Name(last, first): Wang, Haoyang

UCLA Computer Science Department

CS 180 **Algorithms & Complexity** ID : 804920901

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.

Algorithm
1. find in-degree of all nodes
2. choose an arbitrary node with 0 in-degree, remove from graph and output
3. update the in-degree of the removed node's neighbors
4. repeat 2 and 3 until every node is outputted

Analysis: since every DAG has at least one node with 0 in-degree, each iteration can be done. And since we are keep outputting node with 0 in-degree meaning with no prior node (incoming node), the output is in topological order.

n nodes, m edges

Runtime: Step 1 takes $O(n)$. For 2 and 3, we update in-degree of neighbors for each edge removed with the node, so the entire iterations take $O(m)$. So, the overall runtime is $O(m+n)$.

Name(last, first): Wang, HaoYang
804920901

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

Recursion like this:

sort (9 3 6 2) — sort (9 3) 3 9
sort (6 2) 2 6
Merge 2 3 6 9

sort (4 1 5 7) — sort (4 1) 1 4
sort (5 7) 5 7
Merge 1 4 5 7

Merge 1 2 3 4 5 6 7 9

Note: for 2 numbers, sort can sort them in one comparison; for 1, no operation is needed. Merge keeps comparing first of sorted lists and put the smaller one to output.

Runtime = Given $T(2) = 1$, $T(1) = 0$ and $T(n) = 2T(n/2)$ + runtime of merge. With n numbers, the final merge run in $2n$ times at worst.

So, $T(n) \leq 2T(n/2) + 2n \leq \log_2 {n/2} \times 2 \times T(2) + 2 \times \log_2 {n/2} \times n \leq c_1 + c_2 n\log n$

So, $T(n) \leq O(n\log n)$, Mergesort runs in $O(n\log n)$.

1 10 21 4 22

4 | 6 7 5 11

6 5

6 5

Name(last, first): Wang, Haoyang

804920901

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:

1. put the entire sequence in, if No, we are done; if Yes, continue.

2. ~~put the subset of the original~~ for each element in the original sequence, put the subset of all numbers excluding it and excluding all prior Yes numbers(s) in; if No, output this element; if Yes, continue, and in the following iteration, we not only exclude the element iterated but also this Yes number and all prior Yes numbers.

The output is the subset whose sum is k

Analysis: Suppose there is Yes in step 1. Then, any No in any iteration of step 2 means without this element, the sequence no longer has a subset whose sum is k. And since with this element it has, this element must be in the subset. ~~So, if we do this on every element, the output result will be the subset~~

And, for any Yes in any iteration of step 2, even though this element might be able to contribute to the subset, since the remaining sequence still has such a subset, we can skip this element and exclude it in later iteration.

we also exclude prior Yes numbers to ensure the output subset has sum k exactly.

Runtime: since it's a linear traverse of all element, in each iteration we use the black box once it runs in O(n).

1 2 3

Name(last, first): Wang, Haoyang
804920901

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: 1. use one extra storage to remember a candidate
and use one extra storage to count vote
2. initially count is 1, candidate is the first in array v.
3. traverse the array, for each element
a. compare with candidate,
if the same, count increase by 1
else count decrease by 1
b. check count, if count reaches 0
candidate is the next element, count is 1;
if no next element, candidate is null
4. in the end, if candidate is not null,
traverse the array v, use count to count the occurance of the candidate; check if count is bigger than half the size of v

Analysis: since removing 2 different votes doesn't affect the condition of majority, the comparison made in this algorithm can have the correct candidate left. And we need to check since it might not be the majority.

Runtime: step a, b both take constant time; so, in the worst case, we will traverse the array v twice, the runtime is $O(n)$ where n is the size of array v.

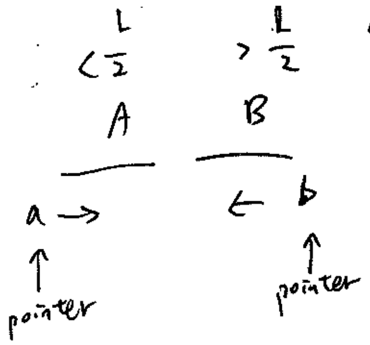
~~1→2~~ ~~3 4~~ ~~5 6~~

Name(last, first): Wang, Haoyang
804920901

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.

Suppose sorted increasingly

1. Use a linear search on the list to exclude all elements larger than L
2. Divide the remain numbers to two sorted groups,
   one s.t. all numbers in it smaller than $\frac{L}{2}$ (A)
   one s.t. all numbers in it larger than $\frac{L}{2}$ (B)
3. use two pointers a b, → smallest
   a points to start of A, b points to end of B largest
4. keep comparing L with sum of *a, *b
   if L larger than that, a points to next bigger element in A
   if L smaller than that, b points to next smaller element in B
   if equals, we find the two



5. if we doesn't find the two when one of the pointers can't move according to rule in 4 or a reaches end of A and b reaches start of B
   no such two exists.

Analysis: By dividing in step 2, we are guaranteed that if such a pair exists, one is from A, the other is from B since any sum of two in A will be ~~larger~~ smaller than L and any two in B sum larger than L. And by using pointer to compare, since it's sorted, we can eliminate multiple potential pairs in one pointer move.

Run time: Since the pointers won't go far away from each other, and in iteration of step 4, one pointer is moved, in the worse case, the pointer movement equals the number of elements. So, this algorithm runs in O(n).

Since the pointer move is always meaningful and correctly eliminate wrong pairs, we will discover the pair or ensure it doesn't exist in the end.

L

if 2 $\le \frac{L}{2}$ $> \frac{L}{2}$

a+b c+d

1 2 3

d/10

1 3 6 17 18

1 3 2 4

1 10 20 100 500 | 1000

1 3 6 17 18

1 3 6 17 20

1 3 6
17 18

3 6

6

1 | | | | | |

20

| | 1 3 4 6 | 17 18 |

| | | | | | | | | | | | | | |

| 1 | 3 6 17 | 18 |