CSE 202 - Homework 1

1.1 Maximum weight connected subgraph of a tree

High level description:
To choose a connected subgraph, we implement a solution where in at each node, we compute the maximum weight subgraph connected to that node and maximum weight subgraph that is not connected to the node. We perform this through recursion and select the maximum weight subgraph in a bottom up manner until we reach the root node.

Algorithm:

- We design recursive algorithm *max_weight_subgraph* which takes a tree node T as a parameter and outputs the maximum weight connected subgraph originating at that node and the maximum weight connected subgraph not originating at that node (can be originating at the child of the node or further below in the subgraph). Let $w_c(T)$ and $w_{nc}(T)$ denote these weights.
- Starting at the node T, whose children are $[c_1, c_2, c_3...]$, $w_{nc}(T)$ is computed as the maximum of $w_c(c_i)$ and $w_{nc}(c_i)$ (subgraphs originating at the child nodes and further below) for all values of i.
- $w_c(T)$ is computed by taking into consideration only those subgraphs originating at T's children whose weight is positive i.e. $w_c(c_i) > 0$ and summing up such weights along with the weight of the root node. Subgraphs originating at the children whose weight is negative are thus discarded.
- The maximum weight subgraph at a node T is thus returned as the maximum of $w_{nc}(T)$ and $w_c(T)$
- Base Case: The base case for this recursion is encountered when we reach the leaf nodes. Since, there are no further nodes, $w_{nc}(T)$ can be returned as 0 and $w_c(T)$ is the weight of the node itself.

Complexity:
At each node the time taken is proportional to the number of children nodes for the node. Summing up over the entire algorithm, the total time taken is thus proportional to the total number of nodes(N) in the Tree T. Therefore the time complexity can be expressed as $O(N)$.

Correctness:
At the base case, max_weight_subgraph algorithm for a leaf node returns the weight of the node if it is positive else it returns 0. Let P be a node whose $level > 0$. Weight of the subtree w_p at P is the sum of weight of all the nodes originating from P (and their further children) including the weight of the node P itself. Since, $w_{nc}(P)$ is computed as the maximum of $w_c(c_P)$ and $w_{nc}(c_P)$ where c_P are the children of P, it keeps track of the maximum weight of such subset of nodes that are connected to a common ancestor below the level of P. In the computation of $w_c(P)$, the subgraphs connected to children contributing a maximum of negative weight are discarded at the level of P. So the cumulative weight w_p at a node P is always less than or equal to maximum of $w_{nc}(P)$ and $w_c(P)$.

1.2 Largest set of indices within a given distance

High level description:
In this approach we first sort the given sequence of numbers in the array $a_1, a_2, a_3...a_n$. Consequently with the help of a sliding window moving across the array - the width (difference between array element at window end and start) of which is always maintained less than or equal to k - we infer the maximum number of arrays elements that can lie in the window.

Algorithm:
For the given array $a_1, a_2, a_3...a_n$ where n ≥ 1, we apply the merge sort algorithm to sort the arrays elements in a non-decreasing manner. We then declare two pointers *start* and *end* which respectively denote the start and end of window of indices under consideration. Initialize both these pointers to 0, the index of the first element of the sorted array. We also declare a variable *max_subset* containing the largest number of indices that can be in the subset at a given point in the traversal.

We first begin by moving the *end* pointer to the right one element at a time. During each of these increments, we compute the absolute difference(*d*) between arrays elements at the *start* index and at the *end* index. If *d* is less than or equal to k, we update the *max_subset* value with the maximum of $(end - start + 1)$ and the current value of *max_subset*. Else if at any point the computed difference *d* becomes greater than k, we keep the end pointer fixed and move the start pointer to the right until the absolute difference of array elements at these two positions again becomes less than or equal to k. We then continue the algorithm by shifting the end pointer and updating the max_subset when necessary. We stop further computation when the end pointer reaches the end of the array. We output the required length of largest subset which is stored in *max_subset* at this point.

Complexity:
Sorting the array of length n using the merge sort algorithm take $O(nlogn)$ time. Since, using the two pointers start and end, we only traverse through the array once i.e. while end pointer moves from 0 to end of the arrays, this step takes linear time $O(n)$. Therefore, overall time complexity is of the order $O(nlogn)$.

Correctness:
We prove the correctness of the above algorithm through contradiction.
Assume for the sorted array $a_1, a_2, a_3...a_n$, the algorithm A_1 as outlined above outputs m_1 as the maximum number of arrays elements within a distance k. For contradiction, assume there exists an algorithm A_2 which gives that the maximum number of arrays elements within a distance k for the same array is m_2 where $m_2 > m_1$. According to A_2, there exist elements $a_i, a_{i+1}, a_{i+2}...a_{i+m_2-1}$ such that:

$$(a_{i+m_2-1} - a_i) \leq k \qquad \text{——} \quad \text{condition 1}$$

This implies that while performing algorithm A_1, when the end pointer was at the index $i + m_2 - 1$, the start pointer was shifted beyond the index i. This happens if and only if the following condition is met:

$$(a_{i+m_2-1} - a_i) > k \quad \text{which is clear contradiction of condition 1.}$$

Hence, there exists no more than the number of indices as obtained from the algorithm A1 for an array $a_1, a_2, a_3...a_n$ within a given distance k.

1.3 132 pattern

High level description:
One approach to find if a 132 pattern exists in an array $a_1, a_2, a_3...a_n$ is to iterate through the indices of the array to find a suitable k. For each k, we check that there exists a number a_j greater than a_k before the index k. This is done using a stack in which we maintain the entries in a decreasing order. We also must check for a_i less than a_k before the index k which is done by computing the minimum so far namely minimum till index $k-1$. Th minimum so far must be less than a_k. When these two conditions are met, there exists a 132 pattern in the array.

Algorithm:
Initialize an empty stack and a variable *min* with arbitrarily high value. For each element a_k in $a_1, a_2, a_3...a_n$ starting with k=1, we process a_k as follows:

- If k is 1, we insert the pair (a_k, min) into the stack.
- For all other k, we pop the top of the stack(t) as long as the first element in the pair t is less than or equal to a_k. If the stack becomes empty or the second element in pair t holds a value greater than or equal to a_k, we simply insert (a_k, min) into the stack and proceed with the next step.
- Else if none of these two conditions are met, i.e for a value of a_k, we found a pair t in the stack such that the first element in t is greater than a_k and the second element is less than a_k, it indicates the presence of 132 pattern and therefore we return a value confirming the pattern.
- If a 132 pattern is not found at the previous step, update the min value with minimum of current min value and a_k.

Correctness:
For an a_k under consideration, presence of 132 pattern is only returned when a pair p exists in the stack such that the first element of p is greater than a_k and the second element is less than a_k. The first element in such a pair is a_j and the second element is a_i. Since an element is pushed into a stack only after it has been searched for being a potential a_k, and k is increased at every step, it is always ensured that $k > j$. Similarly, the minimum is updated at the current element only after the minimum so far is pushed into the stack which ensures that $i < j$.

Complexity:
In the worst case, we traverse the entire array with the k pointer only once which takes linear time $O(n)$.

1.4 Toeplitz matrices

A toeplitz matrix A of size nxn with entries a_{ij} where i and j denote row number and column number respectively is as follows:

$$A = \begin{bmatrix} a_{ij} \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} & . & . & . & a_{1(n-2)} & a_{1(n-1)} & a_{1n} \\ a_{21} & a_{11} & a_{12} & a_{13} & . & . & . & a_{1(n-2)} & a_{1(n-1)} \\ a_{31} & a_{21} & a_{11} & a_{12} & . & . & . & . & a_{1(n-2)} \\ . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . \\ a_{n1} & a_{(n-1)1} & . & . & . & . & . & . & a_{11} \end{bmatrix}$$

We observe that all the elements across each of the $2n - 1$ diagonals (represented by $j - i = k$ where k varies from $-(n - 1)$ to $(n - 1)$) are equal.

1. Adding two toeplitz matrices results in a toeplitz matrix as across any diagonal in the both addends, the elements are equal and hence the elements across the same diagonal in the resultant matrix are equal.

To evaluate the product of two toeplitz matrices, let us consider two toeplitz matrices $A = a_{ij}$ and $B = b_{ij}$ of size nxn. Elements in the longest diagonal of the product matrix ($P = p_{ij}$) p_{ii} are obtained as follows through matrix multiplication:

$$p_{11} = a_{11}b_{11} + a_{12}b_{21} + a_{13}b_{31} + ... + a_{1(n-1)}b_{(n-1)1} + a_{1n}b_{n1}$$
$$p_{22} = a_{21}b_{12} + a_{11}b_{11} + +a_{12}b_{21} + ... + a_{1(n-1)}b_{(n-1)1}$$
$$p_{33} = a_{31}b_{13} + a_{21}b_{12} + +a_{11}b_{11} + ... + a_{1(n-2)}b_{(n-2)1}$$

$$(p_{11} = p_{22} \text{ only if } a_{1n}b_{n1} = a_{21}b_{12})$$
$$(p_{22} = p_{33} \text{ only if } a_{1(n-1)}b_{(n-1)1} = a_{31}b_{13})$$

Therefore values of p_{ii} are not necessarily equal for all possible values of a_{ij} and b_{ij} which is a sufficient condition to establish that the product of two toeplitz matrices is not necessarily a toeplitz matrix.

2. From the structure of the toeplitz matrix, we observe that values across each of the diagonals represented by $j - i = k$ are equal. For an nxn, the value of k varies from $-(n - 1)$ to $(n - 1)$. Hence, there are only $2n - 1$ distinct entries in an nxn matrix. Also, we observe that all these distinct values are present in the first row and the first column of the matrix (these are the elements where the diagonals originate).

An nxn toeplitz matrix can therefore be represented by a single array of size $2n - 1$ containing the values of the first row and the first column without any loss of information.The matrix A can be stored as:

$$A = \begin{bmatrix} a_{ij} \end{bmatrix} = \begin{bmatrix} a_{1n} & a_{1(n-1)} & . & . & a_{11} & a_{21} & a_{31} & . & . & a_{n1} \end{bmatrix}$$

where $n - (j - i)$ represents the diagonal on which the element a_{ij} is present. There are $2n - 1$ diagonals in total for an nxn matrix. The 1^{st} diagonal has only one element which is the right topmost element. The n^{th} is the longest diagonal with n elements. The $(2n - 1)^{th}$ diagonal has only one element which is the bottom left most in the matrix.

The first n values are the first row of the matrix A in the reverse order. The next $n - 1$ values are the values in the first column (excluding the first element, since a_{11} is already present in the first n elements).

Addition of two toeplitz matrices:
This array representation of the toeplitz matrix facilities the addition of two nxn toeplitz matrices in linear time. Since the two arrays are of size $2n-1$, they can be added in $O(2n-1)$ time which can be considered to be of the order $O(n)$.

Reconstructing the matrix from the array representation:
Now, given an array representation of toeplitz matrix, we must also be able to reconstruct the matrix form. Suppose, we know the array representation of a nxn toeplitz matrix $B = b_{ij}$ as:

$$B = \begin{bmatrix} b_1 & b_2 & . & . & . & . & b_{2n-1} \end{bmatrix}$$

For the upper triangular matrix including the longest diagonal (i.e. $j - i \geq 0$), the values are present in the first n elements of the $2n-1$ size array. For the lower triangular matrix excluding the longest diagonal (i.e. $j - i < 0$), the elements are present in the next $n-1$ indices. In the reconstructed matrix RB from B, the 0-based index of elements rb_{ij} $(1 \leq i, j \leq n)$ in B are given by:

$$\text{0-based index of } rb_{ij} = (n-1) + (i-j)$$

3. Multiplication of nxn toeplitz matrix with vector of length n:

Observations in the matrix vector multiplication:
Consider a nxn toeplitz matrix represented by A = $[a_0, a_1, ..., a_{2n-2}]$, this matrix is to be multiplied by a vector B = $[b_0, b_1, ...b_{n-1}]$. The matrix multiplication looks as follows:

$$AB = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ . \\ . \\ p_{n-1} \end{bmatrix} = \begin{bmatrix} a_{n-1} & a_{n-2} & a_{n-3} & . & . & . & . & a_2 & a_1 & a_0 \\ a_n & a_{n-1} & a_{n-2} & a_{n-3} & . & . & . & . & a_2 & a_1 \\ a_{n+1} & a_n & a_{n-1} & a_{n-2} & . & . & . & . & . & a_2 \\ . & . & . & . & . & . & . & . & . & . \\ . & . & . & . & . & . & . & . & . & . \\ a_{2n-2} & . & . & . & . & . & . & . & . & a_{n-1} \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ . \\ . \\ b_n \end{bmatrix}$$

$$AB = \begin{bmatrix} p_0 \\ p_1 \\ p_2 \\ . \\ . \\ p_{n-1} \end{bmatrix} = \begin{bmatrix} a_{n-1}b_0 + a_{n-2}b_1 + ... + a_0 b_{n-1} \\ a_n b_0 + a_{n-1}b_1 + ... + a_1 b_{n-1} \\ . \\ . \\ . \\ a_{2n-2}b_0 + a_{2n-3}b_1 + ... + a_{n-1}b_{n-1} \end{bmatrix}$$

Observing the resultant product matrix, we find that the element p_i is the $(i + (n-1))^{th}$ coefficient in the product of the polynomials $C(X)$ and $D(X)$ given by:

$$C(X) = a_0 + a_1 x + a_2 x^2 + ... + a_{2n-2} x^{2n-2}$$
$$D(X) = b_0 + b_1 x + b_2 x^2 + ... + b_{n-1} x^{n-1}$$

Algorithm:

- Therefore to find the product of a nxn toeplitx matrix $A = [a_{ij}]$ and vector $B = [b_i]$ of length n, we construct two polynomials $C(X)$ and $D(X)$.
- $C(X)$ is a polynomial of degree $2n - 2$ where the coefficients are obtained from the array representation of the toeplitz matrix.
- $D(X)$ is a polynomial of degree $n-1$ where the coefficients are obtained from the given n-dimensional representation of vector B in the same order.
- We can now obtain the polynomial $M(X)$ which the the product of $C(X)$ and $D(X)$ using ***Fast Fourier Transform*** and ***Inverse Fast Fourier Transform***.
- The resultant product $[p_i]$ of A and B can therefore be expressed as:

$$p_i = \text{coefficient of } (i + (n-1))^{th} \text{ degree term in } M(X) \qquad \text{where } i \in [0, n-1]$$

Complexity:
The conversion to toeplitz matrix from matrix form to arrays form and constructing the polynomialś coefficient representation takes linear time of the order $O(2n - 1)$ which can be expressed as $O(n)$ time. The product of the polynomial of degree $2n - 2$ using ***Fast Fourier Transform*** and ***Inverse Fast Fourier Transform*** takes $O((2n - 2)log(2n - 2))$ time which again can be expressed as $O(nlogn)$. Therefore the overall time complexity of the matrix vector multiplication is $O(nlogn)$.

4. Multiplication of two nxn toeplitz matrices:

Consider two toeplitz matrices A and B of size nxn whose array representations are $[a_0, a_1, a_2, ..., a_{2n-2}]$ and $[b_0, b_1, b_2, ..., b_{2n-2}]$ respectively. The product of the matrices is as follows:

$$AB = \begin{bmatrix} a_{n-1} & a_{n-2} & . & . & a_2 & a_1 & a_0 \\ a_n & a_{n-1} & . & . & . & a_2 & a_1 \\ a_{n+1} & a_n & . & . & . & . & a_2 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ a_{2n-2} & . & . & . & . & . & a_{n-1} \end{bmatrix} \begin{bmatrix} b_{n-1} & b_{n-2} & . & . & b_2 & b_1 & b_0 \\ b_n & b_{n-1} & . & . & . & b_2 & b_1 \\ b_{n+1} & b_n & . & . & . & . & b_2 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ b_{2n-2} & . & . & . & . & . & b_{n-1} \end{bmatrix}$$

Consider the elements in the product matrix of size nxn are given by M_{ij} where i and j denote the row and column number respectively $(1 \leq i, j \leq n)$.

We observe the elements in any one of the diagonals in the product matrix. Without loss of generality, considering the longest or n^{th} diagonal, we have the values as:

$$\begin{gathered} M_{11} = a_{n-1}b_{n-1} + a_{n-2}b_n + a_{n-3}b_{n+1} + ... + a_1b_{2n-3} + a_0b_{2n-2} \\ M_{22} = a_nb_{n-2} + a_{n-1}b_{n-1} + a_{n-2}b_n + ... + a_2b_{2n-4} + a_1b_{2n-3} \\ M_{33} = a_{n+1}b_{n-3} + a_nb_{n-2} + a_{n-1}b_{n-1} + ... + a_3b_{2n-5} + a_2b_{2n-4} \\ . \\ . \end{gathered}$$

We observe that :

$$
\begin{gathered}
M_{22} = M_{11} - a_0 b_{2n-2} + a_n b_{n-2} \\
M_{33} = M_{22} - a_1 b_{2n-3} + a_{n+1} b_{n-3} \\
M_{44} = M_{33} - a_2 b_{2n-4} + a_{n+2} b_{n-4} \\
. \\
. \\
M_{ii} = M_{(i-1)(i-1)} - a_{i-2} b_{2n-i} + a_{n+i-2} b_{n-i} \text{ where } 2 \leq i \leq n
\end{gathered}
$$

Observation 1: The above representation implies that if we have the starting element (must be either in the first row or the first column) in the product diagonal, the subsequent elements on that diagonal can be obtained through computing 2 products of elements (one is added and the other is subtracted) at each step. The generalization M_{ij} is as follows:

$$M_{ij} = M_{(i-1)(j-1)} - a_{i-2} b_{2n-j} + a_{n+i-2} b_{n-j} \text{ where } 2 \leq i, j \leq n$$

The first column in the product matrix can be expressed as follows through the rules of matrix multiplication:

$$
\begin{bmatrix} M_{11} \\ M_{12} \\ M_{13} \\ . \\ . \\ M_{1n} \end{bmatrix} = \begin{bmatrix} a_{n-1} & a_{n-2} & . & . & a_2 & a_1 & a_0 \\ a_n & a_{n-1} & . & . & . & a_2 & a_1 \\ a_{n+1} & a_n & . & . & . & . & a_2 \\ . & . & . & . & . & . & . \\ . & . & . & . & . & . & . \\ a_{2n-2} & . & . & . & . & . & a_{n-1} \end{bmatrix} \begin{bmatrix} b_{n-1} \\ b_n \\ b_{n+1} \\ . \\ . \\ b_{2n-2} \end{bmatrix}
$$

Similarly, the first row in the product matrix (represented as column matrix below) can be expressed as follows (Product of matrix B where the diagonals $i - j = k$and $j - i = k$ are swapped and first row of A):

$$
\begin{bmatrix} M_{11} \\ M_{21} \\ M_{31} \\ . \\ . \\ M_{n1} \end{bmatrix} = \begin{bmatrix} b_{n-1} & b_{n+1} & . & . & . & b_{2n-3} & b_{2n-2} \\ b_{n-2} & b_{n-1} & . & . & . & . & b_{2n-3} \\ b_{n-3} & b_{n-2} & . & . & . & . & . \\ . & . & . & . & . & . & . \\ b_1 & . & . & . & . & . & . \\ b_0 & b_1 & . & . & . & . & b_{n-1} \end{bmatrix} \begin{bmatrix} a_{n-1} \\ a_{n-2} \\ a_{n-3} \\ . \\ . \\ a_0 \end{bmatrix}
$$

Observation 2: The first row and the first column of the product matrix M can be expressed as a product of a nxn toeplitz matrix and a vector of size n. The first row of the product matrix can be obtained through the product of toeplitz matrix represented as $[b_{2n-2}, b_{2n-3}, ..., b_1, b_0]$ and vector $[a_{n-1}, a_{n-2}, ..., a_1, a_0]$. The first column of the product matrix can be obtained through the product of toeplitz matrix represented as $[a_0, a_1, a_2, ..., a_{2n-2}]$ and vector $[b_{n-1}, b_n, ..., b_{2n-2}]$.

Algorithm:
Multiply two toeplitz matrices of size nxn given by A = $[a_0, a_1, a_2, ..., a_{2n-2}]$ and B = $[b_0, b_1, b_2, ..., b_{2n-2}]$

- We first compute the first row of the product matrix M which is obtained by multiplying the toeplitz matrix represented as $[b_{2n-2}, b_{2n-3}, ..., b_1, b_0]$ with the vector $[a_{n-1}, a_{n-2}, ..., a_1, a_0]$ as outlined in part 3 of the problem.
- Similarly, we compute the first row of the product matrix M by multiplying the toeplitz matrix represented as $[a_0, a_1, ..., a_{2n-2}]$ with the vector $[b_{n-1}, b_n, ..., b_{2n-2}]$ as outlined in part 3 of the problem.
- The known values in the product matrix are as follows:

$$M = \begin{bmatrix} M_{11} & M_{12} & . & . & . & . & M_{1n} \\ M_{21} & & & & & & \\ M_{31} & & & & & & \\ . & & & & & & \\ . & & & & & & \\ M_{n1} & & & & & & \end{bmatrix}$$

- Evaluate rest of the elements using the formula:

$$M_{ij} = M_{(i-1)(j-1)} - a_{i-2}b_{2n-j} + a_{n+i-2}b_{n-j} \text{ where } 2 \le i, j \le n$$

Complexity:
To compute the first row and first column of the product matrix, we use the method outlined in part 3 of this problem which takes a time complexity of $O(nlogn)$. To obtain the rest of the $(n-1)^2$ elements, we compute two products each and we perform one addition and one subtraction each. Assuming negligible time is take for addition and subtraction operations, the time complexity of this step is of the order $O(2n^2)$. Therefore the overall time complexity is of the order $O(n^2)$
