Algorithms Homework 1

Note: for this class is is generally not necessary to validate input in your pseudocode: if a problem says the input will be in a certain form then write your pseudocode under the assumption that the input is in that form.

Problem 1: *Give a worst-case O(n2) algorithm to determine if a matching is stable. The machinists will be labelled 1 through n and the welders will be numbered likewise. The input to your algorithm should be an array* `match` *so that* `match[n]` *gives the index of the welder machinist number n is matched with, and two 2-D arrays* `prefM` *and* `prefW` *so that* `prefM[i][j]` *gives the index of the welder that machinist number i ranked jth and similarly for* `prefW`*. You may assume that the* `match` *array gives a valid perfect matching. Explain clearly why your algorithm runs in worst case time O(n2).*

```
// Generate array matchW s.t. matchW[n]gives the
index of the machinist that welder n is matched with.
(O(n)).
for n from 0 to N:
    w = match[n]
    matchW[w] = n

// Generate array prefW_lookup s.t. prefW_lookup[i]
[j]  gives welder i's ranking of machinist j.
(O(n^2)).
for i from 0 to N:
    for j from 0 to N:
```

```
        m = prefW[i][j]
        prefW_lookup[i][m] = j

// For each pair m_i and w_j, i.e. machinist i and
welder j:  (O(n))
for i from 0 to N:
    j = match[i]
    // For each w_k preferred by m_i to w_j: (O(n))
    for k from 0 to j-1:
        w_k = prefM[i][k]
        // Look up m_l, w_k's current matching, using
matchW.  (O(1))
        m_l = matchW[w_k]
        // Compare m_i's and m_l's preference ranking
in w_k by looking them up in prefW_lookup. (O(1))
        m_i_rank = prefW_lookup[w_k][m_i]
        m_l_rank = prefW_lookup[w_k][m_l]
        // If m_l > m_i in w_k, then w_k prefers m_l
to her current match m_i, and m_i prefers w_k to his
current match w_j, and we have instability (m_l,
w_k). (O(1))
        if m_l_rank > m_i_rank:
            return False
// No instabilities found
return True
```

The algorithm runs in $O(n^2)$ time because in the worst case, each machinist's current matching is with their least preferred welder, and we have to check for instabilities for $n-1$ preferred welders for each machinist. Additionally the biggest time complexity of generating the complementary arrays takes $O(n^2)$ time, but it allows us to lookup welder pairings and machinist rankings in $O(1)$ time. Thus we have:

$$O(n) + O(n^2) + O(n) * O(n) * O(1) = O(n^2) + O(n^2) = O(n^2)$$

Problem 2: *Consider the special case of the stable matching problem with n machinists and n welders in which all the preferences are the same: each machinist's preference from highest to lowest are w1,w2, ...,wn, and each welder's preferences are m1,m2,...,mn. Prove that in this special case there is a unique stable matching.*

Question: Is an empty set a matching?

Proof: In the special case of the stable matching problem described above, for all $n \geq 1$ there exists a unique stable matching $(m_1, w_1), (m_2, w_2), \ldots (m_n, w_n)$.

In the base case, when $n = 1$, there is only one possible matching (m_1, w_1) and it is stable, as there are no other members to form new pairs.

In the induction case, for some $n = k$ where $k > 1$, suppose we have a unique stable matching
$S = (m_1, w_1), (m_2, w_2), \ldots (m_k, w_k)$. Then for $n = k + 1$, each m now has an additional lowest preference w_{k+1}, and each w has an additional lowest preference m_{k+1}.

We may construct a new matching $S' = S \cup (m_{k+1}, w_{k+1})$. Since:

1. S is stable for $n = k$, so there exists no instabilities with respect to S.
2. m_{k+1} cannot form an instability with respect to S', since they are the lowest preference for every w from 1 to k, i.e., every w besides w_{k+1} is already paired with some m_j whom they prefer to m_{k+1}.

3. Similarly, w_{k+1} cannot form an instability with respect to S', since they are the lowest preference for every m from 1 to k.
4. S' is also a perfect matching, because ...

Therefore from 1, 2, 3, S' contains no instabilities. Hence S' is also stable.

S' is also unique because there are no possible shufflings. Suppose we have S'', another stable matching for $n = k + 1$ where $S'' \neq S'$. We know S'' cannot contain (m_{k+1}, w_{k+1}). If it did, then S'' would contain $S_\beta = S'' - (m_{k+1}, w_{k+1})$, where S_β is stable with respect to k. This is because S'' does not contain any instabilities, and removing (m_{k+1}, w_{k+1}) would not produce any instabilities. But this contradicts the induction hypothesis of S being unique.

Then we have some (m_{k+1}, w_i) and (m_j, w_{k+1}) in S''. But (m_j, w_i) is an instability since both will certainly prefer each other to their lowest ranked partners, w_{k+1} and m_{k+1} respectively. So, there does not exist some distinct stable matching $S'' \neq S'$ for $n = k + 1$. Therefore S' is also unique.

Problem 3: *Describe a Θ(n) algorithm to find a stable matching in the following special case: every welder has the same preference list, and every machinist has one of two preference lists (so perhaps welders are ranked by either speed or accuracy and some machinists prefer fast welders while others prefer accurate welders). The inputs will be:*

- *n, the number of machinists (equal to the number of welders),*
- *a 1-D array prefW where prefW[i] gives the index of the machinist ranked ith by all the welders;*

- *a 2-D array prefM where `prefM[i][j]` gives the index of the welder ranked jth according to machinist preference list i (i=0 or i=1 so that row 0, for example, could store the ordering by speed and row 1 the ordering by accuracy); and*
- *a 1-D array type where type[i] gives either 0 or 1, corresponding to the preference list for machinist i.*

You needn't prove that your algorithm is correct, but you should explain why your algorithm runs in Θ(n) time.

```
matches[N]; // array for matches, length N, stores
pairs of ints
matched_w = [0, 0, 0...] // zero-filled array for
keeping tracked of matched welders, length N, stores
ints
next_w[2]; // counters for machinists preference
lists
next_w[0] = 0 // counter for preference list of type-
0 machinists
next_w[1] = 0 // counter for preference list of type-
1 machinists

for j from 0 to N - 1: // N times // For each
machinist, in order of preference from most
preferred:
    m = prefW[j]
    i = type[m] // Get the type of m. O(1)
    w = prefM[i][next_w[i]] // Get current highest-
ranked welder for type-i machinists. O(1)
    // Find the highest-ranked unmatched welder.
    while matched_w[w] == 1: // O(N) time, amortized
O(1)
        next_w[i] = next_w[i] + 1 // try next w
        w = prefM[i][next_w[i]]
```

```
    // w is now highest-ranked unmatched welder for
type-i machinists
    matched_w[w] = 1 // mark as matched
    matches[j] = (m, w) // add to matches
    next_w[i] = next_w[i] + 1
```

We do not have to rematch machinists since we proceed by order of preference. The outer loop takes $\theta(n)$ time while the iner loop is amortized $\theta(1)$, since overall we will have to iterate through at most n welders across both preference lists. Checking whether a welder is currently matched is $\theta(1$ time.

Therefore the time complexity of the algorithm is $\theta(n)$.

Problem 4: *A squash match ("match" here used in the sense of a contest between two sides) between two teams each with n players consists of n individual matches between players from opposite teams so that each player plays in exactly one individual match. Before the match, the teams determine a sequence of players to play in the individual matches.*

Assuming that the 2n players can be ordered by skill (with no ties) and that a player of higher skill always beats a player of lower skill, is it always possible, no matter the skill order, to create the sequences so that neither team can increase the number of individual matches it will win by unilaterally changing its sequence (so the sequences are stable, in some sense)?

For example, if the players on team 1 are A, B, and C and the players on team 2 are X, Y, and Z, the skill order from most to least skilled is A, B, X, Y, Z, C, and team 1's sequence is A, B, C and team 2's is Z, X, Y then A beats Z, B beats X, and Y beats C, making 2 wins total for team 1, and this does not change if team 1 changes its sequence unilaterally or if team 2 changes its sequence unilaterally, so there are stable

sequences for the given skill order.

Devise an efficient algorithm to find such stable sequences, explain its worst-case running time, and prove that it is correct, or give an example of a skill order so that there are no stable sequences and explain why there are no stable sequences.

Example: The skill order from most to least skilled A-X-B-Y, with players A, B in team 1 and players X, Y in team two, has no stable sequences. There are only two possible matchings:

Case 1: A-X, B-Y. Team 2 may swap its player sequence such that we have Case 2, and will then win one match instead of winning zero matches.

Case 2: A-Y, B-X. Team 1 may swap its player sequence such that we have Case 1, and will then win two matches instead of winning one match.