```
import argparse
from itertools import permutations
# import ortools
# from ortools.constraint_solver import pywrapcp
import random
import copy
import math
import time

"""
======================================================================
  Complete the following function.
======================================================================
"""

def solve(num_wizards, num_constraints, wizards, constraints):
    """
    Write your algorithm here.
    Input:
        num_wizards: Number of wizards
        num_constraints: Number of constraints
        wizards: An array of wizard names, in no particular order
        constraints: A 2D-array of constraints,
                     where constraints[0] may take the form ['A', 'B', 'C']i

    Output:
        An array of wizard names in the ordering your algorithm returns
    """

    #Cost Function - computes how many constratints failed for a specific solution
    def cost(sol,num_constraints,constraints):
        constraints_satisfied = 0
        constraints_failed = []
        output_ordering_map = {k: v for v, k in enumerate(sol)}
        for c in constraints:

            m = output_ordering_map # Creating an alias for easy reference

            wiz_a = m[c[0]]
            wiz_b = m[c[1]]
            wiz_mid = m[c[2]]

            if (wiz_a < wiz_mid < wiz_b) or (wiz_b < wiz_mid < wiz_a):
                constraints_failed.append(c)
            else:
                constraints_satisfied += 1
        return num_constraints - constraints_satisfied

    #Helper function that swaps one element from a given solution list
    def neighbors(sol):
        wiz1 = random.randint(0,num_wizards-1)
        wiz2 = random.randint(0,num_wizards-1)

        new_sol = copy.copy(sol)
        temp = new_sol[wiz1]
        new_sol[wiz1] = new_sol[wiz2]
        new_sol[wiz2] = temp

        return new_sol

    #function that computes the accepted probability
    #based on the old cost and new cost
    #and using an exponent function
    def acceptance_probability(old_cost,new_cost,T):
        exponent = (old_cost - new_cost) / T

        try:
            ans = math.exp(exponent)
        except OverflowError:
            ans = float('inf')
        return ans

    #deals with naive base cases, inputs a solution based on the when do names appear in
first.
    def naive(solution, num_constraints,constraints):
        output_ordering_map = {k: v for v, k in enumerate(solution)}
        ret = []

        for c in constraints:
            if c[0] not in ret:
                ret.append(c[0])
            if c[1] not in ret:
                ret.append(c[1])
            if c[2] not in ret:
                ret.append(c[2])

        return ret

    #Simulated annealing function.
    def anneal(solution,solution2, num_constraints, constraints):


        old_cost = cost(solution,num_constraints,constraints)
        old_cost2 = cost(solution2,num_constraints,constraints)

        T = 1.0
        T_min = 0.000001
        alpha = 0.988
        start_time = time.time()
        while T > T_min:
            i = 1

            while i <= 1000:
                new_solution = neighbors(solution)
                new_cost = cost(new_solution,num_constraints,constraints)

                new_solution2 = neighbors(solution2)
                new_cost2 = cost(new_solution2,num_constraints,constraints)


                if new_cost == 0:
                    print("Minutes It Took To Solve: " + (str(time.time() - start_time/
60.0)))
                    return new_solution,new_cost
                if new_cost2 == 0:
                    return new_solution2,new_cost2


                ap0 = acceptance_probability(old_cost, new_cost, T)
                ap2 = acceptance_probability(old_cost2, new_cost2, T)


                if ap0 > random.random():
                    solution = new_solution
                    old_cost = new_cost

                if ap2 > random.random():
                    solution2 = new_solution2
                    old_cost2 = new_cost2
                i += 1
            T = T*alpha



        print("Minutes It Took To Solve: " + str((time.time() - start_time) /60.0))
        if old_cost < old_cost2:
            return solution, old_cost
        return solution2, old_cost2

    s = copy.copy(wizards)
    s2 = copy.copy(wizards)

    sol = naive(s,num_constraints,constraints)
    if cost(sol,num_constraints,constraints) == 0:
        print("constraints failed: 0")
        return sol


    random.shuffle(s)
    random.shuffle(s2)


    ret = anneal(s,s2,num_constraints,constraints)
    s = ret[0]
    print("Round: " + str(1))
    print("current ret constraints failed: {0}".format(ret[1]))
    print("current ret solution: {0}".format(ret[0]))

    #10 calls to the anneal function to converge on the best answer
    for i in range(2,11):
        if ret[1] == 0:
            break
        random.shuffle(s2)
        new_ret = anneal(s,s2,num_constraints,constraints)
        s = new_ret[0]
        print("Round: " +str(i))
        print("current ret constraints failed: {0}".format(new_ret[1]))
        print("current ret solution: {0}".format(new_ret[0]))
        if new_ret[1] < ret[1]:
            ret = new_ret
    print("constraints failed: {0}".format(ret[1]))


    return ret[0]




"""
======================================================================
   No need to change any code below this line
======================================================================
"""

def read_input(filename):
    with open(filename) as f:
        num_wizards = int(f.readline())
        num_constraints = int(f.readline())
        constraints = []
        wizards = set()
        for _ in range(num_constraints):
            c = f.readline().split()
            constraints.append(c)
            for w in c:
                wizards.add(w)

    wizards = list(wizards)
    return num_wizards, num_constraints, wizards, constraints

def write_output(filename, solution):
    with open(filename, "w") as f:
        for wizard in solution:
            f.write("{0} ".format(wizard))

if __name__=="__main__":
    parser = argparse.ArgumentParser(description = "Constraint Solver.")
    parser.add_argument("input_file", type=str, help = "___.in")
    parser.add_argument("output_file", type=str, help = "___.out")
    args = parser.parse_args()

    num_wizards, num_constraints, wizards, constraints = read_input(args.input_file)
    solution = solve(num_wizards, num_constraints, wizards, constraints)
    write_output(args.output_file, solution)
```