Name(last, first): Luo, Juncai

85

# UCLA Computer Science Department

**CS 180** **Algorithms & Complexity** ID : 304612962

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

Each problem has 20 points.

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

s e s e s s e s e

20 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.

next page

a)
- Assume each interval has a start point, $s_i$, and an end point, $e_i$.
- Assume all intervals are sorted in increasing time order on one timeline.
- So $s_1$ would be the leftmost point on the timeline, and $e_n$ would be the rightmost
- Start from the leftmost point on the timeline, $s_1$, with counter $C_1$
- Start another counter, $C_2$, on the next point on timeline
- If the point is a $s_i$, then we put $C_1$ here & restart the 1st step.
- If the point is a $e_i$, then we increment current_max by one, and put the interval $(s_i, e_i)$ in our subset
- compare current_max with overall_max, which was initially set to 1
- If current_max > overall_max, update overall_max.
- Otherwise, change $C_2$, to the next point on timeline & repeat the process
- Algorithm ends when we've traversed all points.

b)
- Since min value of max non-overlapping intervals is 1 interval, we set overall_max to 1
- Suppose there exists a subset of size K, which is larger than the size, n, output by our algorithm. i.e. $K > n$.
-

1.

a). • we employ greedy algorithm.

• assume intervals are arranged by ~~[x]~~ earliest finishing time.

• Start from the 1st interval (finishes 1st)

• We have an subset of all intervals.

• if there's any interval overlapping with our 1st interval, remove from subset.

• Then go to the next interval that starts after 1st interval.

• Remove overlaping intervals.

• After we've traversed the interval with latest finishing time, We have a subset of max non-overlapping intervals

b) • Assume algorithm O that is an optimal solution.

• ~~Prove by induction.~~

• ~~Base case: 1 interval, our algorithm G, will pick it since it picks earliest finishing time~~

• Suppose G & O are the same until kth step

G has picks an interval ends at time a. O picked interval ends at time b.

and a < b. because G always picks the earliest finishing time

• Prove for k+1 steps.


• the next available interval i that picked by O starts after the interval that O picked at kth step.

• Since a < b, the interval that O picks at k+1 step will be available to choose for G as well.

• So G can pick all the rest of intervals that O will pick

• G will have at least as may intervals as O.

• So G is optimal.

20

**Name(last, first):** Luo, Juncai

elogn $e \cdot n^2$

**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).
- Use a heap structure to store all nodes of graph, with S as root
- All nodes have value of infinity, but S has value 0.
- remove S and put it first in our sequence from heap & look at neighbors of S on the graph.
- Update values of distance from S to its unvisited neighbors, v, into the node in heap.
- Heapify the heap so that the smallest value is at root.
- Remove root & put in our sequence.
- Hepify the heap again so we have a root again.
- Find unvisited neighbors of nodes we moved in sequence & update their values in heap
- repeat this process until we have removed all nodes in heap.
- Algorithm takes $O(m\log n)$.

b).
- Constructing a heap takes $O(n\log n)$. Since putting each node in heap costs $O(\log n)$ since heap has $\log n$ levels. We do this for $n$ nodes.
- Update a value in heap costs $O(\log n)$, since to find a node in heap takes $O(\log n)$
- For each edge, we need to update the value in heap once.
- So in total, we've updated $m$ edges, each takes $\log n$. So $O(m\log n)$.
- Everytime we remove a root, it costs $O(1)$.
- Then heapifying takes $O(\log n)$
- We do this for $n$ nodes. So in total $O(n\log n)$.
- So we have $O(m\log n + 2n\log n)$
- Since $m$ is greater than $n$, we have a overall runtime $O(m\log n)$

20

Name(last, first): Luo, Juncai

Return max_sum

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

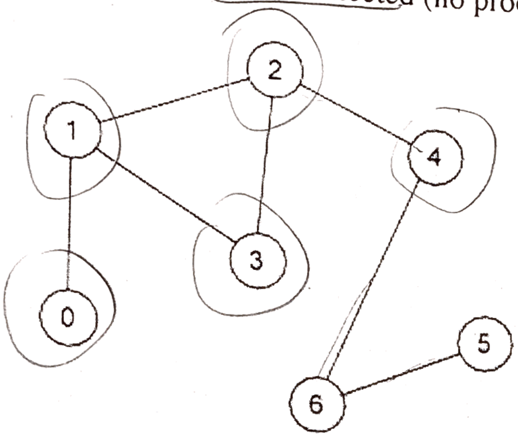
a). • We start from the left of a sequence. keep a counter $C_1$.
- We use $C_2$ to keep track of the end of a subset. Put $C_2$ to the first point in sequence
- We have a current_sum, keeping track of current maximum sum & max_sum, keeping track of overall max sum.
- Initialize max_sum to 0.
- when $C_2$ points to a new number, update current_sum
- If current_sum > max_sum, update max_sum.
- If current_sum < 0, we restart $C_1$ & $C_2$ to the next point in sequence
- repeat those steps until $C_2$ has reached the end of sequence
- O(n)

b). • Since the minimum max sum of the sequence can be any one positive integer, sum < 0 means we can no longer have a max sum with negative sum, meaning the subset will not be the one we're looking for
- Suppose there exists a larger sum b that is larger than our sum.
  b > a
- Then it must because the subset sacrifice adding a negative number to get a bigger number.
- Yet, if the negative number makes our current sum negative, the next number will not be as big as the sum if we restarted the subset from the next number.
- If the negative number still makes our sum positive, our algorithm will not restart a subset
- Contradiction, so our algorithm is optimal.

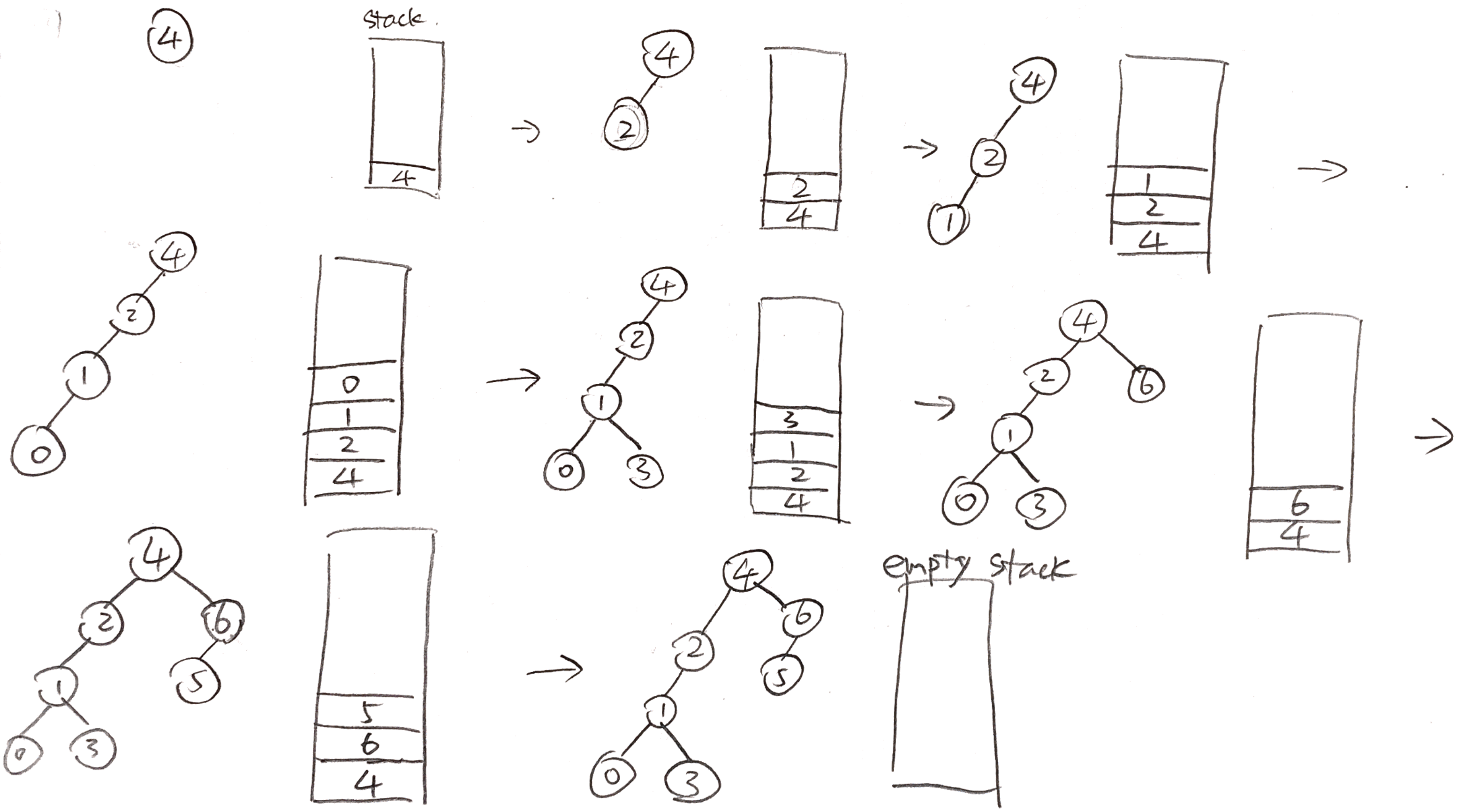
20

Name(last, first): Luo, Juncai

**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). • If the graph is not connected, we need to do DFS on each connected component

• We need to visit all nodes $O(n)$

• We need to visit at most all edges $O(m)$

• So runtime is still $O(m+n)$

3. 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.

- Do a BFS on the binary tree.
- Start from an arbitrary node v and find its unvisited neighbors.
- mark the neighbors $L_1$.
- For each of these neighbors, u, find u's unvisited neighbors & mark them $L_2$.
- Repeat the process, everytime we go to a new level of nodes & find their neighbors, we increment $L_i$ by one.
- After visiting all nodes, we are at $L_k$ where k is the number of levels in the BFS tree.
- Then do DFS on the node we chose first.
- Keep count of edges we go through with $E_j$,
- until we need to back track.
- Diameter $= E_j + L_i$. where i & j are the largest number we recorded

b

- Since BFS & DFS each take $O(m+n)$.
- Solution is still linear time.

BFS

~~it~~ recursively

assume you know the longest path

- 2 operations per node