Algorithms: CSE 202 — Homework I Solutions

Problem 1: Next greater element

Given an array, print the Next Greater Element (NGE) for every element. The Next Greater Element of an item x is the first greater element to its right in the array.

Solution: Next greater element

Algorithm description: Let $a_1, a_2, \ldots, a_n$ be a list of n numbers. Let us assume without loss of generality that a_i is a positive integer for $1 \leq i \leq n$. If a_i does not have any greater element to its right, we define its next greater element to be -1. We compute the next greater element (NGE) for every element a_i by scanning the list from left to right and storing the elements for which we have not yet found the next greater element on the stack.

Here are the details of the algorithm. For each element a_i starting with $i = 1$ and an empty stack, we process a_i as follows until it is pushed onto the stack.

We push a_i onto the stack if the stack is empty or $a_i \leq t$ where t is the top of the stack. Otherwise, if $a_i > t$, we pop t from the stack and mark a_i as its next greater element and repeat the process of pushing a_i onto the stack.

After a_n is processed, we pop all the elements of the stack and mark -1 as their next greater element.

Pseudocode :

Algorithm 1: Next Greater Element

```
input A[1...n]
S ← Empty Stack
for i = 1 to n do
  while S is not empty AND A[S.top] < A[i] do
    t = S.pop()
    NGE(t) = A[i]
  end while
  S.push(i)
end for
while S is not empty do
  t = S.pop()
  NGE(t) = −1
end while
output NGE[1...n]
```

Proof of Correctness: We prove the correctness of the algorithm by induction on the number of iterations of the algorithm. Each iteration exactly involves processing one element of the sequence. More precisely,

Claim 1. For $0 \leq i \leq n$, at the end of processing a_i, the following holds:

1. If the stack is not empty, indexes of the elements on the stack belong to the set $\{1, 2, \ldots, i\}$, are increasing and the values are decreasing as we go from the bottom of the stack to the top of the stack.

2. The elements on the stack do not have a next greater element among $a_1, \ldots, a_i$.

3. If an element e is popped out of the stack during iteration i, then a_i is the next greater element for e.

We regard the beginning of iteration j as the end of iteration $j-1$ for $1 \leq j \leq n$. Although we only push indexes onto the stack, with a slight abuse of the notation, we use the phrase top value on the stack to refer to the value at the index given by the top element of the stack.

Proof.
Base case ($i = 0$): At the end of iteration 0 the stack is empty, so properties $1, 2$, and 3 are satisfied vacuously.
Inductive Step: Assume that the properties hold at the end of iteration j for all $0 \leq j \leq i < n$.

We will prove that the properties hold at the end of iteration $i+1$. Consider the state of the program at the beginning of the iteration $i+1$. There are several possibilities: the stack is empty; if not, a_{i+1} is less than the top value on the stack; or a_{i+1} is greater than the top value on the stack. In all scenarios, we prove that all the three properties will hold.

- The stack is empty: In this case, we simply push a_{i+1} onto the stack during iteration $i+1$. Property 1 holds trivially since there is only one element on the stack. Property 2 holds since the element a_{i+1} cannot have its next greater element in the list $a_1, \ldots, a_{i+1}$. Property 3 holds vacuously since no element has been popped during iteration $i+1$ since the stack is empty at the beginning of the iteration.

- a_{i+1} is less than the top value on the stack: We push a_{i+1} onto the stack in this case. Since $i+1$ is the largest index of all the indices considered and a_{i+1} is smaller than the top value on the stack, it follows from induction hypothesis that Property 1 is satisfied. Again by induction hypothesis and since a_{i+1} is smaller than the top value we conclude that none of the elements on the stack have a next greater element in $a_1, \ldots, a_{i+1}$. Since no elements have been popped from the stack, Proerty 3 holds vacuously.

- a_{i+1} is greater than the top value on the stack: Let $x_1, \ldots, x_k$ be the indices of the elements on the stack with x_1 at the bottom and x_k at the top of the stack. Clearly $i+1 > x_k$ since $i+1$ is largest index we have accessed so far. We pop elements from the stack till the stack is empty or a_{i+1} is less than the top value on the stack. Hence, property 1 still holds at the end of iteration $i+1$.

  Let $a_{x_j}, \ldots, a_{x_k}$ be the elements that were popped during the iteration. This implies that $a_{i+1} > a_{x_j} > \cdots > a_{x_k}$. Additionally, we claim that for every element a_{x_m} where $m \in [j, k]$, there is no other element with index y such that $i+1 > y > m$ and $a_y > a_m$, since from property 2 at the end of iteration i we know that the elements on the stack do not contain their next greater element in the array $a_1, \ldots, a_i$. Therefore all the popped elements have found their next greater element. Since a_{i+1} is smaller than the remaining elements on the stack, we conclude that values on the stack (including a_{i+1}) do not have their next greater element in $a_1, \ldots, a_{i+1}$.

□

Terminating Step: At the end of the loop, if the stack contains any elements (with indices $x_1, x_2, \ldots, x_k$) then $x_1 < x_2 < \cdots < x_k$ and $a_{x_1} > a_{x_2} > \cdots > a_{x_k}$. Since the properties in the claim hold at the end of iteration n, NGE does not exist for the elements on the stack in $a_1, \ldots, a_n$ so we set their NGE values to -1.

Complexity Analysis: Every element in the array is pushed exactly once onto the stack and popped exactly once from the stack. Overall, there are only constant number of operations per element. Therefore, the algorithm runs in $\Theta(n)$ time.

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 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: 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 2: 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 3. 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 the intervals 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: 132 pattern

Given a sequence of n distinct positive integers $a_1, \ldots, a_n$, a 132-pattern is a subsequence a_i, a_j, a_k such that $i < j < k$ and $a_i < a_k < a_j$. For example: the sequence $31, 24, 15, 22, 33, 4, 18, 5, 3, 26$ has several 132-patterns including 15, 33, 18 among others. Design an algorithm that takes as input a list of n numbers and checks whether there is a 132-pattern in the list.

Solution: 132 Pattern

This solution is based on the submission of Yuwei Wang.

High level description

Let $a_1, \ldots, a_n$ be a sequence of distinct integers. We will let $a_0 = \infty$ for the rest of the discussion as it simplifies our definitions without affecting the correctness.
Note: Students are advised to supply all the missing proofs in the following.

For $1 \leq i \leq n$, let $\pi(i) = \max\{j | 0 \leq j < i \text{ and } a_j > a_i\}$. In other words, $\pi(i)$ is the index of the nearest greater element to the left. For all $1 \leq i \leq n$, $\pi(i)$ is defined since $\{j | 0 \leq j < i \text{ and } a_j > a_i\}$ is not empty. Furthermore for $1 \leq i \leq n$, $0 \leq \pi(i) < i$.

For $1 \leq i \leq n$, let $\sigma(i) = j$ where j is such that $a_j = \min_{0 \leq k < i} a_k$. In other words, $\sigma(i)$ is the index of the smallest element to the left of position i. We have $0 \leq \sigma(i) < i$ for all $1 \leq i \leq n$. Note that $\sigma(i) \geq 1$ for all $i \geq 2$ since there is at least one element to the left of position i.

For $i \geq 1$, let $T(i) = (a_{\sigma(\pi(i))}, a_{\pi(i)}, a_i)$.

Fact 0.1. *For $i \geq 3$ such that $\pi(i) \geq 2$, we have $1 \leq \sigma(\pi(i)) < \pi(i) < i$.*

Fact 0.2. *For $i \geq 3$ such that $\pi(i) \geq 1$, if $a_{\sigma(\pi(i))} < a_i$, then $T(i)$ is a 132 pattern.*

For $1 \leq k < j < i$, let the triple $S = (k, j, i)$ be such that (a_k, a_j, a_i) is a 132 pattern, that is, $a_k < a_i < a_j$. We say S is canonical if

- $i \geq 3$ is the smallest index among all such triples,
- $j \geq 2$ is the largest index among all such triples with the smallest i, and
- $k = \sigma(j)$.

Fact 0.3. *If the sequence has a 132 pattern, then there is a canonical triple (k, j, i) of indexes such that (a_k, a_j, a_i) is a 132 pattern.*

For every position $1 \leq i \leq n$, our algorithm checks if the triple $(a_{\sigma(\pi(i))}, a_{\pi(i)}, a_i)$ forms a 132 pattern. In particular we test if $a_{\sigma(\pi(i))} < a_i$. We output true if the test succeeds for some $3 \leq i \leq n$, false otherwise.

For the computation of the previous greater element in linear time, we refer to the solution of the "Next Greater Element" homework question. We will use it here as a black box.

Pseudocode

Algorithm 3: 132Pattern

```
Input  : Numbers a_1, ..., a_n
Output: true if and only if there is a 132 pattern in the input
  π ← findPGE(a_1, ..., a_n)
  σ(1) ← 0
  for i = 2 to n do
    σ(i) ← σ(i − 1) if a_{σ(i−1)} < a_{i−1}, else i − 1
  end for
  for i = 3 to n do
    j ← π(i)
    if j ≥ 1 and a_{σ(j)} < a_i then
      return true
    end if
  end for
  return false
```

Correctness proof

We show that the algorithm returns true if and only if there is a 132 pattern.

We first show that if the algorithm returns true, then there is a 132 pattern. Assume that the algorithm returns true during iteration i for $3 \leq i \leq n$. Consider the triple $(\sigma(\pi(i)), \pi(i), i)$. Let $j = \pi(i)$. We know $j < i$. Since the algorithm returned true during iteration i, we have $j \geq 1$ and $a_{\sigma(j)} < a_i$, which imply $j \geq 2$. Let $k = \sigma(j)$. Since $j \geq 2$, we have $1 \leq k < j$. Combining the inequalities we get $1 \leq k < j < i \leq n$. Since $a_j > a_i$ and $a_k < a_i$ (and thereby $a_k < a_j$), we conclude (k, j, i) forms a 132 pattern.

By Fact 0.3, we know that there is a canonical triple which forms a 132 a pattern if there is a 132 pattern. We show that if $T = (k, j, i)$ is a canonical triple that forms a 132 pattern (that is, $1 \leq k < j < i$ and $a_k < a_j < a_i$), the algorithm return true during iteration i. Since $a_j > a_i$ and j is the largest such index we conclude $j = \pi(i)$. We also have $k = \sigma(j) \geq 1$ since T is canonical. Therefore algorithm returns true during iteration i and would not return true in any earlier iteration since i is the smallest index for which a 132 pattern exists.

Runtime analysis

The algorithm scans the array twice with constant time per iteration. It also calls findPGE once. Since there is a $O(n)$ algorithm for findPGE we conclude that our algorithm runs in time $O(n)$.

Solution: 132 pattern

Algorithm description: Let $a_0, \ldots, a_{n-1}$ be a sequence of distinct positive integers. To find the existence of 132 pattern in the sequence, we employ the following two step algorithm.

- For $0 \leq i \leq n-1$ we compute the smallest number to the left of position i and assign it to $Min_A(i)$. We define $Min_A(0)$ to be ∞. We compute $Min_A(i)$ in linear time by scanning the array from left to right.
- For each location $1 \leq i \leq n-1$ we check if a_i is the middle element of a 132 pattern by scanning the list from right to left while maintaining a balanced binary search tree of all the elements seen so far. For each i we check if a_i is greater than the smallest number to its left, that is, if $a_i > Min_A(i)$. If so, we check if there is an element in between a_i and $Min_A(i)$ in the binary search tree. We do this as follows. We conduct the search for both elements in parallel. If at some point the searches lead to different branches of the tree, we conclude that there is an element between the two and terminate the process. Otherwise, we insert a_i in the binary search tree and continue the scan. If we reach a_0 in the process, we terminate and return False.

Pseudocode :

```
Input  : List A of n positive distinct integers
Output: True if 132 pattern exists in the sequence, False otherwise
1  Function 1:
2  PatternExists(A)
3  Min_A ← empty list of size n
4  minElement ← A(0)
5  Min_A(0) ← ∞
6  for i = 1 to n − 1 do
7  |  Min_A(i) = minElement
8  |  if A(i) < minElement then
9  |  |  minElement = A(i)
10 |  end
11 end
12 BST ← empty BST
13 for j = n − 1 to 1 do
14 |  if Min_A(j) > A(j) then
15 |  |  continue
16 |  end
17 |  if Search(A(j), Min_A(j), BST.root) == True then
18 |  |  return True
19 |  end
20 end
21 return False
22 Function 2: Search(a_j, a_i, root)
23 if root == NULL then
24 |  return False
25 end
26 if root.value > a_i and root.value < a_j then
27 |  return True
28 end
29 if root.value > a_j then
30 |  return Search(a_j, a_i, root.left)
31 end
32 if root.value < a_j then
33 |  return Search(a_j, a_i, root.right)
34 end
```

Proof of correctness:

Claim 4. For $1 \leq i \leq n-1$, at the beginning of the i-th iteration (of the loop in steps 6 through 10) *minElement* is the minimum of the elements in the subarray $A(0), \ldots, A(i-1)$.

Proof: The proof is by induction on the number of iterations, that is, on the value of i. For the base case, consider the state of the execution of the algorithm at the beginning of the first iteration of the loop. At this point, since $i = 1$, the prefix array under consideration has exactly one element, namely, the first element $A(0)$. Also, at this point we have $minElement = A(0)$. The value of *minElement* is indeed the minimum of the subarray $A(0)$. We have thus proved the claim for the case when $i = 1$.

Let $1 \leq x \leq n-1$ be an arbitrary integer. For the inductive step we assume that the claim is true for $1 \leq i \leq x$. Consider the state at beginning of iteration $x+1$. We are considering the subarray $A(0), \ldots, A(x)$ at this point.

We first show that *minElement* is the minimum of the subarray $A(0), \ldots, A(x)$. By the induction hypothesis, at the beginning of the iteration x, *minElement* is the minimum of the subarray $A(0), \ldots, A(x-1)$.

During iteration x we compared it with $A(x)$ and updated it appropriately so we conclude that *minElement* is the minimum of the subarray $A(0), \ldots, A(x)$. Thus, at the beginning iteration $x+1$, the value of *minElement* is equal to the minimum element in the subarray $A(0), \ldots, A(x)$.

For $i < j < k$, we say that (a_i, a_j, a_k) is a *standard* 132 pattern if

- $a_i < a_k < a_j$,
- j is the largest index among all such triples,
- k is such that $\pi(k) = j$, and
- $i = \sigma(j)$.

where π and σ are defined in the previous solution.

Fact 0.4. *If the array has a 132 pattern, then it has a standard 132 pattern.*

Fact 0.5. *Let T be a binary search tree and $\rho_1, \ldots, \rho_l$ be the sorted sequence of its keys. If two queries $q < q'$ (not in the search tree) take the same path in T, then one of the following holds:*

- $\exists 1 \leq i \leq l-1$ *such that* $\rho_i < q < q' < \rho_{i+1}$
- $q < q' < \rho_1$
- $\rho_l < q < q'$

Claim 5. The algorithm returns true if there is at least one 132 pattern in the array.

Proof: Let (a_i, a_j, a_k) be a standard 132 pattern where $i < j < k$, $j = \pi(k)$ and $i = \sigma(j)$. We know that $Min_A(j) = a_i$ and $a_i < a_j$. Consider the iteration when the algorithm is processing the element a_j and assume that the algorithm did not return true prior to this point. If it did, the claim is true. Otherwise, at this point, it will search the binary search tree with the queries a_i and a_j since $Min_A(j) = a_i < a_j$. Since the binary search tree contains at least one element (that is, a_k) which is in between a_i and a_j, the two queries must necessarily disagree at some node in the binary search tree. Let $a_{k'}$ be the key at the node. Since the queries disagreed at the node, the algorithm return true.

Claim 6. If the algorithm returns true, there is at least one 132 pattern in the array.

Proof: If the algorithm returned true, there is some iteration during which it returned true. Let a_j be the element processed during the iteration and $i = \sigma(j)$. Since the algorithm returned during the iteration, the following statements hold:

- $Min_A(j) = a_i < a_j$, and
- during the iteration the algorithm searched the binary search tree with queries a_i and a_j and succeeded in finding a_k where $k \geq j$ and $a_i < a_k < a_j$.

This implies that there are elements a_i, a_j, and a_k in the array such that $i < j < k$ and $a_i < a_k < a_j$, which is a 132 pattern.

Complexity Analysis:

- The first step involves a single scan from left to right with constant time per element. The complexity of this step is $O(n)$.
- The second step involves a single scan from right to left with at most $O(n)$ find and insertion operations on the binary search tree. The complexity of this step is $O(n \log n)$.

Hence, the overall time complexity of the algorithm is $O(n \log n)$.

Problem 5: 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.

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

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

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 $b = (b_0, b_1, \dots, b_{n-1}, 0, 0, \dots, 0)$ and $a = (a_0, a_1, \dots, a_{2n-2})$ 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 $0 \le i \le n-1$.

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

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

Using these two equations we will justify the last step of the algorithm. For $0 \le i \le n-1$, 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 $O(n^2)$.