

# CS180-Fall20 Final Exam 8am

CHARLES XIAN ZHANG

TOTAL POINTS

**92 / 100**

QUESTION 1

**Q1** 20 pts

**1.1 a 10 / 10**

- 0 pts Correct
- 5 pts incorrect answer
- 5 pts incorrect proof
- ✓ - 0 pts the number can be reduced
- 3 pts lack details in the proof

**1.2 b 4 / 10**

- 0 pts Correct
- 10 pts wrong answer
- 3 pts lack details/proof
- 6 pts question asked for how to "maintain" sorted y list in each step, not sort y from scratch
- ✓ - 6 pts didn't answer how to "maintain" sorted y list (ensure y is sorted in each subproblem)

QUESTION 2

**2 Q2 20 / 20**

- ✓ - 0 pts Correct
- 5 pts Wrong answer of 2(a)
- 5 pts 2a proof is not correct or missing
- 2 pts 2a proof is partially correct
- 5 pts Wrong answer of 2(b)
- 5 pts 2b proof is not correct or missing
- 2 pts 2b proof is partially correct

QUESTION 3

**3 Q3 15 / 15**

- ✓ - 0 pts Correct
- 3 pts Partially Correct Algorithm
- 6 pts Incorrect/No Algorithm
- 3 pts Incorrect/No Time Complexity Analysis

- 6 pts Incorrect/No Proof of Correctness

QUESTION 4

**4 Q4 15 / 15**

- Algorithm
- ✓ + 5 pts Correct DP algorithm
  - + 3 pts Mostly correct DP algorithm
  - + 2 pts Attempted algorithm
  - + 0 pts Missing algorithm

Proof

- ✓ + 7 pts Correct proof
- + 5 pts Mostly correct proof
- + 3 pts Attempted proof
- + 0 pts Missing proof

Complexity Analysis

- ✓ + 3 pts Correct complexity analysis
- + 2 pts Mostly correct complexity analysis
- + 1 pts Attempted complexity analysis
- + 0 pts Missing complexity analysis

QUESTION 5

**5 Q5 15 / 15**

- ✓ - 0 pts Correct
- 2 pts Did not specify reducibility or clarify in what direction you were reducing.
- 3 pts Failed to establish polynomial-time verification of solution
- 3 pts Misread of ST-Hamilton (S and T are assigned within the graph, i.e. instead of an unspecified Hamilton route within the graph we're looking for one with a pre-specified start and end)
- 3 pts If reducing Hamilton to ST-Hamilton, one needs to also demonstrate that ST-Hamilton is within NP (going the other way one doesn't need to because Hamilton is already defined as NP complete)

- **5 pts** Failed to establish logical connection between Hamilton and ST-Hamilton in order to prove one is reducible to the other
- **15 pts** Did not attempt
- **3 pts** While an algorithm was developed describing a transformation, the logic that underpins the relationship wasn't explicated.

QUESTION 6

6 Q6 13 / 15

- ✓ - **0 pts** Correct algorithm
- ✓ - **0 pts** Correct time complexity
- ✓ - **0 pts** Correct proof
- **3 pts** Minor time complexity error
- ✓ - **2 pts** No algorithm conclusion given
- **3 pts** Minor error in algorithm
- **5 pts** Wrong algorithm
- **5 pts** Wrong time complexity
- **5 pts** Wrong proof
- **2 pts** Partial proof

Charles Zhang

1a)

- Answer:
  - For each point, we can consider 15 points on the right as a possible new closest-pair
- Proof:
  - In order for there to be a new closest pair, the newly formed pair's distance must be shorter than the current closest pair's distance
  - This current shortest distance is defined by some value  $d$
  - In order for the new closest pair to be less than  $d$  apart from each other, both elements of the pairing must fall within  $d$  of the merge point in relation to the x-axis
  - By similar logic, the right element of the pairing must be within the left point's y-coordinate, plus or minus  $d$  (and vice versa)
  - Combining these requirements, we see we only have to analyze a  $d \times 2d$  space on each side side
  - Within this  $d \times 2d$  space, we know that we can split this space up into  $d/2 \times d/2$  boxes, such that there is only 1 point inside each of these boxes
    - Assume there were 2 points inside one of the boxes
    - The largest distance these 2 points could be from each other is  $\sqrt{2d^2/4}$  or  $(d * \sqrt{2}/2)$
    - This contradicts the setup of our problem, as  $(d * \sqrt{2}/2) < d$ , yet, we defined  $d$  as the smallest distance between any points that fall within the same subproblem
    - Since the theoretical points that share a grid box are in the same subproblem, it is impossible for the distance between them to be less than  $d$
  - This division into boxes results in 8 boxes within the significant region on either side
  - As proved above, we only need to analyze a single point within each of the boxes
  - This results in us needing to analyze 8 total points per side, or 16 total points
  - However, the point we would be comparing to must occupy one of these 16 boxes, so we only need to compare it to 15 other points

**1.1 a 10 / 10**

- **0 pts** Correct
  - **5 pts** incorrect answer
  - **5 pts** incorrect proof
- ✓ - **0 pts** the number can be reduced
- **3 pts** lack details in the proof

1b)

- Discussion:
  - At the beginning of the algorithm, we not only create a global list of the points sorted by x-component, we also create a global list of the points sorted by y-component
  - At each merge, we explained in part 1a why you only need to analyze points that fall within  $d$  units (by x-coordinate) of the merge point
  - As a result, we can simply iterate through our global list that is sorted by y-component to construct a local list containing only points within the current merge's slice, also sorted by y-component
  - From here, we can compare each point in the list to the closest 15 points to it and update our minimum distance accordingly
- Proof:
  - We proved in 1a that only points with an x-component within  $d$  units of the merge point can be possible candidates for closest pair
  - Using this list, we know that only 15 points are possible candidates for closest pair
  - We can simplify this by comparing each point to the next 15 points in our sorted list
  - This is because, by the time we're analyzing a point, we will have already compared it to all 15 points prior to it in the sorting since we're going in order of the sorted list
  - This covers all our bases, as we are guaranteed to have compared each point to its 15 closest neighbors by y-coordinate

1.2 b 4 / 10

- **0 pts** Correct
- **10 pts** wrong answer
- **3 pts** lack details/proof
- **6 pts** question asked for how to "maintain" sorted y list in each step, not sort y from scratch
- ✓ - **6 pts** didn't answer how to "maintain" sorted y list (ensure y is sorted in each subproblem)

2a)

- Answer:
  - Yes, we can just focus on the edges of T and the newly added weights
- Proof:
  - MST theorem states that, when partitioning the graph into 2 partitions, the minimum weight edge that crosses these partitions is part of the MST
  - Take the current tree T as one partition and the vertex X alone in its own partition
  - By MST theorem, the minimum weight edge that connects T to X must be a part of the MST
  - As T is already an MST, adding this newly processed edge to T guarantees a new MST containing X by MST theorem

2b)

- Answer:
  - $T$  is a minimum spanning tree of  $G$
- Proof:
  - Let the number of nodes in the graphs  $G$  and  $G'$  be represented by  $n$
  - This means that there must be  $n - 1$  edges in the MST for either graph, as all nodes are part of the tree and, by definition, the tree cannot have any cycles
  - Due to this, if we decrease the weight of each edge by some constant  $k$ , we're necessarily decreasing the total weight of the MST of that graph by  $k(n - 1)$ , as all  $n-1$  edges in the MST have had their weight reduced by  $k$
  - This tells us that if the MST of  $T'$  is some weight  $W$ , the MST of  $T$  must be  $W - k(n - 1)$
  - Since the difference between  $G'$  and  $G$  is that every edge in  $G$  has  $k$  less weight than its corresponding edge in  $G'$ , an MST made up of edges in  $G$  will be of weight  $k(n - 1)$  less than the corresponding MST in  $G'$
  - This follows exactly with our statement above, as the total weight of the MST in  $G$  will be the total weight of the MST in  $G'$  minus  $k(n - 1)$ , or  $W - k(n - 1)$
  - By sanity check, this proof makes sense as each edge is simply decremented by constant, so the MST in either graph shouldn't change

2 Q2 20 / 20

✓ - 0 pts Correct

- 5 pts Wrong answer of 2(a)
- 5 pts 2a proof is not correct or missing
- 2 pts 2a proof is partially correct
- 5 pts Wrong answer of 2(b)
- 5 pts 2b proof is not correct or missing
- 2 pts 2b proof is partially correct

3)

- Algorithm:
  - Initialize a variable L to 0
  - Initialize a variable R to the size of B - 1
  - While L is less than R:
    - Calculate an index mid using  $\text{floor}([L + R] / 2)$
    - If the value at index mid is greater or equal to the value at index L:
      - Set L to mid + 1
    - Else:
      - Set R to mid
  - Save the value of L in a variable M
  - Reset L to 0 and R to the size of B - 1
  - While L is less than or equal to R:
    - Calculate an index temp using  $\text{floor}([L + R] / 2)$
    - Set a new variable mid equal to  $(M + \text{temp}) \bmod (\text{the size of } B)$
    - If the value at index mid is equal to X:
      - Return true
    - Else if the value at index mid is less than X:
      - Set L equal to temp + 1
    - Else:
      - Set R equal to temp - 1
  - Return false
- Proof:
  - The overall structure of the algorithm is to find how much the array was shifted by, then using that value to modify a basic binary search of the array
  - Since A is sorted, the amount that B was shifted by is equivalent to the index of the minimum element in B
  - This minimum element is differentiated, as it is the only element where the element that directly precedes it is greater than itself
  - Base Case:
    - The size of B is 1
    - If this element is X, then X is in B, otherwise it's not
    - The initial shift calculation should tell us that the array was not shifted ( $M = 0$ )
      - Based on initialization, the loop containing the first binary search will not run
      - L's initial value of 0 will be saved to M
      - Shift calculation passes the base case
    - The following binary search will begin
    - This search should simply check if the element in B is in fact X
    - The loop will run 1 time
    - A mid value will be calculated, and will remain unadjusted, as there have been no shifts

- The first value will be checked, and true will be returned if the value matches X
- If not, the loop will terminate and return false
- This is the expected behavior
- Base case passed
- Inductive Step (Shifting):
  - Assume: The algorithm has executed so that there are n elements to search remaining
  - Prove: The current step will return a subproblem that contains the pivot point
    - A midpoint index will be calculated by a simple average
    - The algorithm will do one of 2 things:
      - It will set L to mid + 1:
        - For this to happen, the algorithm must have found that the value at mid is  $\geq$  to the value at L
        - This tells us that we're still in an increasing part of the array
        - Since the array is non-decreasing from L to mid, we know that the minimum value (pivot point) cannot be located in that subarray
        - Therefore, setting L to the index beyond that subarray is safe, as we know that the minimum must be in the right subarray
        - This is the correct behavior
      - It will set R to mid:
        - For this to happen, the algorithm must have found that the value at mid is  $<$  the value at L
        - Based on this, we know that, at some point in the subarray between L and mid, the values in the array decrease
        - Based on our reasoning from the beginning of the proof, we know that this tells us our minimum must be in this subarray
        - As a result, we move the right bound of R to the end of the subarray
        - This is the correct behavior
      - Both situations exhibit the correct behavior
      - We can inductively extend this logic to say that our resulting subarray contains the minimum element, which means we know how much the array was shifted by
    - Inductive step complete
  - Inductive Step (Searching):
    - Assume: The loop has executed so that there are n elements to search through

- Prove: The current step will return a subproblem that contains the pivot point
  - The algorithm will calculate a temporary midpoint index
  - This temporary midpoint index will be adjusted to reflect the index of the midpoint if the array were sorted
    - This is done by adding the amount that B was shifted ( $M$ ) to the temporary midpoint index and then taking the remainder of that calculation to re-index it into the array B
    - By doing this, each temporary midpoint index we can calculate will correspond to a distinct and valid final midpoint index
  - From here, the algorithm will do one of 3 things:
    - It will find that the value at mid is equal to X
      - At this point, we know X is in fact in B
      - True is returned
      - This is the correct behavior
    - It will find that the value at mid is less than X:
      - Since the array is sorted, we then know that, if X exists, it is located in the right subproblem
      - This is adjusted for by moving the left boundary to the right of mid
      - This is the correct behavior
    - It will find that the value at mid is greater than X:
      - Since the array is sorted, we then know that, if X exists, it is located in the left subproblem
      - This is adjusted for by moving the right boundary to the left of mid
      - This is the correct behavior
  - All 3 situations result in the expected behavior
  - This tells us that, upon completion of the current iteration, the result will have been found, or a subproblem containing the possible solution will have been found
  - We can inductively extend this logic to further subproblems until the target X has been found or there are no more elements to search
  - We will know the answer by the end of execution
    - Inductive step complete
    - Proof by induction complete
- Time Complexity:  $O(\log n)$
- Time Complexity Proof:
  - Outside of the 2 while loops, only constant time operations exist
  - Both while loops consist of binary search, where we can use the equation  $T(n) = T(n / 2) + C$

- Both binary searches are made up of a constant number of calculations and comparisons, resulting in the  $C$  term
- Both binary searches move to the next iteration with a subproblem that is approximately half the size of the previous subproblem, resulting in the  $T(n / 2)$  term
  - If we expand the equation, we see that it is equal to  $T(n) = T(n / 2^i) + Ci$
  - We know that searching an array of size 1 is  $O(1)$ , so we can say that  $n / 2^i = 1$ 
    - This means  $i = \log n$
  - Plugging back in, we have  $T(n) = 1 + C \log n$
  - This reduces to tell us our time complexity is  $O(\log n)$

3 Q3 15 / 15

✓ - 0 pts Correct

- 3 pts Partially Correct Algorithm

- 6 pts Incorrect/No Algorithm

- 3 pts Incorrect/No Time Complexity Analysis

- 6 pts Incorrect/No Proof of Correctness

4)

- Algorithm: // Assume n is the size of A
  - If n is less than 4:
    - Return some failure condition
  - Initialize 4 arrays, a1, a2, a3, and a4, each of size n + 1, with some value representing negative infinity
  - Initialize an iterator i to n - 1
  - While i is greater than or equal to 0: // Get max vals of A[s]
    - Set a1[i] equal to the maximum of a1[i + 1] and A[i]
    - Decrement i by 1
  - Set i to n - 2
  - While i is greater than or equal to 0: // Get max vals of A[s] - A[r]
    - Set a2[i] equal to the maximum of a2[i + 1] and a1[i + 1] - A[i]
    - Decrement i by 1
  - Set i to n - 3
  - While i is greater than or equal to 0: // Get max vals of A[s] - A[r] + A[q]
    - Set a3[i] equal to the maximum of a3[i + 1] and a2[i + 1] + A[i]
    - Decrement i by 1
  - Set i to n - 2
  - While i is greater than or equal to 0: // Get max vals of A[s] - A[r] + A[q] - A[p]
    - Set a4[i] equal to the maximum of a4[i + 1] and a3[i + 1] - A[i]
    - Decrement i by 1
  - Return a4[0]
- Proof:
  - a1 holds the max values of A[s] for each index, a2 holds the max values of A[s] - A[r] for each index, a3 holds the max values of A[s] - A[r] + A[q] for each index and a4 holds the max values of A[s] - A[r] + A[q] - A[p] for each index
  - Proof by induction:
    - For each step in the first loop, 2 things can happen
      - The entry into a1 is the current value of A:
        - This means that the current value of A is the maximum value possible thus far
        - This is confirmed by comparison
        - Correct behavior
      - The entry into a1 is not the current value of A
        - This means that the current value of A is not the maximum value possible thus far
        - This tells us the previous maximum is the optimal solution for this entry
        - This is confirmed by comparison
        - Correct behavior
    - For each step in the second loop, 2 things can happen
      - The entry into a2 is the optimal value of A[s] minus the current value of A:

- This means that the current value of  $A[s] - A[r]$  is the maximum value possible thus far
  - This is confirmed by comparison
  - Correct behavior
- The entry into  $a_2$  does not involve the current value of  $A$ 
  - This means that the current value of  $A[s] - A[r]$  is not the maximum value possible thus far
  - This tells us the previous maximum is the optimal solution for this entry
  - This is confirmed by comparison
  - Correct behavior
- For each step in the third loop, 2 things can happen
  - The entry into  $a_3$  is the optimal value of  $A[s] - A[r]$  plus the current value of  $A$ :
    - This means that the current value of  $A[s] - A[r] + A[q]$  is the maximum value possible thus far
    - This is confirmed by comparison
    - Correct behavior
  - The entry into  $a_3$  does not involve the current value of  $A$ 
    - This means that the current value of  $A[s] - A[r] + A[q]$  is not the maximum value possible thus far
    - This tells us the previous maximum is the optimal solution for this entry
    - This is confirmed by comparison
    - Correct behavior
- For each step in the fourth loop, 2 things can happen
  - The entry into  $a_4$  is the optimal value of  $A[s] - A[r] + A[q]$  minus the current value of  $A$ :
    - This means that the current value of  $A[s] - A[r] + A[q] - A[p]$  is the maximum value possible thus far
    - This is confirmed by comparison
    - Correct behavior
  - The entry into  $a_2$  does not involve the current value of  $A$ 
    - This means that the current value of  $A[s] - A[r] + A[q] - A[p]$  is not the maximum value possible thus far
    - This tells us the previous maximum is the optimal solution for this entry
    - This is confirmed by comparison
    - Correct behavior
- Each step in the algorithm exhibits the correct behavior
- The value in  $a_4[0]$  represents the maximum value of the target expression, given access to the entire array, by definition of our problem construction
- Proof by induction complete

- Time Complexity:  $O(n)$
- Time Complexity Proof:
  - The initialization of the 4 arrays takes  $O(n)$  time
  - Each of the 4 loops iterates through each element of the array, performing constant time comparisons on each
  - Since these loops are organized sequentially, they all contribute to an  $O(n)$  time
  - The overall runtime of the algorithm is  $O(n)$

#### 4 Q4 15 / 15

Algorithm

✓ + 5 pts Correct DP algorithm

+ 3 pts Mostly correct DP algorithm

+ 2 pts Attempted algorithm

+ 0 pts Missing algorithm

Proof

✓ + 7 pts Correct proof

+ 5 pts Mostly correct proof

+ 3 pts Attempted proof

+ 0 pts Missing proof

Complexity Analysis

✓ + 3 pts Correct complexity analysis

+ 2 pts Mostly correct complexity analysis

+ 1 pts Attempted complexity analysis

+ 0 pts Missing complexity analysis

5)

- Proof:
  - Hamiltonian path is NP-complete (Y)
  - Prove that ST-Hamiltonian path is also NP-complete (X)
  - To do this, we must prove that Hamiltonian path is polynomial-time reducible to ST-Hamiltonian path
    - Assume we have algorithms that solve both problems
    - Hamiltonian path takes in an input of a graph
    - Take this graph, and extract all  $nC_2$  combinations of start point and end point
      - $nC_2$  is on the order of  $n^2$ , so this is a polynomial time transformation
      - This is every possible combination of S and T in the graph
    - We can now proceed to pass each graph through our algorithm for ST-Hamiltonian path
    - If any of these inputs are found to have an ST Hamiltonian path, the graph also has a general Hamiltonian path
    - This means we can use a logical statement with order  $n^2$  comparisons to transform the output of ST Hamiltonian path to Hamiltonian path
    - We have shown that Hamiltonian path is polynomial-time reducible to ST-Hamiltonian path
  - Since Hamiltonian path is polynomial-time reducible to ST-Hamiltonian path, it follows that ST Hamiltonian path must be NP-complete, otherwise it would be possible to use a series of polynomial-time transformations to transform the NP-complete Hamiltonian path problem into ST Hamiltonian path, which would result in Hamiltonian path being solvable in polynomial time

## 5 Q5 15 / 15

✓ - 0 pts Correct

- 2 pts Did not specify reducibility or clarify in what direction you were reducing.
- 3 pts Failed to establish polynomial-time verification of solution
- 3 pts Misread of ST-Hamilton (S and T are assigned within the graph, i.e. instead of an unspecified Hamilton route within the graph we're looking for one with a pre-specified start and end)
- 3 pts If reducing Hamilton to ST-Hamilton, one needs to also demonstrate that ST-Hamilton is within NP (going the other way one doesn't need to because Hamilton is already defined as NP complete)
- 5 pts Failed to establish logical connection between Hamilton and ST-Hamilton in order to prove one is reducible to the other
- 15 pts Did not attempt
- 3 pts While an algorithm was developed describing a transformation, the logic that underpins the relationship wasn't explicated.

6)

- Algorithm: // Assumes that such an assignment is possible
  - Create a source node S and a sink node T
  - Create N nodes to represent each team
  - For each node  $N_i$ :
    - Duplicate the node into  $N_{i1}$  and  $N_{i2}$  and connect them from  $N_{i1}$  to  $N_{i2}$  with an edge with capacity  $t_i$  to reflect the size of team i
  - Create an edge with infinite capacity from S to each of the  $N_{i1}$  nodes
  - Create M nodes to represent each table
  - For each node  $M_j$ :
    - Duplicate the node into  $M_{j1}$  and  $M_{j2}$  and connect them from  $M_{j1}$  to  $M_{j2}$  with an edge with capacity  $c_j$  to reflect the number of chairs at table i
  - Create an edge from each node  $N_{i2}$  to each node  $M_{j1}$  with a capacity of 1
  - Create an edge with infinite capacity from each of the  $M_{j2}$  nodes to T
  - Run Ford-Fulkerson on the network
- Proof:
  - Our goal is to represent a matching of players to tables in such a way that no 2 players from the same team are at the same table
  - We enforce the number of players on each team by duplicating the nodes that represent teams, and creating an edge between them with a capacity equal to the number of players on that team
    - This results in us being able to artificially create a bottleneck on each team's node based on the number of players that team has
  - We enforce that each player can only be assigned to a table that none of their other teammates have been assigned to, by creating an edge of capacity 1 between each team and each table
    - In a high-level view of the problem, this essentially means each team can only send 1 player to a table
  - We enforce the number of chairs at each table by duplicating the nodes that represent tables, and creating an edge between them with a capacity equal to the number of chairs at that table
  - We then run Ford-Fulkerson, which will terminate when there are no augmenting edges in the residual network
  - Since there are no augmenting edges in the residual network, it must follow that there exists a cut such that the flow is equal to the capacity of the cut
    - This must be the case because we can create a cut where all nodes reachable from S are in the S partition and all other nodes are in the T partition
    - Since there are no augmenting edges, S and T are disconnected, assuming saturated edges are removed
    - By conservation of flow, all flow that leaves S must end up at T
    - This means that the capacity of the cut is made up of all saturated edges
    - Combining this with our statement on conservation of flow, the capacity of the cut must be equal to the flow in the network

- Since there exists a cut such that the flow is equal to the capacity of the cut, this flow must be the max flow
    - This is because, given a capacity of a cut, the flow in the network must be less than or equal to that capacity
    - Therefore, since we found a flow that is equal to the capacity of a cut, we know that no larger flows can exist, otherwise they would be larger than the capacity of the cut
  - Since this flow is the max flow, we know that our algorithm has finished execution correctly, as players will all be assigned
- Time Complexity:  $O(\sum(t_i)) * (N + M) + NM$
- Time Complexity Proof:
  - Creating all of the nodes requires creating  $2N + 2M + 2$  nodes, which results in an  $O(N + M)$  runtime
  - To connect source to team nodes and table nodes to sink requires  $O(N + M)$  runtime
  - For all  $N$  team nodes, we must create an edge to each of the  $M$  table nodes, resulting in an  $O(NM)$  runtime
  - Running Ford-Fulkerson requires  $O(f(N + M))$  runtime
    - In this case,  $f$  is bounded by the minimum of the summations of  $t_i$  and  $c_j$
    - Since we assume this assignment is possible, we know there are at least as many players as chairs, so  $t_i \leq c_j$
    - As a result, Ford-Fulkerson requires  $O(\sum(t_i) * (N + M))$  runtime
  - Therefore, the overall time complexity of the algorithm is  $O(\sum(t_i)) * (N + M) + NM$

6 Q6 13 / 15

- ✓ - 0 pts Correct algorithm
- ✓ - 0 pts Correct time complexity
- ✓ - 0 pts Correct proof
- 3 pts Minor time complexity error
- ✓ - 2 pts No algorithm conclusion given
- 3 pts Minor error in algorithm
- 5 pts Wrong algorithm
- 5 pts Wrong time complexity
- 5 pts Wrong proof
- 2 pts Partial proof