Charles Zhang

Disc. 1D
Fri. 10am - 1pm
Song

1)

- Counterexample:
  - Denote each set *S* and *T* with each network's show's ratings
  - Given *S* = { 1, 2, 3 } and *T* = { 1.5, 2.5, 3.5 } competing for 3 network spots
  - If either network knows the other network's schedule, it is always possible for them to schedule their shows so that they acquire at least 2/3 network spots
    - This is because each station has at least 2 shows that are higher rated than 2 of the other network's shows
  - Since both networks are able to do this, the network with less network spots will always have a unilateral move that will win it more time slots
  - As a result, a stable solution is impossible, since the "losing" network will always have an option to improve their situation

2a)

- Assumptions:
  - Use some arbitrary standard to break every preference tie (lexicographic order of name, subscript number, etc.)
  - By breaking the ties, the Stable Matching Problem is now back to its original form, as each man and woman have an "ordered" list
- Algorithm:
  - While there is an unpaired man
    - Match a man m with the highest ranked woman that he hasn't proposed to yet
    - If the woman is free, the man and woman are now engaged
    - Otherwise
      - If the woman prefers her current match, the current man remains unpaired
      - If the woman prefers the new man, they become engaged and the old match becomes unpaired
  - Return the final result
- Proof:
  - By artificifically ordering each person's preference list, we arrive at the same initial conditions as the Stable Matching Problem
  - When a woman is engaged, they will be engaged for the rest of the algorithm's runtime
  - As a result, if n women are engaged, n men are also engaged
  - Therefore, a given man cannot reach beyond the end of his preference list, since if the man is unpaired, there must be a woman that is also unpaired
  - By this logic, the GS algorithm always returns a perfect match for the given initial conditions
  - If this perfect match was not a stable match, then a system with 2 pairs, (m, w) and (m', w'), must meet the conditions that m "prefers" w' and w' "prefers" m
  - Assuming m prefers w', he must have proposed to w' already before matching with w, meaning that w' rejected him for another m"
  - This means that w' must prefer her current pairing, m', over m", which implies m' > m" > m, which contradicts our condition for an unstable match
  - The final result must be a stable match

2b)

- Counterexample:
  - Given a set of 2 men, m and m', and 2 women, w and w'
  - m and m' both prefer w over w'
  - w has no preference between m and m'
  - After perfect matching is implemented, one of the men will be matched with w and the other with w'
  - Since the man matched with w' must prefer w and w has no preference, a weak instability is inevitable

3)

- Algorithm:
    - Given a set of n input wires and n output wires
    - Create a “preference list” for each wire, listing the opposite wires in the order you encounter them as you move from the source downstream
    - While there is an input wire with no matched output wire
        - Switch the input wire to the most preferred output wire that it has not yet attempted to switch to
        - If the output wire has not been switched to, switch the input wire to the chosen output wire
        - If the output wire has been switched to:
            - If the current input wire is more preferred by the selected output wire than its previous match, switch the current input wire to the selected output wire
                - Make the previous input wire unswitched
            - If the current input wire is not more preferred than the selected output wire, leave the current input wire unswitched
    - Return the final set of switches between input and output wires
- Proof:
    - Each output wire can only be switched to by a single input wire, since multiple input wires switched onto a single output wire guarantees a crossing of data streams
    - When an output wire is switched onto, they are never unswitched from, so they will be switched onto for the remainder of the algorithm’s runtime
    - As a result of the previous 2 statements, when n output wires are switched to, exactly n input wires have been switched
    - Due to this, it’s impossible to exit the algorithm without every single input and output wire having a unique pair
    - If this solution were to create a result where 2 data streams passed through the same box, then there would have to be pairings (i, o) and (i’, o’) where i would prefer to switch to o’ and o’ would prefer if it was switched to by i
    - In order for this to be true, i must have attempted to switch to o’ first since o’ would be further upstream than o
    - Since i is not paired with o’, it means o’ must’ve found an input wire i’’ that is further upstream than i
    - Since o’ ‘s final pairing is with i’, this implies i’ is further upstream than i’’, and i’’ is further upstream than i
    - Therefore, the conditions for the solution being invalid cannot be met after this algorithm’s execution
    - This has been proved by contradiction, as we have shown that the final pairing is guaranteed to create a valid solution

4a) n^2

- There are 3600 seconds in an hour, therefore 3.6×10^{13} operations can be performed per hour
- For an n^2 algorithm, an input size of n requires n^2 operations to complete computation
- The maximum size of n can then be derived by solving $n^2 = 3.6 \times 10^{13}$
- $n = (3.6 \times 10^{13})^{1/2}$
- $n = 6 \times 10^6$

4b) n^3

- There are 3600 seconds in an hour, therefore 3.6×10^{13} operations can be performed per hour
- For an n^3 algorithm, an input size of n requires n^3 operations to complete computation
- The maximum size of n can then be derived by solving $n^3 = 3.6 \times 10^{13}$
- $n = (3.6 \times 10^{13})^{1/3}$
- n = 33019.27
- n must be an integer, so we floor the result
- n = 33019

4c) $100n^2$

- There are 3600 seconds in an hour, therefore 3.6×10^{13} operations can be performed per hour
- For a $100n^2$ algorithm, an input size of n requires $100n^2$ operations to complete computation
- The maximum size of n can then be derived by solving $100n^2 = 3.6 \times 10^{13}$
- $n = (3.6 \times 10^{11})^{1/2}$
- n = 600000

4d) n log n

- There are 3600 seconds in an hour, therefore 3.6×10^{13} operations can be performed per hour
- For an n log n algorithm, an input size of n requires n log n operations to complete computation
- The maximum size of n can then be derived by solving $n \log n = 3.6 \times 10^{13}$
- $n \log n = n (\ln n / \ln 2)$
- $\ln n \sim n^{1/2} - 1/n^{1/2}$
- $n^{3/2} - n^{1/2} = (\ln 2) \times 3.6 \times 10^{13}$
- $n^{1/2}(n - 1) = 2.495 \times 10^{13}$
- $n = 9.06 \times 10^{11}$

4e) 2^n

- There are 3600 seconds in an hour, therefore 3.6×10^{13} operations can be performed per hour
- For a 2^n algorithm, an input size of n requires 2^n operations to complete computation
- The maximum size of n can then be derived by solving $2^n = 3.6 \times 10^{13}$

- $\log_2(2^n) = \log_2(3.6 \times 10^{13})$
- $n = \log_2(3.6 \times 10^{13})$
- $n = \ln(3.6 \times 10^{13}) / \ln 2$
- $n = 45.03$
- n must be an integer, so we floor the result
- $n = 45$

4f) $2^{2^\wedge n}$

- There are 3600 seconds in an hour, therefore 3.6×10^{13} operations can be performed per hour
- For a $2^{2^\wedge n}$ algorithm, an input size of n requires $2^{2^\wedge n}$ operations to complete computation
- The maximum size of n can then be derived by solving $2^{2^\wedge n} = 3.6 \times 10^{13}$
- $\log_2(2^{2^\wedge n}) = 3.6 \times 10^{13}$
- $2^n = \log_2(3.6 \times 10^{13})$
- $2^n = \ln(3.6 \times 10^{13}) / \ln 2$
- $\log_2(2^n) = \log_2(45.03)$
- $n = \ln 45.03 / \ln 2$
- $n = 5.49$
- n must be an integer, so we floor the result
- $n = 5$

5a)

- Proof:
  - Base case: $1 = 1(1 + 1) / 2$ is true
  - Assume: $(1 + 2 + \ldots + n) = n(n + 1) / 2$
  - Prove: $(1 + 2 + \ldots + n + [n + 1]) = (n + 1)(n + 2) / 2$
  - $(1 + 2 + \ldots + n) + (n + 1) = (n + 1)(n + 2) / 2$
  - $n(n + 1) / 2 + (n + 1) = (n + 1)(n + 2) / 2$
  - $n(n + 1) / 2 + 2(n + 1) / 2 = (n + 1)(n + 2) / 2$
  - $[n(n + 1) + 2(n + 1)] / 2 = (n + 1)(n + 2) / 2$
  - $(n + 1)(n + 2) / 2 = (n + 1)(n + 2) / 2$
  - Assuming $P(k)$ is a valid solution, we proved that $P(k + 1)$ is as well
  - $(1 + 2 + \ldots + n) = n(n + 1) / 2$ has been proved by induction

5b)

- Assumption:
  - $1^2 + 2^2 + 3^2 + \ldots + n^2 = ??$
  - First step: find ??
    - $a^3 - b^3 = (a - b)(a^2 + ab + b^2)$
    - Let $a = n$ and $b = n - 1$
    - $n^3 - (n - 1)^3 = (n - n + 1)(n^2 + n^2 - n + n^2 - 2n + 1)$
    - $n^3 - (n - 1)^3 = 3n^2 - 3n + 1$
    - Repeat this process until $n - 1 = 0$ and sum each equation, and all terms on the left will cancel out except for n^3
    - $n^3 = 3\sum n^2 - 3\sum n + n$
    - $\sum n = n(n + 1) / 2$
    - $n^3 = 3\sum n^2 - 3n(n + 1) / 2 + 1$
    - $\sum n^2 = ?? = n(n + 1)(2n + 1) / 6$
- Proof:
  - Base case: $1^2 = 1(2)(3) / 6$ is true
  - Assume: $1^2 + 2^2 + 3^2 + \ldots + n^2 = n(n + 1)(2n + 1) / 6$
  - Prove: $1^2 + 2^2 + 3^2 + \ldots + n^2 + (n + 1)^2 = (n + 1)(n + 2)(2n + 3) / 6$
  - $(1^2 + 2^2 + 3^2 + \ldots + n^2) + (n + 1)^2 = (n + 1)(n + 2)(2n + 3) / 6$
  - $n(n + 1)(2n + 1) / 6 + (n + 1)^2 = (n + 1)(n + 2)(2n + 3) / 6$
  - $n(n + 1)(2n + 1) / 6 + n^2 + 2n + 1 = (n + 1)(n + 2)(2n + 3) / 6$
  - $(2n^3 + 3n^2 + n) / 6 + (6n^2 + 12n + 6) / 6 = (n + 1)(n + 2)(2n + 3) / 6$
  - $(2n^3 + 9n^2 + 13n + 6) / 6 = (n + 1)(n + 2)(2n + 3) / 6$
  - $(2n^3 + 9n^2 + 13n + 6) / 6 = (2n^3 + 9n^2 + 13n + 6) / 6$
  - Assuming $P(k)$ is a valid solution, we proved that $P(k + 1)$ is as well
  - $1^2 + 2^2 + 3^2 + \ldots + n^2 = n(n + 1)(2n + 1) / 6$ has been proved by induction

6)

- 200 Step Assumptions:
  - In order to solve the problem, the first step is to decide on a partition size for your egg drops
  - Since the problem is strictly about the worst case, focus on that
  - Everytime you go up 1 partition, 1 extra step is added to the problem
  - If the partitions are each the same size, then the bulk of the problem's efficiency is located in the early partitions, while the upper partitions are disproportionately inefficient
  - To minimize the worst-case scenario, the most optimal partition size must be dynamic as the ladder is ascended
  - Since each partition ascended is +1 step, we decrement the size of each partition by 1
  - Assuming the initial partition size is m, the obvious distribution using this method satisfies $m + (m - 1) + (m - 2) + \ldots + 1 = 200$
  - This can be solved for m using $m(m + 1) / 2 = 200$
  - This results in $m = 19.51$
  - Since the partition size must be an integer, and we need to make sure we at least make it to the 200th step, we must ceiling this value
  - Therefore, our initial partition size would be 20
  - Since we've designed this method to balance the partition size and number of partitions, our worst case is also 20 steps
- 200 Step Algorithm:
  - Take $h = 20$ as the initial height and $x = 20$ as the initial partition size
  - While the first egg remains intact
    - Drop the egg from the current height h
    - If the egg breaks or the height surpasses 200, break out of the loop
    - If the egg remains intact, increment h by $x - 1$ and decrement x by 1
  - Decrement current height h by current size of partition x
  - While the second egg remains intact
    - Drop the egg from the current height h
    - If the egg breaks, break out of the loop
    - If the egg remains intact, increment h by 1
  - Return the value $h - 1$, since h is the lowest point at which the egg breaks
- N Step Assumptions:
  - The same assumptions apply here as in the 200 step variation
  - The partitions should satisfy $m + (m - 1) + (m - 2) + \ldots + 1 = N$
  - We can then use $m(m + 1)/2 = N$ to solve for m in terms of N
  - Like in the 200 step variation, this value of m must then be ceilinged
  - Our worst case is ceil(m) steps
- N Step Algorithm:
  - Take h = ceil(m) as the initial height and x = ceil(m) as the initial partition size
  - While the first egg remains intact
    - Drop the egg from the current height h

        - If the egg breaks or the height surpasses N, break out of the loop
        - If the egg remains intact, increment h by x - 1 and decrement x by 1
    - Decrement current height h by current size of partition x
    - While the second egg remains intact
        - Drop the egg from the current height h
        - If the egg breaks, break out of the loop
        - If the egg remains intact, increment h by 1
    - Return the value h - 1, since h is the lowest point at which the egg breaks