James V. Talwar
CSE 202

Homework 1

10/6/2022
Section B00

1) Definitions:
A tree T is a fully connected acyclic graph with $n-1$ edges, where n is the number of nodes (or vertices) in the graph

Key Idea: Graph traversal: BFS/DFS (Breadth First Search/Depth First Search)

Claim: The diameter of a tree T must be between to nodes in the tree with a degree of 1.
- Proof: Contradiction
  Let the following statement be true: A largest path (or tree diameter here) can exist b/w a node of degree ≥ 2 + any other node in the tree. Note that
  It can easily be shown that this statement is FALSE by definition. Observe that for any degree ≥ 2 node u + its diameter terminus v the diameter is $\delta(u,v)$. However from u there must be a node of distance $d \geq 1$ not captured in $\delta(u,v)$. Thus if the diameter were $\delta(u,v)$ then the leaf node of distance d from u, not on the diameter would have distance $\delta(u,v)+d$! This violates the definition of diameter + thus the diameter must be between two degree 1 nodes.

Algorithm: Randomly select a degree 1 node l from the set of degree 1 nodes L. Note this assumes we have direct access to node degrees + can formulate such a set L. However in the event this is not the case we can randomly select a node from T as l. From l run either BFS or DFS to find the furthest node from l. Denote this farthest node as u. From u run BFS or DFS to find its furthest node (i.e., the largest path). Call this node v. The length of the path from $u \to v$ is the diameter of T. Note that the second graph traversal (i.e., from u) the lengths are computed + thus can be directly accessed at the end of the traversal.

Time Complexity: Note this algorithm requires 2 graph traversals: 1) $l \to u$ and 2) $u \to v$. BFS + DFS have a complexity of $O(\|V\| + \|E\|)$. Here $\|V\| = n$ + $\|E\| = n-1$ yielding a single traversal = $O(2n-1) \to O(n)$. We require 2 graph traversals which is a constant and thus still yields a time complexity of n for this algorithm.

Proof:

To prove this must show that U, found from l is a terminus of the tree. More formally that U is an endpoint of the tree diameter. If U is a terminus, then by definition the farthest node from U, V, will be the other terminus + thus the shortest path length between these two nodes will equate to the diameter.

Again employ contradiction. Let the following statement be true: From the randomly selected node l the farthest node from l is $l' \neq U, V$. Use the following notation $l \to l'$ is the longest shortest path length between l and l'. We can formalize this as

$$l \to l' \geq l \to U + \underbrace{U \to V}_{\text{tree diameter}}$$

Since l lies on the path between l' and U we can subtract $l \to U$ from both sides and get $l' \to U \geq \underbrace{U \to V}_{\text{tree diameter}}$.

This is invalid by definition + thus l' must $= U$, the first found diameter terminus. Intuitively the reasoning follows that if a longer path existed $l' \to U$ this would exceed the diameter, and therefore l' must $=$ a diameter endpoint.

2) Definition(s):

Given a $n \times n$ matrix M where each row and column is sorted in ascending order. Denote $M[i,j]$ as the element in M at the i^{th} row and j^{th} column in a zero-indexed manner.

Key Idea: The row & column sorted nature of M forms a conceptual graph. That is we can treat either the top right entry of M (or the bottom left entry) and walk-through the matrix until the desired element is found or no valid moves are left (i.e. proceeding would exceed the bounds of M. For the purposes of the algorithm select the top right element of the matrix as the start point (i.e., start = $M[0, n-1]$

Algorithm: Assign a matrix search location s to the starting point $\rightarrow s = M[0, n-1]$ which is the top right corner of the matrix. Denote the element searching for as e. If $e > s$ (which is the value of M) proceed along the rows. Specifically move down 1 row which by definition is greater than s. If e is $< s$ proceed left along the columns. Specifically move 1 column to the left (while keeping the row the same). By definition we know this left entry $< s$. If $s == e$ return true (i.e., the element e is in M) & the row, column location if so desired. In the event of the previous two cases (i.e., $e < s$ or $e > s$) update s to the left or downward traversal (as previously stated) & update the index accordingly. Specifically if current $s = M[i,j]$ & $e < s$ update s to $s = M[i, j-1]$; if current $s = M[i,j]$ & $e > s$ update s to $s = M[i+1, j]$. Continue in this manner until there are no valid increments/decrements left (i.e., exceed the bounds of M). In this case return False as e is not in M.

Time Complexity: Note that the traversal is only forwards. Specifically in the above algorithm we only move along the rows from $0 \rightarrow m-1$ and along the columns $n-1 \rightarrow 0$. Thus at most this approach takes $n+m$ steps yielding a time complexity of $n+m$.

Proof: Induction

At step 1 $s = M[0, n-1]$ as specified by the algorithm. By matrix definitions know all elements $<$ are left & all elements $>$ are below.

At step t $s = M[i,j]$ at step $t+1$ $s = M[i+1, j]$ or $s = M[i, j-1]$

• Note here at step t the bounds of the matrix at i,j form a submatrix in M with the same ordered properties as M. Thus each progression (time step) reduces the Matrix search space.

Specifically note that if $s = M[i,j]$ all entries from $i \to n-1$ (along rows) are greather than s and all entries along the columns $j \to 0$ are $< s$ + thus can step in the direction of the algorithms check against.

3) Definitions and Problem Formulation:

A 132 pattern in a sequence of n #s is a subsequence of #s such that $i<j<k$ and $a_i < a_k < a_j$

Key Idea: Find a sequence maximum and perform a left-right split around the LAST incidence of the sequence maximum. Employ divide and conquer using left + right of this last max.

- This ensures that all elements to the right of this element is strictly less than

Algorithm: Scan through a sequence to find the maximum, updating the index maximum if a position value in S is $\geq$ current maximum.

- In this evaluation of the maximum + index we can also assess whether S (the sequence is sorted) by checking if the maximum (or maximum index) is changed at each step. If so can return False → no 132 pattern exists
- If the last maximum occurs at index z scan through all elements from $S_0 \rightarrow S_{z-1}$ and find the minimum. Similarly scan through all elements from $S_{z+1} \rightarrow S_{n-1}$ and find the maximum. Compare the minimum of left ($S_0 \rightarrow S_{z-1}$) to the maximum of right ($S_{z+1} \rightarrow S_n$). If minimum of left < max(right) return True that a 132 pattern exists. Otherwise (including the case when z occurs at n-1 + right is empty), repeat the same process assigning $S = S_{LEFT}$ until either a 132 pattern has been found, or there are < 3 elements in S, in which case return False.

Proof of Correctness: Contradiction

- A 132 pattern exists if $\min(S_{LEFT}) > \overset{LAST}{\max}(S_{RIGHT})$
- Assume a 132 pattern does not exist in splitting to $S_{LEFT} + S_{RIGHT}$.
- This violates a 132 pattern by definition. Specifically i is in left < j + right = k > j by index. Since $S_j >$ all in right the maximum in right is < global max. This means if some case exists in $S_{LEFT} < \max(S_{RIGHT})$ have a 123 pattern.

Time Complexity:

Average Case → Time complexity = n

Worst Case → Time complexity = n^2

The justification for these complexities is detailed on the subsequent page.

In the worst case while we check for a sorted list we can still have an alternating list (e.g. [99, 98, 99...]). This will require $c \sum_{k=0}^{n-1} n-k = (n+1)\frac{n}{2} \cdot c$ complexity which is n^2. Note though that this is still an improvement over checking all possible triplets → n choose 3 has an order of n^3 → $[n \cdot (n-1)(n-2)]/3!$

However the average case is far better. Note that a maximum sampled uniformly from all n integers will fall between index $\frac{n}{4}$ & $\frac{3n}{4}$ half the time and thus give a complexity $T(\frac{3n}{4}) + O(n)$ ⇒ which yields a time complexity on the order of n

· Note $T(\frac{3n}{4})$ comes from the fact that half the time we are reducing the search space by at least a quarter.

4) Definitions:

The problem states that little endian places the least significant digit at the lowest array index. Define the lowest array index to be 0-indexed (classically the left most entry of an array A). Thus position $i=n-1$ in array A would be the most significant digit.

Key Idea: Naive computation of the valuation of the digits in base b_1 requires an iterative (across n) computation of b_1 multiplied with the current running total) + value of current index digit (Note this proceeds from right to left). This can be used to give a b_2 representation by converting b_1 + the value in array A to the value equivalent in b_2 (i.e. reframing the problem to be in terms of b_2). However this iterative process takes $O(n^2)$ + fails to leverage an implicit structure in the multiplications. Specifically that we can divide and conquer array A to be:

$$A=\boxed{A_{Left}}\ \boxed{A_{right}} \rightarrow A = A_{Left} + b_1^{n/2} A_{Right} \leftarrow \text{this is in little endian format}$$

Here we partition the problem into 2 parts of half the size:

$$T(n) = 2T\left(\frac{n}{2}\right) + \underbrace{O(n\log n)}_{\text{Where this accounts for the FFT multiplications}}$$

Or we can partition the problem into 3 parts of half the size using Gauss' trick:

$$xy = (x_L + b_1^{n/2} x_R)(y_L + b_1^{n/2} y_R) = b_2^n x_R y_R + b_1^{n/2}\left[(x_L+x_R)(y_L+y_R) - x_L y_L - x_R y_R\right] + x_L y_L$$

Which gives a time recurrence relation of $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$

Time Complexity:

FFT + Divide and Conquer: There are $n\log n$ operations occurring at all $\log n$ levels of the recurrence tree. This gives a time complexity of $n\log^2 n$

Gauss Trick + Divide and Conquer: There are 3 subproblems at each level of the recursion tree each of size $n/2$ which yields a time complexity of $n^{\log_2 3}$.

Given the better time efficiency of the FFT approach, the algorithm is outlined on the next page

Algorithm: Given an array A in little endian format of b_1 coefficients (i.e these #s can be conceptualized as coefficients of a polynomial where $b_1 = x$ in the classical sense) partition A into left and right halves. Represent A as $A = A_{LEFT} + b_1^{n/2} A_{RIGHT}$. Note that b_1 & A should be preprocessed such that the value equivalency is in b_2 to which we are trying to convert (conceptually this can be interpreted as if $b_1 > b_2$ b_1 must be rewritten to a valid b_2 representation).

Having preprocessed, then partitioned, recurse on both A_{LEFT} & A_{RIGHT} updating the value of n at each layer deeper in the recursion tree as $\frac{n}{2}$. Evaluate the multiplications using the FFT from the preprocessed b_2 representation. At the base case return the multiplication value $(b_1^i \cdot \underbrace{val(a_i)}_{\text{value of } a_i \text{ in base } b_2})$. Merging can be done through summation in this instance and gives the desired b_2 representation (in little endian format as desired).

Proof of Correctness: Induction

base case: A is empty of a single element. If A is empty return ϕ as an empty array in b_1 is an empty array in b_2. If A is a single element multiply it (via FFT) with its corresponding b_1^k (where k is the power of the element in A bounded b/w $0 \to n-1$).

Induction:

At recursive step t assume that base conversion has completed $b_1 \to b_2$. At recursive step $t+1$ we note that each subpartition is in and of itself a valid conversion from $b_1 \to b_2$ & that the sum of these $b_1^k A_{RIGHT} + A_{LEFT}$ equates to the value at the layer above the tree.