CS 515
Tenghuan Li; 933638707; liten@oregonstate.edu
Liangjie Zheng 934056203; zhengli@oregonstate.edu
Xingjian Su; 933647032; sux@oregonstate.edu

HW2:

Problem 1:

As the problem say, we can call the $L(i,j)$ be the longest of the longest convex subsequence of the array $A[1 \ldots n]$. As the question say, we need to convex if 2 $2X[i] <$ X[i-1] + X[i+1] for all i.

Assuming that $i < j < k$; the recurrence relation will be:

$L(i,j) = \max(L(i,j), 1 + L(j,k))$, when the $j < k \leq n, and\ A[i] + A[k] > 2A[j]$.

Then, the base case for the length call $l = 0$, So, we can get the longest for the longest convex subsequence of the array $A[1 \ldots n]$ will be the $\max(l, L(i,j))\ when\ 1 \leq i < j \leq n$.

For the dynamic programming ways to fill the values at the $L[i][j]$.

$l \leftarrow 0;$
$for\ (i = n-1; i \geq 1; i--)\{$
  $for\ (j = n; j \geq i+1; j--)\{$
    $for(k = j+1; k \leq n; k++)\{$
      $if(2A[j] < (A[i] + A[k]))$
        $L[i][j] = \max(L[i][j], 1 + L[j][k]);$
    }
    $l = \max(L[i][j], l);$
  }
}
return $l$;

As the pseudocode shows, we can see that there are three levels of loop. The running time is $T(n) = O(n^3)$.

Proof:

This algorithm traverses the array A from the n to the 1.

So, $i \leftarrow n-1\ to\ 1, j \leftarrow n\ to\ i+1, and\ k \leftarrow j+1\ to\ n,$ because we need to meet the conditions $2X[i] < X[i-1] + X[i+1]$ for all $i$. For example, if the first case is {4, 2, 2}. Then we can know 2 * 2 < 4 + 2, it meets the conditions. If we add a new number to this, {5,

4, 2, 2}, then we need to consider 5, 4, the first 2 is meet or not meet or 5, 4, the second 2 is meet or not meet, and we still need to consider 5, 2, 2 is meet or not. So, I designed a 3-layer loop to go through all the cases. And use the dynamic programming ways to fill the values at the $L[i][j]$. In the third loop, the max is to find the maximum value when k increases to n and meet the conditions $2X[i] < X[i-1] + X[i+1]$, then save to $L[i][j]$. When the loop of the third layer ends, what is stored in our L[I][j] must be the maximum value under the condition of the current i and j. Select the max value between the array $L[i][j]$ and $l$. When all three layers of the loop are finished, it will get the longest length $l$ which is the answer.

Problem 2:

For each line segment k in the set L, set its end point at y=0 as $X_k$, its corresponding endpoint on y=1 is also $X_k$. For the endpoint on y=0, there is an string S0={$X_1$...$X_n$}; for the endpoint on y=1, there is an string S1={$X_a$...$X_b$}(Uncertain order). To make each pair of line segments in subset L0 intersect, the points in S1 should be arranged in the reverse order of the points in S0. For the reversed string S2 of S1, you need to find the Longest Common Sequence of S2 and S0. LCS{S0, S2} is the largest subset of L in which every pair of segments intersects.

Algorithm

```
int LCS(String s0, String s2)
{
    int[][] c = new int[s0.length() + 1][s2.length() + 1];

    for (int i = 1; i <= s0.length(); i++)
        for (int j=1; j <= s2.length(); j++)
        {
            if(s0.charAt(i-1) == s2.charAt(j-1))
                c[i][j] = c[i-1][j-1] + 1;
            else if(c[i][j-1] > c[i-1][j])
                c[i][j] = c[i][j-1];
            else
                c[i][j] = c[i-1][j];
}
    return c [s0.length()][s2.length()];
}
```

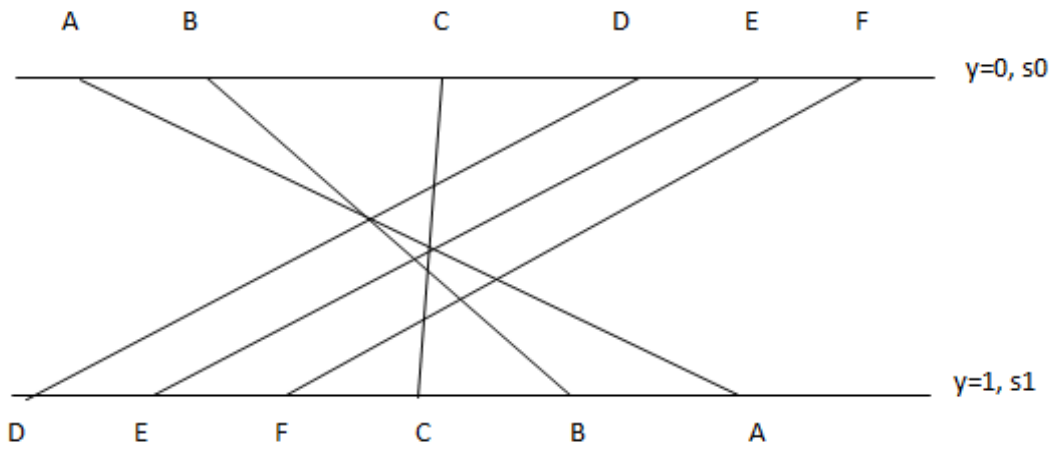
since each c[i,j] is calculated in constant time, and there are n*n elements in the array T(n) = O($n^2$).

Proof:

As the problem says, all 2n endpoints are distinct and we need to find the longest subset of L which every pair of segments intersects. So, we know that all line between y0 and y1 are unique, and call l = {A, B, C, D, E….}. Assume the line A end point at y=0 call A, end point at y=1 call A too. So, endpoint on y=0 is call s0 = {A, B,…..,N} and endpoint on y=1 use the same rule call s1 = {…}. When we reverse order of all endpoints on s1 to get the s2.

When c[i][j] = c[i-1][j-1] + 1; (s0[i] == s2[j]), assume c[i][j] > c[i-1][j-1] + 1, then we can get the paradoxical result that c[i-1][j-1] is not the longest.

When s0[i] != s2[j], assume c[i][j] > max(c[i-1][j], c[i][j-1]); then we can get c[i][j] > c[i-1][j], so the s0[i] it must in the c[i][j] sequence. Because s0[i] != s2[j], so s2[j] it must not in the sequence of c[i][j]. So, this assume can get the c[i][j] = c[i][j-1]. This result contradicts the hypothesis.




As this picture shows, when we reverse the s1 we can get s2 = {A, B, C, F, E, D}. s0 = {A, B, C, D, E, F}. So, the length that intersects the most is also the length of the longest common subset of S0 and S2. The longest L = 4 ({A, B, C, D}).

Problem 3:

For binary tree, the minimum number of rounds to broadcast a message to all nodes is equal to the height of the binary tree or height + 1(when height of left subtree = height of right subtree).

Algorithm

```
int sum_round(Tree k)
{
    int round = 0;
    if (k)                          //Tree exists
    {
        int left_tree = sum_round(tree -> left);        //Rounds of the left subtree
```

```
        int right_tree = sum_round(tree -> right);    //Rounds of the right subtree
        if (left_tree == right_tree)
            round = right_tree + 2;            //Height + 1 + root node
        else
            round = 1 + max(left_tree, right_tree);        //Height + root node
    }
    return round;
}
```

Since the algorithm traverses all the nodes of binary tree once, the running time is O(n), n is the number of nodes in the binary tree.

Prove:

This algorithm regards the left and right subtrees of tree k as two new complete trees, left_tree and right_tree. When the round numbers of left_tree and right_tree are not equal, the round number of tree k is equal to the maximum value of the round numbers of left_tree and right_tree + The root node, that is max(sum_round(left_tree),sum_round(right_tree)) + 1; when the round numbers of left_tree and right_tree are equal, the round number of tree k is equal to the maximum value of the round numbers of left_tree and right_tree + root node + 1. Because the root node needs an extra round to transmit information to another tree.

Regarding the root node of the subtree as a new tree, call the sum_round function on it to get the number of rounds required by the subtree. For the base case, that is, when tree A has only the root node, sum_roundA=(0+0+1)=1. Therefore, for tree k, sum_round(k) traverses all nodes and obtains the maximum round number of information transmission round

Problem 4:

For the DAG G = (V, E) it is a directed acyclic graph. We guess this G has n vertices and each vertices has $V \to R$ specify numbers and G has m edges. $L[v]$ is the longest directed path of the increasing sequence of numbers on the vertices that endpoint at v. Set the $C(v)$ for the each point V: for edges $u \to V, has \forall u \in C(V)$. Because in the title didn't specify that the edges of our directed graph have weighted, so we default it is 1.

For the topology, iterate through each point of the topology and find the L for each point. The largest $L$ which is this problem answer. $L[v] = \max_{u \in C(V)} L[u] + 1$.

```
    for all vertex v in E{
        for all vertex u ∈ C(V){
            L[v] = 1;
            If (a[v] > a[u]){
```

$$L[v] = \max(L[v], L[u] + 1)$$

}

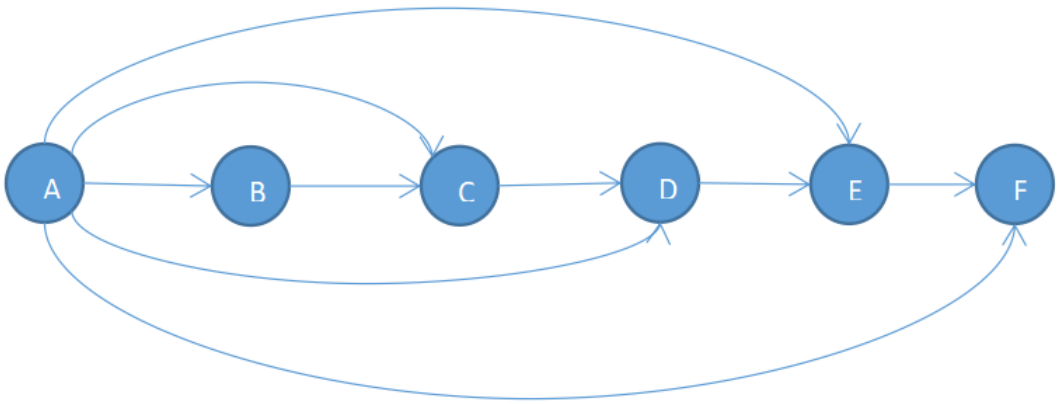
}

}

return max($L[v]$);

For this algorithm, we compute L for every points V in and cost $\sum_{i=1}^{n} C(i)$. and in that summation each edge of the graph is counted once, so it is O(E), therefore, the complexity is O(V*E).

For the special case of this question, as shown below:




In this case, the solution of the DAG in this problem can be equivalent to the longest increasing subsequence of sequence {A,B,C,D,E,F}. Similarly, for any sequence {A,.....,N}, we can get a DAG to describe it. Therefore, this problem is a generalization of the increasing subsequence problem that we have seen in class.