90

Name(last, first): [redacted]

UCLA Computer Science Department

CS 180 **Algorithms & Complexity** **ID :** [redacted]

Midterm **Total Time: 1.5 hours** **October 31, 2017**

Each problem has 20 points.

All algorithm should be described in English, bullet-by-bullet

1 Consider a set of intervals I1, …, In. **a.** Design a linear time algorithm (assume intervals are sorted in any manner you wish) that finds a maximum subset of mutually non-overlapping intervals. **b.** Prove the correctness of your algorithm.

a) • Assume intervals are sorted where the first interval finishes first

Algorithm:

- Pick the interval that finishes first, and add to subset
- Remove all overlapping intervals with this interval
- Repeat above 2 steps until no intervals are left

b) Proof:

- Our algorithm creates a subset H with k intervals and there is some optimal subset O
- O must have at least k+1 intervals to be better than H
- Basis: since our algorithm picks the job that finishes first, O cannot pick an interval that finishes earlier. Our algorithm is at least as efficient as O at this point
- Step: – Assume first i intervals are the same in H & O subsets.
  - i+1 interval that H picks will finish before or at the same time as O's i+1 interval
  - H is again at least as efficient as O, and repeating above steps will result in k intervals for O
  - CONTRADICTION: O cannot have k+1 intervals since H manages to stay ahead or on pace with O, so our algorithm is optimal
- Our algorithm only scans through the intervals once so it runs in linear time O(n).

Name(last, first): [redacted]

10

2 . a. Design an efficient algorithm better than O (n^2) to be used in sparse graphs for finding the shortest path between two vertices S and T in a positive weighted graph. **b.** Justify the correctness of your runtime analysis.

a) Algorithm:

- Start at S and add it to set Z, and add all neighboring weighted vertices to set Y
- Heapify set Y so the smallest weighted vertex is at the top
- Follow the smallest weight to vertex v, (here if v = T (if so, end algorithm), then check if v is already in Z
- If v is not in Z, add v to Z and neighboring weighted vertices to heap Y and reheapify

-10

- Repeat above 2 steps until T is discovered, then follow path from S to T in Z

*Runtime for m edges and n vertices: O (m log n)

b) Proof:

- Obtaining min item from heap takes O(1) time and reheapifying takes O(log n) time
- Each time we check an edge and obtain a path to vertex v, we reheapify so we get O(m log n)

. This is better than $O(n^2)$ in sparse graphs because $m \leq n^2$, but in sparse graphs like trees, m is closer to n-1 (linear n). Thus, O(m log n) would be comparable to O(n log n) which is more efficient than $O(n^2)$.

Name(last, first): [redacted]

20

3. Consider a sequence of positive and negative (including zero) integers. Find a consecutive subset of these numbers whose sum is maximized. Assume the weight of an empty subset is zero. **a.** Design a linear time algorithm. **b.** Prove the correctness of your algorithm.
Example: For the sequence ~~2~~ 4 -3 5 -12 the maximum sum is ~~4~~ 6.

a) Algorithm:

- For each element have a sum variable, and have a global maxSum variable
- For each element:
  - check if sum of last element plus value of this element is negative. If so, set sum of this element to 0
  - If not, set sum of this element to its value plus the sum of the last element
  - If this element's sum is greater than maxSum, set maxSum to sum
- Return maxSum

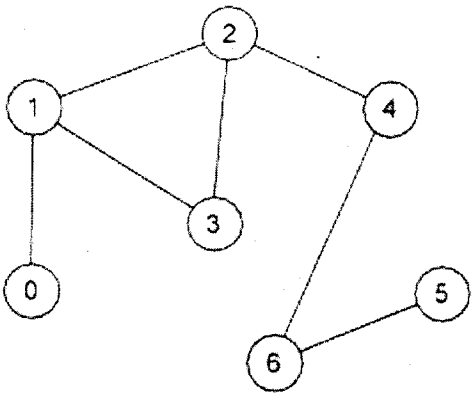
b) Proof:

- First i elements of sequence are positive or 0 elements so they can all be summed together w/o consequence
- 4 cases:
  - i+1 element is positive: simply add to sum & move on
  - i+1 element is negative and all elements after would make the sum smaller: each negative element's sum is smaller until it goes below 0, or rest of numbers don't add up to maxSum, either way maxSum is unaffected & remains correct
  - i+1 element is negative, but future elements will make sum bigger: negative elements temporarily reduce their sums value, but future positive elements bring it up past maxSum, still obtaining correct value
  - i+1 element is negative, and desired maxSum's subset lies entirely ahead: sum for each negative element would bring it down past 0, but sum would be set to 0 so it doesn't affect the sum of future positive elements. maxSum still remains accurate & unaffected by negatives.
- Since our algorithm only looks at each element once in its linear scan, it runs in O(n) time so it is a linear time algorithm

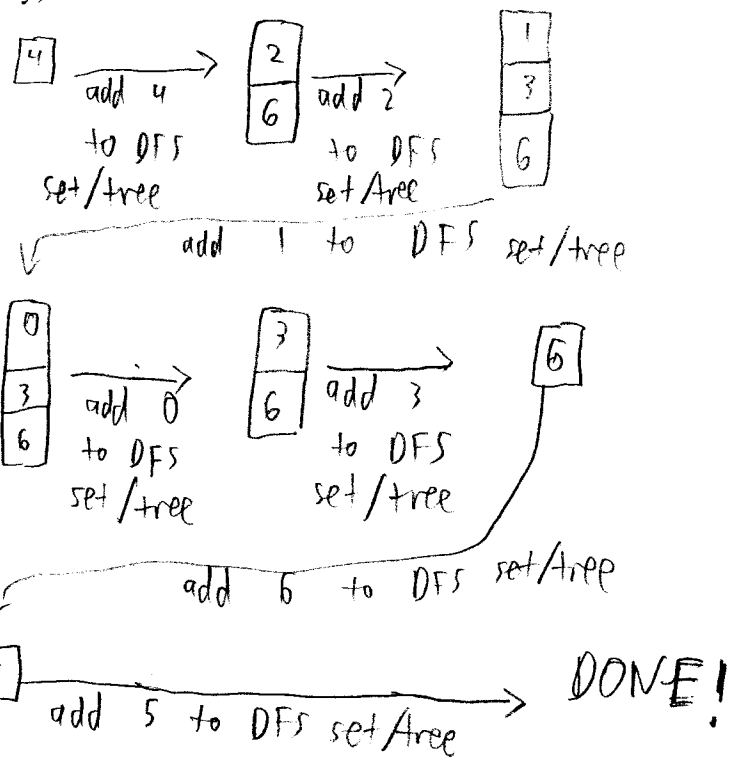
Name(last, first): [redacted]

20

4. Consider an unweighted graph G shown below. **a.** Starting from vertex **4,** show every step of DFS along with the corresponding stack next to it. **b.** What is the run time of DFS if the graph is not connected (no proof is necessary)?



a)



b. For m edges and n vertices, runtime is:

$$O(m + n)$$

20

Name(last, first): [redacted]

5. Consider a binary tree (it is not necessarily balanced). The tree is not rooted. Its diameter is the distance between two vertices that are furthest from each other (distance is measured by the number of edges in a simple path). Design a linear time algorithm that finds the diameter of a binary tree.

Algorithm:

- Start at any arbitrary vertex and perform BFS to get a BFS tree
- Take a vertex in the last layer of BFS tree and perform BFS from that vertex to create new BFS tree
- Diameter is equal to the number of the last layer (distance to the furthest vertex from this vertex) in this new BFS tree

Justification:

- Performing BFS on a tree does not remove any edge since there is only one possible option, so each BFS tree in this algorithm is identical to the other ones but merely shifted, so diameter of one equals diameter of others
- The first BFS will give a vertex that is at one end of the tree. The second BFS from that point will give the other end since we are starting BFS at the first end.
- BFS gives the shortest (and in this case only) distance between 2 vertices, so BFS between 2 ends will give us the maximum distance, which is the diameter.
- Performing BFS takes linear time $O(m+n)$ in any general case for m edges and n vertices. However, since we have a tree, $m = n-1$, so BFS runs in $O(n)$ time. So total rest of our algorithm is $O(n) + O(n) = O(n)$ so we have a linear time algorithm to find the diameter.