Homework 1

Wenjun Zhang

A53218995

Problem 1: Next greater element

Algorithm description:

We are given an array and asked to return the Next Greater Element (NGE) for every element. We assume the output of the elements which don't have NGE to be ∞.

Scan the array from left to right, and use a stack to store the indices of the elements which don't have NGE yet. At each time, there are rules as below:

1. If the stack is empty or the current element is smaller than the element corresponding to the top index in the stack, push the index of the current into the stack.
2. If the current element is greater than the element corresponding to the top index in the stack, the output of this top index is equal to the current number, and we pop the top index. Then we continue to compare this current element to the new top index element.
3. At the end, the indices left in the stack should all have the output ∞ because they don't have a NGE.

Proof of correctness:

To argue the correctness of the algorithm, we first check that the algorithm can terminate since it's only a traversal of the array from the left to the right.

Next, according to the rules mentioned in the algorithm description, only when the stack is empty or the element is smaller will the index of the element be pushed into the stack. So the element corresponding to the top index in the stack will always have the smallest value. If the element corresponding to the top index cannot find a NGE, let alone the other elements with the indices in the stack.

When we meet an element greater than the top index element, it is the NGE of this top index element, because the direction of traversal is from left to right and the indices in the stack are those who haven't found their NGEs. Then we pop the stack and continue to check the element with the new top index element.

By the end of the traversal, the indices left in the stack are those who haven't found their NGEs, and they won't have NGE since the array has reached to the end. So their NGEs will be set to ∞.

Time complexity:

The algorithm runs in time O(n) since it only traverses the array once.

Problem 2: Sorted matrix search

Algorithm description:

We are given an m × n matrix in which row and column is sorted in ascending order, and we are asked to find an element.

Let's assume the element we want to find is k, the matrix is A, and each element in the matrix can be called as A[i, j], where $0 \leq i \leq m-1, 0 \leq j \leq n-1$. Considered that the matrix is sorted, we first compare k with A[0,0] and A[m-1, n-1], if k is outside the range [A[0,0], A[m-1, n-1]], we return {-1, -1} because the element cannot be found.

If the element is within the range, we can start from $i = m-1, j = 0$. In each comparison, there are conditions as below:

1. If A[i, j] = k, then the target k is found, and i, j are its corresponding indices.
2. If A[i, j] > k, we make i minus 1 and go back to condition 1.
3. If A[i, j] < k, we make j add by 1 and go back to condition 1.
4. If we reach to conditions like i < 0 or j > n-1, then it means the target cannot be found, and we return {-1, -1}.

Proof of correctness:

To argue the correctness of the algorithm, we first check that the algorithm can terminate since we either find the target element, or j exceeds m-1 or i is smaller than 0.

Next, the algorithm traverse in right or up direction based on the comparison of A[i, j] to the target k. Since the matrix is sorted in both row and column in ascending order, if A[i, j] > k, it means the target element is in the left or up direction of the current element. Because we start from the left bottom side of the matrix, the target will only be in up direction of the current target, so we make $i = i - 1$. Under the same theorem, if A[i, j] < k, the target will only be on the right direction of the current element, so we make $j = j + 1$.

If we reach to a position where A[i, j] = k, then {i, j} is the output. If j exceeds m-1 or i is smaller than 0, there is no other column or row to search, so it will return {-1, -1} as not finding the element.

Time complexity:

The algorithm runs from the left bottom side from the matrix, the worst case is to run until the reach of the right ceiling side, so the worst time complexity is O(m+n).

Problem 3: Maximum overlap of two intervals

Algorithm description:

We are given a list of intervals $[a_i,\ b_i]$ for $1 \le i \le n$ and we are asked to output the maximum overlap of two distinct intervals in the list.

We first sort the list based on the following rules:

1) If $a_i \le a_j$, then interval $[a_i,\ b_i]$ is placed ahead of $[a_j,\ b_j]$.

2) If $a_i = a_j$ and $b_i \ge b_j$, then interval $[a_i,\ b_i]$ is placed ahead of $[a_j,\ b_j]$.

Then we traverse the list from left to right by first setting $Overlap = 0$ and $Endpoint = b_1$. For each interval $[a_i,\ b_i]$ for $2 \le i \le n)$, we compare it with the *Endpoint* and do the following in order:

1. Try to update $Overlap$ with $Overlap = \max(\min(Endpoint,\ b_i) - a_i,\ Overlap)$.
2. If $b_i > Endpoint$, we update $Endpoint = b_i$ for the next interval.
3. If i > n, we return the current $Overlap$ as output.

Proof of correctness:

To argue the correctness of the algorithm, we first check that the algorithm can terminate since we only sort the intervals and then traverse from the first interval to the last once.

Next, the algorithm compares the interval with the current *Endpoint* to try to update the maximum overlap, and then try to update *Endpoint* for the next interval based on the comparison of current *Endpoint* and the interval's endpoint. So what we need to prove is that we only need to consider the current interval with the largest interval's endpoint so far.

Since we have sorted the intervals using the rules mentioned in algorithm description, the startpoint of each interval is always greater than or equal to the intervals in front of it, and we can write it as $a_i \ge a_j$ for $1 \le j < i$. So the maximum overlap of this interval with the intervals in front of it will be no more than $\min(Endpoint,\ b_i) - a_i$ since $\min(Endpoint,\ b_i)$ is the greatest point of the current interval or the intervals ahead. Then we can compare the result with the current integer *Overlap* to determine whether to update it or not.

At the end, the integer *Overlap* is the maximum overlap so far and it will be the output when i is greater than n.

Time complexity:

The sort algorithm takes O(nlogn) time complexity and a single traversal from left to right takes O(n) since it does no more than 3 operations at each position. So the time complexity is

$$T(n) = O(n\log n) + O(n)$$

which is essentially O(n logn).

Problem 4: 132 pattern

Algorithm description:

We are given a sequence of n distinct positive integers $a_1, \dots, a_n$ and we are asked to check if there is a 132-pattern in the list.

The algorithm begins with checking the size of the sequence, if the size is smaller than 3, it returns false. Otherwise, let's assume the 3 elements we need to find are (s_1, s_2, s_3), where $s_1 < s_3 < s_2$. The idea is to traversal the sequence from right to left to find the appropriate (s_2, s_3) pairs. We can use a stack to store the greatest valid s_3. When a greater element appears, the former greatest element becomes candidate of s_3. The detailed implementations are as below:

1. Before each time we push an element into the stack, we first pop all elements in the stack that are smaller than that element. The elements which are popped out become candidates for s_3.
2. We keep track of the maximum valid s_3, which will always be the most recent popped out element from the stack.
3. Once we meet an element smaller the current s_3, we return true.
4. If after a whole traversal the program hasn't returned true, we return false since no such 132 pattern exists.

Proof of correctness:

To argue the correctness of the algorithm, we first check that the algorithm can terminate since we can either find the 132 pattern or we traverse the whole sequence and return false.

Next, since we push an element into the stack only when the stack is empty or the current element is smaller than the top of the stack, if the current element is greater than the top of the stack, the last popped element will be the maximum valid s_3 when considering the current element as s_2.

As we scan the sequence from right to left, we can always keep track of the greatest s_3 of all (s_2, s_3) candidates that we encounter so far. At each time we compare a_i with the current greatest valid s_3. Therefore, if the program returns false, there must be no 132 sequence.

Time complexity:

The algorithm runs in O(n) time complexity since it's a single traversal of the sequence.

Problem 5: Toeplitz matrices

1. The sum of two Toeplitz matrices is necessarily Toeplitz. We can define two Toeplitz matrices as $A = (a_{ij})$ and $B = (b_{ij})$, and the sum of A and B is $C(c_{ij})$, where $c_{ij} = a_{ij} + b_{ij}$. Because $a_{ij} = a_{i-1j-1}$ and $b_{ij} = b_{i-1j-1}$ for $i = 2,3,\dots,n$ and $j = 2,3,\dots,n$, so $c_{ij} = c_{i-1j-1}$ for $i = 2,3,\dots,n$ and $j = 2,3,\dots,n$.

The product of two Toeplitz matrices is not necessarily Toeplitz, for anti-example as below:

$$\begin{bmatrix}1 & 2\\ 3 & 1\end{bmatrix}\begin{bmatrix}1 & -1\\ -1 & 1\end{bmatrix} = \begin{bmatrix}-1 & 1\\ 2 & -2\end{bmatrix}$$

2. Due to the property of Toeplitz matrices, an n × n Toeplitz matrix will have at most 2n – 1 distinct elements. Using hash mapping, for a Toeplitz matrix below we can write it in a vector as Eq (1).

$$\begin{bmatrix} a_n & a_{n-1} & \dots & a_1 \\ a_{n+1} & a_n & \dots & a_2 \\ \vdots & \vdots & \ddots & \vdots \\ a_{2n-1} & a_{2n-3} & \dots & a_n \end{bmatrix}$$

$$[a_1,\ a_2, \dots,\ a_n, \dots,\ a_{n-2},\ a_{n-1}\ ] \tag{1}$$

Therefore, we can add two Toeplitz matrices in the vector form as above and we only need to add 2n – 1 times, which runs in O(n) time.

3.

Algorithm description:

We are asked to give an O(nlogn) time algorithm for multiplying an n × n Toeplitz matrix with a vector of length n. Let's assume the matrix is A and the vector is x, shown as below:

$$A = \begin{bmatrix} a_n & a_{n-1} & \dots & a_1 \\ a_{n+1} & a_n & \dots & a_2 \\ \vdots & \vdots & \ddots & \vdots \\ a_{2n-1} & a_{2n-2} & \dots & a_n \end{bmatrix} \qquad x = [b_1 \quad b_2 \quad \dots \quad b_n]^T$$

Therefore, the product $y = Ax$.

The algorithm is based on FFT algorithm. We first construct $a = (a_1 \quad a_2 \quad \dots \quad a_{2n-1})$ and $b = (b_1 \quad b_2 \quad \dots \quad b_{n-1} \quad 0 \quad \dots \quad 0)$ both of length 2n – 1, then we compute the convolution c of a and b using FFT algorithm. Finally, we output the element $y_i = c_{n-1+i}$ for $1 \le i \le n$.

Proof of correctness:

The expressions for c_i and y_i are shown as below:

$$c_i = \sum_{k=1}^{i} a_{i-k+1} b_k \text{ for } 1 \le i \le 2n-1 \tag{2}$$

$$y_i = \sum_{k=1}^{n} a_{n+i-k} b_k \text{ for } 1 \le i \le n \tag{3}$$

Because $b_k = 0$ for $k > n$, using the two equations above, for $1 \le i \le n$, we can show that:

$$c_{n-1+i} = \sum_{k=1}^{n-1+i} a_{n-1+i-k+1} b_k$$
$$= \sum_{k=1}^{n} a_{n+i-k} b_k$$
$$= y_i$$

Time complexity:

The length of a and b are both 2n – 1, considering that we are using the FFT algorithm, the time complexity is O(n logn).

4.

Algorithm description:

We are asked to give an efficient algorithm for multiplying two n × n Toeplitz matrices. We can design an algorithm based on the algorithm proposed in part 3. Let's assume the two Toeplitz matrices are A and B, and the product of them is C. the elements in the matrices can be written as $a_{i,j}$, $b_{i,j}$ or $c_{i,j}$ for $1 \le i \le n, 1 \le j \le n$. Using the algorithm mentioned in part 3, we can compute the first row and column in C as below:

$$c_{1,j} = \sum_{k=1}^{n} a_{1,k} b_{k,j} \quad \text{for } 1 \le j \le n \tag{4}$$

$$c_{i,1} = \sum_{k=1}^{n} a_{i,k} b_{k,1} \quad \text{for } 1 \le i \le n \tag{5}$$

Then we can compute the rest elements in C using the equation (6) shown below:

$$c_{i,j} = c_{i-1,j-1} + a_{i,1} b_{1,j} - a_{i-1,n} b_{n,j-1} \quad \text{for } 2 \le i \le n, 2 \le j \le n \tag{6}$$

Proof of correctness:

To prove the correctness of the algorithm, we first check that the algorithm can terminate since we use the algorithm to compute the first row and column of C and then we just compute the elements in C until we reach the right bottom side of C.

Next, since we have proved the correctness of how to compute the first row and column in part 3, what we need to prove is to prove the equation (7).

Since A and B are Toeplitz matrices, for any $2 \le i \le n, 2 \le j \le n$, we have:

$$c_{i,j} = \sum_{k=1}^{n} a_{i,k} b_{k,j}$$
$$= a_{i,1} b_{1,j} + \sum_{k=2}^{n} a_{i-1,k-1} b_{k-1,j-1}$$
$$= a_{i,1} b_{1,j} + (\sum_{k=1}^{n} a_{i-1,k} b_{k,j-1}) - a_{i-1,n} b_{n,j-1}$$
$$= c_{i-1,j-1} + a_{i,1} b_{1,j} - a_{i-1,n} b_{n,j-1}$$

Time complexity:

To compute the first row and first column of matrices C both takes O(n logn) time complexity, and to compute the rest elements in C, we need (n – 1)*(n – 1) computations with each using the same equation, so we can compute the time complexity as below:

$$T(n) = O(n \log n) + O(n \log n) + O(n^2) = O(n^2)$$

So the total time complexity for this algorithm is $O(n^2)$.