Homework 1

Tejas Kamtam
305749402
CS 180 - Fall 2023

Problem 1

- Exercise 3, Page 22
  For every set of TV shows and associated ratings, there is **not** always a stable pair of schedules. We can prove this by analyzing a contradictory set of TV shows and ratings.
  Consider the case when $n = 2$ (each network, A and B have 2 shows each and are competing for 2 time slots). WLOG: suppose A has a set of TV shows with ratings s.t. $A = \{a_1 = 10, a_2 = 15\}$ and $B = \{b_1 = 11, b_2 = 16\}$.
  Now, we can consider the following cases:

1. A pair of schedules $(S, T) = \Big(S = (10, 15), T = (11, 16)\Big)$. In this pair of schedules, network B wins both time slots, so A will unilaterally change their schedule to win more slots, like so:
2. A new pair of schedules $(S', T) = \Big(S' = (15, 10), T = (11, 16)\Big)$. In this situation, A and B each win 1 time slot. But, B can now alter their schedule to win more time slots like so:
3. A new pair $(S', T') = \Big(S' = (15, 10), T' = (16, 11)\Big)$. Now in this scenario, B once again wins both time slots. So, A will change their time slots once more.
4. The final unique pair $(S, T') = \Big(S = (10, 15), T' = (16, 11)\Big)$. Now, A wins back 1 time slot from B and B can change their time slot back to the (S, T) situation to win both time slots.
   This process will continue to repeat endlessly, and based on the definition of

a "stable pair of schedules," this counterexample has no stable pairs as they can oscillate between schedules that always have the possibility for a unilaterally better schedule for one network. Thus, there is not **always** a stable pair of schedules.

Problem 2

- Exercise 4, Page 22
  We can consider the following algorithm then prove it always results in stable assignments:

1. Arbitrarily pick one of the hospitals, m_1, and find their most preferred student, s_1. This is a constant time $O(1)$ operation.
2. If s_1 is not assigned to a hospital, assign that student to m_1 (also an $O(1)$ operation. If s_1 is already assigned to some hospital m_2, check s_1's preference list (both conditionals are $O(1)$ with the right data structure implementation e.g., hashmap mapping hospitals' preferences)
   1. If s_1 prefers m_1 to m_2, assign s_1 to m_1 and add m_2 back to the list at either the beginning or end (such that m_2 **will** have an assignment by the end of the first iteration through the list of hospitals).
   2. Else, cross out s_1 from m_1's preference list and find the next most preferred student, s_2, on m_1's preference list, and try step 2 again. Continue until m_1 is assigned a student.
3. Do steps 1 & 2 for each hospital so that each hospital is assigned at most 1 student per iteration through the list of m hospitals.
4. Finally, continue to loop through the hospitals until every hospital's empty positions are filled (remembering to maintain step 3's rule of at most 1 student assignment per iteration through m hospitals).
   This algorithm will **always** produce stable assignments due to its adherence to the following properties:
5. The algorithm will terminate
   1. Each iteration through the list of m hospitals, each hospital will "propose" to m students.
   2. Because hospitals can take more than 1 student and there are more students than hospitals, in the worst case, each hospital looks for some

large k number of students. In the worst case k is a constant such that $mk = n - 1$ because there must be a surplus of students.
3. So, in the worst case, hospitals will make **at most** $O(mkn)$ proposals (because we can assume k is some constant or each hospital and $k < n$ by what we observe in point 2). By then, every hospital will have an assignment and will exit the loop.
6. The algorithm does not create type 1 instability:
   1. Because each hospital looks for students going down their preference list, the preference of students for each hospital decreases each iteration.
   2. So, it is not possible for any remaining surplus of students with no assignments to have been preferred more by any hospital than any of their assigned students.
7. The algorithm does not create type 2 instability:
   1. In the case of assignment conflicts, students move down their hospital preference list, so any hospital that a student moves to in a conflict will be their most preferred, available hospital.
   2. Because hospitals also move down their student preference list in decreasing order, it is not possible for both a hospital to have preferred some student over their currently assigned student AND that student prefer the hospital more than their currently assigned hospital.

Problem 3

- Exercise 6, Page 25
  A possible algorithm to find a set of truncations that adhere to the definition in the problem statement ("stable truncations," indicated by which port the ship should remain at) can be the following:

1. Assign each ship a priority list, d of size m consisting of the locations of that ship for each of the m days e.g., from the example, Ship 1's priority list would be: [port P_1, at sea, port P_2, at sea] (we assume this is pre-computation).
2. Using an iterator i to represent the day of the month s.t. $i \leq m$, arbitrarily pick a ship s_1 and check its priority list and get its day i priority d_i (a const.

time operation $O(1)$):

    1. If d_i is at sea, do nothing.
    2. If d_i is a port p and p **has not** been assigned to any other ship, assign it to ship s_1.
    3. If d_i is a port p and p **has** been assigned to some other ship s_2, assign p to s_1 and leave s_2 unassigned.

3. Repeat step 2 for each ship.
4. Now, repeat steps 2 & 3 for each day of the month for a total of m iterations. The resulting port assignments represent the location at which to truncate that ship's schedule to.

This algorithm always results in "stable truncations" because it adheres to the following principles:

1. The algorithm terminates.
    1. The algorithm loops through m days and, at most, loops through each ship's priority list of a finite number of ports.
    2. This is, in the worst case, an $O(nm)$ operation and will exit the loop and terminate after m days.
2. The algorithm does not allow for two ships at the same port on the same day. Each ship's schedule will be truncated for maintenance.
    1. The algorithm assigns ports one-to-one s.t. a port is assigned to a ship only if the port has not already been assigned to a ship, or if it has, it will remove the other ship's assignment.
    2. So, there can only be one port assignment per ship.
    3. Although ships can be unassigned because each ship must visit each port exactly once in m days (while following the regular schedule), a ship can only become unassigned if it has already been to the port it was assigned to and a new ship has come to visit. So, the unassigned ship will not have yet visited at least 1 port the incoming ship has visited.
    4. Thus, by the end of the algorithm, there will never be a ship that is not docked at a port for maintenance nor will there be more than 1 ship per port.

Problem 4

- Exercise 4, Page 67
  Functions ranked in ascending order of time complexity (fastest to slowest) in the long run (justification is trivially 1st order derivatives; to prove we can plot each and expand to large n):

1. $g_1(n) = 2^{\sqrt{\log n}}$
2. $g_4(n) = n^{4/3}$
3. $g_3(n) = n(\log n)^3$
4. $g_5(n) = n^{\log n}$
5. $g_2(n) = 2^n$
6. $g_7(n) = 2^{n^2}$
7. $g_6(n) = 2^{2^n}$

Problem 5

Part a

- Prove (by induction) that the sum of the first n integers are $n(n+1)/2$, i.e.

$$1 + 2 + \ldots + n \stackrel{?}{=} \frac{n(n+1)}{2} \tag{1}$$

Base case:

$$n = 1 \implies \frac{n(n+1)}{2} = 1$$

This is true for the given sum.

Inductive assumption:
For the following calculations, we assume (1) is true, then we can suggest

$$1 + \ldots + n + (n+1) \stackrel{?}{=} \frac{(n+1)((n+1)+1)}{2} = \frac{(n+1)(n+2)}{2} \tag{2}$$

Induction:

Substituting the known sum using the inductive assumption and simplifying, we get

$$\frac{n(n+1)}{2} + (n+1) = \frac{n(n+1) + 2(n+1)}{2}$$

Now, factoring out the $(n+1)$ we arrive at (2):

$$\frac{n(n+1) + 2(n+1)}{2} = \frac{(n+1)(n+2)}{2}$$

Therefore, the proposed sum equivalence (1) is, in fact, true QED.

Part b

- Using induction, find:

$$1^3 + 2^3 + \ldots + n^3 \stackrel{?}{=}$$

Let's take a few examples of n to derive a pattern:

$$n = 1 \implies 1^3 = 1$$
$$n = 2 \implies 1^3 + 2^3 = 9 = (1+2)^2$$
$$n = 3 \implies 1^3 + 2^3 + 3^3 = 36 = (1+2+3)^2$$

Now, we're beginning to see a pattern that can be generalized as:

$$1^3 + 2^3 + \ldots + n^3 \stackrel{?}{=} (1 + 2 + \ldots + n)^2$$

We can use the expressions we proved in part (a), specifically (1) to get an expression that we can prove using induction:

$$1^3 + \ldots + n^3 \stackrel{?}{=} \left(\frac{n(n+1)}{2}\right)^2 \tag{3}$$

Base case:

$$n = 1 \implies \left(\frac{n(n+1)}{2}\right)^2 = 1$$

This is true for our proposed equation.

Inductive assumption:
We assume (3) is true, so we can suggest:

$$1^3 + \ldots + n^3 + (n+1)^3 \stackrel{?}{=} \left(\frac{(n+1)((n+1)+1)}{2}\right)^2 \tag{4}$$

Induction:
Now, we can use the assumption from (3) to simplify (4) to:

$$\left(\frac{n(n+1)}{2}\right)^2 + (n+1)^3 \stackrel{?}{=} \left(\frac{(n+1)(n+2)}{2}\right)^2$$

We can expand and simplify this to get

$$\frac{n^2(n+1)^2}{4} + (n^3 + 3n^2 + 3n + 1) = \frac{n^4 + 6n^3 + 13n^2 + 12n + 4}{4}$$

Now, we can expand the right side to get

$$\frac{(n+1)^2(n+2)^2}{4} = \frac{n^4 + 6n^3 + 13n^2 + 12n + 4}{4}$$

This is precisely the LHS of (4), thus we have proved that our proposed solution, (3) is true.

Problem 6

- Given an array A of N positive integers, write an algorithm to find the largest element with minimum frequency.

1. We can loop through the array and store the elements as keys in a hashmap, B, and its frequency as its value. Doing this for each element in the array is of $O(N)$ time and space complexity.
2. Next, we can iterate through each element of the hashmap B to find the minimum frequency value. This is of $O(N)$ time complexity and constant space complexity (to store the minimum frequency).
3. Now, we can iterate through the hashmap B once more and save all elements (keys) for which the frequency (values) equals the minimum

frequency we found from step 2 to another array C. This is of $O(N)$ time and space complexity.

4. Finally, we can iterate through the new array C that contains all values of the minimum frequency and find and return the maximum value. In the worst case, this takes $O(N)$ time complexity (in the case that all N element occurs once, so every element of A is now in the array C) and constant, $O(1)$, space complexity.
   In the worst case, this would take a total of 4 passes through the N elements which is $f(N) = 4N = O(N)$ time complexity. This would also require one $O(N)$ space complex data structure and---
   area: ucla
   quarter: Y3Q1
   created: 2023-10-08 18:56
   updated: Wednesday 11th October 2023 09:47:53
   course:
   📕 courses:

---

2 constant space variables, which comes out to a total of $O(N)$ space complexity.

An implementation of this in Python would look something like this:

```python
from collections import defaultdict
def largest_min_freq(A):
        # Step 1
        B = defaultdict(int)
        for x in A:
                B[x] += 1
        # Step 2
        min_freq = min(B.values())
        # Step 3
        C = [key for key in B if B[key]==min_freq]
        # Step 4
```

```
        largest = max(C)
        return largest
```