Algorithms: CSE 202 — Homework 1 Solutions

Problem 1: Diameter of a tree (CLRS)

The *diameter* of a tree $T = (V, E)$ is given by

$$\max_{u,v \in V} \delta(u, v)$$

where $\delta(u, v)$ is the shortest path length between the vertices u and v. That is, the diameter is the largest of all shortest-path length in the tree. Give an efficient algorithm to compute the diameter of a tree, and analyze the running time of your algorithm.

Solution: Diameter of a tree (CLRS)

First recall that a tree has a unique path between any two vertices. Therefore, the longest of all shortest paths between any two nodes must either run through the root of the tree or be entirely contained in a subtree rooted at one of the root's children. This leads to an intuitive recursive formulation. At each level of the recursion, we keep track of the longest path contained in that subtree and the longest path from the root of the subtree to any other node in the subtree. When we merge our solutions, we compare the maximum of the subtrees contained in each child's subtree with the maximum path that can be formed by combining one or two children's paths with the edges to the root node. This will output the longest path due to our observations.

Complexity: Because of the recursive formulation of this algorithm, we have only a constant amount of operation per node. Hence, this is a linear time algorithm.

In more detail, if it is a full binary tree then we have the recurrence relation $T(n) = 2T(n/2) + O(1)$, where n is the number of nodes, which give a runtime of $O(|V|)$.

In the more general case we can see that our recurrence tree is exactly the tree we are given. We visit each node once and do a constant amount of work at the node giving $O(|V|)$ total work.

Proof of Correctness: Consider the case when the tree consists of only one node. In this case, we have $l = 0$ given by our algorithm which is correct. When we have more than one nodes, we root it arbitrarily and call the *MaxShortestPaths(.)* function. Assume that this function works for a node at height i. We will show that this function will also work for a node at height $i + 1$.
Consider a node V_1 at height $i + 1$. By the assumption in the induction step, we know correctly the longest path contained in each subtree and calculate the longest leaf-to-root path in each subtree. Since by comparing the following three quantities - lengths of the longest path contained in each subtree and sum of the length of the longest leaf-to-root path in each subtree gives us the diameter of the tree rooted at V_1, we are done.
Inductively, we can deduce that this algorithm will work for any tree T.

---

1 **Algorithm:** Tree Diameter
Input: $T = (V, E)$
Output: Diameter of T
2 $(\ell_1, \ell_2) =$ **MaxShortestPaths**(T)
3 **return** ℓ_1

```
Algorithm: MaxShortestPaths
Input: T = (V, E)
Output: Length of longest path in T, Length of longest path to root of T
if |V| = 1 then
    return (0, 0)
end
else
    u ← Root(T)
    maxsubpath ← 0
    maxrootpath ← 0
    nextrootpath ← 0
    for v ∈ u.children do
        (ℓ1, ℓ2) = MaxShortestPaths(subtree(v))
        if ℓ1 > maxsubpath then
            maxsubpath ← ℓ1
        end
        if ℓ2 + w((u, v)) > maxrootpath then
            nextrootpath ← maxrootpath
            maxrootpath ← ℓ2 + w((u, v))
        end
        else if ℓ2 + w((u, v)) > nextrootpath then
            nextrootpath ← ℓ2 + w((u, v))
        end
    end
end
```

Problem 2: Sorted matrix search

Given an $m \times n$ matrix in which each row and column is sorted in ascending order, design an algorithm to find an element.

Solution: Sorted Matrix Search

Algorithm description:
Let $A_{i,j}$ be a 2-dimensional matrix where $0 \leq i \leq m-1$ and $0 \leq j \leq n-1$ such that every row and every column of A is sorted in increasing order. We search for an element x in A by following a walk in the matrix as follows:

- Let c be the element in the top right corner of the matrix.
- If $x = c$, we declare that x is found.
- If $x < c$, then we move one column to the left while remaining at the same row and search in the submatrix obtained by deleting the last column.
- If $x > c$, then we move one row down while remaining at the same column and search in the submatrix obtained by removing the first row of the matrix.
- This process is repeated until either the element is found or the matrix is an empty matrix without any rows and columns.

Proof of Correctness: If x is not in A it is clear that the algorithm is correct. If x is in A, then the correctness of the algorithm follows from this claim.

Claim 0.1. *Let c be the element in the top right corner of a matrix whose rows and columns are sorted in increasing order and x be an element in the matrix. If $x < c$, then x is in the submatrix obtained by deleting the last column. If $x > c$, then s is in the submatrix obtained by removing the first row of the matrix.*

We argue this claim by considering the two cases seperately. If $x < c$, then x is strictly less than every element in the last column of the matrix since the column is sorted in increasing order and c is its first element. Hence, it must be in the submatrix obtained by deleting the last column of the matrix.

Now suppose instead that $x > c$. Then x is greater than every element in the first row of the matrix since the row is sorted in increasing order and c is its last element. Hence, it must be in the submatrix obtained by deleting the first row of the matrix.

Pseudocode:

A = given matrix
x = element to be searched for
$i \leftarrow 0$
$j \leftarrow n - 1$
while ($i \leq m - 1$ and $j \geq 0$)
    if($A_{i,j} == x$)
        return true
    else if($A_{i,j} > x$)
        $j = j - 1$
    else
        $i = i + 1$
return false

Complexity: In this algorithm, at every step we eliminate either a row or a column or terminate if the target is reached. So, we look at a maximum of $m + n$ elements. So the time complexity is $O(m + n)$. Since, we do not use any extra space, the space complexity is $O(1)$.

Problem 3: Maximum overlap of two intervals

Design an algorithm that takes as input a list of intervals $[a_i, b_i]$ for $1 \leq i \leq n$ and outputs the length of the maximum overlap of two distinct intervals in the list.

Solution: Maximum overlap of two intervals

High-level description of the algorithm

Let (s_i, f_i) denote the i-th interval where s_i and f_i respectively are its start point and end points for $1 \leq i \leq n$. We sort intervals according to the increasing order of s_i. We process the sorted list of intervals from left to right while maintaining the following two quantities:

- the largest end point among all the intervals processed so far (which is initialized to f_1) and
- the largest overlap between any two distinct intervals from among the intervals processed so far (which is initialized to 0).

For each interval, we compute its overlap with the previous interval with the largest end point and update the largest overlap accordingly. We also update the largest end point by comparing it with the end point of the interval.

The key idea is that the overlap of the i-interval with any previous interval, say, the j-th interval where $s_j \leq s_i$, is given by $\max(0, \min(f_i, f_j) - s_i)$. To maximize the overlap it is sufficient to keep track of the previous interval which extends farthest to the right, that is, the largest f_j such that $s_j \leq s_i$.

Pseudocode

Algorithm 1: Maximum Overlap

Input : List of intervals $(s_1, f_1), (s_2, f_2), \ldots, (s_n, f_n)$
Output: Maximum overlap of any two intervals (s_i, f_i) and (s_j, f_j)

```
sort intervals by s_i
maxoverlap ← 0
maxf ← f_1
for i = 2 to n do
    overlap ← max(0, min(f_i, maxf) − s_i)
    maxoverlap ← max(overlap, maxoverlap)
    maxf ← max(f_i, maxf)
end for
return maxoverlap
```

Correctness Proof

We prove by induction on the number of iterations that *maxoverlap* computes the maximum overlap between any two distinct intervals and that *maxf* is the maximum end point of any interval from the set of intervals processed until the beginning of the iteration.

More precisely, we prove the following claim to establish the correctness of the algorithm.

Claim 0.2. *For $1 \leq i \leq n$, at the beginning of the iteration i of the loop,*

- *maxf* $= \max_{j=1}^{i} f_j$.
- *maxoverlap is the maximum overlap between any two distinct intervals from among the first i intervals.*

Note that the loop variable is one more than the iteration number.

Proof. For the base case, $i = 1$, we have *maxoverlap* equal to zero, which by definition is the maximum overlap when we have only one interval. We also have *maxf* $= f_1$, which is clearly the maximum value of the endpoints of the set of intervals under consideration.

Assume that the claim is true for all $1 \leq j \leq i$. At the beginning of iteration i, we have by induction hypothesis *maxf* $= \max_{j=1}^{i} f_j$. During iteration i, we compare the current value of *maxf* with f_{i+1} and update it accordingly so *maxf* holds $\max_{j=1}^{i+1} f_j$ at the beginning of iteration $i+1$, thus establishing the first part of the claim.

To establish the second part of the claim, we argue that the computed value of *overlap* during iteration i is the maximum overlap of interval $i+1$ with any earlier interval. Since *maxf* is the largest end point of any earlier interval, there is an interval $p_{j^*} = (s_{j^*}, f_{j^*})$ with $f_{j*} =$ *maxf* where $1 \leq j^* \leq i$. The length of the overlap between (s_{i+1}, f_{i+1}) and (s_{j^*}, f_{j^*}) is exactly $\max(0, \min(f_{i+1}, \textit{maxf}) - s_{i+1})$. Furthermore, for any other $j < i+1$ we have $f_j \leq f_{j^*}$. Hence $\min(f_{i+1}, \textit{maxf}) - s_{i+1} \geq \min(f_{i+1}, f_j) - s_{i+1}$, which completes the proof that *overlap* is the maximum overlap of (s_{i+1}, f_{i+1}) with any earlier interval.

By induction hypothesis, at the beginning of iteration i, *maxoverlap* is the maximum overlap of any two distinct intervals from among the first i intervals. Since the maximum overlap of the first $i+1$ intervals is the maximum of the maximum overlap of any two distinct intervals from among the first i intervals and the maximum overlap of the interval $i+1$ with the previous intervals, the update operation for *maxoverlap* during iteration i ensures that it has the claimed value at the beginning of the iteration $i+1$.

□

Runtime Analysis

This algorithm requires $O(n \lg n)$ time to sort the intervals using mergesort. Then it takes $O(n)$ time to scan the list of intervals. Therefore the total runtime is $O(n \lg n)$.

Solution: Maximum overlap of two intervals

Algorithm 2. In this section we sketch an alternative divide and conquer algorithm. First sort the list of starting points s_i. Now for the set S of intervals, if $|S| \leq 3$, simply compare all pairs and output the maximum overlap. Otherwise, split S evenly into two parts S_1 and S_2 such that the intervals in S_1 all have starting points before (or at the same time) as the starting point of any interval in S_2.

Observe that the largest overlap between an interval $(s_1, f_1) \in S_1$ and $(s_2, f_2) \in S_2$ must involve the interval (s_1, f_1) which has the largest ending point. We omit the proof of this fact here, but the proof is essentially the same as used in solution 1. Since the maximum overlap either involves two intervals from S_1, two intervals from S_2 or one interval each from S_1 and S_2, we can recurse on each of S_1 and S_2 separately. Afterward we find the interval (s_1, f_1) from S_1 that has the largest ending point and compare its overlap with each interval in S_2. Finally we return the maximum of the recursive calls and the largest overlap of (s_1, f_1) with an interval in S_2.

Correctness follows from our observation that we can merge the two subproblems by only comparing the interval with the largest endpoint from S_1 with all intervals in S_2.

Sorting takes $O(n \lg n)$ time using mergesort. The base cases take $O(1)$ time. During the merge of the recursion it takes $O(n)$ time to find the interval with the largest endpoint from S_1 and compare it to each interval in S_2. Therefore we have the recurrence relation $T(n) = 2T(\frac{n}{2}) + O(n)$ which gives a runtime of $O(n \lg n)$.

Problem 4: Toeplitz matrices

A *Toeplitz matrix* is an $n \times n$ matrix $A = (a_{ij})$ such that $a_{ij} = a_{i-1,j-1}$ for $i = 2, 3, \ldots, n$ and $j = 2, 3, \ldots, n$.

1. Is the sum of two Toeplitz matrices necessarily Toeplitz? What about the product?
2. Describe how to represent a Toeplitz matrix so that two $n \times n$ Toeplitz matrices can be added in $O(n)$ time.
3. Give an $O(n \lg n)$-time algorithm for multiplying an $n \times n$ Toeplitz matrix by a vector of length n. Use your representation from part (b).
4. Give an efficient algorithm for multiplying two $n \times n$ Toeplitz matrices. Analyze its running time.

Solution: Toeplitz matrices

Part 1

Yes, the sum of two Toeplitz matrices is also Toeplitz. Let matrix $C = A + B$, where $A = (a_{i,j})$ and $B = (b_{i,j})$ for $1 \leq i, j \leq n$ are two arbitrary $n \times n$ Toeplitz matrices. Let $C = (c_{i,j})$ for $1 \leq i, j \leq n$. We have for $2 \leq i, j \leq n$

$$\begin{aligned} c_{i,j} &= a_{i,j} + b_{i,j} \\ &= a_{i-1,j-1} + b_{i-1,j-1} \text{ since } A \text{ and } B \text{ are Toeplitz matrices} \\ &= c_{i-1,j-1} \end{aligned}$$

which proves that C is Toeplitz.

No, the product of two Toeplitz matrices is not always a Toeplitz matrix. Here is a counterexample :

$$\begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix} \times \begin{bmatrix} 1 & 3 \\ 4 & 1 \end{bmatrix} = \begin{bmatrix} 9 & 5 \\ 4 & 1 \end{bmatrix}$$

Part 2
A Toeplitz matrix can be represented by its first row and first column. Rest of the entries are determined by the entries of the first row and first column.

Since the first row (column) of the sum of two matrices A and B is the sum of the first rows (columns) of A and B, we can compute the representation of the sum of two Toeplitz matrices in linear time, given the representations of A and B.

Part 3
Idea :
Convolution of two vectors $a = (a_0, \ldots, a_{n-1})$ and $b = (b_0, \ldots, b_{n-1})$ is a vector $c = (c_0, \ldots, c_{2n-1})$ where c_i is given by

$$c_i = \sum_{k=0}^{i} a_k b_{i-k}$$

where we assume $a_j = b_j = 0$ for $j \geq n$.

Define $A(x) = \sum_{j=0}^{n-1} a_j x^j$, $B(x) = \sum_{j=0}^{n-1} b_j x^j$, and $C(x) = \sum_{j=0}^{2n-1} c_j x^j$. We know that $C(x) = A(x)B(x)$ can be computed in $O(n \log n)$ time using FFT. Consider the following Toeplitz matrix-vector multiplication problem.

$$\text{Input : } A = \begin{bmatrix} a_{n-1} & a_{n-2} & \ldots & a_0 \\ a_n & a_{n-1} & \ldots & a_1 \\ . & . & \ldots & . \\ . & . & \ldots & . \\ a_{2n-2} & a_{2n-3} & \ldots & a_{n-1} \end{bmatrix} \text{ and } x = (b_0, b_1, \ldots, b_{n-1})$$

$$\text{Output : } y = Ax^T$$

Using the structure of A we will show that y can be obtained from the convolution of two linear size vectors, thereby obtaining an $O(n \log n)$ algorithm for the Toeplitz matrix-vector product problem.
Algorithm :

Step 1: Construct $b = (b_0, b_1, \ldots, b_{n-1}, 0, 0, \ldots, 0)$ and $a = (a_0, a_1, \ldots, a_{2n-2})$ each of length $2n - 1$
Step 2: Compute the convolution c of vectors b and a.
Step 3: Output $y_i = c_{n-1+i}$ for $0 \leq i \leq n - 1$.

Correctness :
From the definitions for convolution and matrix multiplication, we get

$$c_i = \sum_{k=0}^{i} b_k a_{i-k} \text{ for } 0 \leq i \leq 2n-2 \text{ and}$$
$$y_i = \sum_{k=0}^{n-1} b_k a_{n-1+i-k} \text{ for } 0 \leq i \leq n-1$$

Using these two equations we will justify the last step of the algorithm. For $0 \leq i \leq n-1$, we have

$$\begin{aligned} c_{n-1+i} &= \sum_{k=0}^{n-1+i} b_k a_{n-1+i-k} \\ &= \sum_{k=0}^{n-1} b_k a_{n-1+i-k} \\ &= y_i \end{aligned}$$

Complexity :
Since we are computing the convolution of two vectors of size $2n-1$, the time required is $O(n \log n)$.

Part 4 →
Idea :
The product of two Toeplitz matrices is not necessarily a Toeplitz matrix. However, the product matrix has sufficient structure which enables us to compute it in $O(n^2)$ time.

Algorithm :
In the first two steps we will compute the first row and the first column of the product matrix by multiplying the corresponding rows and columns of the input matrices. In the last step we will compute the other entries incrementally using previously computed entries.

$$\text{Step 1: } c_{0,j} = \sum_{k=0}^{n-1} a_{0,k} b_{k,j} \text{ for } 0 \leq j \leq n-1$$
$$\text{Step 2: } c_{i,0} = \sum_{k=0}^{n-1} a_{i,k} a_{k,0} \text{ for } 1 \leq j \leq n-1$$
$$\text{Step 3: } c_{i,j} = c_{i-1,j-1} + a_{i,0} b_{0,j} - a_{i-1,n-1} b_{n-1,j-1} \text{ for } 1 \leq i, j \leq n-1$$

Correctness :
Since steps 1 and 2 simply follow the definition of matrix multiplication the only thing left is to prove the correctness of step 3.

For all $i > 0$ and $j > 0$:

$$\begin{aligned} c_{i,j} &= \sum_{k=0}^{n-1} a_{i,k} b_{k,j} \\ &= a_{i,0} b_{0,j} + \sum_{k=1}^{n-1} a_{i-1,k-1} b_{k-1,j-1} \\ &= a_{i,0} b_{0,j} + \sum_{k=0}^{n-2} a_{i-1,k} b_{k,j-1} \\ &= a_{i,0} b_{0,j} + \sum_{k=0}^{n-1} a_{i-1,k} b_{k,j-1} - a_{i-1,n-1} b_{n-1,j-1} \\ &= c_{i-1,j-1} + a_{i,0} b_{0,j} - a_{i-1,n-1} b_{n-1,j-1} \end{aligned}$$

The second equation follows from the fact that A and B are Toeplitz. The rest of the equations are simply derived by adjusting the indices involved in the summation.

Complexity :

Steps 1 and 2 takes $O(n^2)$ time. In step 3 we are computing $O(n^2)$ entries and each entry takes constant amount of computation. So, the total complexity of the algorithm is $O(n^2)$.