CS180 midterm

Junhong Wang

TOTAL POINTS

96 / 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 **18 / 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

QUESTION 5

5 problem 5 **18 / 20**

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

Name(last, first): WANG JUNHONG

UCLA Computer Science Department

CS 180 **Algorithms & Complexity** **ID:** 504941113

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. Iterate over every edge to find out indegees and out degrees of each vertex
2. Find out the sources (i.e. vertices with indegree 0)
3. Print one of the sources (Let's call this vertex v)
4. Remove v. (i.e. Check v's edges and decrement its children's indegree by 1)
5. If a vertex becomes indegree 0, add it to the list of sources
6. Repeat 3-5 until every vertex is printed
7. The order of printed vertices is the topological order

[prove]

Topological sort is a way to sort vertices such that no vertex has an edge pointing to vertices before it. (i.e. This can't happen). Suppose on the contrary, there exists a vertex that has an edge pointing backward. However, this can't happen because because we always choose vertex with indegree 0.

[Run Time Analysis]

V denotes # of vertices, and E denotes # of edges

Step 1 takes $O(E)$ because we iterate over all edges once

Step 2 takes $O(V)$ because we iterate over all vertices once

Step 3 takes $O(1)$

Step 4 takes $O(E)$ througout the entire algorithm

Step 5 takes $O(1)$

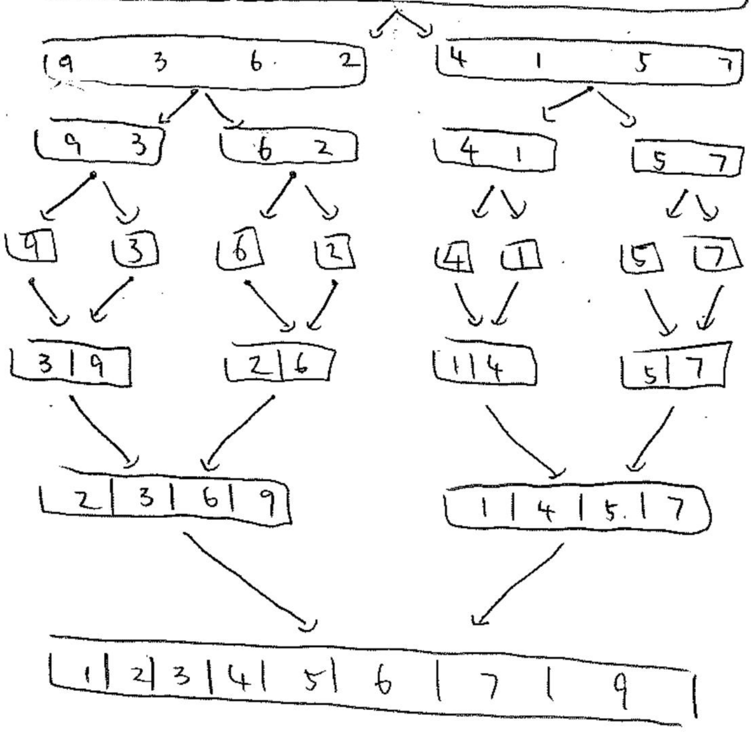
We do step 3. - step 5 for every vertex (i.e. $O(V)$)

Thus the time complexity of this algorithm is $O(V+E)$

Name(last, first): WANG JUNHONG

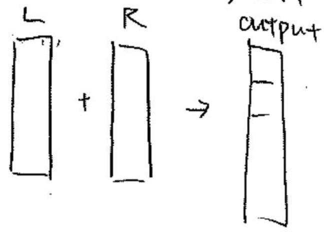
Problem 2: Run Merge sort on the following set of numbers. Show every step. Analyze the time complexity of merge sort on a set of **n** numbers (show every step)

9	3	6	2	4	1	5	7



[Run Time Analysis]

The algorithm of merge sort is to divide an array into half and merge them recursively



Claim: It takes $O(n)$ to merge n elements

proof of claim: Since left array and right array are already sorted, it takes 1 comparison to fill out each entry in the merged array. Since there are n elements, it takes $O(n)$ □

Let $T(n)$ be time complexity of this algorithm. Then $T(n) = 2T(\frac{n}{2}) + cn$ (c is constant)

Notice $T(n) = 2T(\frac{n}{2}) + cn = 2^{\log n} T(1) + cn \log n = n + cn \log n$ ($\because T(1) = O(1)$)

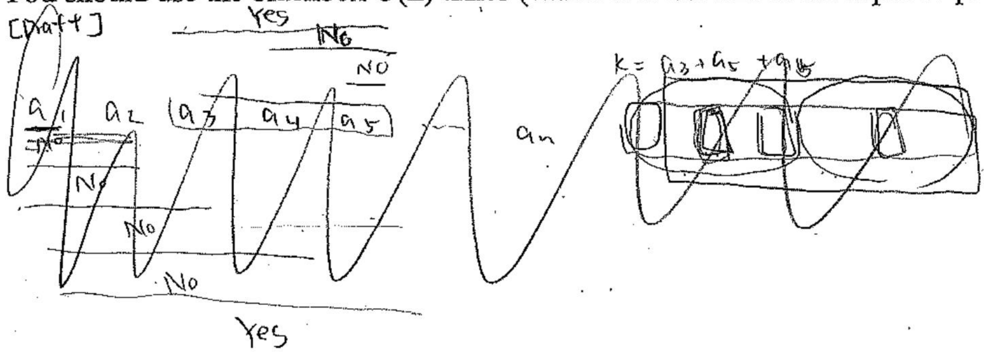
Thus time complexity of this algorithm is $O(n \log n)$

Name(last, first): WANG JUNHONG

may not be consequtive !

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).

[Draft]



[Algorithm]

1. Let $a = [a_1, \dots, a_n]$ be the sequence of numbers we are interested in.
2. Iterate over every element in a.
3. During each iteration i, we run black box algorithm on $a \setminus \{a_i\}$
4. If it returns NO, then a_i must be one of the subset. Add a_i to the list of subset
5. After the loop, if the list of subset is empty, then there's no subset whose sum is k.
6. Otherwise return the list of subset.

[proof]

By the property of the blackbox algorithm, if any element in the subset we want to find is missing from the arguments for blackbox algorithm, then the blackbox algorithm will return No. Thus if we feed in $a \setminus \{a_i\}$ and it says No, that means a_i must be one of the elements to sum up to k.

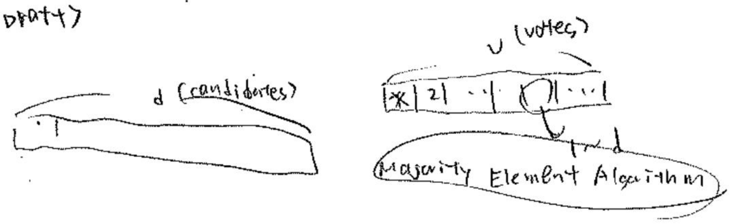
[Run Time Analysis]

Since we iterate over the array once and perform the blackbox algorithm once during each iteration, we are indeed using the blackbox O(n) times.

Name(last, first): WANG JUNHANG

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.

[Draft]



[Algorithm]

1. Arbitrary pick two different elements, with different values in votes
2. Delete them from votes
3. Repeat 1-2 until there's no elements left in votes or all elements in votes are the same value.
4. If there's no element left, there's no majority.
5. If there's at least one element left, pick that element as majority candidate
   (Note that we can do this because if there are two or more elements left, they all have same value due to step 3)
6. Check the original votes and see if the value we picked appear more than $\frac{v}{2}$ times
7. Return the value if it appears more than $\frac{v}{2}$ times. Otherwise there's no majority.

[proof]

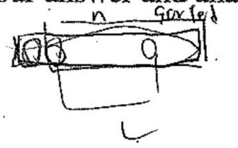
If there's an array of size n and there exists a majority element, then there must at least $\frac{n}{2}+1$ such element. We remove two different elements (different values) from the array. There's at most one majority element in the two. Thus there are still at least $\frac{n}{2}+1-1=\frac{n}{2}$ majority elements in the array. Note the array size is now $n-2$. To be a majority element in an array of size $n-2$, it must appear at least $\frac{n-2}{2}+1=\frac{n}{2}$ times. Thus the original majority element is still a majority elements after removing the two elements.

[Run Time Analysis]

We need a slot for a number and a counter to keep track of how many times that number appeared. We iterate over the array once. During each iteration i, if the slot is empty, we assign the slot to be the ith number in the votes. Set the counter to be 1. If we see the same number again, we increment the count. If we see different number, we decrement the count. If the count is 0, we assign new number at next iteration. This way we can virtually delete two numbers with only two slots (i.e. constant storage) Then we need to check the number slot and see if it's actually majority (i.e. another $O(n)$) Thus, this algorithm is $O(n)$.

Name(last, first): WANG, JUN HANG

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.



nlogn

[Algorithm]

1. Let a be the array
2. Iterate over each element in a.
3. During each iteration i, we perform binary search on the entire array to find L - a[i].
4. If we find such element a[j] and i ≠ j, return a[i] and a[j].
5. Otherwise we continue the loop
6. If no numbers are returned in the loop, then there aren't such elements, whose sum is L.

[Proof]

For each element a[i], if a[i] + a[j] = L, then a[j] = L - a[i].

Thus for every element a[i] we want to find if a[j] exists in the array,

Since we want to find two numbers, i ≠ j.

[Run Time Analysis]

During each iteration we are running binary search, which takes O(logn)

Since there are n elements, the time complexity of this algorithm is O(nlogn)