Name	Srijith (srijithka72@tamu.edu)
Problem Statement	THE “CONDITION SATISFIABILITY PROBLEM”
Algorithm	We need to define an algorithm to assign the values to x1 to xn variables such that it satisfies all the lead to conditions and false exists conditions. As stated in the hint, we start with assigning all the values to false for all the variables. Then when we start the program, We keep a map/ dictionary to keep track of the lead to clauses where the variable xi is appearing. So this map has a key as Xi and Values is a list of values of lead to clauses where this variable is present. We start iterating the lead to conditions, and when we encounter a true condition on LHS, we check if the variable on the RHS is set to true or false. If it is false, we set it to true and then we call a recursive function on all the lead to clauses in which the variable on the RHS is present on the LHS part in the already processed lead to conditions. This function recursively iterates the condition on the LHS and does the same thing. We eventually stop when we process all the lead to conditions and the base condition is that the variable is already set to true, we never check the clauses in which this variable is present. When all the leads to clauses are evaluated, we verify the false must exist conditions. If any of the conditions fails, we report that there is no solution for the problem. If all conditions are satisfied, we report the values back of all the variables.
Pseudocode	`  // recursive_call does the same check as calc` `def recursive_call(mp, j, lead_to_list_sizes, leads, var_list):` `    pos_lhs = 1` `    if(var_list[leads[j][-1]] == 1):` `      return` `    for k in range(lead_to_list_sizes[j]):` `      pos_lhs = pos_lhs and var_list[leads[j][k]]` `      if(pos_lhs == 0):`

```
        break
    res = 0
    if(pos_lhs == 1):
      if(var_list[leads[j][-1]] == 1):
        res = 1
      var_list[leads[j][-1]] = 1
    else:
      return

    if leads[j][-1] not in mp or res == 1:
      return

    for k in mp[leads[j][-1]]:
      if(var_list[leads[k][-1]] == 1):
        continue
      recursive_call(mp, k, lead_to_list_sizes, leads,
var_list)




def calc(n, lead_size, false_size, lead_to_list_sizes,
false_must_exist_sizes, leads, falses)
    // lead_size is the number of lead to conditions
    // false_size is the number of flase must exist conditions
    // lead_to_list_sizes is the size of each lead to
ccondition
    // false_must_exist_sizes is the size of each false must
exist
    // leads contains each variables in lead to conditions
    // falses contains all the variables in false must exist
conditions
    var_list = [0] * n // Here n is the number of variables
given
    mp = {} // Initializing the empty map


    for i<-0 to lead_size: // iterating each lead to
conditions
        lhs = 1
        for j<-0 to lead_to_list_sizes[i]: // iterating all
the variables in lead to conditions
```

	`` lhs = lhs & var_list[leads[i][j]] // Determining the LHS with & operation mp[leads[i][j]].insert(i) // to keep track of element xi appears in which clause for future computing res = 0 if lhs == 1: // if lhs is true if(var_list[leads[i][-1]] == 1): // to keep track whether RHS is true, if it is we don't need to evaluate res = 1 var_list[leads[i][-1]] else: continue if leads[i][-1] not in mp or res == 1: // res is used here and if there is no clause in which this varaible is present we skip continue for j in mp[leads[i][-1]]: // iterate the clauses in which the concerned variable is present if(var_list[leads[j][-1]] == 1): // i that clause RHS is set, we don't need to continue recursive_call(mp, j, lead_to_list_sizes, leads, var_list) // We perform the same LHS check and verify with recursive call ```
Proof of Correctness	To prove this algorithm, we will use induction. Since we are greedy, we assign the false values to all the variables. This allows us to get all the false must exist conditions to be satisfied. This is our priority, otherwise, we won't find the solution. We assign true to variables if and only if, the LHS part of the Lead to conditions evaluate to true. Otherwise this lead to condition will fail. Consider the base case, where there is a degenerate lead to condition, we need to set that particular variable to true. Then we iterate through the false must exist conditions and check whether the collection exists. For this case, if

	there is any false condition, then there will be no solution. If there are no false must exist conditions then the value is evaluated correctly. Consider in the base case when there is no degenerate lead to conditions, then all the variables will be set to false and it will satisfy the false conditions. Now consider the solution is correct(like we find or we don't find) for K lead to conditions. Now we need to find the solution for k + 1 lead to conditions. So now the condition which might be added can be degenerate and at that time, we set the variable xi to true and evaluate all the clauses(before k clauses) and try to reevaluate to see anything sets to true and if it does, we check for RHS to be true. If none of the conditions are affected by using the (k + 1) condition, then we set it to true , without touching the rest of the k variables. We at the end check the false must exist conditions to verify whether the solution we got after processing all the k + 1 lead to conditions, to verify it is valid or not. So we can prove we can get the solution for the k + 1 lead to condition. Thus by induction hypothesis, we prove that the given algorithm is correct.
Time Complexity	The time complexity can be evaluated as follows, Consider n be the size of the lead to conditions times number of variables in each lead to condition and the number of false conditions times the size of each false condition. Let p be the number of the lead to conditions. Since we are iterating each lead to conditions and in each iteration we check the clauses in which the RHS is present, in the worst case, we check all the p lead to conditions. So time complexity is O(n * p).