[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:

- Label every node with a count of its number of incoming edges. Push every node with count = 0 into a queue
- START • Pop the queue, call it node v (no incoming edges) and add that to the list.
- Decrement the count of each of v's neighbors by one (and delete the edges to them)
- If any neighbors have a count of 0, push them onto the queue
- Repeat by going back to START till the queue is empty

Proof:

- A node with 0 incoming edges at the start is not dependent on any other nodes. We call these nodes source nodes.
- At every iteration of the loop, by picking a source node, we know that that node depends only on the nodes already added to the list
- The node can't be dependent on any other node not in the list because then it would have an incoming edge and wouldn't be a source
- Therefore, this algorithm always produces a valid topological ordering in a DAG

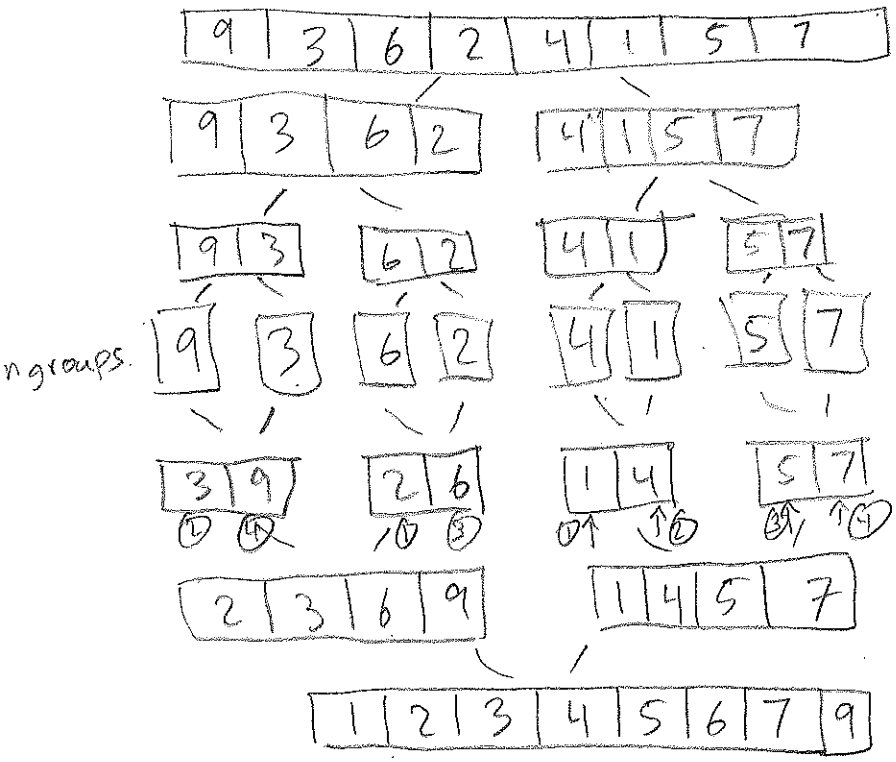
Runtime:

- labeling every node at the beginning is O(e) since we are counting edges
- Step 2 is constant time.
- Over the course of the loop we visit every node by following every edge
- This takes O(v+e) so the runtime of this algorithm is O(v+e).

[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



First recursively split each list in half till they have a size of 1 each

An array with size 1 is already sorted.

To merge 2 sorted arrays, put a pointer at the start of each and the lower element to the combined list. Repeat till all elements of both arrays are added

Time complexity:

- Merging 2 sorted arrays takes $O(n)$. This is because of the sliding 2 pointer approach.
- Splitting the arrays in half till they are size 1 takes $O(\log n)$.
- Since we have to merge 2 arrays at every level and every level has n elements even if they are split into up to n groups, we multiply the time complexities since we are doing $O(n)$ work at every level and there are $O(\log n)$ levels.

- Therefore the time complexity of merge sort is $O(n \log n)$.

3 4 6 1 2 (17)

3 4 6 1 x

3 4 6 2 → 3 4 1 2 3 6 2 ✓

[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).

3 3 3/3 6

Algorithm:

- Run the blackbox on the entire input sequence to make sure there is a valid subset, otherwise return
- For every value i in the input sequence:
  - Run the blackbox on the input sequence without i
  - If the blackbox answers YES, delete i from the input sequence completely
  - If the blackbox answers NO, add i to the list containing the subset to be returned and keep i in the input sequence
- Return the list containing the subset

Proof:

- At each iteration, we check if there is a subset to sum to k in the input sequence with a number
- If it is possible, we don't need that value left out so we can remove it altogether
- If it isn't possible, we know that value is part of the subset.
- This means for every number in the input sequence we decide if it is absolutely necessary for it to be in the subset.
- If there was some number n that must be in the subset but isn't in this algorithm, then it means Blackbox(input - {that number}) returned YES. But if that number must be in the subset then the Blackbox should have returned NO so by contradiction every number that must be in the final subset, is in that subset in this algorithm. →

- If there is a number in the returned subset that shouldn't be there then in the iteration of that loop without that number, blackbox must have returned NO or else the algorithm wouldn't add it to the subset.
- However the only reason blackbox would have returned NO is if that number had to be in the subset which is a contradiction.
- So the returned subset will contain all numbers that are required to sum to k and no extras.

Runtime:

- Constant call to blackbox at beginning which is O(1)
- We call blackbox for every iteration of the loop but the loop runs n times so we have O(n) calls to the blackbox
- The conditional checks in the while loop are constant time
- Therefore, this algorithm runs in O(n) time

[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:

- Take every pair of votes $\{0,1\}, \{2,3\} \ldots \{v-1, v\}$ (or $\{v-2, v-1\}$ if v is odd.
- For every pair, ignoring the odd card compare the values. If they are different, remove both of them, if they are the same, keep one and remove the other.
- Repeat this till there are 0 or 1 votes left
- If there are 0 votes left, there is no majority
- If there is 1 vote left, compare it against all votes and check if it is a majority.

Proof:

- If there was a majority in v votes, then by removing every pair of cards that was different, there will still be a majority
- Removing 1 card in a pair that are the same will still keep the majority
- This means at every iteration of the loop, if there was a majority, it will still exist after removing the cards
- However when we get to one card, we know that card would be the majority IF it was a majority in v so we must check it with every vote in v to make sure it is a majority

Runtime on back →

Runtime analysis:

- Initially we make $\frac{v}{2}$ comparisons.
- But from these comparisons we delete at least $v/2$ cards.
  - → 2 cards are same → delete 1 → at least half will be deleted
  - → 2 cards different → delete both
- This means at every iteration we delete $v/2$ cards
  So our n_{total} comparisons are $v/2 + v/4 + v/8 \ldots + 1 \approx v$
- We also compare the last card to all v votes which takes $O(v)$
- So our runtime is $v+v$ which is $O(v)$

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

i, j are indexes

(i) (j)

Algorithm:

- Start with a pointer at index 0 and index n-1 in the list
- While i is not equal to j
  - Sum list[i] + list[j]
  - Check if it is equal to L
  - If equal
    - Break out of loop and return these numbers
  - If L > sum
    - Decrease j by one
  - Otherwise increase i by one
- Return that there are no 2 numbers that sum to L

Proof:

- Since we know the list is sorted, we can reduce the number of comparisons from n^2 to n
- At some index i & j in this algorithm, either the sum is equal to L so we return
  - If it is not equal to L, we either need to increase or decrease sum (if sum < L) or decrease (if sum > L)
- Since the list is sorted, if we want to increase, we can either increase i or j. But the only way to get to this j is if using j+1 resulted in a sum > L so we must increase i
- The same logic works for decreasing the sum

Runtime on back →

Runtime analysis:

- At every iteration of the loop, we decrease the size of $j-i$ by 1
- Initially $j-i = n-1$ so if we decrease by 1 every iteration, we have $O(n)$ iterations.
- Inside the loop, there are constant time comparisons
- Therefore, the runtime of this algorithm is $O(n)$.