

CS180 midterm

Aurora Yeh

TOTAL POINTS

98 / 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 overall 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

QUESTION 5

5 problem 5 20 / 20

✓ - 0 pts Correct

UCLA Computer Science Department

CS 180

Algorithms & Complexity

ID: 305110110

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:

- run through DAG, labeling each node with number of incoming edges
- select a node with 0 incoming edges. if there are more than one, then choose one arbitrarily
- save node to topological sort.
- delete node from DAG. Update labels of children nodes because they have one less incoming edge
- repeat until no nodes are left in DAG

proof:

1) all nodes will be sorted

- aka, there will never be a time when no nodes have 0 incoming edges
- assume by contradiction that there are nodes left in the graph but they all have incoming edges
- if we trace the edges backwards, we can trace infinitely because every node has at least one incoming edge
- once we traced $n+1$ edges, by pigeonhole principle, we must have visited at least one of the n nodes twice
- this means the DAG had a cycle, which is a contradiction
- therefore, the algorithm will sort all nodes.

2) sorting is correct

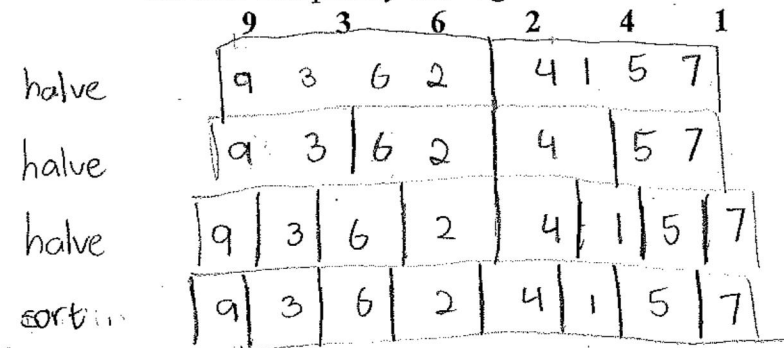
- assume node n_i has an incoming edge from n_j and $i < j$ (aka assume algorithm did incorrect sort)
- by algorithm, node n_i was selected because n_i had no incoming edges
- if n_i had incoming edge from n_j , n_j must have been selected and deleted before n_i , so $j < i$
- by contradiction, our sort must be correct

time complexity

- $O(e+n)$

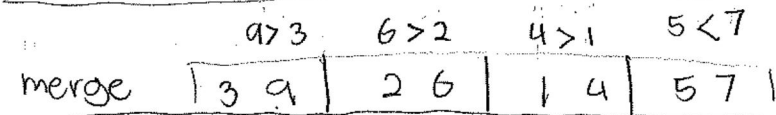
- visit every node twice (once to label, once to delete) ¹
- visit every edge twice (count incoming edges, update labels)

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)



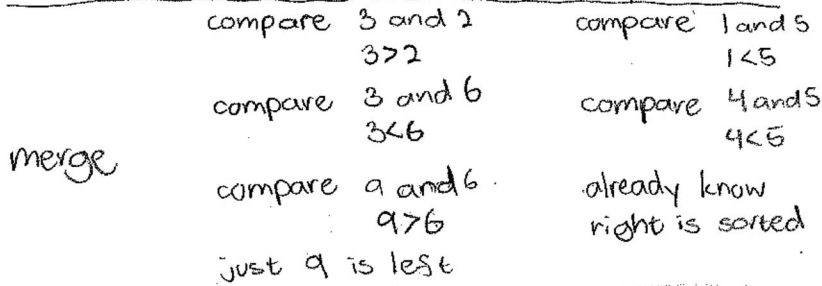
$\log n$ splits

since one element, already sorted

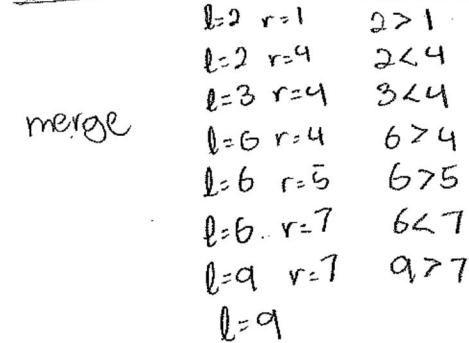
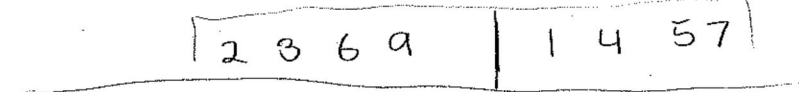


worst case $n/2$ comps

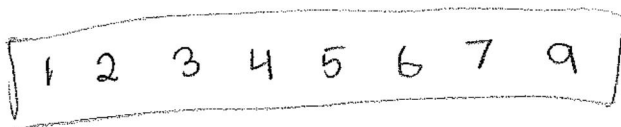
worst case $n - n/4 = 3n/4$ comps



- take and compare first elements of left and right (already sorted)
- take smaller value and move index to next number of subarray
- is equal, take either
- repeat until one subarray empty
- copy rest of other subarray (because it's sorted, so if first term > then rest >)



eg. if $left[l] < right[r]$ and l is last in left, we know $right[l+1] \geq right[r]$; $n > 0$ so $left[l] < right[r] \leq right[l+n]$



time complexity - there are $\log n$ layers
 each layer has $O(n)$ comparisons
 $O(n) \cdot \log n = O(n \log n)$

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).

Given sequence S , find subsequence S' whose sum is k

algorithm -

- put S and k in box
- if box is NO, there is no solution
- for each num n in S
 - delete n from S , put $S-n$ in box
 - if box is YES, continue (don't put n back)
 - if box is NO, put n back, continue
- remaining nums in S are solution

time - $O(n)$ because delete each num once

proofs -

1) result will sum to k

- induction: box says YES for S initially
- assume box is YES for some S with num n
- inductive step - n is deleted
 - if box still says YES, S still has sum to k
 - if box says NO, putting n back will make box say YES
- therefore, box will say YES at end of algorithm

2) sum is exactly k

- assume extra term by end of algorithm
 - since extra, deleting it and putting the rest in box will get YES
 - then by algorithm, term would have already been deleted since all terms checked, so contradiction

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:

- pair up votes in v (ignore if there are extras)
- compare, if different then remove both from consideration
- if same then remove one
- repeat until only one vote in v is left
- count number of votes in v that have same vote (constant extra storage)
- if count $> v/2$ return the candidate
- else no majority candidate

proof:

- if there is a majority, majority will stay majority
- from each pair, at most one of majority is removed
 - if pair match, then one is removed
 - if pair mismatch, can't both be majority so one majority vote and one nonmajority is deleted
- aka, at most half of majority is deleted, but \geq half of votes are deleted
- since $\text{maj} > \frac{1}{2}v$, then $\frac{1}{2}\text{maj} > \frac{1}{4}v = \frac{1}{2}(\frac{1}{2}v)$
- so maj is still maj after deletion

time - $O(v)$

log v pairs in worst case
 each pair does 1 comparison
 final checks all v votes

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

- have two indices, $i=0$ and $j=1$
- while $(\text{list}[i] + \text{list}[j] < L \text{ and } j < n)$, increment j
- while $(\text{list}[i] + \text{list}[j] \neq L \text{ and } i < j)$
 - is $\text{list}[i] + \text{list}[j] < L$ and $j+1 = n$, increment i
 - is $\text{list}[i] + \text{list}[j] > L$, decrement j
- is $i \geq j$, no solution

proof

1) we find solution

- assume it exists and we don't find it
 - $\text{list}[0] + \text{list}[1]$ is smallest sum, which is checked
 - j will increase until end of list or until $\text{list}[i] + \text{list}[j] > L$
 - From then on, every valid pair in between will be checked
 - L must be in between because increasing either i or j would increase sum and sum is already $> L$
 - if $\neq L$, then must be $<$ or $>$
 - if $<$ then can only increment i
 - if $>$ then can only decrement j

time $O(n)$

interval increases at most n times
 interval decreases at most n times

