CS180 midterm

[redacted]

TOTAL POINTS

100 / 100

QUESTION 1

1 problem 1 **20 / 20**

✓ **- 0 pts** Correct

QUESTION 2

2 problem 2 **20 / 20**

✓ **- 0 pts** Correct

QUESTION 3

3 problem 3 **20 / 20**

+ **3 pts** basic understanding of the question
✓ **+ 5 pts basic understanding of the question is correct**
✓ **+ 10 pts Correct algorithm**
+ **8 pts** Partially correct algorithm
+ **3 pts** Partially correct algorithm
✓ **+ 5 pts runtime analysis and justification**
+ **0 pts** wrong approach
+ **0 pts** no answer
+ **3 pts** Some clues were right but the overal approach was not correct

QUESTION 4

4 problem 4 **20 / 20**

✓ **+ 5 pts Complete proof of correctness**
✓ **+ 5 pts Complete complexity analysis**
✓ **+ 10 pts Correct algorithm**
+ **3 pts** Correct complexity with analysis error
+ **3 pts** Proof of correctness had minor errors
+ **8 pts** Good algorithm, minor errors
+ **5 pts** Incomplete algorithm
+ **0 pts** Algorithm uses non constant storage
+ **0 pts** Complexity analysis is wrong
+ **0 pts** Proof of correctness is wrong
+ **0 pts** Algorithm is wrong

QUESTION 5

5 problem 5 **20 / 20**

✓ **- 0 pts** Correct

Name(last, first): [redacted]

UCLA Computer Science Department

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

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

- calculate the in-degree of every node n in G
  - → for every edge in G, increment the in-degree of the node that the edge points to
- maintain a set S of all nodes whose in-degree is 0
- while S is not empty:
  - output some arbitrary node v in S
  - decrement the in-degree by 1 of all nodes u s.t. (v, u) is an edge
  - if the in-degree of some node u is now 0, move u to S
  - delete v and all edges (v, u) from G
- final output is now a topological ordering

Proof

- Assume, by contradiction, there exists some edge (v, u) s.t. u appears before v in the ordering
- for our algorithm to have outputted u, u must have had an in-degree of ϕ
- however, b/c v hadn't been outputted & deleted yet, u must have had an in-degree of at least 1
- therefore, CONTRADICTION – u would not be ordered before v

Runtime

- calculating in-degree's initially takes O(m) since we look at every edge
- we must output n nodes – O(n)
- we delete each edge once & decrement count of the second node in each edge – O(m)

- total – O(m+n)

Name(last, [redacted]

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.

divide: 9 3 6 2 | 4 1 5 7

divide: 9 3 | 6 2 | 4 1 | 5 7

div: 9 | 3 | 6 | 2 | 4 | 1 | 5 | 7

merge: 3 9 | 2 6 | 1 4 | 5 7

merge: 2 3 6 9 | 1 4 5 7

merge: 1 2 3 4 5 6 7 9

* at this step, each partition is sorted

* merge by maintaining a pointer to each list (2 lists)
* choose smaller item and add to merged list
* increment pointer from list that had smaller element
* repeat until both lists are empty

Runtime

- O(n log n)
- each step for the divide phase of the algorithm breaks each array of size n into 2 arrays of size n/2 until we get to one element each
- this therefore takes log n iterations
- merging also takes log n iterations to recombine all elements into one array
- each iteration of merging takes n steps since we must look at every element in each list to put them in order
- log n merges x n steps per merge = O(n $\log_2 n$)

1 $\boxed{-1}$ 5 2 $\boxed{4}$ -6 10 1 $S = \{5, 4, -6$

Name(last, first): [redacted]

$k = 3$

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).

- let $i = 1$ and intialize empty sequence $S = \{\}$
- let input sequence be $\{a_1, a_2, \dots, a_n\}$
- while $i <= n$
  - input the union of S & $\{a_i, \dots, a_n\}$ and k into blackbox
    - if answer is YES
      → increment i by 1
    - if answer is NO
      → if $i = 0$, there is no subset w/ sum k
      → else, append a_{i-1} to S and increment i by 1
- input S and k into the blackbox
  - if answer is NO, append a_n to S
- final solution subset is S

Runtime: $O(n)$ b/c i traverses initial sequence linearly and calls blackbox at each step so $O(n)$

d=3

2 1 2 2 3 3 1 1 2 Name(last, first) [redacted]

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

- let the list of votes be $\{a_1, a_2, \ldots, a_v\}$
- let $i=1$, $j=2$
- while $j < v$
  - if a_i and a_j are distinct elements, delete both from the list, let i & j point to next 2 elements
  - if a_i and a_j are the same,
    - keep both of them and let j point to next element in the list
- after this first pass, we either have no elements in the list or all identical element(s)
  - case no elements: there is no majority
  - case identical element(s): → majority candidate
    - let this element be X, keep counter $c=0$
    - for each a_i in original list:
      - if a_i equals X, increment c by 1
    - if $c > v/2$, candidate X has a majority
    - otherwise, no majority

Runtime

- finding majority candidate takes O(n) to traverse entire list once
- checking count of candidate also takes O(n) to compare every a_i to X
- total: O(n)

★ Proof on back →

Proof

- case 1)
  - assume there is some majority m w/ > v/2 votes but our algorithm does not identify m as a majority candidate (either identifies some other m' or no majority)
  - thus all m votes must have been eliminated
  - however, all votes are eliminated in pairs that are 2 distinct elements so there must be more than v/2 votes that are not m
  - this adds up to more than v votes - contradiction
  - m must be identified as a majority candidate
  - m will then be confirmed as a majority when we count all m's votes in linear time

- case 2)
  - assume there is no majority but our algorithm declared n a majority
  - the second phase of the algorithm counts all votes for the candidate n
  - if n is not a majority, the count c will be <= n/2 and n will not be declared a majority - contradiction

1 3 4 6 7 9 Name(last, first) [redacted]

$L = 9$

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(n²) algorithm would be trivial by considering all possible pairs). Justify your answer and analyze its time complexity.

Algorithm
- let list = $\{a_1, a_2, \cdots, a_n\}$
- start at $i = 1$ and $j = n$
- while $i < j$:
  - if $a_i + a_j$ equals L, a_i & a_j are the solution & we're finished
  - else if $a_i + a_j < L$, increment i by 1
  - else if $a_i + a_j > L$, decrement j by 1
- no solution found

Runtime
- O(n) b/c we look at every element at most once
- i traverses list linearly from the left and j traverses list linearly from the right and we stop when they meet in the middle

Justification
- since the list is sorted, if some $a_i + a_j < L$, we know adding either a_i or a_j w/ a smaller number will yield an even smaller sum
  - thus, we increment i b/c all elements at or before a_i are smaller and we can ignore those
- symmetrically, if $a_i + a_j > L$, we decrement j b/c any element at or above a_j will make the sum too large and we can ignore those
- eventually, we will either find the sum or i will equal j and we'll know there's no solution.