Homework 1

Q1:

For each vertex *v* in V(G), we will do breadth-first search(BFS) starting at *v* and find the longest path for vertex v.

First, we define a list: *Boolean[] visited=new Boolean[n]* and *n* is the number of vertexes and initialize it with all false to represent that all nodes are not been visited. We also define a queue *q_BFS* to realize BFS and another queue *q_distance* to record the shortest distance $\delta(u, v)$ between vertex *v* and the corresponding node u in *q_BFS*. The process is as below:

1. We queue in the vertex *v* and queue in 0 to *q_distance* to show the shortest path between *v* and *v* itself is 0;
2. Queue out the first vertex in *q_BFS* and the first integer *i* in *q_distance*;
3. Queue in all of v's descendant nodes to *q_BFS* and queue in *(i+1)* simultaneously to *q_distance*, update all the descendent nodes as visited to avoid visiting again;
4. We repeat the step 2 and 3 until all the vertexes have been visited, that is, *q_BFS* is empty.

The last integer that *q_distance* queued out is the diameter for vertex *v*. We do the same BFS for all vertexes in V(G) and record the maximum value of *n* diameters.

Proof of correctness:

We first verify that the distance *q_distance* queued out between vertex *v* and *u* is the shortest. Assume that we have two path between vertex *v* and vertex *u*. Path 1: *v->s->u* and path 2: *v->s->t->u*. After queueing out *v*, we push node *s* into the queue and when we retrieve all descendant vertexes of *s*, vertex *t* and *u* will be pushed into the *q_BFS* with the same distance. Because *u* has been visited, we will no longer push *u* into the queue when retrieving vertex *t*. Thus only the smallest distance between vertex *v* and any other node will be computed. Because we use a queue to preserve the visited node, after we queued out v from *q_BFS*, we will push all nodes whose distance is one edge away from *v* into the queue. When polling out these nodes, we push all nodes that are two edges away from *v* and these nodes and ignore those nodes that have been visited. Those nodes that are three edges away from v have no direct edge with any node that are one edge way from *v*, otherwise, they will be pushed into *q_BFS* with corresponding distance as two. Through traversing, we are assured that each node is at the least distance in *q_distance*.

After we did the same breadth-first search to all vertexes, we will get the diameter for every vertex. And we could compare and find the maximum to be the diameter of the tree.

Time complexity:

For a certain vertex *v,* we will visit at most all the edge in the graph, that is, E. Since we do searching for all vertexes, the entire run time is O(|V||E|). (V is the number of vertexes and E is the number of edges).

Q2:

Assume the element we need to find is *x*. If *m=n=1*, we compare *M[1][1]* with target *x*, if *M[1][1]=x*, we return a two-elements array *[1,1]* to show target x's position. Otherwise, we return *[-1,-1]* to represent no element is found.

For *1≤a≤c≤m* and *1≤b≤d≤n*, we define a submatrix *SM* of the m*n matrix *M* which is bounded by *M[a,b]* and *M[c,d]*.

Firstly, we define *a=1, b=1* and *c=m, d=n*. We will use binary search to find the target *x*. We could define two integer variables *mid_row=(a+c)/2* and *mid_col=(b+d)/2* and then compare *M[mid_row][mid_col]* with target *x*.

1) *M[mid_row [mid_col]=x   =>   return [mid_row, mid_col];*
2) *M[mid_row][mid_col]>x   =>   c=mid_row, d=mid_col;*
3) *M[mid_row][mid_col]<x   =>   a=mid_row, b=mid_col.*

After comparison, we need to compare *a* with *mid_row* and *b* with *mid_col*. If *a=mid_row* and *b=mid_col*, we need to search all elements in submatrix SM determined by *M[a,b]* and *M[c,d]* which is at most 4 elements. if no element=*x* is found, we return *[-1,-1]*. Otherwise, we repeat the process shown above.

Proof of correctness:

Clearly, if the matrix only has one element, we could compare it with *x* to get *x*'s position. When *m>1* or *n>1*, we define *M[mid_row][mid_col]* and compare to narrow the scope of finding *x*.

If *M[mid_row] [mid_col]=x*, we find the right location and return it. In addition, we did not consider if there are same elements in the matrix, in that case, we should find out if the elements near *M[mid_row] [mid_col]* is still same to *x*.

If *M[mid_row] [mid_col]>x*, then *x* must be in the left and upper part of *M[mid_row] [mid_col]* and include *M[mid_row][j]* for *j<mid_col* and *M[i][mid_col]* for *i<mid_row* because M is sorted. So we narrow the scope by change *c=mid_row* and *d=mid_col* to

avoid redundant comparison with the part of larger values. The same is to *M[mid_row][mid_col]<x*. We always guarantee target *x* within submatrix *SM* determined by *M[a,b]* and *M[c,d]*.

It's important to note that we could not make *c=mid_row-1, d=mid_col-1* for *M[mid_row] [mid_col]>x*, because *x* could be in the same line of *mid_row* or *mid_col*. We could find *x* if *M[mid_row] [mid_col]=x* or compare *x* with elements in submatrix determined by *M[a][b]* and *M[c][d]* . The last round will be stopped when *M[mid_row] [mid_col]=x(type 1)* or *c-a≤1 and d-b≤1 (type 2)* . In *type 2*, we need to compare at most 4 elements with *x* to find the location of *x*.

Time complexity:

If *M[mid_row] [mid_col]=x*, the loop will stop. So it costs longer time when no element is found in which case the loop will stop when *a=mid_row and b=mid_col*. Since the row have nothing to do with column, the running time is the maximum of *logm* and *logn*. Time complexity: *O(max(logm, logn))*.

Q3.

First, we sort the intervals according to their left boundary and get a list of intevals $[a_i,b_i]$ *for* $1\leq i\leq n$, and $a[i]\leq a[i+1]$ *for* $1\leq i<n$. If there is only one interval, we return 0.

There are three scenarios that could happen to two adjacent intevals $[a_i,b_i]$ and $[a_{i+1},b_{i+1}]$:

1) $a_i\leq a_{i+1}\leq bi,\ b_{i+1}>b_i;$
2) $a_i\leq a_{i+1}\leq b_{i+1}\leq b_{i;}$
3) $a_{i+1}>b_i$

We also define a variable *max_overlap* which initialized as 0.

For case 1), the overlap *OLP* will be b_i-a_{i+1}.

For case 2), the overlap *OLP* will be $b_{i+1}-a_{i+1.}$

For case 3), the overlap *OLP* will be 0.

We update the *max_overlap=max(max_overlap, OLP)*.

We define a array *prev=[x,y]*. *[x,y]* will be initialized as $[a_1,b_1]$. Then for intervals $[a_i,b_i]$ in sorted intervals[2:n], we compare Interval $[a_i,b_i]$ with *prev: [x,y]* and obtain three conditions:

4) $x\leq a_i\leq y,\ b_i>y;$

5) $x \le a_i \le b_i \le y$;

6) $a_i > y$

For case 4) and 6), we change *prev=[a_i,b_i]*, otherwise, prev is still *[x,y]*. At the same time, we update the *max_overlap* by *max_overlap=max(max_overlap, OLP)*. After comparison with all intervals in sorted intervals, we return the *max_overlap* as the result.

Proof of correctness:

In our algorithm, we only need to compare each interval with its previous preserved interval *[x,y]*. We could illustrate this in a small example. Assume that we have a previous interval *I1:[x,y]* and its descending two intervals *I2:[a_i,b_i]* and *I3:[a_{i+1},b_{i+1}]*. Because our intervals have been sorted, it is clear that $x \le a_i \le a_{i+1}$.

If $x \le a_i \le y$, $b_i > y$ (case 4), the overlap of interval *I1* and *I2* begins at a_i and ends at y, which is $[a_i,y]$. Since $a_{i+1} > a_i$, the overlap of interval *I1* and *I3* will be:

7) $[a_{i+1}, y]$ if $a_{i+1} < y$ *and* $b_{i+1} > y$

8) $[a_{i+1}, b_{i+1}]$ if $b_{i+1} < y$

9) 0 if $a_{i+1} > y$

Since $y < b_i$, the overlap between *I1* and *I3* should be the sub-overlap between *I2* and *I3*.

In case 6), *I1* and *I2* do not have overlap, so there is no overlap between *I1* and *I3* since $a_{i+1} > a_i > y$. Above all, the overlap between *I2* and *I3* is always larger than that between *I1* and *I3*. So we could update *[x,y]* with *[a_i,b_i]* to compare with the subsequent intervals.

If $x \le a_i \le b_i \le y$ (case 5), the overlap between *I1* and *I2* is *[a_i,b_i]* which is a sub-interval of *I1*. So the overlap between *I1* and *I3* should always larger or same to the overlap between *I2* and *I3*. So we could compare *[x,y]* with *[a_{i+1},b_{i+1}]* directly to save time. By determining the best previous interval, we could always obtain the maximum overlap between current interval and its previous all intervals. At the same time, we record the maximum overlap.

Time complexity:

We have to sort the intervals based on their left boundary. We could do sorting in *O(nlogn)* by using merge sort or quick sort. After we sort the intervals, all we need to do is a for-loop to compare subsequent intervals with previous interval *[x,y]* which needs *n* rounds. In each round, only constant operations are needed. So the comparison process only takes *O(n)*. So the overall time complexity is *O(nlogn)*.

Q4:

1. If the dimensions of two Toeplitz matrices are the same, the sum of two Toeplitz matrices is necessarily Toeplitz but the product is not.
   1) Consider two Toeplitz matrix $A=(a_{ij})$ such that $a_{ij}=a_{i-1,j-1}$ for $i=2,3,..n$ and $j=2,3,...n$ and $B=(b_{ij})$ such that $b_{ij}=b_{i-1,j-1}$ for $i=2,3,..n$ and $j=2,3,...n$.

   The sum of A and B is C:

   $$\begin{aligned} c_{ij} &= a_{ij}+b_{ij} \\ &= a_{i-1,j-1}+b_{i-1,j-1} \\ &= c_{i-1,j-1} \end{aligned}$$

   for $i=2,3,...n$ and $j=2,3,...n$, so the sum of Toeplitz matrices is Toeplitz.
   2) The product of A and B is D: $d_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$, $d_{i-1,j-1} = \sum_{k=1}^{n} a_{i-1,k}b_{k,j-1}$. Because a_{ik} is not equal to $a_{i-1,k}$, $d_{ij}\neq d_{i-1,j-1}$. So the product of two Toeplitz matrices is not Toeplitz matrix. A counter-example:

   $$\begin{bmatrix} 1 & 2 \\ 3 & 1 \end{bmatrix}\begin{bmatrix} 1 & 4 \\ 5 & 1 \end{bmatrix} = \begin{bmatrix} 11 & 6 \\ 8 & 13 \end{bmatrix}$$

2. We represent Toeplitz matrix $A=(a_{ij})$ like shown in the table. $a_{ij}=x_{j-i}$ and we only need to know x_m for $1-n\leq m\leq n-1$ and then we could know all a_{ij} for $1\leq i\leq n$ and $1\leq j\leq n$. Similarly, we represent Toeplitz matrix $B=(b_{ij})$ as $b_{ij}=y_{j-i}$. The sum of A and B is C. For every element in C: $c_{ij}=a_{ij}+b_{ij}=x_{j-i}+y_{j-i}$. Because C is a Toeplitz matrix like proven in part (1), we only need to compute z_{j-i} ($z_{j-i}=c_{ij}$) for $1-n\leq j-i\leq n-1$. So the total addition will occur $(2n-1)$ times. The time complexity is $O(n)$.

x_0	x_1	…	X_{n-1}
x_{-1}			
…			
X_{1-n}			

3. We define the multiplication as $v=Au$. We represent u as $u=[u_n,u_{n-1},...,u_1]^T$.
   Then $v_i = \sum_{k=1}^{n} a_{ik}u_{n+1-k} = \sum_{k=1}^{n} x_{k-i}u_{n+1-k}$. It's a convolution. And then we use FFT to compute the convolution and extract the n desired coefficients. The FFT 's time complexity is O(nlogn).
4. We could treat the second matrix as a sequence of n*1 vectors and multiply the first matrix with each vector to get the result like shown in part 3. Because there are n vectors, the time complexity is $O(n^2logn)$.