CS180 Midterm

Siddharth Joshi

TOTAL POINTS

81 / 100

QUESTION 1

1 Problem 1 20 / 25

✓ **- 5 pts** Updated score based on the comments from the professor

QUESTION 2

2 Problem 2 25 / 25

✓ **- 0 pts** Correct

QUESTION 3

3 Problem 3 25 / 25

✓ **- 0 pts** Correct

QUESTION 4

4 Problem 4 11 / 25

✓ **- 5 pts** (a). property should be about the bottleneck edge or lexicographic order

✓ **- 9 pts** (b). Not a linear time algoirthm

CS 180: Introduction to Algorithms and Complexity
Midterm Exam

Feb 20, 2019

Name	Siddharth Joshi
UID	105032378
Section	[redacted]

1	2	3	4	Total

★ **Print your name, UID and section number in the boxes above, and print your name at the top of every page.**

★ **Exams will be scanned and graded in Gradescope. Use Dark pen or pencil. Handwriting should be clear and legible.**

- The exam is a closed book exam. You can bring one page cheat sheet.
- There are 4 problems. Each problem is worth 25 points.
- Do not write code using C or some programming language. Use English or clear and simple pseudo-code. Explain the idea of your algorithm and why it works.
- Your answer are supposed to be in a simple and understandable manner. Sloppy answers are expected to receiver fewer points.
- Don't spend too much time on any single problem. If you get stuck, move on to something else and come back later.

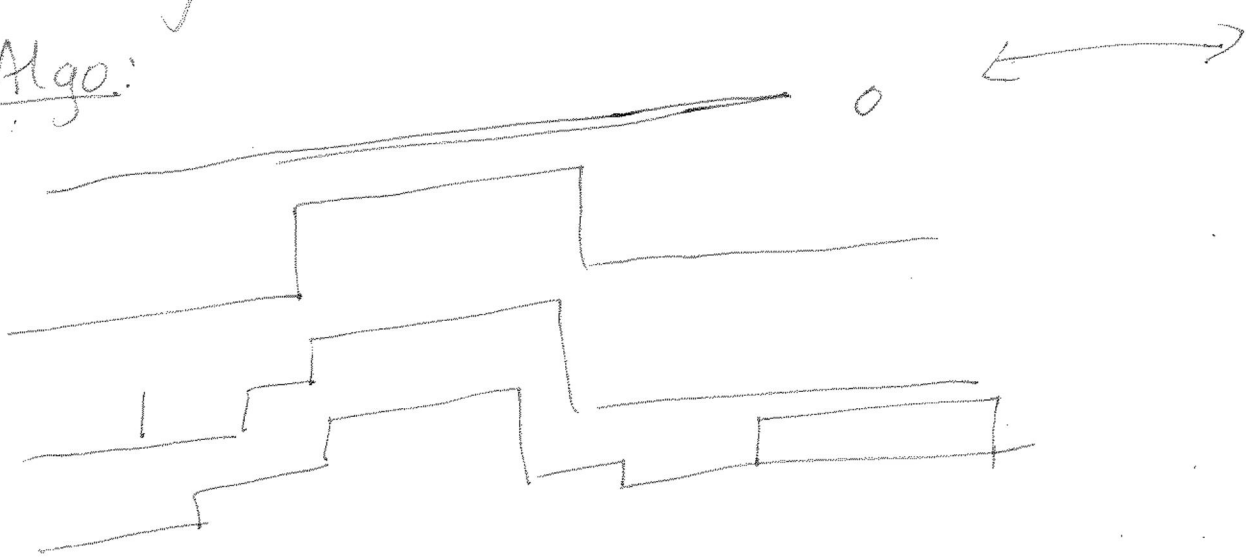
ALG:
sort by start time
~~max flow~~ Q
pressure [earliest start time] = ~~start~~ max (requests starting at earliest start time) max current = argmax ( )

until next ~~start time~~ max current has not ended):
H is this while (
pressure [i] = pressure at time
if (max Current (end end Time = i)
then max Current

unless we sort by
max flow in which case
max flow for all
start i ≤ i < end: if IST = ... if i
and so and so

sort by max flow not start time

Algo:



1. A water utility has to adjust its pressure according to the maximum rate of flow any of the customers need at the time, i.e., at time 3 pm the pressure has to be proportional to accommodate the maximum flow rate among the customers flow-rate demands. The Utility wants to plan ahead for the next day. The clients are n companies. Each submits a triple (start-time$_i$, end-time$_i$, flow-rate-required$_i$), $i = 1, \ldots, n$. The output of the utility produces is a graph whose axis is time, say 12 AM to 11:59 PM of the pressure at any time t that corresponds to the maximum flow-rate-required$_i$ over all i such that start-time$_i \leq t \leq$ end-time$_i$. Since the function jumps from fixed value to another fixed value (piece-wise constant), it can be described by at most about $3n$ values just telling the next value at the next point of time the value switches to another value, and the time of the switch.

   At perhaps the cost of sorting at the beginning, produce the graph of the function as described above for the Utility, incrementally proceeding from 12 AM to 11:59 AM.
   The cost of your algorithm should be $O(n \log n)$. (25 pts)

Algorithm:

→ Let customers be the list of 3-tuples containing the requests of each customer s.t. customers[i] = (start time$_i$, end time$_i$, flow-rate-required$_i$) and such that customers[i].start = start time$_i$, customers[i].end = end time$_i$ and customers[i].flow = flow-rate-required$_i$

→ Sort customers by comparing the flow rates required s.t. customers[i].flow ≤ customers[j].flow ∀ i<j s.t. i, j ∈ [1...n]

→ Let pressure be a list such that pressure[k] = value that pressure at time 12AM + k minutes must be i.e. max flow rate at time 12AM + k minutes

→ Set all values in pressure to 0

→ For i in (1...n):
- let req = customers[i]
- int sTime = req.start - 12AM // minutes from 12AM
- while (sTime < (req.end - 12AM)):
  - if (pressure[sTime] < req.flow)
    - → pressure[sTime] = req.flow
  - sTime++

→ Let graph be an empty list of 2-tuples s.t. graphs[i].first = next value at next point of time val switches
. second = time switches occurs

→ Insert

→ :0

→ 12:00 AM):

→ [lastTime])

→ return graph

Algorithm:

→ Let customers be the list of 3 tuples s.t. customer[i] corresponds to the 3 (startTime, endTime, flow-rate req) and customers[i].start = startTime, customers[i].end = endTime, customers[i].flow = flow rate req.

→ Sort customers key comparing flow-rate s.t. customers[i].flow ≤ customers[j].flow. ∀ i < j, i, j ∈ [1...n]

→ sUsed = ∞, eUsed = −∞, graph = list of 2-tuples indicating changes

→ for each request in (sorted) customers:

```
if (req.start > sUsed)
    if (req.end ≤ eUsed)
        do nothing
    else if (interval [req.flow, eUsed] is unused)
        insert (req.flow, eUsed) and mark any intervals that are subsets of [eUsed, req.end] as used
        eUsed = req.end
else
    if (req.end > sUsed and if interval [req.start, sUsed] is unused)
        insert (req.flow, req.start) and mark any intervals that are a subset of [req.start, sUsed] = used
    else if (interval req.end to req.start is unused)
        insert (req.flow, req.start) into graph
        mark [req.end, sUsed] as unused
```

→ For all unused intervals add change of flow

insert values to indicate flow to be 0 into graph

→ return graph

Correctness: same requests are sorted according to flow

a request j cannot be the max in any interval that is being used by requests [j+1 ... n] and this is kept track of using both sUsed, eUsed the used/unused intervals. It also must be max in all unused intervals as it's flow > flow of all reqs j+1 ... n

Time complexity: max time is taken by sorting as the for loop the runs in O(n) ∴ Time = O(n log n)

2. Same as the problem above only that now you solve the same problem with the same complexity using divide-and-conquer. (25 pts)

→ Algorithm:
→ If only 1 request exist then the graph is simply 0 in the interval [0, start of request] and [end of request, end of day], & it is also known to be = request's flow rate in [start of req, end of req]
Return this graph

→ Divide requests into 2 sets of ~~equal~~ R_1, R_2 equal size (or 1 more than other, if odd # requests)
→ Graph 1 = ~~Algorithm~~ Recurse with set of req requests = R_1
→ Graph 2 = Recurse with set of requests = R_2
→ Return Merge (graph 1, graph 2)

Definition of Merge function:
Merge (Graph 1, Graph 2)
Return a graph where value at any instant is max of (value at that instant, value at that instant in graph 2) in graph 1.

Correctness:
Base case is trivially correct when there is only 1 request, the value of the graph is simply the value at instant = flow if that time is in the interval and 0 if not.
The merge step: The two graphs represent at every instant the max value that any request in R_1 and R_2 take respectively. But the max value any request $\in R_1 \cup R_2$ takes is the max value of some request in R_1 or R_2 and hence the value at any instant in the combined graph = max (value of that instant in the 2 graphs for R_1 & R_2 respectively).

Time complexity:
There are $\log n$ subproblems as we divide # of request into 2 at every step and each subproblem takes $O(n)$ time as merge would take $O(n)$ time.
∴ Total runtime = time/subproblem × # of subproblems
$= O(n \log n)$

3. You are given n item types $x_1, \ldots, x_n$ each of integer volume value, and each type has infinite multiplicity (as many items of the type as you wish). In addition to a volume, an item of type x_i has a weight $w_i > 0$. Item of type x_1 has a volume 1. You are asked to fill a knapsack of integer volume V to carry a total of V cumulative volume of items, but you want to minimize the total weight you carry.

   Give a pseudo-polynomial algorithm to solve the problem. Write the recursion, and argue that it is amenable to Dynamic-Programming treatment. Outline your algorithm and analyze its complexity. (25 pts)

   For an added protection, if you did not solve the problem or just made a mistake, you will get partial credit for naming the problem by a name that you might have heard for the case when $w_i = 1$ for all types.

The case when $w_i = 1$ is the minimum coin change problem with value of coins = V desired

Algorithm:

→ Let DP be an array of size V+1

→ DP[0] = 0

→ For j in (1 ... V):

- min = +∞
- for all x_i s.t. j − volume of $x_i \geq 0$
  - if (w_i + DP[j − volume of x_i] < min)
    - min = w_i + DP[j − volume of x_i]
- DP[j] = min

→ Return DP[V]

Correctness: This uses bottom-up dynamic programming where solutions for all values {0 ... j−1} are computed before j so that the array access DP[...] never results in an error. Also the actual recurrence relation that the algo is uses is

$$DP[j] = \min_{\forall i \in \{1 \ldots n\},\ j - \text{volume}(x_i) \geq 0} (w_i + DP[j - \text{volume}(x_i)])$$

and this tries out all combinations
and hence is correct.

Time Complexity: $O(V * n)$
where V is the value and n is the
number of items as outer for loop = $O(V)$
and inner for loop = $O(n)$

O(V+|E|) - Dijkstra

4. Given a connected undirected graph $G = (V, E)$, with edge costs that you may assume are all distinct. A particular edge $e = \{v, w\}$ of G is specified. Give an algorithm with running time $O(|V| + |E|)$ to decide whether e is contained in the unique (why?) minimum spanning tree (MST) of G, or not. Notice that the complexity required is too low to produce the MST and check whether e is in it, or not.

Dijkstra

from v to all w ∈ G

(a) Give a property of the edge that determines if and only if the edge $e = \{v, w\}$ is in the MST. (10 pts) (Hint: Recall that we have seen in the homework that a MST is also the lexicographic MST and therefore in the MST, the unique path between v and w is the lexicographically smallest path.)

(b) Give an algorithm and argue it is of the complexity required. (15 pts)

a) An edge e=(v,w) of G is in MST if and only if v and w cannot be joined by a path consisting entirely of edges who are cheaper than e. This is seen naturally from the fact that (v,w) is the shortest path from v to w in the MST and hence if e=(v,w) ∈ MST it must be the shortest path from v to w, implying that any v and w cannot be joined by any combination of cheaper edges (otherwise e=(v,w) would not be the shortest path from v to w anymore, contradicting our initial assumption)

b) Algorithm:-
→ Run Dijkstra's algorithm to determine the shortest path from v to w
→ If shortest path has weight > weight of edge e = (v,w) then e is not in MST and if shortest path is equal to weight of e, then e is in MST

Time Complexity = $O(|V|+|E|)$
as Dijkstra's algorithm is the only non-constant work done and it's runtime is $O(|V|+|E|)$.

Correctness: From part a) it is clear that if an edge is to be in the MST it must be the shortest path from v to w or equal to weight of shortest path in G from v to w