CS180 midterm

Wilson Jusuf

TOTAL POINTS

94 / 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 **18 / 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): Jusuf, Wilson

U C L A Computer Science Department

CS 180 **Algorithms & Complexity** ID: 404997407

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.

the Topsort algorithm creates an ordering of vertices $\{v_1, v_2, \dots, v_n\}$ such that if the edge (v_i, v_j) i.e. $(v_i \to v_j)$ exists in the DAG, then v_i appears before v_j in the Topsort ordering (i.e. $i < j$).

algorithm: Suppose we have DAG ~~V~~ $G = (V, E)$

1.) for every edge $e \in E$, $(v_i \to v_j)$ increment v_i outdegree by 1 and increment v_j indegree by 1. This is $O(|E|)$

2.) Find the sources in V (i.e. ~~out~~ indegree = 0). Put all these sories in the topsort ordering first.

3.) Delete ~~#~~ said sources from V. ~~to~~ Also decrement indegrees of the sources' neighbors.

4.) repeat 2 with this newly updated V, until V empty. Use the newly formed sources from the old sources' (~~out~~ now deleted) neighbors.

Correctness proof:

Suppose towards contradiction that i~~n the ordering~~, ∃ edge $v_i \to v_j$ in the ~~DAG~~ & where v_i appears after v_j in the topsort (i.e. $i > j$). Then, when the topsort also is being executed, ~~so~~ v_j becomes a source before v_i since $i > j$. However, this edge $v_i \to v_j$ exists, so v_j not a source as v_i not deleted yet. Contradiction.

Complexity

on step 1), we looked at every edge to see in and out degrees. This is $O(|E|)$. Then, in 2 to 4, we loop through all vertices until V empty, which is $O(|V|)$. Hence, complexity is $O(|E|+|V|)$.

Name(last, first): Jusuf, Wilson

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

[9 3 6 2] [4 1 5 7] (Split to halves)

[9 3] [6 2] [4 1] [5 7] (Split again)

[9] [3] [6] [2] [4] [1] [5] [7] (final split - note that this is sometimes skipped. go straight to sorting each [i j].)

[3 9] [2 6] [1 4] [5 7] (merge each pair to form 2 elements)

[2 3 6 9] [1 4 5 7] (merge to form 4 elements in each array)

[1 2 3 4 5 6 7 9] (finally, merge to form the whole sorted array)

<u>time complexity</u>

Note that in merge sort, we split the set into 2 halves and keep doing so until it is the smallest granularity. Then, we merge, which takes 2n (O(n)) as we go through 2 sorted lists ~~bo~~ each of length n.

Here, we have the following recurrence relation:

$$T(n) = T(\tfrac{n}{2}) + T(\tfrac{n}{2}) + cn = 2T(\tfrac{n}{2}) + cn = 2\left(2T(\tfrac{n}{4}) + cn\right) + cn = 4T(\tfrac{n}{4}) + 3cn$$

$$= 4\left(2T(\tfrac{n}{8}) + cn\right) + 3cn = 8T(\tfrac{n}{8}) + 7cn = \text{etc} \cdots = 2^i T(\tfrac{n}{2^i}) + 2^i cn \text{ where } \frac{n}{2^i} = 1$$

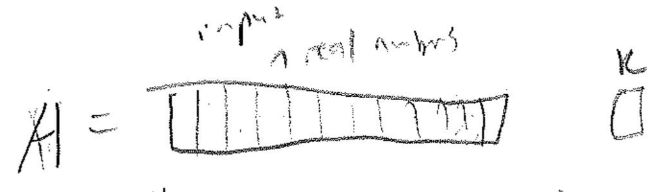
so $2^i = n$

$i = \log_2 n$

$$= n + cn\log n \Rightarrow \underline{O(n \log n)}$$

Name(last, first): Jusuf, Wilson

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).



The naive way is to do a $\binom{n}{1}+\binom{n}{2}+\binom{n}{3}+\dots+\binom{n}{n}$ algorithm, which is to compare every possible combination. This of course is horrible (2^n).

~~the~~

Suppose we have a function
Check(L, R, k) which outputs YES if a subset in L ~ R (0-indexed, inclusive), otherwise NO.

Our algorithm is as follows:

```
1 -> if check(0, len(A)-1, k) == "NO" : Then conclude that no such subset exists.
2 -> let Subset = {Ø} (empty set)
     R = len(R)-1
3 -> loop len(R)-1 times, or until k == 0 ;
4 ->    if check(0, R-1, k) == "YES":
            (Then A[R+1] not important)
5 ->    else if check(0, R-1, k) == "NO":
            This means A[R-1] was important for our subset.
            add A[R-1] to our subset.
            Set k = k - A[R-1] (loop again with this new k value)
        R = R-1
     return Subset.  <- note, if at this point, if "Subset" = {φ}, then it is simply A[0].
                        i.e. A[0] = k.
```

Note that we are looping though the loop n times, and we are using the check (blackbox) function once (result saved). Thus, it is O(n).

Name(last, first): Just, Wilson

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.

majority means $> \frac{1}{2}$ the votes. so if for a $P = P_1 \cup P_2$, some i is the majority, Then it must be that ~~for~~ ~~one~~ in at least one of P_1 or P_2, i is the majority as well.

so if P_1 has a majority; then we check whether enough people in P_2 like i to make i a majority in $P = P_1 \cup P_2$.

aha

algorithm

note: the base case is when there is 1 candidate, In which case, the majority is this candidate's own ~~vote~~ vote.

```
1.) get_majority(V):
      partition V = V_1 ∪ V_2 where |V_1| = |V_2| ≈ |V|/2 (roughly)
2.) if get_majority(V_1) = i exists ...
        check if i is a popular enough in V_2. if so:
            return i
3.) else if get_majority(V_2) = j exists ...
        Check if j is popular enough in V_1. if so:
            return j
    else:
        return none.
```

note: 'popular enough' ~~means~~ in V_1 or V_2 doesn't mean majority. It means that there is enough people in the other V_1 / V_2 such that the candidate is majority in $V_1 \cup V_2 = V$.

We keep track of the # of votes whenever we return so that we can do this. This is constant storage since ~~*~~ it is within recursive call returns.

proof of correctness

follows from above.

Suppose towards contradiction that for $P = P_1 \cup P_2$, where candidate i majority, that i is not majority for P_1 and P_2. Then i votes $< \frac{1}{2}$ in each P_1 and P_2. So we have $< \frac{|P_1| + |P_2|}{2} = \frac{|P|}{2}$ votes which is a contradiction to definition of 'majority'.

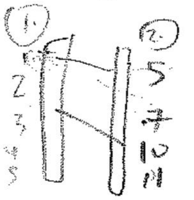
Complexity.

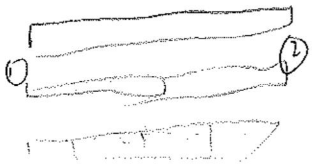
we have $T(n) = 2T(\frac{n}{2}) + Cn$ where Cn is the "check ~~for~~ if popular enough ~~in~~" part.

Hence, this is $O(n \log n)$ like merge sort.

Name(last, first): Jusuf, Wilson

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.

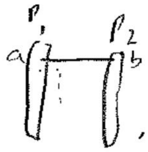




Idea: split into 2 halves.
compare ~~#~~ in $O(m+n)$ time return
if L ~~ex~~ sum exists (since it's sorted).
If not exists, split again. recursively, ...
until we get final level. In which case,
no L sum found.

we ~~give~~ use a ~~ca~~ divide and conquer algorithm as follows:

1.) split list into 2 halves P_1 and P_2, where P_1's values are ~~s~~ ≤ P_2 values.



2.) compare P_1 and P_2 as follows: checking if $a+b=L$. if $a+b<L$, advance one of a or b to a larger amount. if $a+b>L$, then ~~decrease~~ go back one of a or b. This takes $O(\frac{n}{2}+\frac{n}{2})=O(n)$ time since we can advance it ~~at most~~ until the end of P_1 and P_2. more

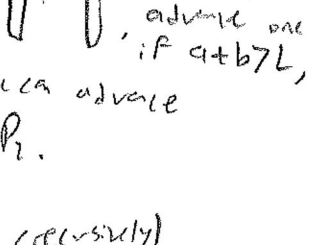
If ~~a~~ $a+b=L$ exists, return a, b. otherwise,

3.) repeat 1) on P_1, and on P_2 (recursively). Recurse until we have 1 element in the base case. If ~~a~~ $a+b=L$ still not found, then return not found.

Justification

This algorithm compares pairs between splits. Note that if $\exists a,b$ ~~∈~~ such that $a+b=L$, then there exists a split P_1, P_2 where $a \in P_1$, $b \in P_2$ since we recurse until the base level. Thus our algorithm would detect it.

Concretely, if we consider $a \in P_i$ and $b \in P_j$ where $a+b=L$, & we suppose that each P partition is a node in a binary tree. every ~~node~~ node has a common ancestor node. Thus our algorithm would detect this $a+b=L$.



Complexity

This is identical to mergesort, which is $T(n) = 2T(\frac{n}{2}) + cn$. We proved earlier that this is $O(n \log n)$.