Name(last, first): Seo, Amy

ID (rightmost 4 digits): 8863

U C L A Computer Science Department

CS 180 **8 am** **Algorithms & Complexity**

Final Exam **Total Time: 3 hours** **December 16, 2018**

***** Write all algorithms in bullet form (as done in the past) *****

You need to prove EVERY answer that you provide.

There are a total of 8 pages including this page.

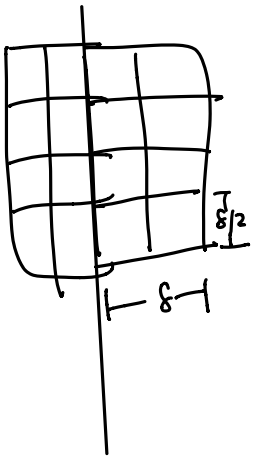
You need to upload ONE file in PDF to Gradescope.

You can include at most 15 pages in your PDF.

Name(last, first): Seo, Amy

1. (20 points: each part has 10 points)

a. Consider an instance of the closet pair problem. In the merge step, for every point on the left, how many points do we consider and compare? Prove your answer.



- In the merge step, we consider 16 points max because each point in S (the set of midpoints) must be at least δ/2 apart, or else we would have found a closer pair w/ distance δ which would form a contradiction.
- Then we calculate the distance of each of these points, and find the minimum.

Proof: Suppose by contradiction, there were more than 16 points to consider in the merge step for every point in the left

- Then, there must be 2 points that are less than δ/2 apart, which would make them a closer pair than the current left point being merged
- Thus, the closest pair wouldn't include the current point being considered, which is a contradiction.

∴ we only consider a max of 16 points

Name(last, first): Seo, Amy

b. Consider an instance of the closest pair problem. How do we maintain a list of points sorted in the y-direction at each step? Discuss and prove the details.

- We can maintain 2 sorted lists:
  1) X can be an array holding sorted points in the X-direction
  2) Y can be an array holding sorted points in the y-direction

  Both lists can have pointers to their respective positions in the other array.

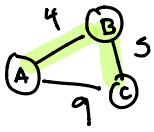
- In this way, we can partition our points in both ways.
- At each step of the algorithm, we can create 2 new subarrays of the sorted lists in both directions:

snippet of algorithm:
- Base case: if size of the X or Y array ≤ 3, find closest pair by directly computing euclidean distances
- otherwise, partition X into 2 halves: P & Q

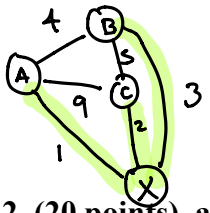
have pointers to position in the array:
- sort P by increasing X-coordinate into P_X
- sort P by increasing Y-coordinate into P_Y
- sort Q by increasing X-coordinate into Q_X
- sort Q by increasing Y-coordinate into Q_Y

Proof: By contradiction, suppose we don't create 2 subarrays, such that we only have the sorted coordinates in one dimension.
- Then, we would be unable to compare smallest candidates relative to <u>both</u> the X & Y direction.
- By creating 2 arrays sorted in X & Y direction w/ pointers to its candidate in the other array, we can keep track of distances relative to both directions, allowing us to calculate closest pairs by euclidean distances.

Example:



G



compare 4 to 1
compare 5 to 3

Name(last, first): Seo, Amy

2. (20 points) a. Consider a weighted connected graph G and a Minimum spanning tree T on G. Assume a new vertex X and a set of weighted edges from X to other vertices are added to G. The new graph is called G'. To find an MST of G' can we just focus on edges of T and the newly added weights? (that is, can we ignore all edges of G that are not part of T?) Prove your answer.

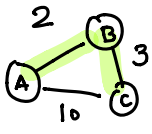
b. We have a weighted graph G. We increase each weight by a number K to obtain a graph G'. We then find an Minimum Spanning Tree T' of G'. We decrease weight of each edge in T' by K to obtain a tree T. Is T a minimum spanning tree of G? Prove your answer.

a) Proof: Suppose we have a weighted connected graph G and an MST T on G where we add a new vertex X and a set of weighted edges from X to other vertices, calling this new graph G.

- suppose, by contradiction, we do consider edges of G not in T.
- then, we would be considering edges that reach a certain node K when it is already known that a smaller edge e reaches that node, since the edge is not in T
  → considering edges not in T in G would only consider larger edges
- This is a contradiction as we would only care about the min. edges that reach node k, so we would compare the min edge in T to the new edges from X.

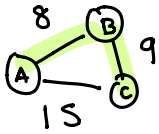
∴ When building T', you would only focus on T and the newly added weights by contradiction.

b)



G

K=6



G'

Proof claim: T is a MST of G

- suppose by contradiction, T is not an MST of G. then, there must be at least one edge e connecting a node A that can be replaced by a smaller edge e' where e'<e
- since K is a constant, if e'<e, then e'+k<e+k
- But if e'+k<e+k, our MST would have chosen e'+k and not e+k.
- This is a contradiction, so T is always an MST of G.

Name(last, first): Seo, Amy

3. (15 points: Each part has 10 points)

Consider a sorted sequence A = (a1, a2, an). B is obtained from A by a cyclic shift. So B is (aj, a(j+1), …, an, a1, a2, … a(j-1)). Design an O(log n) time algorithm that finds if an element X is in B.

Example: If A= (2 , 4 , 7, 9) then a cyclic shift of it can be B = (7, 9, 2, 4).

↑ assuming return boolean

Algorithm: modified binary search?

- Find pivot point of cyclic shift:
  - initialize L=R=0
  - while L<R:
    - m = (L+R)/2
    - if B[m] > B[R]
      - set L = m+1
    - else
      - set R = m
- perform binary search on rotated array
  - let p = L for # of pivots
  - set L=0 and R= length of B - 1
  - while L<=R:
    - m = (L+R)/2
    - calculate real mid accounting for rotation: m' = (m+p)% ℓ (let ℓ be length of B)
    - if B[m'] == X
      return true
    - if B[m'] < X
      L = m+1
    - else
      R = m-1
  - return false

Time Complexity:
O(log n) because of binary search reducing search space in half

Proof: Suppose by contradiction, our algorithm incorrectly finds if X is in B

Part 1: Suppose by contradiction, our algorithm does not find the right pivot point P

- Then, B[P+1] < B[P] and B[P-1] < B[P] if its incorrect.
- However, our loop reduces the right boundary if B[m] <= R and increases the left boundary otherwise. so B[P+1] < B[P] is not possible

[5, 6, 7, 1, 2, 3]

- It is guaranteed we find the pivot point by contradiction.

Part 2:

Case 1: X is in B, but algo says its not
- once we find the pivot point, we perform a modified BFS.
- we would have performed a binary search for X, and if X is not in here, it cannot be in B as we would have found it
  ⇒ contradiction!

Case 2: X is not in B, but algo says it is
- our algorithm only looks at elements in the array, so if it finds X at some index m, X must be in the array B which is a contradiction if X ∉ B.

∴ our algorithm always finds X if it is in B.

Name(last, first): Seo, Amy

4. (15 points)

Use dynamic Programming: Given an array of integers A[], find maximum value of the expression

(A[s] – A[r] + A[q] – a[p]) such that s > r > q > p

Example A[] = [3,9,10,1,30,40] Maximum value is 46 : (40 – 1 + 10 – 3)

(40 = largest, 1 = smallest, 10 = second largest, 3 = second smallest)

Algorithm: let l = length of A

- initialize empty array D to hold differences between array elements
- For r = 0 to j = l-2
  - For S = 1 to i = l-1
    D[S,r] = A[S]-A[r]
- For s = 3 to s = l-1:
  - For q = s+2 to l-3:
    M = max(D[s,r] for s>r>q + D[q,p] for q>p)
- return m

Proof:

- Base case: when there are 4 elements in the array, the max of the expression must be A[l-1] - A[l-2] + A[l-3] - A[0] where l is the length of the array (4).
- Inductive Hypothesis: Suppose we have n elements in the array, such that the max of A[s] - A[r] + A[q] - A[p] is known
  s > r > q > p
- Inductive step: suppose we have n+1 elements in the array.
  - we must calculate the difference between the n+1th element and all other elements in the array
  - once that is stored, we can compare it to all previously calculated differences from the inductive hypothesis.
  - It follows that either s increases, causing p, q, or r to possibly increase, or one or all of p, q, r to decrease
  - we must consider all possibilities, but because all differences are precomputed, we can do this in $O(n^2)$ time.

∴ our algorithm always returns the maximum of the given expression.

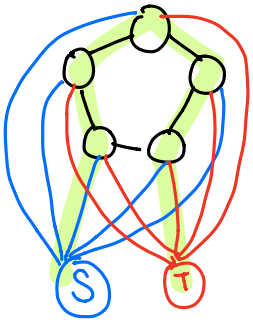
Runtime:

- precomputing D = $O(n^2)$
- for loops = $O(n^2)$ x recurrence = $O(n^2)$ = $O(n^4)$ total runtime

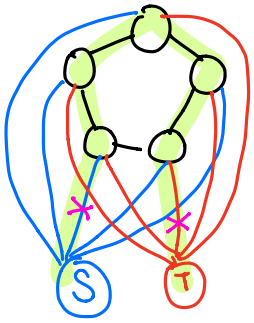
Name(last, first): Jeo, Amy

5. (15 points) A **Hamiltonian path** in a graph with **n** vertices is a path of length **n−1**, i.e., it is a path that visits all vertices of the graph exactly once (which also means no edges can be repeated). Hamiltonian path problem is known to be NP-Complete.
An ST-Hamiltonian path problem is a version of the Hamiltonian path problem where we have to start at given vertex **s** and end at a given vertex **t**. Prove that ST-Hamiltonian path problem is also NP-Complete.

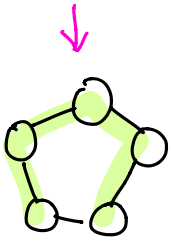
Hamiltonian path $\leq_p$ ST-Hamiltonian path



1) transformation:
- suppose we are given a graph G
- we can create a new graph G' where there is a start node S and end node T such that both S and T have edges connecting them to each node in G
- This transformation can happen in polynomial-time O(n+e) where there are n nodes and e edges



2) Proving Hamiltonian path $\leq_p$ ST- Hamiltonian path
- If we have a blackbox to solve ST-Hamiltonian path, we can perform the above transformation to G on the Hamiltonian path problem and use the blackbox to solve S-T Hamiltonian path
- Afterwards, we can remove S and T and its edges from the found path to find the original Hamiltonian path.

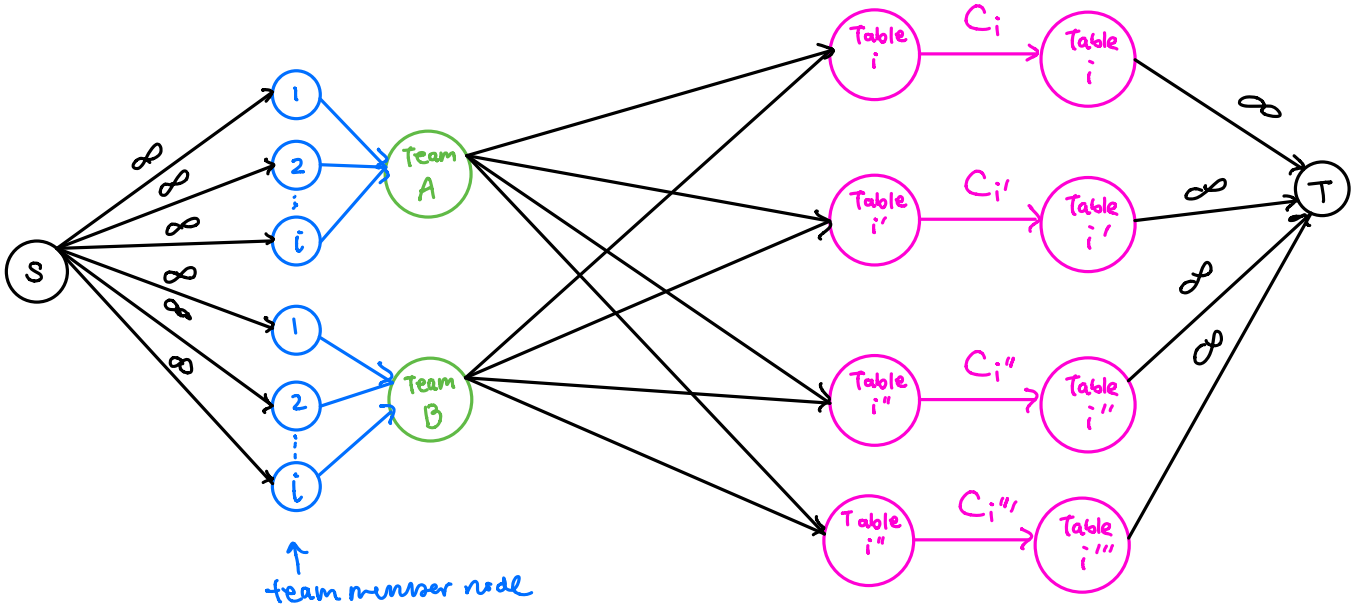


- Therefore, Hamiltonian path $\leq_p$ ST-Hamiltonian path and since Hamiltonian path is NP-complete, ST-Hamiltonian path is also NP-complete.

Name(last, first): Seo, Amy

6. (15 points) N teams attend a dinner. Team **i** has **ti** members. There are **M** tables at the dinner, with **M ≥ N**. Table **i** has **ci** chairs. We wish to seat all teams such that no two team members are at the same table. Design an algorithm for solving this problem. Prove its correctness. Analyze its time complexity.

* all unlabeled edges have weight = 1



Algorithm:

- create a start node S and an end node T with infinite capacities as shown
- let S have outgoing edges to each team member node
- let each team member node have an outgoing edge of capacity 1 that goes into a team node
- let each team node have M outgoing edges where M is the number of tables such that each outgoing edge has capacity 1 and each edge goes to each table node
- Let the table nodes be connected to a copy of itself with a capacity C_i, where C_i is the number of chairs at this table.
- let each end node of the table have an outgoing edge of ∞ capacity to end node T
- run Ford & Fulkerson

Proof: By Ford & Fulkerson, we know we have reached the max # of people that can be seated, we must show the network set up is correct

(1) Suppose our algorithm returns the incorrect number of people at each table such that 2 members of the same team share a table
- Our algorithm only sends one person max from each team to each table, thus this is impossible.

(2) Suppose our algorithm seats a table beyond its capacity.

- our setup creates an edge between the 2 table nodes for each table with a capacity of how many people it can seat.
- Following this, there can never be a flow higher than this capacity, so there can never be an instance where more people are seated than the table's number of chairs

∴ Since our algorithm does not match same team members to the same table nor seat tables beyond its capacity, we can seat all team members on the same team at different tables, assuming there are enough tables.

Runtime: $O(|f| \cdot (n+e))$

- It takes polynomial time $O(n+e)$ where n are all the nodes (team members, teams, tables etc. as shown in the diagram) and e are all the edges to setup the network diagram
- Running ford & fulkerson takes $O(|f| \cdot (n+e))$ time

∴ Total = $O(|f| \cdot (n+e))$