Problem 1 (10 points):
Let \( I = (M, W) \) be an instance of the stable matching problem. Suppose that the preference lists of all \( m \in M \) are identical, so without loss of generality, \( m_i \) has the preference list \([w_1, w_2, \ldots, w_n]\). Show that there is a unique solution to this instance.

Solution 1:
The stable matching always exists. Without losing generality, we assume in one of these stable matchings \( m_i \) is matched with \( w_i \) for all \( i \in \{1, 2, \ldots, n\} \). Let’s call this matching \( M \). We claim that \( w_i \) prefers \( m_i \) to \( m_j \) for all \( i, j \in \{1, 2, \ldots, n\} \) and \( i < j \). This is because otherwise \((w_i, m_j)\) would be an unstable pair in \( M – w_i \) prefers \( m_j \) to \( m_i \), and at the same time \( m_j \) prefers \( w_i \) to \( w_j \).

Suppose that there exists another stable matching \( M' \). Let \( i \) be the smallest index in \( \{1, 2, \ldots, n\} \) so that \( m_i \) is matched with \( w_j \) for some \( j > i \) in \( M' \). Let \( w_i \) be matched with \( m_k \) in \( M' \). We claim that \( k > i \). This is because \( k \neq i \) as \( m_i \) is already matched with \( w_j \), and \( k < i \) contradicts to the minimality of \( i \). Consider the unmatched pair \((m_i, w_i)\) in \( M' – m_i \) prefers \( m_j \) to \( m_i \), and at the same time \( w_i \) prefers \( m_i \) to \( m_k \) because \( i < k \) and because of the argument at the end of the above paragraph. Therefore \((m_i, w_i)\) is an unstable pair in \( M' \) which contradicts to the assumption that \( M' \) is a stable matching.

Therefore, there exists a unique stable matching when \( m_i \) has the same preference list \([w_1, w_2, \ldots, w_n]\) for all \( i \in \{1, 2, \ldots, n\} \).

Problem 2 (10 points):
Show that the stable matching problem may have an exponential number of solutions. To be specific, show that for every \( n \), there is an instance of stable matching on sets \( M \) and \( W \) with \(|M| = |W| = n\) where there are at least \( c^n \) stable matchings, for some \( c > 1 \). (Hint: Suppose you have an instance of size \( n \) with \( k \) solutions, show that you can create an instance of size \( 2n \) with \( k^2 \) solutions.)

Solution 2 (Version 1):
We first show the lemma in the hint. Suppose we have an arbitrary matching instance with size \( n \) and \( k \) solutions. Men are named \( m_1, m_2, \ldots, m_n \) and women are named \( w_1, w_2, \ldots, w_n \). Now we create an additional \( n \) men and women named \( m'_1, m'_2, \ldots, m'_n \) and \( w'_1, w'_2, \ldots, w'_n \), respectively. The preference lists for \( m' \) and \( w' \) are same as the preference lists for \( m \) and \( w \), except that each \( m \) and \( w \) on a preference list becomes its \( m' \) and \( w' \) counterpart.

To concatenate the two solutions together, we need to extend all the preference lists so that they each contain all \( 2n \) men/women. The general idea is to let each man/woman to dislike the women/men
from the other group. For each \(m_i\) and \(w_i\), we extend their preference lists by appending \(w'\) and \(m'\) in random order to their original lists. The preference lists of \(m'_i\) and \(w'_i\) are extended in an analogous way (by appending \(w\) and \(m\), respectively).

Then we will observe that the combined matching instance has size \(2n\) and \(k^2\) solutions. We can think this as combining the solutions in the original group and in the new group. Since both groups have \(k\) stable matchings, they will produce \(k^2\) different combinations. All \(k^2\) combinations are valid stable matchings because there cannot be unstable pairs. It is obvious that there are no intra-group unstable pairs (\(m - w\) or \(m' - w'\)) because all preference lists come from the original matching instance. Also, there cannot be cross-group unstable pairs (\(m - w'\) or \(m' - w\)) because according to our design, a man/woman prefers his/her partners in his/her group than all women/men in the other group. Therefore, we have successfully created a matching instance with size \(2n\) and \(k^2\) solutions from an instance with size \(n\) and \(k\) solutions.

It is then easy to verify that among all reasonable growth functions, only the exponential function matches this growth pattern \((n, k) \rightarrow (2n, k^2)\). Suppose \(f(x) = c^x\) and \(f(n) = c^n = k\). Then we know that \(f(2n) = c^{2n} = (c^n)^2 = k^2\).

Solution 2 (Version 2):
We show by induction that for \(n = 2k\) there is a configuration with at least \(2^k\) solutions. This yields \(c^n\) solutions for \(c = \sqrt{2}\).

The base case, \(k = 1\) is:

\[
\begin{align*}
m_1 : & \quad w_1 \quad w_2 \\
m_2 : & \quad w_2 \quad w_1 \\
w_1 : & \quad m_2 \quad m_1 \\
w_2 : & \quad m_1 \quad m_2
\end{align*}
\]

This has two stable matchings: \((m_1, w_1), (m_2, w_2)\) and \((m_1, w_2), (m_2, w_1)\).

The Induction step is to assume we have a configuration of \(n = 2k\) that has \(2^k\) solutions. We now add \(m_{2k+1}, m_{2k+2}, w_{2k+1} \) and \(w_{2k+2}\) with each other as their top two preferences in a manner equivalent to the base case. There are two matchings for \(m_{2k+1}, m_{2k+2}, w_{2k+1} \) and \(w_{2k+2}\). Since these are independent of how the first \(n\) pairs are matched, there are \(2^{k+1}\) configurations.

There are a number of different ways to handle the odd numbers. One is to start with a base case of \(n = 3\) that has three solutions. Another is to just match \(m_{2k+1}\) with \(w_{2k+1}\) and have \(2^k\) solutions for \(n = 2k + 1\) and argue that this is growing at \(c^n\) for some constant \(c\). [Note from instructor: This is a detail, I did not mean to cover, and should have worded differently. If you lost points for the odd case, this was a miscommunication between instructor and TAs, so send mail, and we will fix it.]

Problem 3 (10 points):
(Adapted from text, page 28, exercise 8.) For this problem, we explore the issue of truthfulness in the Gale-Shapley algorithm for Stable Matching. Show that a participant can improve its outcome by lying about its preferences. Consider \(w \in W\). Suppose \(w\) prefers \(m\) to \(m'\), but \(m\) and \(m'\) are low on \(w\)'s preference list. Show that it is possible that by switching the order of \(m\) and \(m'\) on \(w\)'s preference list, \(w\) achieves a better outcome, e.g., is matched with an \(m''\) higher on the preference list than the one if the actual order was used.
Solution 3:
Consider the following preference lists with \( n = 3 \):

\[
\begin{align*}
M_1 : & \quad W_1 \quad W_2 \quad W_3 \\
M_2 : & \quad W_1 \quad W_3 \quad W_2 \\
M_3 : & \quad W_2 \quad W_1 \quad W_3 \\
W_1 : & \quad M_3 \quad M_1 \quad M_2 \\
W_2 : & \quad M_1 \quad M_3 \quad M_2 \\
W_3 : & \quad M_1 \quad M_2 \quad M_3
\end{align*}
\]

Running the Gale Shapely algorithm results in \( W_1 \) being matched with \( W_1 \). Now consider the following preference list where \( W_1 \) lies and swaps \( M_1 \) and \( M_2 \) in its preference list:

\[
\begin{align*}
M_1 : & \quad W_1 \quad W_2 \quad W_3 \\
M_2 : & \quad W_1 \quad W_3 \quad W_2 \\
M_3 : & \quad W_2 \quad W_1 \quad W_3 \\
W_1 : & \quad M_3 \quad M_2 \quad M_1 \\
W_2 : & \quad M_1 \quad M_3 \quad M_2 \\
W_3 : & \quad M_1 \quad M_2 \quad M_3
\end{align*}
\]

Now running the Gale Shapely algorithm results in \( W_1 \) being matched with \( M_3 \), \( W_1 \)'s first preference.

Programming Problem 4 (10 points):
Implement the stable matching algorithm.

You are free to write in any programming language you like (but Java is recommended). The quality of your algorithm may be graded (but you can use the one in the book), but the actual quality of the code will not be graded. The expectation is that you write the algorithmic code yourself - but you can use other code or libraries for supporting operations. You may use a library to generate random permutations (although this can be done as a four-line algorithm.) Submit your code as a PDF.

Make sure that you test your algorithm on small instance sizes, where you are able to check results by hand. A collection of sample instances are provided.

Run your algorithm on the following instance of size \( n = 4 \). (You can just hard code this as an input into your program.) The preferences for \( M \)'s are given by the following matrix (where the \( i \)-th row in the ordered list of preferences for \( m_i \)).

\[
\begin{bmatrix}
2 & 1 & 3 & 0 \\
0 & 1 & 3 & 2 \\
0 & 1 & 2 & 3 \\
0 & 1 & 2 & 3
\end{bmatrix}
\]
and the preferences for the $W$’s are given by the matrix:

$$
\begin{bmatrix}
0 & 2 & 1 & 3 \\
2 & 0 & 3 & 1 \\
3 & 2 & 1 & 0 \\
2 & 3 & 1 & 0 \\
\end{bmatrix}
$$

Give the resulting matching that is found, along with the list of proposals performed by the algorithm.

==== PYTHON IMPLEMENTATION =====

```python
import numpy as np

men_pref = np.array([[2, 1, 3, 0], [0, 1, 3, 2], [0, 1, 2, 3], [0, 1, 2, 3]])
women_pref = np.array([[0, 2, 1, 3], [2, 0, 3, 1], [3, 2, 1, 0], [2, 3, 1, 0]])

N = 4

women_match = np.array([-1 for i in range(N)])

free_men = list(range(N))
men_pref_idx = [0 for i in range(N)]

while len(free_men) > 0:
    m = free_men.pop(0)
    pref_w_idx = men_pref_idx[m]
    w = men_pref[m, pref_w_idx]
    men_pref_idx[m] += 1
    if women_match[w] == -1:
        # woman is free
        women_match[w] = m
        print("Man", m, "propose to woman", w, ", accepted")
    else:
        # woman is not free
        m_prime = women_match[w]
        w_pref = women_pref[w]
        m_prime_idx = np.where(w_pref == m_prime)[0][0]
        m_idx = np.where(w_pref == m)[0][0]
        if m_idx < m_prime_idx:
            # w prefers m to m'
            women_match[w] = m
            print("Man", m, "propose to woman", w, ", accepted")
            free_men.append(m_prime)
```

```
else:
    # w prefers m' to m
    print("Man", m, "propose to woman", w, ", rejected")
    free_men.append(m)

print()
print(\"Final result:\")
print(\"Woman Man\")
for i in range(N):
    print(i, women_match[i])

Programming Problem 5 (10 points):
Write an input generator which creates completely random preference lists, so that each M has a random permutation of the W's for preference, and vice-versa. The purpose of this problem is to explore how “good” the algorithm is with respect to M and W. (There is an interesting meta-point relating to algorithm fairness that can be made with this problem.)

We define “goodness” of a match as the position in the preference list. We will number positions from one (not zero as is standard for array indexing.) Note that lower numbers are good. To be precise, suppose m is matched with w. The mRank of m (written mRank(m)) is the position of w in m’s preference list, and the wRank of w is the position of m in w’s preference list. We define the MRank of a matching to be the sum of all of the mRank(m) and the WRank of w to be the sum of all of the wRank(w). If there are n M’s (and n W’s), we define the MGoodness to be MRank/n and the WGoodness to be WRank/n.

As the size of the problem increases - how does the goodness change for M and W? Submit a short write up about how the goodness varies with the input size based on your experiments. Is the result better for the M’s or W’s? You will probably need to run your algorithm on inputs with n at least 1,000 to get interesting results.

Solution 5:

==== PYTHON MAIN ====
==== DIFFERENT CODE FROM PROBLEM 4 ====

from p3_classes import Man, Woman
from p3_algorithm import GSAlgorithm
from random import sample
from statistics import mean

class ProblemInstance:
    def __init__(self, N):

self.N = N
self.M = self.generateInput(Man)
self.W = self.generateInput(Woman)
self.result = dict()
self.run()

def generateInput(self, constr):
    return [ constr(i, sample(range(self.N), self.N)) for i in range(self.N) ]

def run(self):
    GSAlgorithm(self.M, self.W)

    # saving results
    self.result['matchings'] = { w.engaged_to.idx: w.idx for w in self.W }
    self.result['m_g'] = sum([elt._goodness() for elt in self.M])
    self.result['w_g'] = sum([elt._goodness() for elt in self.W])

if __name__ == '__main__':
    report = dict()
    for i in range(50, 1001, 50):  # 50 1001 500
        print(f'processing {i}')
        ps = [ProblemInstance(i) for _ in range(10)]
        report[i] = [ p.result for p in ps ]
        avg_m_g = mean(res['m_g']/i for res in report[i])
        avg_w_g = mean(res['w_g']/i for res in report[i])
        total_g = avg_m_g + avg_w_g
        print(f'avg_m_g: %.3f, avg_w_g: %.3f, total_g: %.3f' % (avg_m_g, avg_w_g, total_g))

==== PYTHON CLASSES ====
from collections import deque
class Man:
    def __init__(self, idx, pref_list):
        self.idx = idx
        # using deque for constant time popleft operation
        self.pref_list = deque(pref_list)
        self.N = len(pref_list)

    def nextWomanIdx(self):
        return self.pref_list.popleft()

    def _goodness(self):
        # used only for result gathering
return self.N - len(self.pref_list)

class Woman:
    def __init__(self, idx, pref_list):
        self.idx = idx
        # invert the preference rank and index of men for the constant time access comparison
        self.pref_list = [pref_list.index(i) for i in range(len(pref_list))]
        self.engaged_to = None

    def engage(self, m):
        self.engaged_to = m

    def isFree(self):
        return self.engaged_to is None

    def consider(self, m):
        # returns true iff m is prefered than current matching
        return self.pref_list[m.idx] < self.pref_list[self.engaged_to.idx]

    def _goodness(self):
        # used only for result gathering
        return self.pref_list[self.engaged_to.idx] + 1

====== PYTHON ALGORITHM ======
def GSAlgorithm(M, W):
    free_M = list(M)
    while free_M:
        m = free_M.pop()
        w = W[m.nextWomanIdx()]  # deque popleft, list get: constant time
        if w.isFree():  # constant time
            w.engage(m)
            break
        elif w.consider(m):  # list get: constant time
            free_M.append(w.engaged_to)
            w.engage(m)
        else:
            free_M.append(m)