Francis' answer to the Paintshop Problem

How to approach

One of the approaches to solving this problem is to retrieve all combinations from the space created by *2N of colors* and *M of customers*. The point is to efficiently find possible combinations **using given constraints**.

Here I introduced a *trie*, where each depth of the trie indicates a customer and each node represents a preferred color batch of a customer. When construction of a trie, I sorted it in advance in order to efficiently retrieve as follows:

- At each depth, sort the batches by *glossiness* and *color* (smaller index of color and glossy first)
- Sort customers by the size of batches each customer has (smaller size first)

Then, we can easily solve this using a recursive function (namely, with the help of call stack: the trie search is performed while winding and unwinding of the stack). An answer firstly satisfying constraints of the problem will be the answer we are looking for.

Note: A drawback of this solution is that it uses call stack. (But we can get a cleaner and simpler code) Surely, an overflowing stack should not happen. However, the limits given in the problem for both small dataset and large dataset are enough to cover with this solution, in general. However, the Python interpreter limits the depths of recursion to avoid infinite recursions (by default `1000`). The default value is sufficient, but I conservatively adjusted this a bit in my code.

How to run

```
# Python 3.6 or later is recommended
# INPUT_FILE in problem-defined format
$ python3 answer.py INPUT_FILE

# self-prepared case test
$ python3 test.py -v
```

Some comments with code

```
# Note: here assumed that ONLY VALID DATA will be given.
# In order to focus on the algorithm itself, I skipped all validations of
# input files and command-line arguments.

def parse_input(lines):
    """parse the given file lines and sort it as a pre-process
    """
    lines = lines[::-1]
    data = []
    for _ in range(int(lines.pop())):
        num_colors = int(lines.pop())
        num_customers = int(lines.pop())
        customers = []
        for _ in range(num_customers):
            pairs = [ int(x) for x in lines.pop().split() ]
            customer = zip(pairs[1::2], pairs[2::2])
            customers.append(sort_customer(customer))
        data.append((num_colors, sort_customers(customers)))
    # data in forms of [(NUM_COLORS, [[(COLOR, MATTE), ...], ...]), OTHER CASE, ...]
    return data
```

```python

def sort_customer(customer):
    """sort 'smaller color index' and 'glossy' first
    among (X: color, Y: glossy | matte) pairs that a customer has.
    """
    return sorted(sorted(customer, key=lambda x: x[0]), key=lambda x: x[1])

def sort_customers(customers):
    """sort 'smaller length of color set' first among all the customers
    """
    return sorted(customers, key=len)

def traverse_trie(trie, track, depth, num_customers, num_colors):
    if depth == num_customers: return track
    for color, matte in trie[depth]:
        # skip searching this path if the same color(but different glossiness) is in track
        if (color, matte ^ 1) in track: continue

        # required to clone 'track' to be branched out here
        new_track = set(track)
        if not (color, matte) in track: new_track.add((color, matte))

        # skip tracing further if determined as not a solution
        if len(new_track) > num_colors: continue

        # using recursive call, trace all of possible combinations until the answer is found
        result = traverse_trie(trie, new_track, depth+1, num_customers, num_colors)
        if result: return result    # exit and unwind stack immediately if found a answer
    return None

def evalute_result(result, nth, num_colors):
    output = f'Case #{nth}: '
    if not result:
        output += 'IMPOSSIBLE'
    else:
        matte = [ '1' if (i, 1) in result else '0' for i in range(1, num_colors+1) ]
        output += ' '.join(matte)
    return output

def solve_each_case(nth, case):
    num_colors, trie = case
    result = traverse_trie(trie, set(), 0, len(trie), num_colors)
    return evalute_result(result, nth, num_colors)

def run(filename):
    with open(filename, 'r') as f:
        data = parse_input([ line.strip() for line in f if line.strip() ])
        return [ solve_each_case(i+1, case) for i, case in enumerate(data) ]


if __name__ == '__main__':
    import sys
    sys.setrecursionlimit(5000)   # adjust stack size for the reason above
    print('\n'.join(run(sys.argv[1])))
```