Lecture 22: Sub-linear Time Algorithms
Today’s goal

• Motivation and models
• Classical approximation problems
  – Diameter of a point set
• Property testing: approximation for decision problems
  – “Sortedness” of a list
  – Connectedness of a graph
How can we understand?
Vast data

• Impossible to access all of it

• Potentially accessible data is too enormous to be viewed by a single individual

• Once accessed, data can change
Connected world phenomenon

- each “node” is a person
- “edge” between people that know each other
- Is the underlying graph connected?
Does earth have the connected world property?

• How can we know?
  – data collection problem is immense
  – unknown groups of people found on earth
  – births/deaths
The Gold Standard in 6.046

- Linear time algorithms!
- Are they adequate?
What can we hope to do without viewing most of the data?

• Can’t answer “for all” or “exactly” type statements:
  – Exactly how many individuals on earth are left-handed?
  – Are all individuals connected?

• Maybe can answer?
  – approximately how many individuals on earth are left-handed?
  – is there a large group of connected individuals?
What can we hope to do without viewing most of the data?

• Change our goals?
  – for most interesting problems: algorithm must give approximate answer

• we know we can answer some questions...
  – e.g., sampling to approximate average, median values
What types of approximation?

• “Classical” approximation for optimization problems: output is number that is close to value of the optimal solution for given input. (not enough time to construct a solution)

• Property testing for decision problems: output is correct answer for given input, or at least for some other input “close” to it.
I. Classical Approximation Problems
First:

• A very simple example –
  – Deterministic
  – Approximate answer
  – And (of course).... Sub-linear time!
Approximate the diameter of a point set

- Given: \( m \) points, described by a distance matrix \( D \), s.t.
  - \( D_{ij} \) is the distance from \( i \) to \( j \).
  - \( D \) satisfies triangle inequality and symmetry.
    (note: input size \( n = m^2 \))
- Let \( i, j \) be indices that maximize \( D_{ij} \) then \( D_{ij} \) is the diameter.
- Output: \( k, l \) such that \( D_{kl} \geq D_{ij}/2 \)
Algorithm

- Algorithm:
  - Pick $k$ arbitrarily
  - Pick $l$ to maximize $D_{kl}$
  - Output $D_{kl}$

- Why does it work?
  
  \[
  D_{ij} \leq D_{ik} + D_{kj} \quad \text{(triangle inequality)}
  \]
  
  \[
  \leq D_{kl} + D_{kl} \quad \text{(choice of $l$ + symmetry of $D$)}
  \]
  
  \[
  \leq 2D_{kl}
  \]

- Running time? $O(m) = O(n^{1/2})$
II. Property testing
Main Goal:

- Quickly distinguish inputs that have specific property from those that are far from having the property

**Benefits:**
- Natural question
- Just as good when data constantly changing
- Fast sanity check to rule out very "bad" inputs (i.e., restaurant bills) or to decide when expensive processing is worthwhile
Property Testing

• Properties of any object, e.g.,
  – Functions
  – Graphs
  – Strings
  – Matrices
  – Codewords

• Model must specify
  – representation of object and allowable queries
  – notion of close/far, e.g.,
    • number of bits/words that need to be changed
    • edit distance
A simple property tester
Sortedness of a sequence

• Given: list \(y_1 y_2 \ldots y_n\)
• Question: is the list sorted?

• Clearly requires \(n\) steps – must look at each \(y_i\)
Sortedness of a sequence

- Given: list \( y_1 y_2 \ldots y_n \)

- Question: can we quickly test if the list close to sorted?
What do we mean by ``quick’’?

• query complexity measured in terms of list size \( n \)

• Our goal (if possible):
  – Very small compared to \( n \), will go for \( \text{clog } n \)
What do we mean by “close”? 

Definition: a list of size $n$ is $\varepsilon$-close to sorted if can delete at most $\varepsilon n$ values to make it sorted. Otherwise, $\varepsilon$-far.

($\varepsilon$ is given as input, e.g., $\varepsilon=1/10$)

Sorted:  1  2  4  5  7  11  14  19  20  21  23  38  39  45
Close:    1  4  2  5  7  11  14  19  20  39  23  21  38  45
          1  4  5  7  11  14  19  20         23        38  45
Far:       45 39 23  1  38 4  5  21  20  19  2  7  11  14
            1         4    5                              7  11  14
Requirements for algorithm:

- Pass sorted lists
- Fail lists that are $\epsilon$-far.
  - Equivalently: if list likely to pass test, can change at most $\epsilon$ fraction of list to make it sorted
  
  Probability of success $> \frac{3}{4}$
  
  (can boost it arbitrarily high by repeating several times and outputting “fail” if ever see a “fail”, “pass” otherwise)

- Can test in $O(1/\epsilon \log n)$ time
  
  (and can’t do any better!)
An attempt:

• Proposed algorithm:
  – Pick random $i$ and test that $y_i \leq y_{i+1}$

• Bad input type:
  – $1,2,3,4,5,\ldots, n/4$, $1,2,\ldots, n/4$, $1,2,\ldots,n/4$, $1,2,\ldots,n/4$
  – Difficult for this algorithm to find “breakpoint”
  – But other tests work well…
A second attempt:

- Proposed algorithm:
  - Pick random $i<j$ and test that $y_i \leq y_j$

- Bad input type:
  - $n/4$ groups of 4 decreasing elements
    - $4, 3, 2, 1, 8, 7, 6, 5, 12, 11, 10, 9..., 4k, 4k-1, 4k-2, 4k-3,...$
  - Largest monotone sequence is $n/4$
  - must pick $i,j$ in same group to see problem
  - need $\Omega(n^{1/2})$ samples
A minor simplification:

• Assume list is distinct (i.e. \( x_i \neq x_j \))

• Claim: this is not really easier
  – Why?
    Can “virtually” append \( i \) to each \( x_i \)
    \( x_1, x_2, \ldots, x_n \) \( \rightarrow \) \((x_1,1), (x_2,2), \ldots, (x_n,n)\)
    e.g., \( 1,1,2,6,6 \) \( \rightarrow \) \((1,1),(1,2),(2,3),(6,4),(6,5)\)
  Breaks ties without changing order
A test that works

• The test:

Test $O(1/\varepsilon)$ times:
  • Pick random $i$
  • Look at value of $y_i$
  • Do binary search for $y_i$
  • Does the binary search find any inconsistencies? If yes, FAIL
  • Do we end up at location $i$? If not FAIL
    – Pass if never failed

• Running time: $O(\varepsilon^{-1} \log n)$ time
• Why does this work?
Behavior of the test:

• Define index $i$ to be good if binary search for $y_i$ successful

• $O(1/\varepsilon \log n)$ time test (restated):
  – pick $O(1/\varepsilon)$ $i$’s and pass if they are all good

• Correctness:
  – If list is sorted, then all $i$’s are good (uses distinctness)
    • So test always passes
  – If list likely to pass test,
    • Then at least $(1-\varepsilon)n$ $i$’s are good.
    • Main observation: good elements form increasing sequence
      – Proof: for $i<j$ both good need to show $y_i < y_j$
        • let $k =$ least common ancestor of $i,j$
        • Search for $i$ went left of $k$ and search for $j$ went right of $k$ $\rightarrow$$y_i < y_k < y_j$
      • Thus list is $\varepsilon$-close to monotone (delete $< \varepsilon n$ bad elements)
Testing connectedness of a graph

• Given graph G
  – n vertices
  – Max degree d
  – Adjacency list representation

• Is G connected?
Connected world phenomenon

• Is the underlying graph close to connected?
Close to connected

• Def: $G$ is $\epsilon$—*close* to connected if can add $< \epsilon dn$ edges and transform it to connected
  – Today: ok to violate max deg $d$ requirement
Property tester:

• Input: $\epsilon$ and G

• Output:
  – If G connected, output “PASS”
  – If G not $\epsilon$-close to connected, output “FAIL” with probability $\geq 3/4$

  – (note: if G not connected, but is close, then ok to output either “PASS” or “FAIL”)
Idea:

• If G far from connected, lots of nodes must be in small components!
• More specifically...
  – Will show that if G far from connected
  – Then must have many connected components
  – So many components must be small
  – And there must be many nodes in small components
Algorithm:

• Do $O\left(\frac{1}{\epsilon d}\right)$ times:
  – Pick random node $s$, and run BFS from $s$ until:
    • $\geq \frac{2}{\epsilon d}$ distinct nodes seen
    • OR see that $s$ is component of size $< \frac{2}{\epsilon d}$ nodes, in which case output “FAIL” and halt

• If reach this point, output “PASS”

Runtime: $O\left(\frac{1}{\epsilon d}\right)$ loops, each does $O\left(\frac{1}{\epsilon d}\right)$ steps of BFS, using $O(d)$ time per step – total is $O\left(\frac{1}{\epsilon^2 d}\right)$
Behavior

- Lemma 1: If $G \in\epsilon$-far from connected, then has $\geq \epsilon dn$ components.
- Lemma 2: If $\geq \epsilon dn$ components then $\geq \epsilon dn/2$ components of size $< \frac{2}{\epsilon d}$.
- Observation: If $\geq \epsilon dn/2$ components of size $< \frac{2}{\epsilon d}$ then $\geq \epsilon dn/2$ nodes in components of size $< \frac{2}{\epsilon d}$.

These cause tester to FAIL!
Behavior

• Putting it together: If $G$ $\epsilon$-far from connected, then $\geq \epsilon d / 2$ fraction of nodes cause algorithm to fail!
  
  – So $\text{Prob}[\text{tester fails in one of } \frac{c}{\epsilon d} \text{ loops}]$ is

$$\geq 1 - \left(1 - \frac{\epsilon d}{2}\right)^{\frac{2c}{2\epsilon d}} \geq 1 - e^{\frac{c}{2}} \geq \frac{3}{4} \text{ (for big enough } c)$$
Lemma 1

If $G \varepsilon$-far from connected, then has $\geq \varepsilon dn$ components

Proof: if $<\varepsilon dn$ components, can add $<\varepsilon dn$ edges to connect
Lemma 2

If $\geq \epsilon dn$ components then $\geq \epsilon dn/2$ components of size $< \frac{2}{\epsilon d}$

(see notes for proof)
Observation:

If $\geq \epsilon dn/2$ components of size $< \frac{2}{\epsilon d}$ then $\geq \epsilon dn/2$ nodes in components of size $< \frac{2}{\epsilon d}$

Why? Each small component has at least one node.