorithm s.t.
    \[
    \Pr[(1/c)F_0 \leq F'_0 \leq cF_0] \geq 1 - 2/c
    \]

  ➢ Exactly counting $F_0$ requires $\Omega(n)$ space:
    
    o Once the stream is processed, the algorithm acts as a membership tester. On new element $x$, the count increases by 1 iff $x$ was not part of the stream.
    
    o Algorithm must have enough memory to distinguish between all possible $2^n$ states
Frequency Moments

• Goal: Given $\epsilon, \delta$, find $F'_k$ s.t.

$$\Pr[|F_k - F'_k| > \epsilon F_k] \leq \delta$$

• Seminal paper by Alon, Matias, Szegedy [AMS’99]

  - $k = 0$: For every $c > 2$, $O(\log n)$ space algorithm s.t.
    $$\Pr[(\frac{1}{c})F_0 \leq F'_0 \leq cF_0] \geq 1 - \frac{2}{c}$$

  - State-of-the-art is “HyperLogLog Algorithm”
    - Uses hash functions
    - Widely used, theoretically near-optimal, practically quite fast
    - Uses $O(\epsilon^{-2} \log \log n + \log n)$ space
    - It can estimate $> 10^9$ distinct elements with 98% accuracy using only 1.5kB memory!
Frequency Moments

• Goal: Given $\epsilon, \delta$, find $F'_k$ s.t.
  $$\Pr[|F_k - F'_k| > \epsilon F_k] \leq \delta$$

• Seminal paper by Alon, Matias, Szegedy [AMS’99]
  - $k > 2$: The $\Omega(m^{1-5/k})$ bound was improved to $\Omega(m^{1-2/k})$ by Bar Yossef et al.
    - Their bound also works for real-valued $k$.
  - Indyk and Woodruff [2005] gave an algorithm that works for real-valued $k > 2$ with a matching upper bound of $\tilde{O}(m^{1-2/k})$. 
AMS $F_k$ Algorithm

• The basic idea is to define a random variable $Y$ whose expected value is close to $F_k$ and variance is sufficiently small such that it can be calculated under the space constraint.

• We will present the AMS algorithm for computing $F_k$, and sketch the proof for $k \geq 3$ as well as the improved proof for $k = 2$. 
AMS $F_k$ Algorithm

• Algorithm:
  ➢ Let $s_1 = 8\varepsilon^{-2}k m^{1-1/k}$ and $s_2 = 2 \log^{1/\delta}$.
  ➢ Let $Y = \text{median}(Y_1, ..., Y_{s_2})$, where
  ➢ $Y_i = \text{mean}(X_{i,1}, ..., X_{i,s_1})$, where
    o $X_{i,j}$ are i.i.d. random variables that are calculated as follows:
    o For each $X_{i,j}$, choose a random $p \in [1, ..., m]$ in advance.
    o When $a_p$ arrives, note down this value.
    o In the remaining stream, maintain $r = |\{q | q \geq p \text{ and } a_q = a_p\}|$.
    o $X_{i,j} = m(r^k - (r-1)^k)$.

• Space:
  ➢ For $s_1 \cdot s_2$ variables $X$, log $n$ space to store $a_p$, log $m$ space to store $r$.

• Note: This assumes we know $m$. But it can be estimated as the stream unfolds.
AMS $F_k$ Algorithm

• We want to show: $E[X] = F_k$, and $Var[X]$ is small.

• $E[X] = E[m(r^k - (r - 1)^k)]$
  - The $m$ different choices of $p \in [m]$ have probability $1/m$.
  - Thus, $E[X]$ is just the sum of $r^k - (r - 1)^k$ across all choices of $p$.
  - For each distinct value $i$, there will be $m_i$ terms:
    $$(m_i^k - (m_i - 1)^k) + ((m_i - 1)^k - (m_i - 2)^k) + \cdots + (1^k - 0^k) = (m_i)^k$$
  - Thus, the overall sum is $F_k = \sum_i (m_i)^k$.

• Thus, $E[Y] = E[X] = F_k$
AMS $F_k$ Algorithm

• To show: $\Pr[|Y_i - F_k| > \epsilon F_k] \leq \frac{1}{8}$
  ➢ Median over $2 \log \frac{1}{\delta}$ many $Y_i$ will do the rest.

• Chebyshev’s inequality:
  ➢ $\Pr[|Y_i - E[Y_i]| > \epsilon E[Y_i]] \leq \frac{\text{Var}[Y]}{\epsilon^2 (E[Y])^2}$
  ➢ $\text{Var}[Y_i] \leq \frac{\text{Var}[X]}{s_1} \leq \frac{E[X^2]}{s_1}$, and $E[Y] = E[X] = F_k$.
  ➢ Thus, probability bound is:
    $$\frac{E[X^2]}{s_1 \epsilon^2 (F_k)^2} \leq \frac{E[X^2]}{8\epsilon^{-2} km^{1-1/k} \epsilon^2 (F_k)^2}$$
  ➢ To show that this is at most $1/8$, we want to show:
    $$E[X^2] \leq km^{1-1/k} (F_k)^2$$
  ➢ Show that: $E[X^2] \leq kF_1F_{2k-1}$, and $F_1F_{2k-1} \leq m^{1-\frac{1}{k}} (F_k)^2$
Sketch of $F_2$ improvement

• They retain $s_2 = 2 \log 1/\delta$, but decrease $s_1$ to just a constant $16/\varepsilon^2$.
  
  ➢ The idea is that $X$ will not maintain a count for each value separately, but rather an aggregate.
  
  ➢ $Z = \sum_{t=1}^{n} b_t m_t$, then $X = Z^2$
  
  ➢ The vector $(b_1, ..., b_n) \in \{-1,1\}^n$ is chosen at random as follows:
    o Let $V = \{v_1, ..., v_h\}$ be $O(n^2)$ “four-wise independent” vectors
    o Each $v_p = (v_{p,1}, ..., v_{p,n}) \in \{-1,1\}^n$
    o Choose $p \in \{1, ..., h\}$ at random, and set $(b_1, ..., b_n) = v_p$. 
Majority Element

• Input: Stream $A = a_1, \ldots, a_m$, where $a_i \in [n]$
• Q: Is there a value $i$ that appears more than $m/2$ times?

• Algorithm:
  ➢ Store candidate $a^*$, and a counter $c$ (initially $c = 0$).
  ➢ For $i = 1 \ldots m$
    o If $c = 0$: Set $a^* = a_i$, and $c = 1$.
    o Else:
      • If $a^* = a_i$, $c \leftarrow c + 1$
      • If $a^* \neq a_i$, $c \leftarrow c - 1$
Majority Element

- **Space:** Clearly $O(\log m + \log n)$ bits
- **Claim:** If there exists a value $v$ that appears more than $m/2$ times, then $a^* = v$ at the end.
- **Proof:**
  - Take an occurrence of $v$ (say $a_i$), and let’s pair it up:
    - If it decreases the counter, pair up with the unique element $a_j$ ($j < i$) that contributed the 1 we just decreased.
    - If it increases the counter:
      - If the added 1 is never taken back, QED!
      - If it is decreased by $a_j$ ($j > i$), pair up with that.
  - Because at least occurrence of $v$ is not paired, the “never taken back” case happens at least once.
Majority Element

• **Space:** Clearly $O(\log m + \log n)$ bits

• **Claim:** If there exists a value $v$ that appears more than $m/2$ times, then $a^* = v$ at the end.

• **A simpler proof:**
  - At any step, let $c' = c$ if $a^* = v$, and $c' = -c$ otherwise.
  - Every occurrence of $v$ must increase $c'$ by 1.
  - Every occurrence of a value other than $v$ either increases or decreases $c'$ by 1.
  - Majority $\Rightarrow$ more increments than decrements in $c'$.
  - Thus, a positive value at the end!
Majority Element

• **Note 1:** When a majority element does not exist, the algorithm doesn’t necessarily find the mode.

• **Note 2:** If a majority element exists, it correctly finds that element. However, if there is no majority element, the algorithm does not detect that and still returns a value.
  - It can be trivially checked if the returned value is indeed a majority element if a second pass over the stream is allowed.
  - Surprisingly, we can prove that this cannot be done in 1-pass. (Next lecture!)