Algorithm

SAT Reduction: We reduced this problem to SAT. Each constraint (W1 W2 W3) means that W3 cannot be between W1 and W2. In other words:

$$\neg(W1 < W3 < W2 \lor W2 < W3 < W1) \equiv \neg(W1 < W3 < W2) \land \neg(W2 < W3 < W1)$$
$$\equiv \neg(W1 < W3 \land W3 < W2) \land \neg(W2 < W3 \land W3 < W1)$$
$$\equiv (\neg(W1 < W3) \lor \neg(W3 < W2)) \land (\neg(W2 < W3) \lor \neg(W3 < W1))$$

Notice that the last step of this simplification results in two clauses in CNF, where each (Wi < Wj) can be thought of as its own variable, A (so, (Wj < Wi) would be ¬A in this construction). Thus, each constraint is transformed into two separate CNF clauses.

We must also take care to ensure transitivity in our constraints as follows:

$$(W1 < W2) \land (W2 < W3) \Rightarrow (W1 < W3) \equiv \neg((W1 < W2) \land (W2 < W3)) \lor (W1 < W3)$$
$$\equiv \neg(W1 < W2) \lor \neg(W2 < W3) \lor (W1 < W3)$$

The last step of this simplification results in a clause in CNF. In order to maintain transitivity in all cases, we must create such clauses for all (Wi, Wj, Wk) ordered triplets, which means we add n * n-1 * n-2 additional clauses, where n = number of wizards.

Graph Construction and Topological Sort: We use a SAT solver to find variable assignments that satisfy all of these clauses. We then construct a graph whose vertices are each of the wizards in our input file. There is a directed edge from Wi to Wj iff (Wi < Wj) is true or (Wj < Wi) is false.

Observe that as an optimal ordering must exist that satisfies all the constraints, the graph that results must necessarily be a DAG (if it weren't, then no such optimal ordering could exist). By simply doing a topological ordering of the graph that we construct, we are able to find an optimal ordering of the wizards such that all of the constraints are satisfied.

Greedy Solver: For files with 200+ wizards, a SAT reduction is too slow. Instead, we use a greedy approximation, where on each iteration of the greedy algorithm, we choose the swap of two wizards (among all possible swaps) in the current ordering that results in the fewest number of remaining unsatisfied constraints. Once fewer than 15% of the constraints remain unsatisfied by the current ordering and the algorithm has reached a local minimum, the algorithm returns the current ordering. If the algorithm is at a local minimum with > 15% unsatisfied constraints, it slightly shuffles the current ordering by a random amount and continues.

Runtime: As there are 2*(#constraints) + n(n-1)(n-2) clauses for the SAT solver to satisfy, the runtime of the SAT Solver algorithm is $O(2^{2*(\#constraints) + n(n-1)(n-2)})$. In the worst case, the greedy algorithm examines all possible orderings of the wizards, which takes O(n!) time.

Non-standard Libraries Instructions: We used pycosat (https://pypi.python.org/pypi/pycosat) and networkx (https://pypi.python.org/pypi/networkx/2.0), with python3. To install pycosat, use "pip install pycosat"; to install networkx, use "pip install networkx". We used these libraries because they were well-documented Python packages and seemed to work until we hit 200 constraints.

Code

```
import argparse
import pycosat
import itertools
import sys
import random
import networkx as nx


def solve(num_wizards, num_constraints, wizards, constraints, output_file):
    """
    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
    """
    if num_wizards < 200:
        opt_ordering = sat_reduction(wizards, constraints)
    else:
        opt_ordering_map = greedy_solver(wizards, constraints, output_file)
        # as the opt_ordering_map is a dict of (name : position), we need to convert it to an ordered
list
        rev_dict = {}
        for key in opt_ordering_map:
            rev_dict[opt_ordering_map[key]] = key
        opt_ordering = []
        for i in range(num_wizards):
            opt_ordering.append(rev_dict[i])
        return opt_ordering
    return opt_ordering


def sat_reduction(wizards_list, constraints):
    # all the clauses for SAT
    cnf = []

    # maps inequality to var used in SAT (note clark < bruce is represented as (clark, bruce);
variables are 1, 2, etc.)
    inequality_to_var = {}

    # same map except reversed
    var_to_inequality = {}
```

```
# create the clauses from the constraints
for constraint in constraints:
    wizard1, wizard2, wizard3 = constraint[0], constraint[1], constraint[2]
    if wizard1 == wizard2 or wizard2 == wizard3 or wizard1 == wizard3:
        continue

    # clause 1: NOT (wizard1 < wizard3) or NOT (wizard3 < wizard2)
    var1 = get_or_make_var(wizard1, wizard3, inequality_to_var, var_to_inequality)
    var2 = get_or_make_var(wizard3, wizard2, inequality_to_var, var_to_inequality)
    cnf.append([-var1, -var2])

    # clause 2: NOT (wizard2 < wizard3) or NOT (wizard3 < wizard1)
    var3 = get_or_make_var(wizard2, wizard3, inequality_to_var, var_to_inequality)
    var4 = get_or_make_var(wizard3, wizard1, inequality_to_var, var_to_inequality)
    cnf.append([-var3, -var4])

# create the transitivity clauses
for perm in itertools.permutations(wizards_list, 3):
    # NOT (wizard1 < wizard2) or NOT (wizard2 < wizard3) or (wizard1 < wizard3)
    wizard1, wizard2, wizard3 = perm[0], perm[1], perm[2]
    var1 = get_or_make_var(wizard1, wizard2, inequality_to_var, var_to_inequality)
    var2 = get_or_make_var(wizard2, wizard3, inequality_to_var, var_to_inequality)
    var3 = get_or_make_var(wizard1, wizard3, inequality_to_var, var_to_inequality)
    cnf.append([-var1, -var2, var3])

# solve the SAT
solution = pycosat.solve(cnf)

# make the directed graph
DG = nx.DiGraph()
DG.add_nodes_from(wizards_list)

# use the solution to SAT to figure out the graph edges
for var in solution:
    if var < 0:
        # the inequality is false, so the reverse statement is true, so add edge w2 -> w1
        inequality = var_to_inequality[-var]
        DG.add_edge(inequality[1], inequality[0])
    else:
        # the inequality is true, so add edge from w1 -> w2
        inequality = var_to_inequality[var]
        DG.add_edge(inequality[0], inequality[1])
# topological sort
top_sort = nx.topological_sort(DG)
opt_ordering = []
for wizard in top_sort:
    opt_ordering.append(wizard)
```

```
    return opt_ordering


def get_or_make_var(wizard1, wizard2, inequality_to_var, var_to_inequality):
    """
    Given wizard1, wizard which form the inequality wizard1 < wizard2, checks to see if there is a
variable
    already mapped to this inequality or wizard2 < wizard1. If not, makes one
    :param wizard1:
    :param wizard2:
    :param inequality_to_var:
    :param var_to_inequality:
    :return: the variable number
    """
    cur_inequality = (wizard1, wizard2)
    rev_inequality = (wizard2, wizard1)
    if cur_inequality in inequality_to_var:
        # the mapping already exists
        return inequality_to_var[cur_inequality]
    elif rev_inequality in inequality_to_var:
        # the reverse mapping exists, so return the negation of the inequality
        return -1 * inequality_to_var[rev_inequality]
    else:
        # the mapping doesn't exist, so make a new variable, which is 1 greater than the max
number used so far
        if len(var_to_inequality) == 0:
            new_var = 1
        else:
            new_var = max(var_to_inequality) + 1
        inequality_to_var[cur_inequality] = new_var
        var_to_inequality[new_var] = cur_inequality
        return new_var



def greedy_solver(wizards_list, constraints, output_name):
    # generate a list of all possible swaps, i.e. pairs of indices
    possible_swaps = list(itertools.combinations([a for a in range(num_wizards)], 2))
    # randomly shuffle list
    random.shuffle(wizards_list)
    # convert wizard list to map using positions in random shuffling order
    node_map = {k: v for v, k in enumerate(wizards_list)}
    # baseline number of failed constraints
    failures = constraints_unsatisfied_map(node_map, constraints)
    seen_failures_map = {}
    best_map_ever_seen = node_map
    best_failures = len(constraints)
    try:
        while failures > 0:
```

```
        rev_dict = {}
        for key in node_map:
            rev_dict[node_map[key]] = key
        best_map = node_map
        for i in possible_swaps:
            # for each swap, we make the swap and determine if this new ordering results in the
fewest failures
            swap_a = rev_dict[i[0]]
            swap_b = rev_dict[i[1]]
            cur_map = dict(node_map)
            cur_map[swap_a], cur_map[swap_b] = cur_map[swap_b], cur_map[swap_a]
            cur_fail = constraints_unsatisfied_map(cur_map, constraints)
            if cur_fail < failures:
                failures = cur_fail
                best_map = cur_map
        node_map = best_map
        if failures < best_failures:
            # keep track of the best ever seen ordering in case of early termination
            best_map_ever_seen = dict(node_map)
            best_failures = failures
        if failures in seen_failures_map:
            seen_failures_map[failures] += 1
            # if we have seen this particular number of failures repeatedly, we are stuck in a local
minimum
            if seen_failures_map[failures] > 3:
                fail_perc = (failures * 1.0) / len(constraints)
                if fail_perc > 0.15:
                    # if more than 15% of the constraints remain unsatisfied, choose a random
number of swaps
                    num_random_swaps = random.randint(5, 25)
                else:
                    # if <= 15% of the constraints remain unsatisfied, just give up
                    return node_map
                for i in range(num_random_swaps):
                    # make the random number of swaps to get out of the local minimum
                    rev_dict = {}
                    for key in node_map:
                        rev_dict[node_map[key]] = key
                    swap = random.randint(0, len(possible_swaps) - 1)
                    swap = possible_swaps[swap]
                    swap_a = rev_dict[swap[0]]
                    swap_b = rev_dict[swap[1]]
                    node_map[swap_a], node_map[swap_b] = node_map[swap_b],
node_map[swap_a]
                    failures = constraints_unsatisfied_map(node_map, constraints)
                seen_failures_map = {}

        else:
```

```
                seen_failures_map[failures] = 1
            print(failures)
        return node_map
    except KeyboardInterrupt:
        # if the approximation is taking too long to get below 15%, just write the best known
ordering to file
        rev_dict = {}
        for key in best_map_ever_seen:
            rev_dict[best_map_ever_seen[key]] = key
        lst = []
        for i in range(num_wizards):
            lst.append(rev_dict[i])
        write_output(output_name, lst)
        sys.exit(0)


def constraints_unsatisfied_map(node_map, constraints):
    num_failed = 0
    for constraint in constraints:
        wiz_a = node_map[constraint[0]]
        wiz_b = node_map[constraint[1]]
        wiz_mid = node_map[constraint[2]]
        if (wiz_a < wiz_mid < wiz_b) or (wiz_b < wiz_mid < wiz_a):
            num_failed += 1
    return num_failed


"""
==============================================================================
=
   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, args.output_file)
    write_output(args.output_file, solution)
```