CS180 midterm

[redacted]

TOTAL POINTS

98 / 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

+ **2 pts** Prof graded

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

+ **2 pts** Add 2 points

+ **15 pts** Prof graded

+ **20 pts** Prof graded

QUESTION 5

5 problem 5 18 / 20

✓ **- 2 pts** Not best time complexity

Name(last, first): [redacted]

UCLA Computer Science Department

CS 180 **Algorithms & Complexity** **ID:** [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.

- Iterate through all the edges in the DAG and record the degree of each node. If degree = 0, add to set of nodes to be used as next step.
- While there are still nodes in the DAG, that are not in the topological ordering
  - Arbitrarily pick a node with degree 0 and add it to our topological ordering.
  - Remove all edges from the node and decrement the degree of all adjacent nodes by 1. If any have degree 0, add to our set of available nodes.

Proof:

- We know that a DAG must have at least one node with degree 0
  - Suppose that each node had at least one incoming edge.
  - Then, if we travel backwards n+1 times, we have visited a node twice (by Pigeonhole principle) meaning that we have a cycle ⇒ not a DAG
- Removing a node does not add edges, so we cannot create cycles through removal
- This means that at every step in the algorithm, we have some node with degree 0
  ⇒ Algorithm works until ordering is complete.
- We must add a node with no incoming edges, otherwise some node that we added to our topological ordering later will have an edge to that node, which contradicts our definition of topological ordering (only previous nodes can point to nodes)

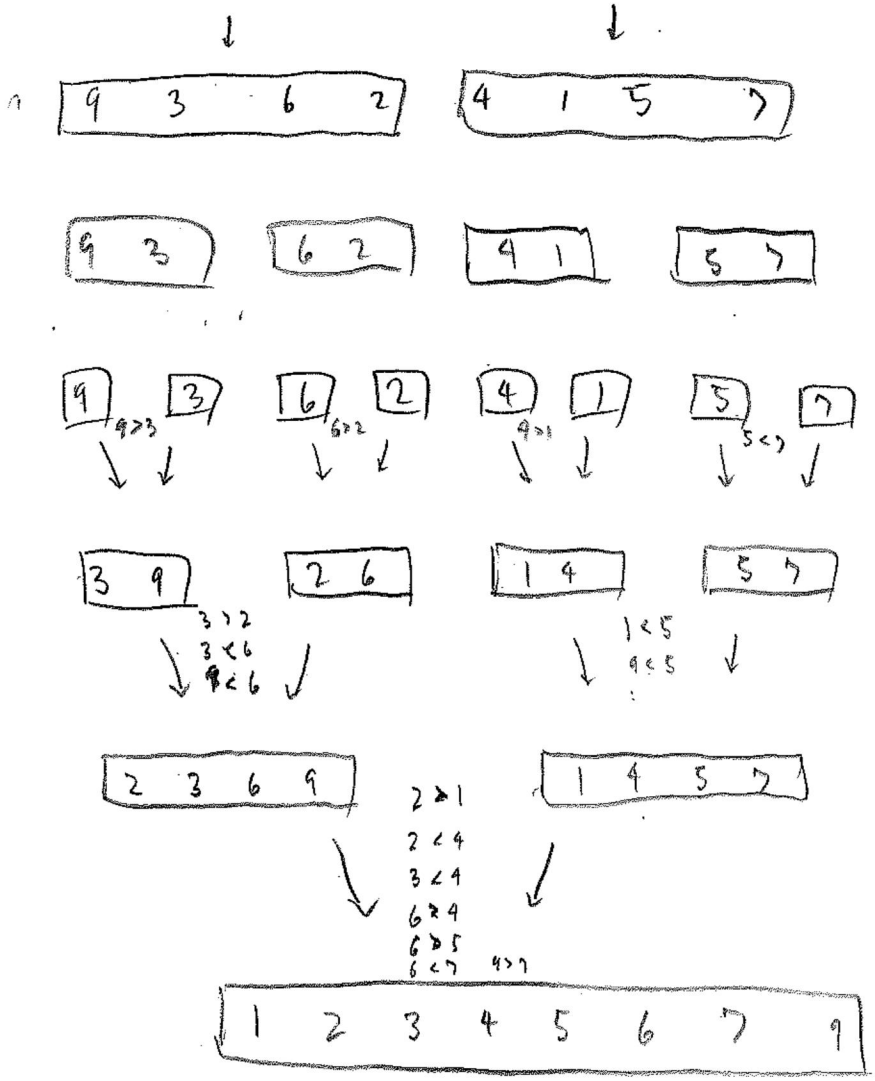
Complexity:

- Recording degrees of each node can be done in $O(e)$ time.
- We explore each node and edge at most once because we remove after ⇒ $O(n+e)$
- In total, runtime is $\boxed{O(n+e)}$

Name(last, first): [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



$T(n) = T(\frac{n}{2}) + T(\frac{n}{2}) + cn$ ← merging takes constant time because each spot is filled w/ one comparison (divide)

$= 2T(\frac{n}{2}) + cn$

$= 2(2T(\frac{n}{4}) + c\cdot\frac{n}{2}) + cn$

$= 2^2 T(\frac{n}{2^2}) + 2cn$

$= 2^3 T(\frac{n}{2^3}) + 3cn$

$= 2^i T(\frac{n}{2^i}) + icn$

we break it down to one-element chunks

$\frac{n}{2^i} = 1 \Rightarrow 2^i = n \Rightarrow i = \log n$

$\Rightarrow 2^{\log n} T(1) + cn\log n$

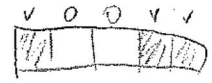
$= n(1) + cn\log n$

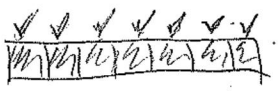
$= \boxed{O(n\log n)}$

Name(last, first): [redacted]

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





- Use the blackbox on the entire array. If answer is NO, then return that the subset doesn't exist.
- Initialize all array elements as required.
- For every element i in the array, use the blackbox with every required element, but exclude i regardless of its label
  - If the answer is YES, mark i as not required.
  - Otherwise, check next element.
- Return the elements marked required.

Proof

- There are 3 cases to consider:
  - If there was no subset to begin with, so blackbox would immediately return NO
  - If there was exactly one subset whose sum is k, removing any element in this subset would mean that there is no longer any subset adding up to k, so these elements are required. Removing any other element would mean there is still some subset adding up to k.
  - If there were multiple subsets adding up to k, eliminating one of the elements in one of the subsets would do nothing because we still have some subset adding up to k.

Complexity

- We run the blackbox $n+1$ times: once to check if there actually is a subset and once excluding each element, making it $\boxed{O(n)}$

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

- Look at the first vote. Set the indicated candidate as our majority candidate and set count to 1.
- For each remaining element in the array,
  - If the vote is for the majority candidate, increment count by 1.
  - Otherwise if the vote is for some other candidate and count is not zero, decrement count by 1.
  - Otherwise, replace the majority candidate with the current voted candidate and increment count.
- Reiterate through the array and count the number of votes for the majority candidate.
- If # votes > $\frac{n}{2}$, return candidate, otherwise return no candidate

Proof

- Suppose we had a majority. If we discard two votes, each for a different candidate, we are left with 2 cases:
  1) If neither vote was for the majority candidate, then there are still $> \frac{n}{2}$ votes for that candidate and $n-2$ total votes, meaning the majority is maintained.
  2) If one of the votes was for the majority candidate, then there are $> \frac{n}{2} - 1$ votes for that candidate and $n-2$ total votes, which means that the majority is still maintained. ($\frac{\frac{n}{2}-1}{n-2} = \frac{1}{2}$ and we have strictly greater than $\frac{n}{2}-1$ votes)
- Either way, the majority is maintained, so we can remove pairs of votes and still maintain the status of the majority candidate. By incrementing and decrementing count, we are essentially adding potential pairings and cancelling two votes →

- However, we could also create a majority by removing votes. For example, if we had

a	b	c	c

, removing a and b could create a majority candidate c.

- Therefore, we must check whether or not the candidate actually has a majority vote by counting the number of votes for that candidate.

Complexity

- We iterate through the voting array twice: once to find a potential majority and then again to confirm that the candidate has a majority.

$T(n) = 2n \Rightarrow \boxed{O(n)}$

Name(last, first): [redacted]

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). nlogn. Justify your answer and analyze its time complexity.

Algorithm:

- For each integer x in the list
  - If $x > L - x$, set left to beginning of array and right to element of array preceding x.
  - Otherwise, set left to element of array after x and right to end of array
  - Recursively until left ≥ right
    - Find the middle element of subarray designated by left and right
      - If the middle element > L - x, set right to element preceding middle
      - Otherwise if middle < L - x, set left to element after middle
      - Otherwise return (x, L - x)
  - There is no pair for this x, so continue with next element in list
- No pair was found, so return some indication of that

Proof:

- If we know one element of the pair of numbers whose sum is equal to L, then we know the other element, given that the number exists in the list.
- We know that the list is in sorted order, meaning that any numbers to the left of a number are less than or equal to that number and any numbers to the right of that number are greater than or equal to that number, so our binary search algorithm to check whether the second number in the pair exists in the list works (comparison guarantees when an element cannot be)

Complexity:

- Binary search reduces the array size by a factor of 2 every time, meaning that its runtime is O(logn). We perform binary search for every number in list ⇒ O(nlogn)