75

Name(last, firs [redacted]

UCLA Computer Science Department

CS 180

Midterm

Algorithms & Complexity

Total Time: 1.5 hours

[redacted]

October 31, 2017

Each problem has 20 points.

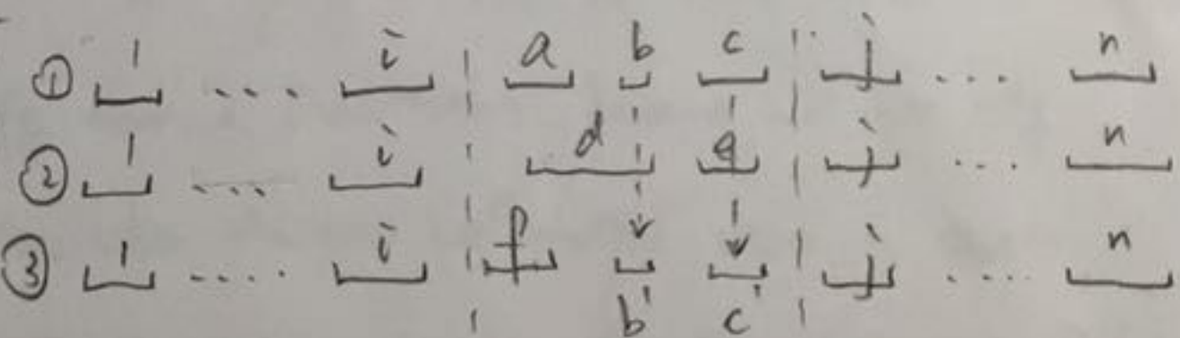
All algorithm should be described in English, bullet-by-bullet

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.

20

a) • Consider the whole set of interval be S and an empty set of A.
• Find the interval that ends the first from S extract it from S and add it to A.
such that the subset S now contains that interval and S alone not.
• Remove all other intervals that overlaps with that interval in S.
• Find the interval that ends the first from the remaining intervals from S
delete it from S and add it to A. and remove other overlapping intervals
with the extracted interval from S.
• repeat the above step until there is no more intervals in S
• return A, which is the maximum subset of mutually non-overlapping interval.

b) Consider the graph in the right assume that the intervals labeled 1 to i and j to n are exactly the same ($i \le j \le n$) Note that it is possible that the subset (1 … i) and (j … n) to be empty. We assume that choice ① is the optimal



If the first end interval is chosen.
in this case. f. then we delete all other overlapping intervals. a and d, then we can at least choose the intervals b and c, which makes my algorithm not worse than the optimal, if not better. We can exchange the interval in the optimal solution from a to f which still makes the solution optimal., if not better. After we swap all such violations, we can see that the optimal solution is in fact my solution.

10

Name(last, first): Fang Kaijie

O(E+n)

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.

Dijstra's Algorithm

a). - Initialize all vertices with infinity and the starting node S to be zero. Mark all vertices to be 'unvisited'.

- for all the neighboring vertices of the current node calculate the total weight from the starting node to that neighboring node by adding the value in that node with the value of the edge connecting that node to the neighboring node. If the value is larger than the value in the neighboring node, update it and move on to that neighboring node, mark the vertex to be visited. If the value is smaller than or equal the value in the neighboring node, do not modify it.

-5

- continue this process until we mark the destination T. and we can return the path from S to T which is the shortest.

b) $O(n^2)$ is a pessimistic calculation. In the my algorithm, we write each edge only once and we are not going to look at them again, this is O(e). and each vertex we only modify them a linear times. because we are only going to update the value in vertices when the new calculated value is smaller than the original value. Therefore, the time complexity of the solution is O(e+n) which makes it O(n)

-5

5' Name(last, first): Fang Yujie

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 -3 5 -12 the maximum sum is 4. 6

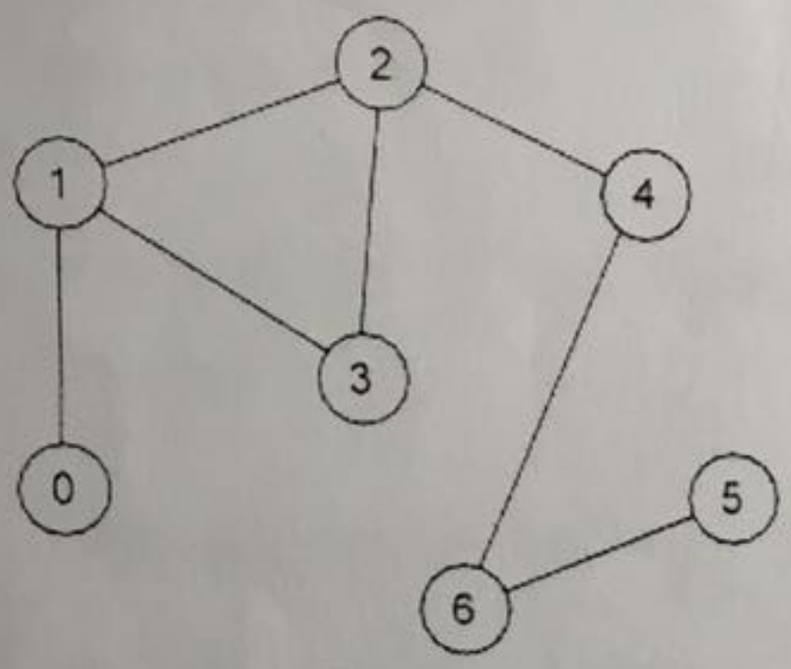
-3 4 -3 5 -12
-3 1 2 3 -9

a) - Add the first element of the sequence to zero. put it into a list which start with 0.
In the example, the sum is 4 [0|4|...] Set a negative flag1=-1. and flag2=-1
- Add the next element of the sequence to the previous total weight. put it into the list.
In the example, the sum is 1 [0|4|1|...] If the sum number is positive, set the flag1 to 0.
If the element itself is positive, set flag 2 to 0
- repeat the above steps until we reach the last element of the sequence
In the example, we got the list [0|4|1|6|-8] -6 wrong It takes O(n-1)=O(n) time steps.
- Go through the list from the start to the end, find the maximum value if flag1=0 and flag2=0
Else if flag2=0 go through the list from the second number to the end + find the maximum value
Else, go through the original sequence and return the largest value directly. It takes O(n)
- Go through the list again from the start and stop at the maximum point in the previous step.
(This takes O(n) the maximum) find the minimum value kmin
- The maximum sum of such a consecutive subset is kmax - kmin.
- This algorithm takes O(n+n+n)=O(n) which is a linear time.

b) The time complexity correctness is proven in part a). To show that it is indeed the maximum sum, we can use contradiction.
Assume that the result is a lager value than my maximum value, we either have a value kmax' larger than kmax or kmin' smaller than kmin such that (kmax'-kmin)> (kmax-kmin)
If there is a kmax' which is the largest value in the list, we would already chosen it to be the end point of the sequence Thus. kmax is at least equal if not smaller to kmax. If there is a kmin' which is the smallest value in the list from the starting value to the maximum value, we would already chosen it as kmin by my algorithm. Thus. kmin' is at least equal if not lager than kmin By contradiction. my algorithm is correct.

Name(last, first): Fang Kejie

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). unmark all vertices

start from 4. mark it as visited → 4

find 4's child. 6. put in stack, mark → 6 4

find 6's child, 5. put in stack, mark → 5 6 4

find 5's child. no child, pop it → 6 4

find 6's child. no child pop it. → 4

find 4's child 2. put in stack, mark → 2 4

find 2's child. 3. put in stack, mark → 3 2 4

find 3's child. 1, put in stack, mark → 1 3 2 4

find 1's child 0, put in stack, mark → 0 1 3 2 4

find 0's child, no. pop it → 1 3 2 4

find 1's child, no, pop it → 3 2 4

find 3's child, no, pop it. → 2 4

find 2's child, no, pop it → 4

find 4's child, no, pop it

stack

☆ NOTE: mark means mark it to be visited

child here means unvisited child.

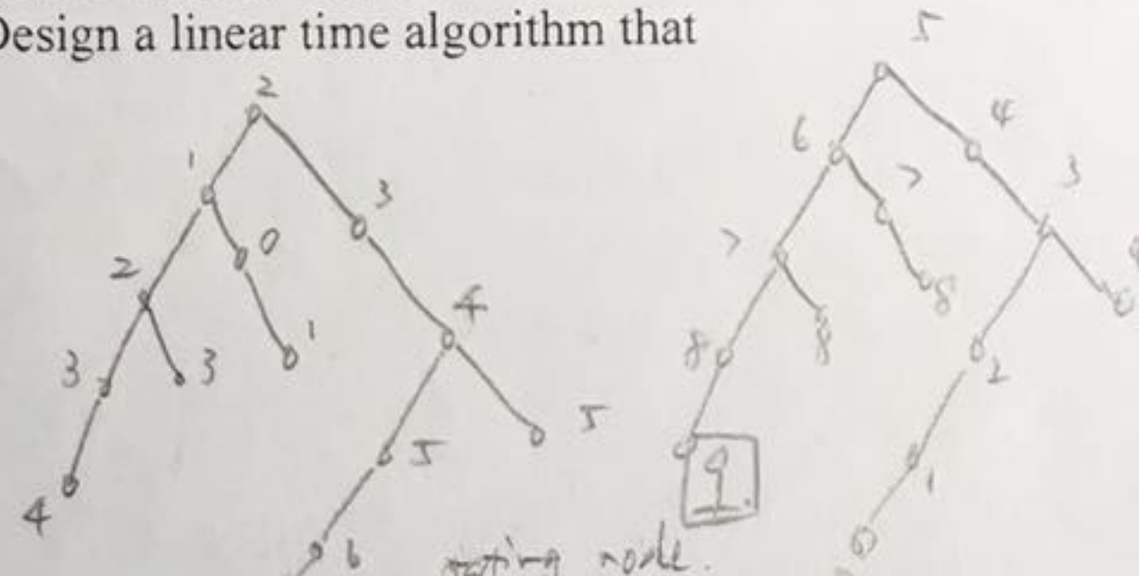
4 → 2 → 3 → 1 → 0.
↓ ↓
6 5

b). O(n+e)

20

Name(last, first): Fang Yujie

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.



(the graph help explains)

- we can use BFS to help to solve this question because it asks for the largest path from two points.

- starting from an abitrary vertex v. mark it with integer 0. and visited.
- for v's unvisited neighbor vertex w. mark it as visited and increment its integer by 1.
- repeat the above step until all vertices are marked as visited.
- we can find the vertex that is marked the the largest value.

- set this vertex as the root and we run BFS again.
- unmark all vertices first. and the root vertex with integer 0
- starting from that root, for each of it's unvisited neighbor vertex. mark it as visited and increment their integer by 1.
- from the neighbor vertices, find the neighboring vertices of the neighboring vertices mark them as visited and increment their integer by 1
- repeat the above step until we have all vertices marked as visited and the largest integer appended to any vertex in the tree will be the diameter of the binary tree.
- Since we only run BFS twice and update making status as well as integer status twice for all vertices. the runtime is $O(\underbrace{e+n}_{\text{this is BFS}} + \underbrace{2n}_{\text{mark}} + \underbrace{2n}_{\text{integer updates}}) = O(e+n)$. we have a linear time algorithm.