Name(last, first): _

U C L A Computer Science Department

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

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.

- Each node will keep track of the number of edges going into it and going out of it
- Find all source node(s) and add them to the result queue, Q
- Remove these source node(s) from the graph to form G'
- For all removed source nodes, decrement the number of incoming edges for each of their children by 1
- Recall this algorithm on G' until all nodes are in Q
- The ordering of Q is the topological sort result

source node = node with 0 incoming edges

proof: - since there are no cycles, there is always at least 1 source node (no incoming edges)

- Once we remove a source node, we create G' which is guaranteed to also be a DAG since G' is just a sub-graph of the original DAG.
- So if the algorithm correctly removes sources from G, it will do the same for G'

Time complexity:

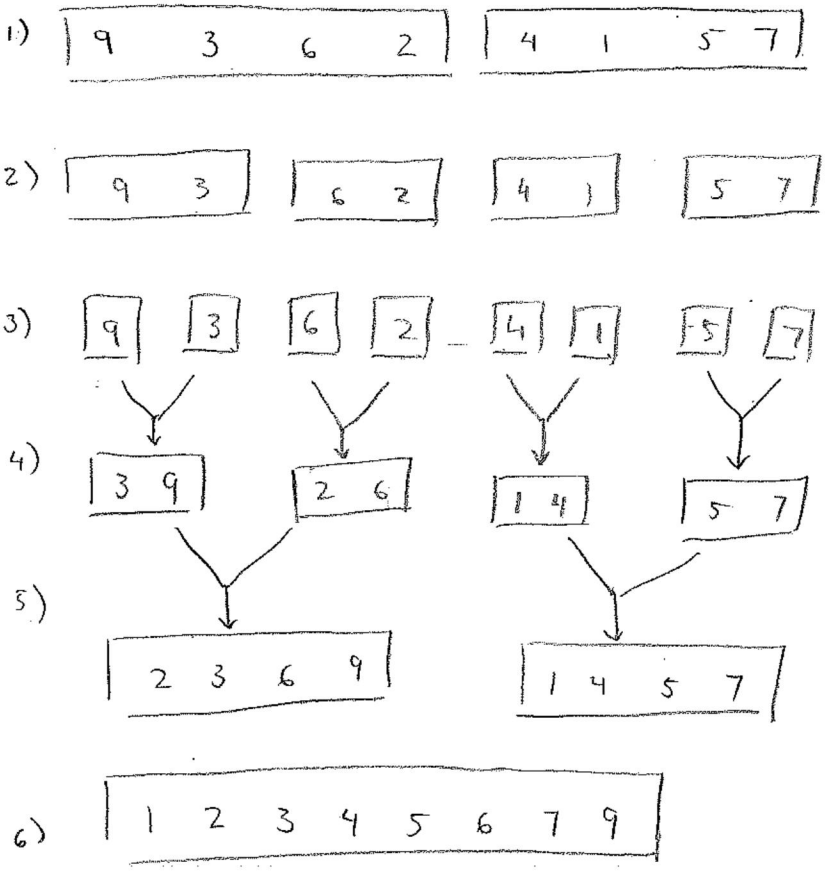
- we look at each node once, and that is only when it's a source and we remove it (O(1))
  - since we do this for n nodes, it is $n \cdot O(1) = O(n)$.
- We also need to keep track of all edges going in/out of nodes and to do this, we need to at least look at each edge once → O(e)

- Total: $O(n) + O(e) = \boxed{O(n+e)}$

Name(last, first)

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 set into S_1 and S_2 each with $\lfloor \frac{n}{2} \rfloor$ and $\lceil \frac{n}{2} \rceil$ values respectively
- continue until each set has 1 number
- Merge numbers by adding smaller number befor larger ones into output-array.
- continue until output-array is same size as input array.

Time complexity: For n numbers.

- we know $T(1) = O(1)$
- we divide n numbers in half repeatedly until we can operate on 1 value

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

- cn repnsents the time it takes to compare and insert all n values into output array
- we continue to divide $T(n)$ until it equals $T(1)$

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

- General Form: $T(n) = 2^i T(\frac{n}{2^i}) + i(cn)$
- For $T(\frac{n}{2^i}) = T(1)$, $\frac{n}{2^i} = 1 \rightarrow n = 2^i \rightarrow i = \log_2(n) = \log(n)$
- So $T(n) = n\,T(1) + \log(n)(cn)$

$$= O(n) + O(n\log n)$$

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

Name(last, first):

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).

not necessarily consecutive

- sort all integers in non-decreasing order.
- input 1st element of sort-array into blackbox.
  - If YES:
    - Inserted subset sums to k
  - If NO:
    - Add next element from sorted array
    - Recall blackbox.
- If subset sum > k after adding element x
  - Reset subset from element x
  - Recall algorithm
- Repeat until subset is found or until no more integers to add from sorted sequence

- input 1st element
- call box
- if YES: yay
- if NO:
  - add next element
  - if sum > k
    - ~~delete~~ sequence at element x
    - ~~next~~

-3	4	5	7

11? 12?

Proof:

- This is a greedy algorithm
- Assume a subset exists but is not found
- That means the algorithm did not consider this subset at all, since if it did, it would have returned it
- But that cannot be true since the algorithm continuously adds elements, so each one is looked at
- And since we reset our subset whenever the sum exceeds k and we re-evaluate, all we consider subsets that are always less than or equal to k.
- So by contradiction, we always consider the subset at some point so the subset must be found by our algorithm if it exists

Time

- Sorting takes O(n log n) (mergesort)
- Adding element is O(n) in worst case
- Reset subset is O(1)
- In total: O(n log n))

Name(last, first)

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.

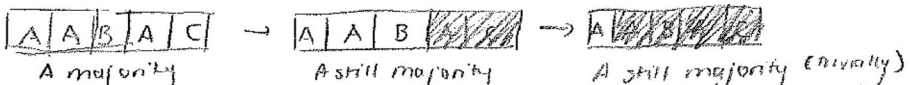
- Create a numVotes counter, initialized to 0
- Set currMajorityCandidate as the first element of v (v[0])
- Increment numVotes by 1
- Iterate along the array v
  - If the next vote is for currMajorityCandidate:
    - Increment numVotes by 1
  - If the next vote is not for currMajorityCandidate
    - If numVotes is currently 0
      - Set currMajorityCandidate as this new person.
      - Increment numVotes by 1
    - Otherwise
      - Decrement numVotes by 1
- After iterating through all of v, if numVotes > n/2
  - Then currMajorityCandidate has a majority of votes!
- Otherwise no majority exists

* Extra Storage
- numVotes
- currMajorityCandidate

= Both constant sized variables *

Proof:

- The proof relies heavily on the fact that votes of different candidates can "cancel" each other out without disturbing majority



- So by decrementing numVotes for each candidate that's not currMajorityCandidate, we are "cancelling" votes out while still maintaining majority
- We reset currMajorityCandidate when numVotes = 0 because that means majority has been totally "cancelled out"

Time Complexity

Back page.

Time complexity

- Since we iterate through all v votes to find a candidate and only look at each vote once, the complexity is $O(v)$.
- Then once we find a candidate, we check if numvotes $> n/2$ which is $O(1)$
- Total: $O(v) + O(1) = \boxed{O(v)}$

Problem 5: Consider a sorted (nondecreasing order) 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.

- Create a pointer, p1, that starts at first element of list
- Create a pointer, p2, that points at last element of list
- If p1 + p2's values sum is equal to L
  - Then return p1 and p2 to give 2 integers that sum to L
- Otherwise
  - If p1+p2's values sum is less than L
    - Move p1 forward to next integer in list
  - If p1+p2's values sum is more than L
    - Move p2 backwards to previous integer in list
- Continue until either:
  - The 2 integers that sum to L are found

  OR
  - until p1 and p2 meet, meaning no such 2 integers exist in the list.

Proof:

- Since list is sorted in non-decreasing order, we know that moving left along the list decreases integers and moving right increases them
- Using this to our advantage, we can readjust our sum to get as close to L as we can.
- We know that if our current sum is less than L, then we need to consider larger integers by moving to the right
- We know if our current sum is more than L, then we need to consider smaller integers by moving to the left
- Since we never skip any integers, we will consider sums that potentially add to L and either find result or return no such 2 integers exist

Time Complexity

- In the worst case scenario, we will scan through every element in the list → O(n)
- We never scan more than n because our algorithm stops when p1 and p2 meet so they'll never cross over each other and rescan something the other pointer has already looked at.
- We compute p1 + p2's sum → O(1)
- We compare sum with L → O(1)

} overshadowed by O(n)

- Total: O(n)