Name(last, first): Xie, Hongyi

UCLA Computer Science Department

CS 180 **Algorithms & Complexity** **ID: 604998533**

Midterm **Total Time: 1.5 hours** **November 6, 2019**

Each problem has 20 points.

All algorithm should be described in English, bullet-by-bullet (with justification) You cannot quote any time complexity proofs we have done in class: you need to prove it yourself.

Problem 1: Describe the topological sort algorithm in a DAG. Prove its correctness. Analyze its complexity. Assume n nodes, m edges

Algorithm & Complexity

- Create an empty list.
- Loop through all nodes to get their number of incoming edges. This takes O(n).

loop starts →

- Arbitrarily append a node with no incoming edges into the list.
- Remove the node and all of its outcoming edges.
- Update number of incoming edges for the nodes that can be reached directly from the just-deleted node.

Takes O(m) for the whole loop.

- Repeat until all nodes are in the list. The order of nodes in the list is the topological order.
- If at any point there is no node with no incoming edges, the graph is not a DAG.

Takes O(n) for the whole loop.

loop ends →

The algorithm takes O(m+n) in total.

Proof of Correctness

The algorithm is correct if in the resulting list, there is no pair of nodes i, j such that j comes after i but there is a path from j to i. We'll proove by contradiction.
Assume the is such pair. Then at the time i is added into the list, it must have at least one incoming edge since j still presents in the graph, which contradicts with how we run the algorithm. So no such pair exists and the algorithm is correct.

Name(last, first): Xie, Hongyi

Problem 2: Run Merge sort on the following set of numbers. Show every step. Analyze the time complexity of merge sort on a set of **n** numbers (show every step)

9 3 6 2 4 1 5 7

"|" indicates dividing the list.

1. 9 3 6 2 | 4 1 5 7

2. 9 3 | 6 2 | 4 1 | 5 7

3. Sort within each sublist.

→ 3 9 | 2 6 | 1 4 | 5 7

4. start merging.

• 3>2

→ 2 3 9 | 6 | 1 4 | 5 7

• 3<6, 9>6

→ 2 3 6 9 | 1 4 | 5 7

• 1<5, 4<5

→ 2 3 6 9 | 1 4 5 7

• 2>1

→ 1 2 3 6 9 | 4 5 7

• 2<4, 3<4

→ 1 2 3 4 6 9 | 5 7

• 6>5

→ 1 2 3 4 5 6 9 | 7

• 9>7

→ 1 2 3 4 5 6 7 9

Continues at back.

Complexity

- Merge of two sorted lists (each of length $\frac{n}{2}$) takes $O(n)$, since each comparison puts one element in the final list.
- We divide the initial list until we can do sort in constant time ($O(1)$) within each sublist. The resulting number of sublists is $\log n$.
- So we do $O(\log n)$ times of merge in the algorithm and the resulting complexity is $O(n \log n)$.

Name(last, first): Xie, Hongyi

Problem 3: Suppose that you are given an algorithm as a blackbox. You cannot see how it is designed. The blackbox has the following properties: If you input any sequence of real numbers, and an integer **k**, the algorithm will answer YES or NO indicating whether there is a subset of the numbers whose sum is exactly **k**. Show how to use this blackbox to find the subset whose sum is **k**, if it exists.
You should use the blackbox O(**n**) times (where **n** is the size of the input sequence).

Algorithm

- On the original set, consult if such subset exists. If no, we are done.
- If yes, loop through all numbers i in the set. Initialize $k' = k$.
- For each number i, extrait it from the set. Ask if there exists a subset whose sum is $k' - i$ in the remaining set.
- If no, i cannot be in the targeting subset. Don't change k' in this case.
- If yes, put i in the targeting subset. Change k' to $k' - i$.
- We have the targeting subset after we loop through all numbers. At this time k' should be 0.

We use the blackbox once per number in the set so it's $O(n)$.

Name(last, first): Xie, Hongyi

Problem 4: You have been commissioned to write a program for the next version of electronic voting software for UCLA. The input will be the number of candidates, **d**, and an array votes of size **v** holding the votes in the order they were cast where each vote is an integer from 1 to **d**. The goal is to determine if there is a candidate with a majority of the votes (more than half the votes) . You can use only a constant number of extra storage (note that **v** and **d** are not constants).Prove the correctness of your algorithm and analyze its time complexity.

Algorithm & Complexity

- Change the array into a linked list. This takes $O(v)$.
- loop starts → Partition the list into sublists of length 2.
- Compare the votes within each sublist.
- If they are equal, disregard one of them, i.e. save it's pointer but remove it from the list
- If they are not equal, disregard both of them.
- Now we have a list of half of the initial length. Repeat until we have only 1 (if the last 2 votes agree) or 2 (if the last 2 votes don't agree) votes left in the list.

The whole loop takes $\frac{1}{2}n + \frac{1}{4}n + \ldots = O(n)$

- loop ends → For the remaining votes, loop through all votes we have at the beginning to see if it's a majority vote. This takes $O(n)$. If no, there is no majority vote.

The whole algorithm takes $O(n)$.

Correctness

- If there is a majority vote, it has to be voted for n times where $n > \frac{1}{2}V$.
- According to how we disregard votes in the algorithm, we disregard at most 1 vote from the majority vote at a time.

Continue next page.

- So when each loop finish, we have at most $\frac{1}{2}v$ votes remain, in which there are at least $\frac{1}{2}n$ votes from the majority vote.
- Since $n > \frac{1}{2}v$, $\frac{1}{2}n > \frac{1}{2}(\frac{1}{2}v)$, so the majority is still a majority in the remaining list.
- Therefore the last remaining vote(s) are the only candidate of majority. We just do a check at last.

Alternative Algorithm

- For each candidate d, loop through the array to count it's frequency. Maintain a sum of this count.
- If any count is greater than $\frac{1}{2}v$, that vote is majority.
- If at any point the sum of count is greater than $\frac{1}{2}v$, we know there cannot be a majority.
- This takes $O(vd)$.

Name(last, first): Xie, Hongyi

Problem 5: Consider a sorted list of **n** integers and given integer **L**. We want to find two numbers in the list whose sum is equal to **L**. Design an efficient algorithm for solving this problem (note: an $O(n^2)$ algorithm would be trivial by considering all possible pairs). Justify your answer and analyze its time complexity.

Algorithm

- Loop from the start of list.
- For each integer i, calculate L - i.
- Perform binary search on the list to find if (L - i) is in the list. If yes, we are done. If no, repeat for the next integer in the list. Note we don't even have to run binary search if the largest integer in the list is less than L - i, or if the smallest is greater than L - i.
- If we loop through all integers in the list and no such pair is found, we conclude no two integers in the list add up to L.

Correctness

The algorithm is correct due to correctness of binary search on a sorted list, i.e. it always finds L - i if it exists, and i + (L - i) = L which is what we want.

Complexity

Binary search takes O(log n) in each loop, and we loop for n times, so resulting complexity is O(n log n).