```python
"""
Name: Hongda Li
Class: cse 417 winter 2020

This is a homework assignment for cse 417 winter 2020, it's implementing the Gale-Shapley
algorithm, problem 4.

It needs python 3.6 or above to run.

* The core part is: the "produce_stable_match(M, W)" function.

Here is the console outputs produced when running it with the given example:
----------------------------------------------------------------------------------------------------------------------
----
3 proposes to 0: [0,-1] Accepted
2 proposes to 0: [0,3] Accepted
3 proposes to 1: [1,-1] Accepted
1 proposes to 0: [0,2] Rejected
1 proposes to 1: [1,3] Rejected
1 proposes to 3: [3,-1] Accepted
0 proposes to 2: [2,-1] Accepted
Results: [2, 3, 0, 1]
"""

from typing import List


def produce_stable_match(M:List[List[int]], W:List[List[int]], verbo=True):
    """
    Function will produce a list of tuple representing the stable matching between the set M, W.

    The proposing side is: M.
    :param M
        A n by n matrix,
        the i th row denotes the preference list of m_i
    :param W
        A n by n matrix,
        the W th row denotes the preference list of w_i
    :param verbo:
        Set to False to stop printing out the trace.
    :return
    A list of tuple in the following format:
    [(m_1, W_1), (m_2, w_2)... (m_n, w_2)],
    """
    convert(M) # This is only for sanity checks.
    W = convert(W)
    keypairs = [(I, NodeM(I, M, W)) for I in range(len(M))]
    M = dict(keypairs)
    M_free = [kv[1] for kv in keypairs]
    while len(M_free) > 0:
        m = M_free.pop()
        while not m.propose(verbo=verbo):
            m.increment()
        m = m.engage()
        if m is None:
            continue
        M_free.append(M[m])
    return [M[I].last_propose() for I in range(len(M))]


def convert(M: List[List[int]]):
    """
        This function takes in the preference matrix and convert the value into a dictionary making he look up of
        preference value constant.
        [
            [[<preference list>], [preference list reverse map], current match element/next proposing element]],
            [[<preference list>], [preference list reverse map], current match index/next proposing element]],
            ...
        ]
```

```
        Example:
        M := [[0, 1], [1, 0]]
        then after the conversion, we will have:
        [
            [[0, 1], {0:0, 1:1}, None], # -1 means m_1 hasn't proposed to anyone yet, 0 would mean m_1 has proposed
to
                                        # the first w in its reference list.
            [[1, 0], {1:0, 0:1}, None]
        ]

    :param M:
        A 2d array specified by the problems, I call it M but it really could be M or W.
    :return
        A array of inner arrays, where each inner arrays strictly has the length of 3.
        I will this data structure: a look up table.
    >>> convert([[1, 0], [0, 1]])
    ...
    >>> convert([[2, 0, 1], [0, 2, 1], [1, 0, 2]])
    ...
    >>> convert([[2, 0, 1], [0, 2, 2], [1, 0, 2]])
    """
    output = [[I, bijective_convert(I), None] for I in M]
    return output

def bijective_convert(arr: List[int])-> List[int]:
    """
    Function takes in an array with ints, ranging from 0 to n-1, without repeating elements.
    :param arr:
        int array.
    :return:
        arr such that:
        arr_returned[I] returns the position of I inside arr.
    >>> bijective_convert([2, 0, 1])
    """
    res = [None]*len(arr)
    for V, I in zip(arr, range(len(arr))):
        assert res[V] is None, "Repeated element in preference list. "
        assert V < len(res) and V >= 0, "Invalid value in the preference list. "
        res[V] = I
    return res

class NodeM:
    """
        This class models the nodes in the bipartite graph, it doesn't really depend on whether the node is in W, or
M,
        but it will makes things better for the codes.
    """
    def __init__(self, id: int, M: List[List[int]], W_tbl: List[List]):
        """
        :param id:
            The id of the element m
        :param M:
            The preferential matrix for all m.
        :param W_tbl:
            The reference table produce by convert method for all w in W.
        """
        self.__ID = id
        self.__W = W_tbl
        self.__M = M
        self.__TopChoice = 0

    def propose(self, verbo=True):
        """
            Method will propose to its top choice.
            It will print the trace.
        :param verbo:
        Set it to False to stop printing out the trace.
        Default to True.
        :return:
```

```python
            True if accepted by top choice.
            False if Rejected by top choice.
        """
        assert self.__TopChoice < len(self.__M[0]), "Preference List runs out, grave Error."
        w = self.__M[self.__ID][self.__TopChoice]
        m_competitor = self.__W[w][2]
        trace= f"{self.__ID} proposes to {w}: [{w},{-1 if m_competitor is None else m_competitor}]"
        if m_competitor is None:
            trace += " Accepted"
            if verbo:
                print(trace)
            return True
        competitor_Rank = self.__W[w][1][m_competitor]
        this_Rank = self.__W[w][1][self.__ID]
        assert competitor_Rank != this_Rank, "Internal error. "
        result = this_Rank < competitor_Rank
        if verbo:
            print(trace + (" Rejected" if not result else " Accepted"))
        return result


    def engage(self):
        """
        Method will engage to its top choice, changing the reference table of W.
        :return:
        None if the top choice partner doesn't have any previous partner.
        else
            returns the id of w's previous partner.
        """
        w = self.__W[self.__M[self.__ID][self.__TopChoice]]
        previous_partner_id = w[2]
        w[2] = self.__ID
        self.__TopChoice += 1
        return previous_partner_id

    def last_propose(self):
        """
        :return:
        The ID of the last proposed w.
        """
        return self.__M[self.__ID][self.__TopChoice - 1]

    def increment(self):
        self.__TopChoice += 1


if __name__ == "__main__":
    print("...")
    import doctest
    doctest.testmod()

    print("Try and construct an example for testing the NodeM class. ")
    M = [[1, 0], [1, 0]]
    W = [[0, 1], [1, 0]]
    W_tbl = convert(W)
    m1 = NodeM(0, M, W_tbl)
    m2 = NodeM(1, M, W_tbl)
    assert m1.propose()
    assert m1.engage() is None
    assert m2.propose()
    assert m2.engage() == 0
    print("Ok test passed")

    print(f"Result: {produce_stable_match(M, W)}")
    M = [[2, 0, 1], [2, 0, 1], [2, 1, 0]]
    W = [[2, 0, 1], [0, 2, 1], [2, 0, 1]]
    print(f"Result: {produce_stable_match(M, W)}")
    M = [[3, 1, 0, 2], [3, 0, 1, 2], [3, 1, 2, 0], [0, 3, 2, 1]]
    W = [[3, 0, 2, 1], [3, 1, 2, 0], [0, 1, 3, 2], [3, 0, 2, 1]]
```

```
print(f"Result: {produce_stable_match(M, W)}")
M = [[1, 3, 0, 2], [3, 0, 2, 1], [2, 3, 1, 0], [2, 0, 3, 1]]
W = [[1, 2, 0, 3], [0, 3, 2, 1], [1, 2, 3, 0], [0, 1, 2, 3]]
print(f"Result: {produce_stable_match(M, W)}")
M = [
        [4, 2, 0, 1, 3], [0, 1, 4, 2, 3],\
        [4, 0, 1, 3, 2],\
        [1, 4, 3, 0, 2],\
        [0, 1, 3, 4, 2],\
    ]
W = [
    [2, 3, 4, 0, 1],\
    [2, 4, 0, 3, 1],\
    [1, 4, 2, 3, 0],\
    [3, 0, 1, 4, 2],\
    [0, 3, 1, 2, 4]
]
print(f"Result: {produce_stable_match(M, W)}")
M = [[2, 1, 3, 0], [0, 1, 3, 2], [0, 1, 2, 3], [0, 1, 2, 3]]
W = [[0, 2, 1, 3], [2, 0, 3, 1], [3, 2, 1, 0], [2, 3, 1, 0]]
print(f"Results: {produce_stable_match(M, W)}")
```