Group Name: TheNoobs

Group Member: Ruijie Zhou, Zhiyao Jiang, Yunjie Zhang

The basic idea is to represent the problem with a directed graph, and then use a topological sort to traverse the whole graph and thus we would get the correct ordering of all wizards. The basic idea is to use backtracking to find out the correct ordering.

For the graph, we would like to generate a directed graph according to the constraints. Each wizard is represented as a vertex in the graph. We define an edge as the relationship between age of wizards: the wizard with the smaller age directs to the wizard to larger age. For instance, if A is older than B, then there is an edge pointing from B to A. In this sense, there should not be any cycle in the graph because otherwise it means that someone with a smaller age is older than someone older.

We add the edges in two ways, if there is a constraint “A B C”, then C’s age is not between A and B, which means C’s age is either smaller than that of A and B, or C’s age is larger than that of A and B. Therefore, we have two cases, one graph with edges CA and CB, and one graph with AC and BC. We would use both cases to guarantee a solution.

In the beginning, we only have an empty graph with no vertices and edges. Then, we add all vertices, which are wizards, into the graph. Now, we have a graph with all vertices.

Then iterate through all constraints. For each constraint, we have two cases, with edges CA, CB or AC, BC. We make a deep copy of the original graph, and now we have two graphs exactly the same. We add two pairs of edges to two graphs respectively, and call them graph1 and graph2. Check if there is a cycle in graph2. If so, then it means our current assignments are in conflict and thus this graph should be discarded. If there is no cycle in graph2, then it means this graph can be used as a backup. Store this graph2 in the backup list of all graphs called “branches” and store the corresponding constraint index in the list “constraintIndices” for backup.

Now we check the graph1. If there is a cycle in graph1, it means we have incorrect assignment. Therefore, we discard this graph and take the latest backup graph with latest constraint index from the backup lists. And we will use this backup for further operation.

Finally, if we reach the end of the list of constraints, it means we have got a graph satisfying all constraints, and thus this graph contains the solution desired. Then we jump out of the iteration and run topological sort to get the correct ordering of the wizards.

Code is in next page:

```
import argparse
from constraint import *
import operator
import collections
import copy

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

GRAY, BLACK = 0, 1

def checkCycle(g):
        """Return True if the directed graph g has a cycle.
        g must be represented as a dictionary mapping vertices to
        iterables of neighbouring vertices. For example:

        >>> cyclic({1: (2,), 2: (3,), 3: (1,)})
        True
        >>> cyclic({1: (2,), 2: (3,), 3: (4,)})
        False

        """
        path = set()
        visited = set()

        def visit(vertex):
                if vertex in visited:
```

```
            return False
        visited.add(vertex)
        path.add(vertex)
        for neighbour in g.get(vertex, ()):
            if neighbour in path or visit(neighbour):
                return True
        path.remove(vertex)
        return False

    return any(visit(v) for v in g)

def topological(graph):
    order = collections.deque()
    enter = set(graph)
    state = {}

    def dfs(node):
        state[node] = GRAY
        for k in graph.get(node, ()):
            sk = state.get(k, None)
            if sk == GRAY: raise ValueError("cycle")
            if sk == BLACK: continue
            enter.discard(k)
            dfs(k)
        order.appendleft(node)
        state[node] = BLACK

    while enter:
        dfs(enter.pop())
    return order



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']
Output:
    An array of wizard names in the ordering your algorithm returns
"""
graph = {}
branches = []
constraintIndices = []

for wizard in wizards:
    graph[wizard] = set()
index = 0
while (index < len(constraints)):
   constraint = constraints[index]
    a = constraint[0]
    b = constraint[1]
    c = constraint[2]

    graph2 = copy.deepcopy(graph)

    graph[c].add(a)
    graph[c].add(b)
    graph2[a].add(c)
    graph2[b].add(c)

    if not checkCycle(graph2):
        branches.append(graph2)
```

```
                constraintIndices.append(index)

            if not checkCycle(graph):
                index += 1
            else:
                graph = branches.pop()
                index = constraintIndices.pop() + 1

        result = topological(graph)
        return list(result)

""""
==========================================================
====================
    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)
```