Charles Zhang

Disc. 1D
Fri. 10am - 12pm
Song

1)

- Algorithm: // **Assuming *n* and *H* could be equal to 0**
  - Initialize a 2-D array of size (*n* + 1) x (*H* + 1), *opt*
  - Initialize an iterator *i*
  - For all values of *i* from 0 to *n*:
    - Set *opt*[*i*][0] to $f_i(0)$
  - For all values of *i* from 0 to *H*:
    - Set *opt*[0][*i*] to 0
  - Initialize an iterator *j* to 1
  - Reset *i* to 1
  - While *i* is less than or equal to *n* and *j* is less than or equal to *H*:
    - Set *opt*[*i*][*j*] to -1
    - Initialize an iterator *k*
    - For all values of *k* from 0 to *j*:
      - Set *opt*[*i*][*j*] to the maximum of its current value and $f_i(k)$ + *opt*[*i* - 1][*j* - *k*]
        - If the maximum changes, pair the current value of *k* with the entry
    - If *j* is equal to *H*:
      - Increment *i* by 1
      - Reset *j* to 1
    - Else:
      - Increment *j* by 1
  - Set *i* to *n* and *j* to *H*
  - Initialize a list *res* of size *n*
  - While *i* is greater than 0:
    - Insert the paired value of *k* of *opt*[*i*][*j*] into *res*
    - Decrement *i* by 1
    - Decrement *j* by the paired value of *k*
  - Reverse *res*
  - Return *res* // ***res* contains the # of hours spent on each class in order**
- Proof:
  - Mathematically, maximizing average grade means you must maximize the total grade you earn between all courses
  - Base Case:
    - *n* = 1
    - Regardless of how many hours are given in *H*, the optimal solution will always be to spend all of those hours in course 1

    - Since all functions $f_i(h)$ are guaranteed to be non-decreasing, whenever we make the comparison to find the optimal solution, we will always pick the highest value of *k* to pair with the entry
    - At the end, we will simply place this highest value of *k* into *res*
    - Based on our logic above, this must be the correct answer
    - Base case passed
  - Inductive Step:
    - Assume: We have found the optimal solutions for all entries, scanned in row-major order, up until some arbitrary entry (*i*, *j*)
    - Prove: We can now find the optimal solution for entry (*i*, *j*)
      - When we process this entry, we are looking for the optimal solution (highest grade) for the subproblem where we're given *i* courses and *j* hours of studying
      - Our algorithm processes all possible time allocations to course *i*, by using an iterator *k* to find the optimal solution when *k* hours are spent on course *i*
        - This can be done by using the function $f_i(k)$ to calculate the current course's contribution to the total grade
        - Once this contribution is found, it is added to the optimal solution of *i* - 1 courses and *j* - *k* hours
          - This is because we have used *k* hours to study for the current course *i*, so we must reference back to a the previously memoized solution for *i* - 1 courses, given *j* - *k* time
        - Since we exhaustively search all of these possible allocations, we are guaranteed to find the optimal solution, as we consider all possible solutions
      - We will have the optimal solution for entry (*i*, *j*) by the end of the iteration of the loop
      - We can extend this logic until we have entry (*n*, *H*), which represents the optimal solution given access to all *n* courses and *H* hours of studying
    - Inductive step passed
  - Proof by induction complete
- Time Complexity: $O(H^2n)$
- Time Complexity Proof:
  - We initialize and process all elements of an *n* x *H* array
    - This results in a time complexity of $O(Hn)$
    - Each time we calculate the entry into the array, we must iterate through a maximum of *H* elements to process all possible times
    - Since all *Hn* elements take $O(H)$ time to finalize, the main loop structure takes $O(H^2n)$ time to execute

- The final loop that gathers the final time distributions for each course is guaranteed to run n times, as each iteration processes a single row of the 2-D array, resulting in an $O(n)$ time complexity
- The reversal of the *res* array takes $O(n)$ time
- The overall time complexity of the algorithm is $O(H^2n)$

2)

- Algorithm:
    - Initialize an array of size (k + 1) x (n + 1), *opt*
    - Initialize iterators i and j
    - For all values of i from 0 to k:
        - Set *opt*[i][0] to 0 and pair it with a null pair
    - For all values of i from 0 to n:
        - Set *opt*[0][i] to 0 and pair it with a null pair
    - For all values of i from 1 to k
        - Set a variable *pMax* to some value representing negative infinity
        - For all values of j from 1 to n - 1:
            - Set *pMax* to the maximum of its current value and *opt*[i - 1][j - 1] - p(j - 1)
                - If it was set to *opt*[i - 1][j - 1] - p(j - 1), pair it with j - 1
            - Set *opt*[i][j] to the maximum of *pMax* + p(j) and *opt*[i][j - 1]
                - If it was set to *pMax* + p(j), pair it with *pMax*'s paired value and j
                - Else, pair it with *opt*[i][j - 1]'s pairing
    - Set i to k and j to n - 1
    - Initialize a list *res*
    - While i is greater than 0:
        - If *opt*[i][j]'s paired pair is not null:
            - Place *opt*[i][j]'s paired pair into *res*
            - Set j equal to the first value in *opt*[i][j]'s paired pair
        - Decrement i by 1
    - Reverse *res*
    - Return *res*
- Proof:
    - Each array entry (i, j) represents the optimal solution given i transactions and j days
    - Base Case:
        - There is 1 day:
            - No transactions can be made
            - The main loop won't process anything
            - No pairs are created, so the backtracking will simply output nothing
            - This is the correct behavior
        - There are 2 days:
            - There are 2 possibilities:
                - Buying on day 1 and selling on day 2 makes a profit
                - Buying on day 1 and selling on day 2 makes no profit
                - Both possibilities are checked when we compare maximums within the nested for loop
                - If a profit does exist, a pair will be created

            - The backtracking will therefore either output he only possible pairing or a null pairing, based on if the profit was correct
        - This is the correct behavior
    - Base case passed
  - Inductive Step:
    - Assume: We have found the optimal solutions for all entries, scanned in row-major order, up until some arbitrary entry (*i*, *j*)
    - Prove: We can find the optimal solution for entry (*i*, *j*)
      - When we come across the entry, there are 2 possibilities:
        - We sell on day *j*:
          - This means we gain profit equal to the maximum possible profit gained from selling on day *j*
          - We exhaustively check all possible options for this profit by maintaining the variable *pMax*, and comparing it to the profit gained from buying on day *j* - 1
          - This comparison allows us to find the maximum possible profit when selling on day *j*, by exhaustively searching all possible buy days
          - This is covered in both max-comparisons
        - We don't sell on day *j*:
          - This means we gain no extra profit, so the optimal solution for (*i*, *j*) is the same as the optimal solution for (*i*, *j* - 1)
          - This is covered in the second max-comparison
        - The maximum profit of both entries will be placed into *opt*[*i*][*j*]
        - By definition, this maximum profit will be the optimal solution, so this is the correct behavior
    - Inductive step passed
  - Proof by induction complete
- Time Complexity: O(*kn*)
- Time Complexity Proof:
  - We initialize a *k* x *n* (ish) array
  - For each element of the array, we perform a constant number of comparisons and array accesses
  - Based on this, we know that filling this array using dynamic programming costs O(*kn*) time
  - After we finish filling the array, we backtrack through our solution in order to get the actual *k*-shot strategy
  - Since each iteration is guaranteed to process one row of the *opt* array, and there are *k* rows, there are guaranteed to be *k* iterations, each performing constant time operations, so this backtracking takes O(*k*) time

- The list is then reversed, resulting in another O(*k*) time operation
- The overall algorithm is therefore O(*kn*) time

3)

- Algorithm: **// Assumes *n* will be an even number, party votes are stored in an array where *P*[*i*][0] is the number of votes for party *A* in precinct *i* and *P*[*i*][1] is the number of votes for party *B* in precinct *i***
  - Initialize variables *v* and *party* to 0
  - For all precincts:
    - Add the number of votes for party *A* to *v*
  - If *v* is less than *m* / 2: **// Get majority party's total votes into *v***
    - Set *v* to *m* - *v*
    - Set *party* to 1
  - Initialize a 3-D array of size *n* x (*n* / 2) x (*v* - *nm* / 4 - 1), *dp*
  - For all values of *x* from 0 to *n*: **// Base case**
    - Set *dp*[*x*][0][0] to true
  - For all values of *i* from 1 to *n*:
    - For all values of *j* from 1 to *n* / 2:
      - For all values of *k* from 1 to *v* - *nm* / 4 - 1:
        - If *i* and *j* are equal to 1 and *k* is equal to *P*[1][*party*]:
          - Set *dp*[*i*][*j*][*k*] to true
        - Else if *dp*[*i* - 1][*j* - 1][*k* - *P*[*i*][*party*]] is true:
          - Set *dp*[*i*][*j*][*k*] to true
        - Else if *dp*[*i* - 1][*j*][*k*] is true:
          - Set *dp*[*i*][*j*][*k*] to true
        - Else:
          - Set *dp*[*i*][*j*][*k*] to false
  - For all values of *x* from *nm* / 4 + 1 to *v* - *nm* / 4 - 1:
    - If *dp*[*n*][*n* / 2][*x*] is true
      - Return true
  - Return false
- Proof:
  - Each array entry (*i*, *j*, *k*) represents if it is possible to gerrymander using *j* out of the first *i* precincts with exactly *k* majority party votes
  - We know that for a party to win a district, it must be true that the number of votes for that party in that district is greater than or equal to *nm* / 4 + 1
  - In order to win both districts, it must also hold that this is true for the other district as well
  - If we say that the total number of votes for this majority party is *v*, we can conclude that gerrymandering is possible for some number of votes *x*, where *nm* / 4 + 1 <= *x* <= *v* - *nm* / 4 - 1
  - Base Case:
    - *n* is 0:
      - This is true by vacuous truth
      - There are 0 possible majority party votes, and 0 precincts to access
      - This is represented by an entry (0, 0, 0)

            - This entry is set to true in the for loop that handles this base case
            - This is the correct behavior
        - *n* is 1:
            - This is true, as long as there is a majority party in the single district
            - The bounds for this condition are checked by the inequality presented above
            - The entry (1, 1, *k*), where *k* is equal to the number of majority votes in the single precinct will be set to true by the first check in the nested for loop
            - If *k* satisfies the above inequality, it will cause the algorithm to return true from within the backtracking for loop
            - This is the expected behavior by the logic above
        - Base cases passed
    - Inductive Step:
        - Assume: we've filled our array such that we know all values up until *dp*[*i* - 1][*j* - 1][*k* - 1]
        - Prove: we can determine the value of *dp*[*i*][*j*][*k*]
            - When we come across this value, we have 2 options:
                - Include P_i in the majority district:
                    - If we choose to do this, we must trace back to see if gerrymandering is possible in the entry *dp*[*i* - 1][*j* - 1][*k* - *P*[*i*][*party*]]
                        - This is because we've included P_i, which means there is *j* - 1 precincts left to use for this entry, and we must account for the majority votes in the current precinct
                        - We are guaranteed access to the referenced value by our assumption
                        - If the referenced value is true, that means we could add the current precinct to the district, and gerrymandering would be possible, so the current value is true
                - Don't:
                    - If we choose to do this, we must trace back to see if gerrymandering is possible in the entry *dp*[*i* - 1][*j*][*k*]
                        - This is because we didn't include P_i, so we still have *j* precincts left to use and, since no majority votes were taken from P_i, we still have *k* votes left to use
                        - We are guaranteed access to the referenced value by our assumption
                        - If the referenced value is true, that means we could skip the current precinct and gerrymandering would be possible, so the

current value is true

            - Both of these conditions are checked, alongside conditions that represent the base cases described above
        - All possibilities are counted for exhaustively
        - (i, j, k) is guaranteed to hold the correct value, given our assumption
        - We can repeat this logic until our array is completely filled
        - By the inequality we originally described, we must see if it is possible to gerrymander with n precincts available, using n / 2 precincts, with x votes, where $nm / 4 + 1 <= x <= v - nm / 4 - 1$
            - This is checked by brute force in our final for loop
    - Inductive step passed
  - Proof by induction complete
- Time Complexity: $O(n^3m)$
- Time Complexity Proof:
  - Getting the majority vote count requires an array access and arithmetic operation for all n precincts, resulting in an $O(n)$ runtime
  - We initialize an array of size $n \times (n / 2) \times (v - nm / 4 - 1)$ and iterate through each element of it once, performing a set of comparisons (constant time) on each
    - This results in $n^2v / 2 - n^3m / 8 - n^2 / 2$ sets of comparisons
    - v is at most m, so this can be rewritten as $n^2m / 2 - n^3m / 8 - n^2 / 2$
    - The n^3m dominates, so the loop processing is $O(n^3m)$ time
  - The final backtrack through a row of our *dp* array takes at most $O(nm)$ time
  - The overall complexity of our algorithm takes $O(n^3m)$

4a)

- Algorithm:
  - Create a source node *S* and target node *T*
  - Create nodes x_O, x_A, x_B, and x_{AB} to represent the supply of each blood type
  - Create nodes y_O, y_A, y_B, and y_{AB} to represent the demand of each blood type
  - Create directed edges from *S* to each of the *x* nodes and assign each of them their corresponding supply as a capacity (edge (S, x_O) has capacity s_O, etc.)
  - Create directed edges from each of the *y* nodes to *T* and assign each of them their corresponding demand as a capacity (edge (y_O, T) has capacity d_O, etc.)
  - Create a directed edge of infinite capacity from x_i to y_j if people of blood type *j* can receive blood of type *i*
    - The edges (x_O, y_A), (x_O, y_B), (x_O, y_{AB}), (x_O, y_O), (x_A, y_A), (x_A, y_{AB}), (x_B, y_B), (x_B, y_{AB}), and (x_{AB}, y_{AB}) will be created
  - Run Ford and Fulkerson on the resultant network
  - For each edge leading into *T*:
    - If the edge's capacity is non-zero:
      - Return false
  - Return true
- Proof: (?)
  - Based on our algorithm design, each edge leading from *S* to x_i will carry a flow equal to the amount of blood of type *i*
    - Since we assign capacity based on our supply, this flow can never exceed the supply of that type of blood
  - Likewise, each edge leading from y_i to *T* will carry a flow equal to the amount of blood of type *i*
    - Since we assign capacity based on our demand, this flow can never exceed the supply of that type of blood
  - The edges connecting our *x* nodes and *y* nodes have a limitless capacity, so there's no worry about an arbitrary limit being set on how much blood can be transferred, other than the supply given to us
  - There are only edges between valid suppliers and demanders, so no blood can be assigned to an invalid recipient
  - Based on this, the flow from each x_i to y_j will correctly represent the amount of blood taken from blood type *i* and given to recipient's blood type *j*
- Time Complexity: O(*f*)
- Time Complexity Proof:
  - The initialization of the algorithm creates a constant number of nodes and edges and assigns the edges weights
  - We then run Ford and Fulkerson, which has an $O(|f|\ (n + e))$ runtime
    - However, since the number of nodes and edges are constant, this simplifies down to an $O(|f|)$ runtime
  - We will then search a constant number of edges in the final loop
  - The algorithm's overall runtime is O(*f*)

4b) **// Assuming there's a typo in the table and B should have a demand of 10**

- Network Explanation:
  - A possible optimal solution is to send 3 units of AB to AB patients, 10 units of B to B patients, 36 units of A to A patients, 45 units of O to O patients, and 5 units of O to A patients
  - We know that the max flow must be equal to the capacity of the min-cut
  - This means that, given a cut, the maximum flow must be less than or equal to the capacity of that cut
  - Based on our network's setup, the cut consisting of *S* and the vertices corresponding to types B and AB in one cut and the rest of the vertices in the other cut has a capacity of 99
  - This means that the maximum total flow in the network is capped at 99, which immediately tells us that we cannot meet a demand of 100
- Simple Explanation:
  - If we look at the total demand for blood types O and A, we find that the total demand is 87
  - The only blood types that can donate to O and A are O and A, respectively
  - That means the total supply for these recipients is the supply of O and A
  - The total supply of O and A is only 86, so there isn't enough supply to cover the demand

5)

- Algorithm:
    - Initialize lists *res* and *cut*
    - Initialize an iterator *i*
    - Run Ford and Fulkerson to arrive at a residual network *G*'
    - Use BFS starting from *s*, where "reachable" edges have a capacity in the residual network that is greater than 0
        - During execution, if an edge that has capacity 0 is found, insert it into *cut* and do not follow it
    - For all values of *i* from 0 to *k* - 1:
        - If *i* is equal to the size of *cut*: **// There were less than *k* edges in the min-cut**
            - Exit the loop
        - Else:
            - Insert *cut*[*i*] into *res*
    - While the size of *res* is less than *k*:
        - Insert an arbitrary edge that is not already in *res* into *res*
    - Return *cut*
- Proof:
    - The termination condition for Ford and Fulkerson is that there is no augmented path from *s* to *t* in the residual network
    - The fact that there is no augmented path tells us that there exists some cut of *G*' such that the flow of the graph is equal to the capacity of the cut
        - This is because we can remove all saturated paths from *G*'
        - This guarantees that *t* will be unreachable from *s*, since we know for a fact that there are no augmented paths in the graph
            - If *t* was reachable after this modification, then there would have to exist some augmented path
        - We place all vertices reachable from *s* into one cut and the rest into another
        - By conservation of flow, the capacity of this cut must be the flow of the network, since anything that comes out of *s* must eventually reach *t*
    - If there exists some cut of *G*' such that the flow of the graph is equal to the capacity of the cut, then that flow must be the max flow
        - By definition, the max flow of a network must be less than or equal to the minimum capacity of any cut in the network
        - Since we have a found a cut that has a capacity equal to the flow, we know that no flow that is greater than this capacity can exist
        - Based on this, the flow we found must be the maximum possible flow
    - We can combine these 2 facts to say that the result of Ford and Fulkerson guarantees that we have found a max flow, and that forming cuts based on vertices that are reachable from *s* allows us to generate a min-cut
    - The removal of an edge that travels between the cuts of the min-cut decreases the capacity of the cut

    - Since the max flow must be less than or equal to the minimum capacity of the cut, decreasing the min-cut's capacity decreases the max flow
    - Based on this, we can decrease the max flow by at most *k* by removing *k* edges from the min-cut
        - This is exactly what we do using BFS and the following traversal
    - If there are less than *k* edges in the min-cut, then we can remove all the edges in the min-cut and the max flow will drop to 0 since the network will be disconnected
        - This is also covered by our BFS and following traversals
    - Proof complete
- Time Complexity: O(*ne*)
- Time Complexity Proof:
    - The Ford and Fulkerson algorithm has a runtime of O(|*f*| (*n* + *e*))
        - Since each edge has a capacity of 1, *f* is at most *e*
        - This means we can simply our runtime to O(*ne* + e^2)
        - Using the relationship between *n* and *e*, we can say that *ne* will be greater than or equal to e^2, allowing us to simplify this even further to O(*ne*)
    - The modified BFS simply adds an extra constant time check for each iteration, so it still has a runtime of O(*n* + *e*)
    - The final loops can run through *k* times, which will be at most *e*, so we have a runtime of O(*e*) for these loops
    - The overall runtime of this algorithm is therefore O(*ne*)

6)

- Algorithm: // **Assume input array is *seq* and its size is *n***
    - Initialize arrays *inc* and *dec* of length *n* with values of 1
    - Initialize an array *res*
    - Initialize iterators *i* and *j*
    - Initialize variables *maxLen* and *maxIndex* to -1 and *flag* to true
    - For all values of *i* from 1 to *n* - 1: // **Get maximum length of inc/dec subarrays**
        - For all values of *j* from 0 to *i* - 1:
            - If *seq*[*i*] is greater than *seq*[*j*]:
                - Set *inc*[*i*] equal to the maximum of *inc*[*i*] and (*dec*[*i*] + 1)
            - Else if *seq*[*i*] is less than *seq*[*j*]:
                - Set *dec*[*i*] equal to the maximum of *dec*[*i*] and (*inc*[*i*] + 1)
    - For all values of *i* from 0 to *n* - 1: // **Find the max length subarray and its index**
        - If the maximum if *inc*[*i*] and *dec*[*i*] is greater than *maxLen*:
            - Set *maxLen* to the maximum if *inc*[*i*] and *dec*[*i*]
            - Set *maxIndex* to *i*
    - If *maxLen* is equal to *dec*[*maxIndex*]: // **Set flag appropriately**
        - Set *flag* to false
    - Append *seq*[*maxIndex*] to *res*
    - For all values of *i* from *maxIndex* - 1 to 0: // **Rebuild subsequence, using *flag* to alternate**
        - If *maxLen* equals 1:
            - Exit the loop
        - If *flag*:
            - If *dec*[*i*] + 1 is equal to *maxLen*:
                - Append *seq*[*i*] to *res*
                - Decrement *maxLen* by 1
                - Invert *flag*
        - Else:
            - If *inc*[*i*] + 1 is equal to *maxLen*:
                - Append *seq*[*i*] to *res*
                - Decrement *maxLen* by 1
                - Invert *flag*
    - Reverse *res*
    - Return *res*
- Proof:
    - Base Case:
        - *n* = 1
        - The only subsequence possible is simply the input
        - The *inc* and *dec* arrays will both be initialized with a single 1
        - The first set of nested for loops will not run
        - The next for loop will run, allowing us to find the correct maximum index and length (0 and 1)
        - The only value in *seq* will be placed into the result

- The result will be returned
- Base case passed

- Inductive Step:
  - *inc*[*i*] represents the maximum length of an increasing subsequence built from the first *i* elements of *seq*
  - *dec*[*i*] represents the maximum length of an decreasing subsequence built from the first *i* elements of *seq*
  - Assume: We have the correct values of *inc*[*i* - 1] and *dec*[*i* - 1]
  - Prove: We can get the correct values of *inc*[*i*] and *dec*[*i*]
    - There are 2 options for each array:
      - We account for *seq*[*i*] in the maximum length:
        - If we account for it in *inc*[*i*], then *seq*[*i*] must have increased from previous elements
        - If we account for it in *dec*[*i*], then *seq*[*i*] must have decreased from previous elements
        - Both options are checked exhaustively for each iteration by iterating through each previous element and maintaining the maximum length found thus far
        - This accounts for the correct behavior if *seq*[*i*] is to be considered in either array
      - We don't:
        - If we skip it in *inc*[*i*], then *seq*[*i*] must have decreased from or been equal to previous elements
        - If we skip it in *dec*[*i*], then *seq*[*i*] must have increased from or been equal to previous elements
        - Both options are checked exhaustively for each iteration by iterating through each previous element and maintaining the maximum length found thus far
        - This accounts for the correct behavior if *seq*[*i*] is to be skipped in either array
      - Both options will exhibit the correct behavior
    - We can get the correct values of *inc*[*i*] and *dec*[*i*] for any value of *i* from 0 to *n* - 1
    - All we need to prove now is that we can reconstruct the subsequence afterwards
    - We do this by maintaining a flag that tells us which array to look in next
      - We reference this flag to determine if we want a decreasing value or an increasing value, then combine this information with an access to the *seq* array to retrieve the appropriate value
      - This can be proved with common sense
    - Since we iterate backwards, we reverse and return the value
  - Inductive step passed

    - Proof by induction complete
- Time Complexity: O(n^2)
- Time Complexity Proof:
    - The initialization of 2 length-n arrays with the value 1 takes O(n) time
    - The first for loop iterates through n elements
        - It contains a nested for loop which, in the worst-case, iterates through n - elements, performing constant time operations on each
        - This loop therefore takes O(n^2) time
    - The for loop that finds the maximum length and index iterates through n values and performs constant time operations on each, resulting in an O(n) runtime
    - The for loop that builds the subsequence runs a maximum of n times, performing constant time operations with each iteration, resulting in an O(n) runtime
    - The reversal of the resulting array can take O(n) time
    - The overall algorithm's runtime is O(n^2)