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.

Algorithm

1. Create a set S
2. for every node in G, store the count of incoming nodes. If the count is 0, add that node to S
3. select an arbitrary node k from S, and append it to the output.
4. for every node which k has an edge toward, decrement the count for that node. If it is 0, add that node to S
5. Remove k from S. If S is not empty, go to step 3
6. Return the output

Proof: This algorithm works by selecting first all the nodes with no incoming edges and selecting arbitrarily among them

- Contradiction: assume there was some node k' before k, but our algorithm selected k. By contradiction, k' could not have come before k (in the graph) since k has no incoming edges.
- By removing a node k, and removing all outgoing edges, the number of incoming edges on all $\{k', \ldots, k^{(n)}\}$ would also decrease, and by rule 2 and 4, show that all incoming edges have already been added to the ordering, so k' can be added.

Runtime

$O(n+e)$ where n is the number of nodes and e the number of edges.

- Counting all edges initially takes $O(e)$ time, since an incoming edge will be counted at most once
- Removing all nodes takes $O(e+n)$ since each node only update the count of its neighbors, since the edges are removed after ordering, this is $O(e+n)$

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

split → 9 3 | 6 2 | 4 1 | 5 7

split (until each contains 1 element) → 9 | 3 | 6 | 2 | 4 | 1 | 5 | 7

merge → [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]

```
merge func (A, B)
result = empty list
index a = 0
index b = 0
if A(a) < B(b)
    append (A(a)) to result
    a++
else
    append (B(b)) to result
    b++
```

Runtime

To perform the "merge" operation takes $O(n)$ for an entire round. Since each comparison always adds an element to the sorted output and a comparison takes constant time. There are only n $O(1)$ operations done.

At each level of merging $T(n)$ refers to the time to sort n items

$T(n) = T(n/2) + T(n/2) + cn$, since the time is time to sort left + time to sort right + constant $\cdot n$ (to merge)

so $T(n/4) = T(n/4) + T(n/4) + cn$

...etc

As such, $T(n)$ can be rewritten as $T(n)$ $2T(n/4) + cn + 2T(n/4) + cn \cdot cn$

$= 4T(n/4) + cn$

Eventually we see $T(1) = O(1)$,

so $T(n) = nT(1) + cn = cn$

Since the input is divided each level, there are $\log n$ levels, each with a $T(n)$ operation, so the total runtime is $O(n \log(n))$

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

1. Starting set S filled with the multiple input sequence
2. input S into the machine
3. if the answer is NO, no such subset exists
4. for each element e in S, ~~repeat~~
   remove e from S
   check the machine.
   if NO, add e back to S
5. Return S

Proof

At each element, we remove and check if the set contains this element sums to k. If previously it did and now doesn't, e must be a part of the subset. If it previously did and now continues to work, e must not have been a part of the subset, as the subsets did not change the result. Therefore we know that any element which alters the answer must be in the subset and any which doesn't is not in the subset.

Runtime.

O(n). Since each element is only removed once, and then the box is calculated on the remains, the calculation is only performed n times at most, giving an O(n) runtime of the black box

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

1. set a candidate P to null
2. set a counter to 0
3. for each element N in votes, if P is null, set P to N
   if P is null, set P to v, and set counter to 1
   elif P = v, increment the counter
   elif P ≠ v, decrease the counter
   if counter = 0, set P to null
4. If P is null, no majority
5. Create a counter set to 0
6. for each vote v, if v = P, increment the counter
7. if the counter is > $\frac{\text{total \# of votes}}{2}$, P is the majority
   otherwise no majority

Proof

If a majority ($n/2+1$) exists in a group of n, removing 2 different candidates must change the majority

Case a: removed a non-majority candidate
- previous majority was $n/2+1$, no change
- previous total was n, now $n-2$
- since $n/2+1 > n/2$ and $n/2 > n-2/2$, it must be true that $n/2+1 > n-2/2$, so there remains a majority

Case b: one of the removed was a majority
- majority now has $n/2$
- total has $n-2$
- a majority of $n-2$ is $> n-2/2+1 = n/2-2/2+1 = n/2$..
- since the old majority has $n/2$, and to be a majority of $n-2$ needs $n/2$ at least, the majority is the same

→

By the majority principle, we can keep a "running majority" as we go through the list. The ~~candidate~~ "running majority" candidate is added to each time it is seen, and we do the "remove 2 different" by subtracting 1 when a different value is seen. This guarantees that a majority could never have been fully removed, so whatever is the P at the end ~~must~~ is the only possibility for a majority. (By definition, only 1 majority can exist in a set). The final pass checks to ensure the running majority is actually a majority

Runtime

$O(n)$. The candidate pass examines each element only 1 time and does a constant # of operations (add, delete, etc) to the counter and majority. The second pass is also $O(n)$ since it only checks each element 1 time, using $O(1)$ if it is the same as the candidate majority, and $O(n) + O(n) + O(1)$

Space complexity is constant, as the only values stored are 2 counters and a potential candidate, which are all fixed size. $O(1)$

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). Justify your answer and analyze its time complexity.

Algorithm

1. create a set S
2. for each element e in the list, insert L-e to the set S
3. for each element e in the list, if e is in S (written) (e, L-e)
4. no such pair exist

Proof:

Any two number j and k that sum to L follow the property j+k=L. Therefore j=L-k. So if an element with value j can be found such that L minus some other value equals j, that other value and j will sum to L. By looking at all possible (L-k) values, we guarantee that if such a j exists, we will find it

Runtime

O(n). Calculating L-k is a constant time operation, and so is inserting it to a set, and this is done n times. Looking through each element again is also constant time per element, as a set lookup takes O(1). Therefore the total runtime is O(n)

(note space use is also O(n))