Algorithms: CSE 202 — Homework 1 Solutions

Problem 1: Maximum weight subtree

The maximum weight subtree is as follows. You are given a tree T together with (not necessarily positive) weights $w(i)$ for each node $i \in T$. A subtree of T is any connected subgraph of T, (so a subtree is not necessarily the entire subtree rooted at a node). You wish to find a subtree of T that maximizes $\sum_{i \in S} w(i)$. Design an efficient algorithm for solving this problem. Note that there is a linear (in the number of nodes of the tree) time algorithm for this problem. You can assume that for each node in the tree T, you are given a list of its children as well as the parent pointer (except for the root node).

Solution: Maximum weight subtree

We will introduce some notation to faciliate our description. For a node v, let T_v stand for the subtree rooted at v. Let Children(v) denote the set of children of the node v. Let p_v denote the weight of the maximum weight subtree of T_v which does not include v. If such a subtree is empty, set $p_v = 0$. Let q_v denote the weight of the maximum weight subtree of T_v which includes v. Our goal is to compute p_v and q_v for each node $v \in T$.

We show how to compute p_v and q_v for a node $v \in T$ assuming that p_u and q_u are computed for each child u of v.

$$\begin{aligned}
p_v &= \max\{p_u | u \in \text{Children}(v)\} \cup \{q_u | u \in \text{Children}(v)\} \\
q_v &= w_v + \{q_u | q_u > 0 \text{ and } u \in \text{Children}(v)\}
\end{aligned}$$

In our recursion, a leaf node is the base case. For a leaf node v, set $p_v = 0$ and $q_v = w_v$.

Finally, for the root r of the input tree we return $\max(p_r, q_r)$.

Complexity analysis: The algorithm takes linear (in the number of nodes in T) time since for each node $v \in T$, the computation of p_v and q_v takes $O(|\text{Children}(v)|)$ time. Note that the number of edges and the number of nodes in a tree differ by 1.

Proof of correctness:

Claim 0.1. *For $v \in T$, p_v and q_v are correctly computed (as per their definition) by the recursive formula mentioned above.*

Proof. We prove this by induction over the height of v. For the base case, v is a leaf (i.e. at height 0) and it is clear that $p_v = 0$ and $q_v = w_v$. Assume that v is a node at height $i > 0$. Consider any subtree S of T_v which includes v. Let w_S denote the weight of S. Then we have

$$w_S = w_v + \sum_{u \in S \setminus \{v\}} w_u$$

By the structure of a tree we have that $T_v \setminus \{v\} = \sqcup_{u' \in \text{Children}(v)} T_{u'}$ where $\sqcup$ denotes a disjoint union of sets. Therefore, we can write the above sum as

$$w_S = w_v + \sum_{u' \in \text{Children}(v)} \sum_{u \in S \cap T_{u'}} w_u$$

Now, note that $S \cap T_{u'}$ is a connected subgraph since every node of the path that connects $u_1, u_2 \in S \cap T_{u'}$ is in S and $T_{u'}$. Thus $S \cap T_{u'}$ is a subtree of $T_{u'}$. Also, if $S \cap T_{u'} \neq \varnothing$, then since S is connected and it contains v, we have that $u' \in S$. Hence $\sum_{u \in S \cap T_{u'}} w_u \leq q_{u'}$. Therefore we have for any subtree S that contains v

$$w_S \leq w_v + \sum_{u' \in \text{Children}(v)} q_{u'} \leq w_v + \sum_{u' \in \text{Children}(v): q'_u > 0} q_{u'}$$

It is easy to see that by picking v along with the maximum weight subtree containing u' for every u' such that $q_{u'} > 0$, we attain this maximum value. Hence $q_v = w_v + \sum_{u' \in \text{Children}(v): q'_u > 0} q_{u'}$.

For any subtree S not containing v, it cannot have nodes u_1 and u_2 such that $u_1 \in T_{u'_1}$ and $u_2 \in T_{u'_2}$ for distinct children u'_1 and u'_2 of v. This is because $T_{u'_1}$ and $T_{u'_2}$ are disjoint sets and a node in one is only connected to the other through v. Hence we have that $S \subseteq T_{u'}$ for some child u' of v and therefore S is a subtree of $T_{u'}$. Therefore we have for some child u' of v, $w_S \leq \max(p_{u'}, q_{u'})$ and thus

$$w_S \leq \max_{u' \in \text{Children(v)}} \max(p_{u'}, q_{u'})$$

It is easy to attain this upper bound by picking S to be the best of the maximum weight subtrees of the children of v. Hence, $p_v = \max\{p_u | u \in \text{Children}(v)\} \cup \{q_u | u \in \text{Children}(v)\}$.

□

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 0.2. *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: Largest set of indices within a given distance

You are given a sequence of numbers $a_1, \ldots, a_n$ in an array. You are also given a number k. Design an efficient algorithm to determine the size of the largest subset $L \subseteq \{1, 2, \cdots, n\}$ of indices such that for all $i, j \in L$ the difference between a_i and a_j is less than or equal to k. There is an $O(n \log n)$ algorithm for this problem.

For example, consider the sequence of numbers $a_1 = 7, a_2 = 3, a_3 = 10, a_4 = 7, a_5 = 8, a_6 = 7, a_7 = 1, a_8 = 15, a_9 = 8$ and let $k = 3$. $L = \{1, 3, 4, 5, 6, 9\}$ is the largest such set of indices. Its size is 6.

Solution: Largest set of indices (Joseph Li)

先排序，后双指针

By Joseph L.

Input: An array A of size n; a number k
Output: The size of the largest subset $L \subseteq A$, such that:

$$\forall x, y \in L,\ |x - y| \leq k$$

Inequality 1. The constraint on L.

Observations: We are told outright that our algorithm should be $O(n \log n)$. This is a valuable hint - $O(n \log n)$ is enough time to sort the array, for example, and ordering seems helpful since the problem can be reworded as follows:

$$\max(L) - \min(L) \leq k$$

Inequality 2. An equivalent constraint on L.

Description
Precomputation: Sort A, taking $O(n \log n)$ time.

Algorithm: We will iteratively check subsets of consecutive elements in the sorted list. Maintain two pointers i and j, both initialized to 1. These pointers mark the left and right endpoints, respectively, for the subsets that that we will evaluate. Initialize L representing the largest-sized subset seen so far, to $\varnothing$ and s_L, its size to 0.

Repeat until $j > n$:

Check $A_j - A_i$.

- If $A_j - A_i \leq k$: Our current subset satisfies *Equation 2*. Update $s_L \leftarrow \max\{j - i + 1, s_L\}$ and L accordingly. Increment j, expanding the subset to the right.
- Else: Our current subset does not satisfy *Equation 2*. Increment i, thus removing our smallest element from the set.

Solution: Return L and s_L as our solution.

Example Consider the provided example: $A = [7, 3, 10, 7, 8, 7, 1, 15, 8]$

We begin by sorting: $A = [1, 3, 7, 7, 7, 8, 8, 10, 15]$

Now, we will give examples of how our algorithm will behave in the two possible scenarios for a given iteration.

Consider $i = 1$ and $j = 2$. We see that our subset $S = [1, 3]$ satisfies *Equation 2*, and the largest subset seen thus far is $L = \{1\}$ from the previous iteration. S is bigger, so we update L to S and s_L to its size: $s_L \leftarrow 2$. Then, we increment j, effectively expanding its range.

In the very next iteration with $i = 1$ and $j = 3$, we see that our subset $S = [1, 3, 7]$ does not satisfy *Equation 2*; as such, we increment i.

Continuing on, we eventually reach a point at which $i = 3$ and $j = 8$, and our subset $S = [7, 7, 7, 8, 8, 10]$. This is the point at which our solution, L and s_L, are determined - note that the next iteration sees 15, and our set will no longer satisfy *Equation 2*.

Complexity

We spend $O(n \log n)$ on a precomputation. We then perform a bounded number of constant-time operations for up to $2n$ times, as i and j are incremented no more than n times each. This results in an additional $O(n)$. The total complexity is $O(n \log n)$, dominated by the precomputation.

Correctness

We will make and prove a series of claims, using the following conventions:

- We will use the term "valid" to describe subsets that satisfy *Equation 2*.
- For any value of i, let L_i be the largest valid subset in which A_i is the smallest element.

Claim 1: L and s_L are only updated when our pointers mark valid subsets.

Proof 1: We update L and s_L only when $A_j - A_i \leq k$. Our list is sorted and $i \leq j$; as such, $A_i \leq A_{i+1} \leq \ldots \leq A_j$. Therefore, if $A_j - A_i \leq k$, then our subset is valid.

Claim 2: Our algorithm moves i only once we have considered L_i or, otherwise, L_i can't be the largest valid subset.

Proof 2: Our algorithm moves i only when $A_j - A_i > k$; suppose we are at the start of such an iteration.

Consider first the case in which which the previous iteration had a valid subset $[A_i, \ldots, A_{j-1}]$ and thus incremented j. Because the list is sorted, we know that if A_j cannot be in the same subset as A_i, then neither can anything to the right of A_j. Therefore, we can add no further elements to this subset, and the subset from the previous iteration is L_i.

Now, consider the case in which the previous iteration *also* did not have a valid subset - i.e., we're incrementing i two or more iterations in a row. Let $\bar{i}$ be the last i value for which $L_{\bar{i}}$ was valid. Since finalizing $L_{\bar{i}}$, we've moved i (shrunken our range) more times than we've moved j (expanded our range). Therefore, whatever L_i is for our current value of i, it's smaller than $L_{\bar{i}}$ and therefore not a candidate to consider (given what we've already found).

Claim 3: j never needs to be decremented.

Proof 3: When we increment i, why don't we need to retry previous values of j with our new left-endpoint? Suppose we've just incremented i; let $\hat{i}$ and $\hat{j}$ be the current values of i and j, respectively.

By *Claim 2*, we know that if $i = \hat{i}$, then we've already seen either $L_{\hat{i}-1}$ or some larger valid subsets that occur before it with the same right-endpoint. Let $\bar{L}$ be the largest of those subsets. Note that the current subset $[A_{\hat{i}}, ..., A_{\hat{j}}]$, if valid, is at most as large as $\bar{L}$, which at the very least includes $[A_{\hat{i}-1}, ..., A_{\hat{j}-1}]$.

Therefore, any sets with endpoints at $i = \hat{i}$ and $j < \hat{j}$ are smaller than $\bar{L}$ and can't be the largest valid subset - we don't need to waste effort considering them.

Proof 4: Our algorithm finds the size of the overall largest, valid subset, L.

Claim 4: Consider the implications of *Claim 2*. For any A_i, we've either considered the largest valid subset in which A_i is the smallest element, or proven that the subset can't be larger than one that we've already seen. Whatever L is, it obviously must have *some* element as its smallest; thus, L is among the subsets that we've considered. We update s_L to the size of the largest valid subset considered thus far so, having necessarily considered L, we necessarily return the correct value of s_L.

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 $1 \le i, j \le n$ are two arbitrary $n \times n$ Toeplitz matrices. Let $C = (c_{i,j})$ for $1 \le i, j \le n$. We have for $2 \le i, j \le n$

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

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 $\rightarrow$
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 $O(n^2)$.