HW1

1. **Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.**

   True or false? In every instance of the Stable Matching Problem, there is a stable matching containing a pair (*m*, *w*) *such that m is ranked first on the preference list of w and w is ranked first on the preference list of m.*

   Solution: False. Consider four individuals: two men (m1, m2) and two women (w1, w2). Their preferences are as follows:

   m1's preference: w1 > w2
   m2's preference: w2 > w1

   w1's preference: m2 > m1
   w2's preference: m1 > m2

   Now, let's analyze this situation.

   If m1 were to pair with the woman ranked first on his preference list (w1), w1 would not be paired with her top choice (m2). Thus, the pair (m1, w1) doesn't satisfy the original claim.

   Similarly, if m2 were to pair with the woman ranked first on his preference list (w2), w2 would not be paired with her top choice (m1).

   Thus, the pair (m2, w2) doesn't satisfy the original claim either.In this scenario, there's no pair (m, w) in which both m is ranked first on w's preference list and w is ranked first on m's preference list. Therefore, no stable matching in this situation can contain a pair that satisfies the original claim.

2. **Decide whether you think the following statement is true or false. If it is true, give a short explanation. If it is false, give a counterexample.**

   True or false? Consider an instance of the Stable Matching Problem in which there exists a man m and a woman w such that m is ranked first on the preference list of w and w is ranked first on the preference list of m. Then in every stable matching S for this instance, the pair (m, w) belongs to S.

   Solution: True. Suppose there exists a stable matching S in this scenario that includes pairs (m, w') and (m', w).

   However, given that w is the top choice on m's preference list, m would favor w over

w'. Similarly, since m is the top choice on w's preference list, w would prefer m over m'.

This creates a contradiction because m and w both prefer each other over their current partners, forming an instability. Therefore, this contradiction indicates that the pair (m, w) must be a part of the stable matching S.

3. **State True/False: An instance of the stable marriage problem has a unique stable matching if and only if the version of the Gale-Shapely algorithm where the male proposes and the version where the female proposes both yield the exact same matching.**

   Solution: True.

4. **A stable roommate problem with 4 students is defined as follows. Each a, b, c, d student ranks the other three in strict order of preference. A matching is defined as the separation of the students into two disjoint pairs. A matching is stable if no two separated students prefer each other to their current roommates. Does a stable matching always exist? If yes, give a proof. Otherwise give an example roommate preference where no stable matching exists.**

   Solution: Consider four individuals (a, b, c, d) with preferences as follows:

   a: b > c > d
   b: c > d > a
   c: a > d > b
   d: a > b > c

   The Gale-Shapley algorithm starts with a proposing to b and c proposing to d, resulting in temporary pairs (a, b) and (c, d). But b prefers c over a, and this preference would lead to the formation of a new pair, (b, c). This leaves a and d, but a would rather be with c than d, leading to an unstable situation. Hence, not all Stable Roommate Problems yield a stable matching.

5. **There are many other settings in which we can ask questions related to some type of "stability" principle. Here's one, involving competition between two enterprises.**

   Suppose we have two television networks, whom we'll call A and B. There are n prime-time programming slots, and each network has n TV shows. Each network wants to devise a schedule—an assignment of each show to a distinct slot—so as to attract as much market share as possible.

   Here is the way we determine how well the two networks perform relative to each other, given their schedules. Each show has a fixed rating, which is based on the number of people who watched it last year; we'll assume that no two shows have

exactly the same rating. A network wins a given time slot if the show that it schedules for the time slot has a larger rating than the show the other network schedules for that time slot. The goal of each network is to win as many time slots as possible.

Suppose in the opening week of the fall season, Network A reveals a schedule S and Network B reveals a schedule T. On the basis of this pair of schedules, each network wins certain time slots, according to the rule above. We'll say that the pair of schedules (S , T) is stable if neither network can unilaterally change its own schedule and win more time slots. That is, there is no schedule S' such that Network A wins more slots with the pair (S', T) than it did with the pair (S, T); and symmetrically, there is no schedule T' such that Network B wins more slots with the pair (S, T') than it did with the pair (S, T).

The analogue of Gale and Shapley's question for this kind of stability is the following: For every set of TV shows and ratings, is there always a stable pair of schedules? Resolve this question by doing one of the following two things:

(a) give an algorithm that, for any set of TV shows and associated ratings, produces a stable pair of schedules; or

(b) give an example of a set of TV shows and associated ratings for which there is no stable pair of schedules.

Solution: Let us consider two television networks A and B with (A1, A2) and (B1, B2) as their TV shows respectively. The ratings out of 10 of these shows can be considered to be as follows:

A1 - 4
A2 - 8

B1 - 2
B2 - 6

Now, for n=2 TV slots both Networks would try to maximise the number of slots that they can get.

Suppose Network A and Network B have the schedules S=(A1, A2) and T=(B1, B2). In this case Network A would win the first slot and A1 would be aired since A1 has a higher rating that B1. Similarly, the second slot would also be won by Network A since A2 has a higher rating than B2.

However, if we considered the schedule T'=(B2, B1) with S, Network B would win the first slot and B2 would be aired instead of A1 and Network A would win the second slot since A2>B1.

The above pair of schedules can still not be called stable as Network A would try to win more slots by changing it's schedule again to S'=(A2,A1) where it would win both slots, leading back to our originally flipped schedule. Thus each schedule would lead to an instability.

This proves that no stable pair of schedules are present for the given set of TV shows and associated ratings

6. **Gale and Shapley published their paper on the Stable Matching Problem in 1962; but a version of their algorithm had already been in use for ten years by the National Resident Matching Program, for the problem of assigning medical residents to hospitals.**

Basically, the situation was the following. There were m hospitals, each with a certain number of available positions for hiring residents. There were n medical students graduating in a given year, each interested in joining one of the hospitals. Each hospital had a ranking of the students in order of preference, and each student had a ranking of the hospitals in order of preference. We will assume that there were more students graduating than there were slots available in the m hospitals.

The interest, naturally, was in finding a way of assigning each student to at most one hospital, in such a way that all available positions in all hospitals were filled. (Since we are assuming a surplus of students, there would be some students who do not get assigned to any hospital.)

We say that an assignment of students to hospitals is stable if neither of the following situations arises.

First type of instability: There are students s and s′, and a hospital h, so that
– s is assigned to h, and
– s′ is assigned to no hospital, and – h prefers s′ to s.

Second type of instability: There are students s and s′, and hospitals h and h′, so that
– s is assigned to h, and
– s′ is assigned to h′, and
– h prefers s′ to s, and
– s′ prefers h to h′.

So we basically have the Stable Matching Problem, except that (i) hospitals generally want more than one resident, and (ii) there is a surplus of medical students.

Show that there is always a stable assignment of students to hospitals, and give an algorithm to find one.

Solution:

- Start with all hospitals H and students S unassigned and keep track of the number of slots available in each hospital.

- While there exists a hospital h that has at least one slot available and has not yet proposed to every student, do the following:

a. Select such a hospital h.
b. Let s be the highest-ranked student on h's preference list that h has not yet proposed to.
c. If s is unassigned, then create a tentative pair (h, s) and decrease the number of available slots in hospital h by one.

d. If s is already assigned to a hospital h', then:
i. If s prefers h' over h, then h remains unassigned and moves on to the next preferred student.
ii. If s prefers h over h', then break the pair (h', s), create a new pair (h, s) and decrease the number of available slots in hospital h by one. The slots of hospital h' increase by one.

- Repeat this process until all slots in every hospital are filled or every hospital has proposed to every student.

- Return the set of hospital-student pairs.

This process guarantees a stable match: no hospital has an unfilled slot with a student they prefer who also prefers that hospital to their current assignment, and no student is assigned to two hospitals.

The modified Stable Matching algorithm, where hospitals propose to students until their slots are filled, exhibits properties of progressive preference. Students continue to receive better offers and once a student receives an offer, they are never without one for the rest of the algorithm. The hospitals' offers may decrease in preference, but all slots are ultimately filled. The algorithm has a time complexity of O(m*n), where m is the number of slots available and n is the number of hospitals.

To prove that the association A is a stable one, let us consider that there exists an instability in the algorithm and then try to contradict the same.

Instability 1: Suppose student s is assigned to hospital h, but another student s' is unassigned and h prefers s' over s. This suggests a contradiction. The algorithm would have assigned s' to h if h preferred s' over s. Since s', not s, is unassigned, this contradicts our assumption, proving it incorrect.

Instability 2: Assume two pairings, (h, s) and (h', s'), exist where h prefers s' to s and h' prefers s to s'. This contradicts the algorithm's progression. If h preferred s', h would have offered s' a slot first. If s' rejected h for another preferred hospital, then h wouldn't prefer s' over its actual assigned student.

Thus, both cases contradict our assumptions, reinforcing the stability of the assignment.

7. **What is the worst-case runtime performance of the procedure below?**

```
c = 0
i = n
while i > 1 do
   for i = 1 to i do
      c = c+1
   end for
   i = floor (i /2)
end while
return c
```

Solution: $O(nlogn)$

8. **Arrange these functions under the O notation using only = (equivalent) or $\subset$ (strict subset of):**

a. 2^{logn}
b. 2^{3n}
c. n^{nlogn}
d. $logn$
e. $nlog(n^2)$
f. n^{n^2}
g. $log(log(n^n))$

Solution:

$\mathcal{O}(\log n) \subset \mathcal{O}(\log\log n) = \mathcal{O}(\log n + \log\log n) \subset \mathcal{O}(n) = \mathcal{O}(2^{\log n}) \subset \mathcal{O}(n\log n) = \mathcal{O}(n\log n^2) \subset \mathcal{O}(2^{3n}) \subset \mathcal{O}(n^{n\log n}) \subset \mathcal{O}(n^{n^2})$

9. **Given functions f_1 , f_2 , g_1 , g_2 such that $f_1(n) = O(g_1(n))$ and $f_2(n) = O(g_2(n))$ For each of the following statements, decide whether you think it is true or false and give proof or a counterexample.**

a. $f_1(n) \cdot f_2(n) = O\left(g_1(n) \cdot g_2(n)\right)$

b. $f_1(n) + f_2(n) = O\left(max\left(g_1(n), g_2(n)\right)\right)$

c. $f_1(n)^2 = O\left(g_1(n)^2\right)$

d. $log_2 f_1(n) = O\left(log_2 g_1(n)\right)$

Solution:

a) **True**. As per the definition of Big O notation, if $f_1(n) = O(g_1(n))$, there exists a constant c1 such that $f_1(n) \leq c1.g_1(n)$ for sufficiently large n. Similarly, if $f_2(n) = O(g_2(n))$, there exists a constant c2 such that $f_2(n) \leq c2.g_2(n)$ for sufficiently large n.

Therefore, $f_1(n) \cdot f_2(n) \leq (c1 \cdot g_1(n)) \cdot (c2 \cdot g_2(n)) = (c1 \cdot c2) \cdot (g_1(n) \cdot g_2(n))$, which implies $f_1(n) \cdot f_2(n) = O(g_1(n) \cdot g_2(n))$.

b) **True**. As per the definition of Big O notation, $f_1(n) \leq c1.g_1(n)$ and $f_2(n) = O(g_2(n))$ for sufficiently large n. Therefore, $f_1(n) + f_2(n) \leq c1 \cdot g_1(n) + c2 \cdot g_2(n)$. Since the sum of the two functions is less than or equal to the sum of the individual upper bounds, and $\max(g_1(n), g_2(n)) \leq g_1(n) + g_2(n)$, it can be concluded that $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$.

c) **True**. Given $f_1(n) = O(g_1(n))$, there exists a constant c such that $f_1(n) \leq c \cdot g_1(n)$ for sufficiently large n. Squaring both sides gives $(f_1(n))^2 \leq (c \cdot g_1(n))^2$, which is equivalent to $(f_1(n))^2 = O((g_1(n))^2)$.

d) **False**. This statement is not necessarily true. Big O notation describes the growth rate of functions, and the growth rate of $\log f_1(n)$ might be different from the growth rate of $f_1(n)$ itself. As a counterexample, let $f_1(n) = 2^n$ and $g_1(n) = n$. Although $f_1(n) = O(g_1(n))$, is true, $log f_1(n) = n$ and $log(g_1(n)) = \log n$ is not $O(\log(g_1(n)))$.

10. **Given an undirected graph G with n nodes and m edges, design an $O(m + n)$ algorithm to detect whether G contains a cycle. Your algorithm should output a cycle if G contains one.**

Solution: Using a DFS search we can find out if the undirected graph has a cycle or not.

Algorithm

Step 1: Traverse all nodes in the graph. Maintain a 'visited' array, visited[], to record the nodes that have been explored and a parent[] array to store the parent of each

node.

Step 2: Execute a Depth-First Search (DFS) on the subgraph connected to the current node. Remember to pass the parent node of the current node during each recursive call.

Step 3: Mark visited[root] as 1, indicating that the root node has been visited.

Step 4: Loop through all nodes adjacent to the current node in the adjacency list and set parent[adjacent_node] = node.

Step 5: If an adjacent node is unvisited, run DFS on it. Return 'true' if the DFS call also returns 'true'.

Step 6: If the adjacent node is visited and it is not the parent of the current node, return ‘true' and print the cycle by backtracking from the current node to the u using the parent[] array.

Step 7: If none of the above conditions are met, repeat all the above steps for all the remaining unvisited nodes.

Step 8: If all nodes are visited, return False.

Time Complexity

The program does a simple DFS Traversal of the graph which is represented using an adjacency list. So the time complexity is O(m+n).