Homework 1
CS 180 Algorithms and Complexity

Q1)
There are two TV networks A and B, which have n prime time programming slots. We want to implement a schedule where a news network wins slots based on the which TV programs have to highest rating(which is a fixed value and there can't exist two TV programs with the same rating). Assume A has a schedule S, and B has a schedule T. The pair (S,T) is stable if neither can win more time slots regardless of the order. The condition that has to be met is that there should exist no pair (S',T) that wins more slots than with (S,T). Similarly, there shouldn't be a pair (

For instance, let A have two shows with ratings 5 and 8. Let B have two shows with rating 7 and 9. Consider the pair (S,T), where A's schedule S is 8, 5, and B's schedule is 7,9. This would be the best pair for A, considering that at most it could get one slot(Considering B has the highest one so it always gets minimal of 1 slot regardless of the ordering). However this schedule would not work out for B as optimally, it would schedule is as 9,7 and would get both the slots, and this would create a conflict of interest with A.

```
A (one slot)    5  8     A(0 slots)       5  8
B (one slot)    7  9.    B(2 slots)       9  7
```

Thus both A and B would never have a stable schedule as they would continuously change their schedules so that they move back and forth.

Q2)

There exists m hospital slots, and n students. It is given that the m < n; that hospitals have multiple positions(a fixed number), and that if all the positions at the hospitals are filled, there will be students who don't get assigned to any hospitals.

Difference from the GS Med school problem is that in this scenario, m is not equal to n, and that hospitals can have multiple positions.

For this scenario, the schools ask the students according to their priority list. So according to the algorithm, if the hospital has a priority list (D, C, F , G, A …), it would ask this in order until all of its available positions are filled. Once it's done, the next hospital would ask students according to their priority list, and if the student is already taken by another hospital, it would either choose to switch hospitals(depending on which is higher on its priority list), or stay with it's assigned hospital. This would keep going on until all hospitals have no positions left.

The pseudocode for this algorithm looks like:

```
While there is a hospital slot open in (H):
        Ask the first student in the priority list if that student is not part of that institution
        If:   the student is not assigned any hospital, then accept
        Else:
                If:    the student is already assigned a school, if H’ ranks lower than H, then the
                Student leaves H’ and chooses H. Decrement the number of positions in H by 1
                And increment number of positions in H by 1.
                Else:   Go around the loop once more and ask the next student
```

Some properties about the algorithm

1) In worst case scenario, every hospital offers a position to a student no more than once, giving the worst case time complexity as (n*m).
2) According to this algorithm and the GS algorithm, the one that does the proposing gets the best choice and the one that accepts the offers will always end at the lower end of choice. Thus, in this scenario the school gets favored in case of selection.

Proof that the algorithm works:

Case 1 - Stability: To prove the stability of this algorithm, we need to show that there are no unstable pairs. In the scenario I’ve described, students may leave one hospital for another based on their priority lists. However, this process aims to ensure that students only switch to a hospital they prefer more than their current one(and thus students won’t choose a worse off pair than the currently assigned one. A student will switch only if the new hospital is ranked higher in their priority list. This ensures that no student will voluntarily switch to a lower preferred hospital, thereby creating instability. Therefore, the algorithm maintains stability.

Case 2: When both a student and a hospital prefer each other over their current assignments, they could switch their assignments. However in my algorithm, a student will only switch to another hospital if that hospital is higher in their priority list. As for hospitals, they only accept students who are unassigned or rank them higher. Thus it again maintains stability

Time complexity: In the worst-case scenario, every hospital offers a position to each student once. This results in a time complexity of **O(n * m**), where n is the number of students and m is the number of hospital slots. This time complexity is due to the need to iterate through each hospital slot and ask the students in their priority order.

Q3)

Peripatetic Shipping Liens owns n ships and provides service to n ports. Each ships has a schedule detailing when it is at what port, and when is it out at sea. A month has m day,s, for some m > n. Each ship visits each port for exactly one day. No two ships can be in the same port on the same day. For ship maintainance, a ship stops at one of the ports and it stays there until maintenance is complete(which takes a month) and this is the only scenario where the ship doesn't visit the port.

Truncation of Sj's schedule will consist of the original schedule up to a certain specified day on which it is in a port O.

Question: To find a truncation such that each of the conditions are met.

Algorithm: For this we are going to use the gayley shapely algorithm in a different manner

Pseudocode steps:

1) We Initialize an empty list which would keep track of the each ships' preferred ports for each day. We do the same for each of the ports, however, the ranking is different
2) After we get the schedule for each ship and each port, we assign the priority of the ships in terms of which ports it is going to, and for the ports we schedule the priority list in reverse order of the ships visiting that port.(there would be blank spaces in between when there are ports which have no ships coming that day or when ships are at sea.
3) In this case, we ask the port to ask the ship whether they can be truncated at their port. While there are available ports/ships(as they are the same number):
    1) Pick an available port, and according to the priority list ask the ship whether it is free or not.
    2) If free, then assign the ship to be truncated at the port, and fill the position of the port. This way, it asks every ship from the last ship assigned to the port whether it is truncated or not
    3) If not free, i.e. the ship is truncated somewhere else already, then if the ship has a higher ranking of the port than what it is already assigned to have, then it would change its assignment. If not, it would keep its assignment and the port would ask the next one on it's priority list.

At the end of this algorithm, there would be a stable matching pair of each of ports where the ships would get a stable pair of matching(earliest it could be truncated), while the ports get a better preference of which ships it gets to choose(in case of a conflict which is explained below). At every point of the step, there would be a stable matching of the pair of ships.

Proof of Stable matching:

A pair (P,S) is unstable if there exists another pair (P', S') such that both of the following conditions are met:

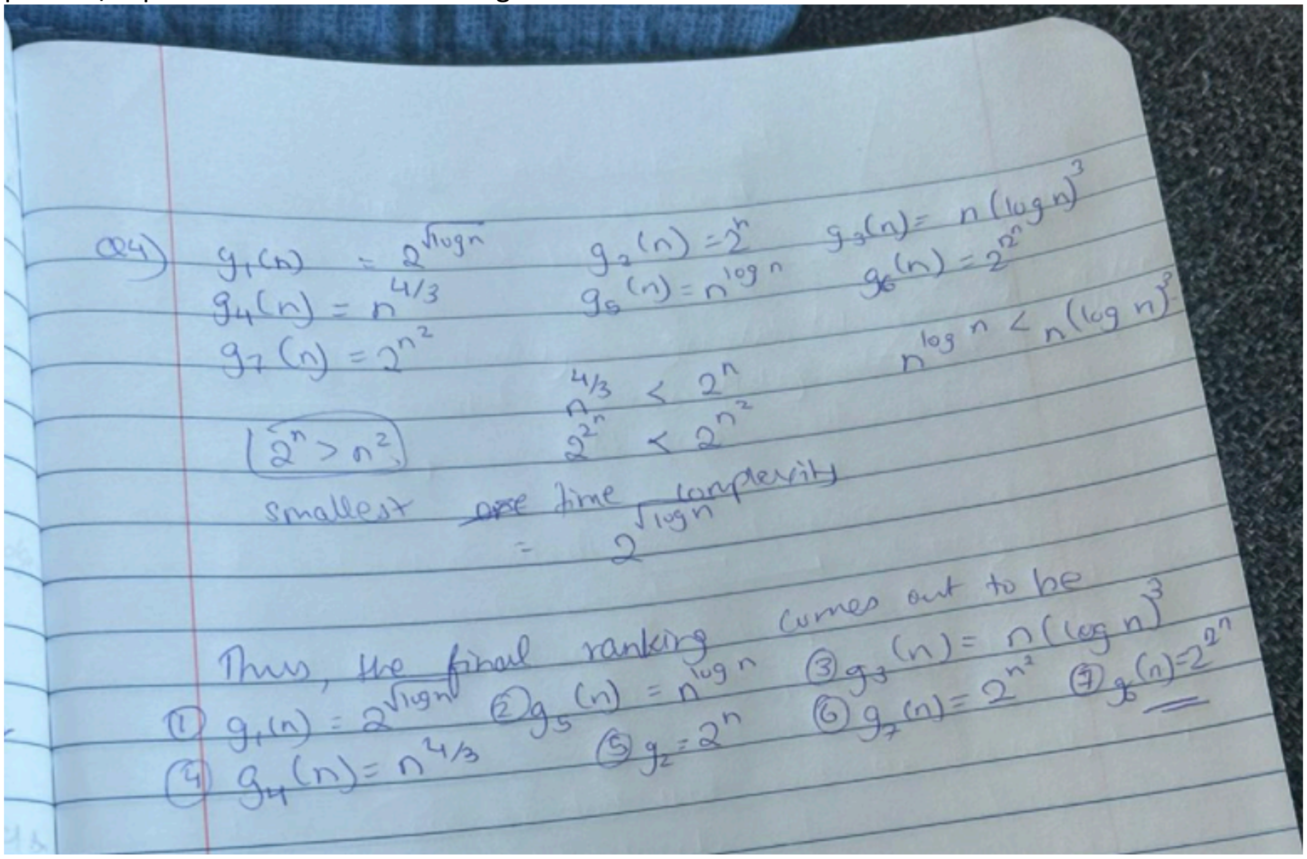
1. P prefers S' over S.
2. S' prefers P over its current assignment (or S' is unassigned).

If there is an unstable pair, according to our algorithm, port P would already have asked S' whether it would want to be truncated at that port. It would have already been truncated Based on it's preference list.
If an unstable pair were to exist, that implies that the port and ship involved had the opportunity to consider each other for truncation but chose to remain with their current assignment. This contradicts the stability of the matching produced by our algorithm, which makes sure that no better alternative pairings can be made while adhering to everyone's preference lists. Thus, there is no unstable matching produced by my algorithm.

In the worst case, each port makes proposals to every ship. This results in 'n' iterations for each of the 'n' ports, which contributes to O(n^2) proposals.

Question 4)

Comparing time complexities-> Polynomials have a slower increasing gradient compared to powers/exponentials. Thus the ranking comes out to be:



Q4) $g_1(n) = 2^{\sqrt{\log n}}$ $g_2(n) = 2^n$ $g_3(n) = n(\log n)^3$

$g_4(n) = n^{4/3}$ $g_5(n) = n^{\log n}$ $g_6(n) = 2^{2^n}$

$g_7(n) = 2^{n^2}$

$n^{\log n} < n(\log n)^3$

$n^{4/3} < 2^n$

$\boxed{2^n > n^2}$ $2^{2^n} < 2^{n^2}$

smallest ~~one~~ time complexity

$= 2^{\sqrt{\log n}}$

Thus, the final ranking comes out to be

(1) $g_1(n) = 2^{\sqrt{\log n}}$ (2) $g_5(n) = n^{\log n}$ (3) $g_3(n) = n(\log n)^3$

(4) $g_4(n) = n^{4/3}$ (5) $g_2 = 2^n$ (6) $g_7(n) = 2^{n^2}$ (7) $g_6(n) = 2^{2^n}$

Question 5)

Prove by induction: p(n) = 1+2+3….+n = n(n+1)/2

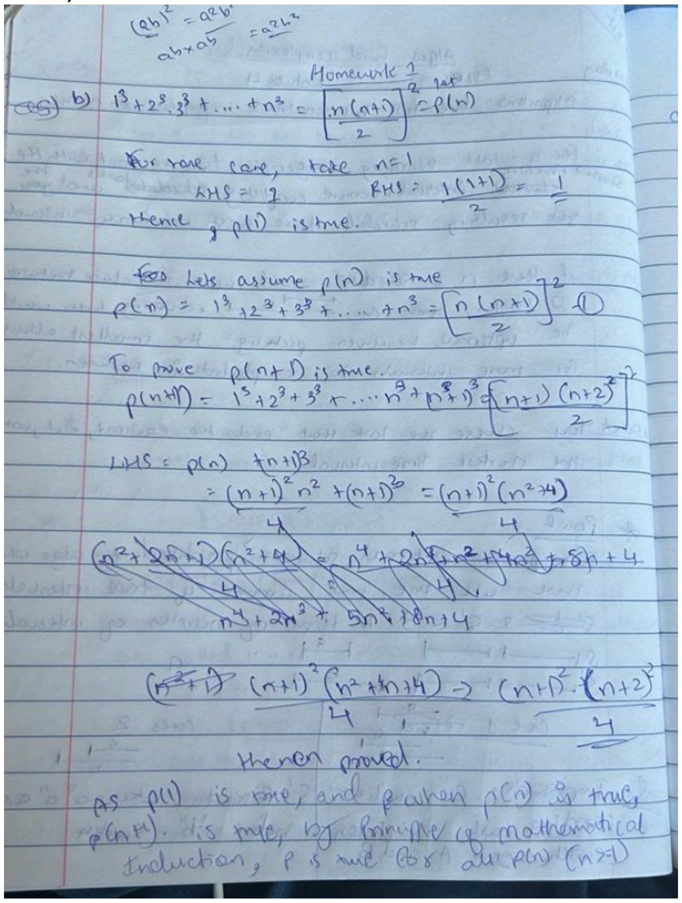
Proving the base case: p(1)
LHS = 1 RHS = 1(1+1)/2 = 1
Thus it is true for p(1)
Let's consider that p(n) is true, thus 1+2+3….+n = n(n+1)/2 ----1

To Prove: p(n+1) is true
P(n+1) = 1+2+3….+n + n+1 which should be equal to n+1(n+2)/2

LHS using equation 1: p(n) + n+1, = n(n+1)/2 + n + 1, = (n^2+n + 2n +2)/2
= (n^2 +3n + 2)/2 , which when factorizing gives (n+1)(n+2)/2.
WHICH IS EQUAL TO RHS.

Thus, according to mathematical induction, and p(1) (the base case) is true, and when p(n) is true, p(n+1) is true, thus through principle of mathematical induction, p(n) is true for all p(n) for all n >= 1.

Q5 b)



$(ab)^2 = a^2b^2$

$ab \times ab = a^2b^2$

Homework 1

~~Q5)~~ b) $1^3 + 2^3 \cdot 3^3 + \dots + n^3 = \left[\frac{n(n+1)}{2}\right]^2 = p(n)$

~~F~~or rare case, take $n = 1$

$LHS = 1$ $RHS = \frac{1(1+1)}{2} = 1$

Hence, p(1) is true.

~~for~~ Lets assume p(n) is true

$p(n) = 1^3 + 2^3 + 3^3 + \dots + n^3 = \left[\frac{n(n+1)}{2}\right]^2$ ①

To prove p(n+1) is true.

$p(n+1) = 1^3 + 2^3 + 3^3 + \dots n^3 + (n+1)^3 = \left[\frac{(n+1)(n+2)}{2}\right]^2$

$LHS = p(n) + (n+1)^3$

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

~~$\frac{(n^2+2n+1)(n^2+4)}{4} = \frac{n^4 + 2n^3 + n^2 + 4n^2 + 8n + 4}{4}$~~

~~$n^4 + 2n^3 + 5n^2 + 8n + 4$~~

~~(n^2+1)~~ $\frac{(n+1)^2(n^2+4n+4)}{4} \rightarrow \frac{(n+1)^2 \cdot (n+2)^2}{4}$

Hence proved.

As p(1) is true, and when p(n) is true, p(n+1) is true, by principle of mathematical induction, P is true for all p(n) (n>=1)

Q6)
Given an array A of size N. The elements of the array consist of positive integers. You have to find the largest element with minimum frequency.

The algorithm that I came up with consists of two major steps. First is sorting the entire list, i.e. using the best sorting algorithm to sort the list in ascending order. For this step we will use a sorting algorithms such as bubble sort, which would have a time complexity of O(N log N) to sort the entire list.

Once the algorithm is sorted in ascending order, we would move from left to right in the array. We would create a struct which has two components, the frequency, and the value associated with that frequency. There would be two of these structs, one with the current frequency(the current value we are going over in the list), and the lowest frequency number, which stores the value with the lowest frequency that we encountered at a given point of time.

We initialize the min_frequency struct such that value is empty(or something not in the list) and the frequency initialized as the (1+length of the list). Then we start from the beginning of the array, where the current frequency stores the value being encountered and keeps a track of the frequency (which starts from 0.) Every time it encounters the same value, it will increment the frequency list, and once it encounters a different value, it will do a comparison with the min_frequency variable and if the frequency is lower or equal to the frequency of min_frequency, it would update min_frequency with the value and the frequency. This continues to go on till the end of the loop, and once at the end of the loop, we have our answer in the min_frequency variable (which we also update at the end of the list depending on the frequency of the last variable.)

The sorting clumps all the similar values together and in an ascending manner, and it becomes much easier to keep track of the current value's frequency.

The big O (time complexity) for this is around N log N for sorting the array , and another N to go over the entire array. Thus the time complexity is N log N + N =
O(N log N) for the algorithm.

However, there could be another way to further reduce the time complexity of this algorithm. Using a hashmap, we could iterate through the entire list, keeping count of the variable and the frequency being used. We could use the value in the list as the key and whenever we encounter the variable, we increment the frequency by 1. Keep on doing this till the end of the array, and each bucket should contain the frequency. Now using an algorithm to iterate over the hashmap and check the lowest frequency(complexity would be O(N)), and track the frequency of values with the lowest ffrequency (For instance, keep a linked list or array of all the lowest frequency pairs up till that point, and if you encountered a new variable with an even lower frequency, delete the linked list and add the value and the frequency to it). Then once you've iterated through the entire hashmap [O(N)], iterate through the final array/linked list to find the highest value in the list, which should be the

highest value with the lowest frequency. Though this is a more complex algorithm with a greater amount of steps, the time complexity of this algorithm would be O[N] .