CMSC 27200 - Spring 2024 — Lorenzo Orecchia
due 5/3/24 via Gradescope

H7.

- The assignment is due at Gradescope on 5/3/24.
- A LaTeX template will be provided for each homework. You are strongly encouraged to type your homework into this template using LaTeX. If you are writing by hand, please fill in the solutions in this template, inserting additional sheets as necessary. This will help facilitate the grading.
- You are permitted to discuss the problems with up to 2 other students in the class (per problem); however, *you must write up your own solutions, in your own words*. Do not submit anything you cannot explain. If you do collaborate with any of the other students on any problem, please list all your collaborators in the appropriate spaces.
- Similarly, please list any other source you have used for each problem, including other textbooks or websites.
- *Show your work.* Answers without justification will be given little credit.
- Your homework is *resubmittable*. Please refer to the course syllabus on Canvas for a more detailed description of this. For any problem that you have not changed from your last submission, please make sure to indicate this in your submission to help our graders grade faster.

PROBLEM 1 (LOADING...) *There are N jobs, and each needs to run from time exactly s_i to t_i. The CS department decided that it wants all jobs to be run, but to do so with the least number of machines, where each machine can only run one job at a time. Find the schedule of jobs that uses the minimum number of necessary machines.*

(a) Describe a greedy algorithm that finds the optimal schedule of jobs that uses the minimum number of machines. Your algorithm may be stated in words or with pseudocode.

(b) Provide a formal proof of its correctness. You can use either "greedy stays ahead" or "exchange argument".

Collaborators:

Solution:

Algorithm 1 Job Scheduling Algorithm

(a)
```
schedule(J_i)
Input: J_i = {(s_1, t_1), ..., (s_N, t_N)}
Output: M
J_i.sort(key = s_i)
M ← {}
for j_i : (s_i, t_i) ∈ J do
    if m ∈ M available after s_i then
        m.addToSchedule(j_i)
    else
        m ← New Machine
        m.addToSchedule(j_i)
        M.add(m)
end for
return M
```

(b) Let D_t be the number of jobs that overlap at a time t. Any optimal algorithm will need at least $D_{max} = \max[D_t]$ machines because no machine can run multiple jobs at once.

We claim the greedy algorithm above optimally uses at most D_{max} machines.

Let t_{max} be the first time at which D_{max} is achieved. Suppose for contradiction that at t_{max}, the algorithm allocates $> D_{max}$ machines. But this means that there were at least D_{max} machines already busy, contradicting the claim that at most D_{max} jobs overlap at any given time.

PROBLEM 2 (CLUSTERING) *The problem of **clustering** a sorted sequence of one-dimensional points $x_1, \ldots, x_n$ entails splitting the points into k clusters (where $k \leq n$ is an input parameter) such that the sum of the squared distances from each point to its cluster mean is minimized.*

For example, consider the following sequence with $n = 5$:

$$3,\ 3,\ 6,\ 16,\ 20$$

Suppose we want to partition it into $k = 2$ clusters. Here is one possible solution:

$$3, 3 \mid 6, 16, 20$$

The mean of the first cluster is $(3+3)/2 = 3$, and the mean of the second cluster is $(6+16+20)/3 = 14$. The cost (total variance) of this clustering is $(3-3)^2 + (3-3)^2 + (6-14)^2 + (16-14)^2 + (20-14)^2 = 104$. This clustering is not optimal because there exists a better one:

$$3, 3, 6 \mid 16, 20$$

The mean of the first cluster is $(3+3+6)/3 = 4$, and the mean of the second cluster is $(16+20)/2 = 18$. The cost of this clustering is $(3-4)^2 + (3-4)^2 + (6-4)^2 + (16-18)^2 + (20-18)^2 = 14$, which is optimal.

*Give a **dynamic programming** algorithm that takes as input an array $x[1..n]$ and a positive integer k, and returns the lowest cost of any possible clustering with k or fewer clusters. The running time should be $O(n^3k)$. Note: $O(n^3k)$ is not necessarily the optimal running time!*

(a) *Define the subproblem that you will use to solve this problem **precisely**, and give a recurrence relation that will help you solve this problem. Formally prove the correctness of the recurrence relation.*

(b) *Describe an algorithm that solves this problem based on your answer to (a), and show that your algorithm achieves the desired running time. Note that a description of any Dynamic Programming algorithm entails:*

- *The recurrence relation,*
- *The base cases,*
- *An explicit ordering in which the algorithm should evaluate the subproblems,*
- *How to extract the final answer from the completed table.*

(c) *Think about how you can improve the running time of your algorithm to $O(n^2k)$. **No need to turn in this part.** If you are curious about the solution, you can discuss with any TA.*
Hint: You can use $\sum_i \left(x_i - \frac{\sum_j x_j}{n}\right)^2 = \sum_i x_i^2 - \frac{(\sum_i x_i)^2}{n}$.

Collaborators:

Solution:

(a) **Notation and Definitions**

Let $\{x\}_j^k = \{x_j, \ldots, x_k\}$ for $j \leq k$.

Let $E(C_i) = \sum_{x \in C_i} (x - \mu_i)^2$ for cluster C_i with mean μ_i.

Let $\mathcal{L}(\{x\}_1^m, \ell)$ be the minimal $\sum_{i=1}^{\ell} E(C_i)$ attainable when clustering $\{x\}_1^m$ into ℓ clusters C_i.

Subproblem

The subproblem is $\mathcal{L}(\{x\}_1^j, k)$, the minimal cost k-clustering of $\{x\}_1^j$ for some j.

Recurrence Relation

The recurrence relation is:

$$\mathcal{L}(\{x\}_1^n, k) = \min_{k \le j \le n} \left[\mathcal{L}(\{x\}_1^{j-1}, k-1) + E(\{x\}_j^n) \right]$$

with base cases $\mathcal{L}(\{x\}_1^c, c) = 0$ and $\mathcal{L}(\{x\}_1^c, 1) = E(\{x\}_1^c)$

Correctness

In the base case of clustering c items into c clusters, the solution attains optimal cost of 0 by putting each element in its own cluster.

In the base case of clustering c items into 1 cluster, there is no option but to put everything in the same cluster.

Now consider the case of clustering n items into k clusters. If the kth cluster is $\{x\}_j^n$ for some j, necessarily $\mathcal{L}(\{x\}_1^n, k) = \mathcal{L}(\{x\}_1^{j-1}, k-1) + E(\{x\}_j^n)$. If this was not the case, we could get a better $\mathcal{L}(\{x\}_1^n, k)$ by using a better clustering of $\{x\}_1^{j-1}$.

The only feasible j's are those between k and n, inclusive, as otherwise there would be an empty cluster or j would be out of bounds, respectively.

(b) **Recurrence Relation** (again)

Our recurrence relation is $\mathcal{L}(\{x\}_1^n, k) = \min_j \left[\mathcal{L}(\{x\}_1^{j-1}, k-1) + E(\{x\}_j^n) \right]$ for $k \le j \le n$

Base Cases

- $\mathcal{L}(\{x\}_1^c, c) = 0$ for any $c \le k$
- $\mathcal{L}(\{x\}_1^c, 1) = E(\{x\}_1^c)$ for any $c \le n$.

Ordering

Let L be a table such that $L[p, q] = \mathcal{L}(\{x\}_1^p, q)$. Note that:

- L is 1-indexed, as the clustering of an empty set and the clustering a set into 0 clusters are both undefined.
- L is lower triangular, as $\mathcal{L}(\{x\}_1^p, q)$ is undefined for $p > q$

To compute L, first fill the diagonal and the left edge using the two base cases, respectively.

For arbitrary entry $L[p, q]$, for all $q \le j \le p$ compute $\mathcal{L}(\{x\}_1^{j-1}, q-1) + E(\{x\}_j^p)$.
But this is $L[j-1, q-1] + E(\{x\}_j^p)$.

Thus, $L[p, q]$ needs all table entries in the $q-1$ column above $p-1$.

Two possible orderings are:

- Top-to-bottom down each column, from leftmost to rightmost column
- Left-to-right across each row, from top to bottom row

Answer Extraction

Query table entry $L[n, k]$. Note that not all entries of L need to be computed to get $L[n, k]$.

Runtime

For a particular entry $L[p, q]$, there are at most n j's such that $q \le j \le p$. Recall that each j corresponds to a subproblem. For each j, one table entry is queried in constant time and $E(\{x\}_j^p)$ is computed in $O(n)$ time. Thus, each entry takes $O(n^2)$.

L up to $L[n,k]$ has n rows and k columns. So there are $O(nk)$ entries that need to be computed before $L[n,k]$ can be computed.

So in total, the algorithm takes $O(n^3k)$ time.

PROBLEM 3 (SCHEDULE PLANNING) *In this question, you will design a dynamic programming algorithm that solves the following problem: Given the schedule of talks for the conference at each hotel, and their associated values* $\{v_{i,h}\}_{i=1}^{n}$ *for* $h = 1, 2, 3$*, as well as the costs of taking Ubers* $\{c_{h,k}\}_{h,k=1,\ldots 3}$ *find the maximum value of any choice of talks to attend.*

Note:

- *The conference might last multiple days, but you have been allocated a room at each of the three hotels, so you can stay for free overnight in any of these three places.*
- *On the first day you can get a free Uber to whatever hotel you want to start in.*
- *You may assume that the price of Uber is independent of the direction* $(c_{h,k} = c_{k,h})$*, though your algorithm will likely generalize to a version of this problem which doesn't satisfy this property.*

Example: *Suppose that the value of the talks are given by the following table:*

Schedule Slot:	*1*	*2*	*3*	*4*
Hotel 1	*1*	*3*	*20*	*1*
Hotel 2	*1*	*30*	*1*	*2*
Hotel 3	*15*	*3*	*4*	*3*

and let the Uber prices be given by: $c_{1,2} = 3$*,* $c_{1,3} = 2$ *and* $c_{2,3} = 4$*. Then the optimal schedule is given by: starting out at hotel 3, moving to hotel 2 for the second talk, moving to hotel 1 for the third talk and then staying there for the last talk. The value of this sequence of talks is* $15 + 30 + 20 + 1 - 4 - 3 = 59$*: the sum of the values of the talks attended minus the cost of the two Ubers. So your algorithm would output 59.*

(a) Define a subproblem and give a recurrence relation that will help you solve the problem, and formally argue it is correct.

(b) Give the pseudocode of an algorithm that solves this problem based on your answer to (a)*. Your algorithm should run in time* $O(n)$ *where n is the number of talks taking place at each hotel (i.e. the number of time slots in which a talk could be scheduled). Sketch a proof that the algorithm you give satisfies this runtime bound. How much space does your algorithm use?*

Collaborators:

Solution: Your solution here.

(a) **Subproblem and Recurrence**

The subproblem is to decide where to go after a talk has finished.

Let $T_h^{(i)}$ be the talk held at hotel h during slot i.
Let $\mathcal{L}_h^{(i)}$ be the value of the optimal schedule to get to $T_h^{(i)}$.
Then:

$$\textbf{Recurrence:} \quad \mathcal{L}_h^{(i)} = \max_{h,g}[\mathcal{L}_g^{(i-1)} - c_{g,h} + v_{i-1,h}]$$

$$\textbf{Base Case:} \quad \mathcal{L}_h^{(1)} = \max_{h}[v_{1,h}]$$

Correctness

First, we claim that the recurrence relation correctly computes the optimal schedule to get to $T_h^{(i)}$. We use induction on the slot number i.

In the **base case** where $i = 1$, Uber rides are free for the first hotel so the only thing we need to consider is talk value. Thus, it is optimal to choose the most valuable talk in slot 1.

In the **inductive case**, suppose we know all $\mathcal{L}_g^{(i-1)}$. Now, the optimal schedule to attend some $T_h^{(i)}$ must necessarily go through some $T_g^{(i-1)}$. The sub-schedule to $T_g^{(i-1)}$ must be optimal. If it wasn't optimal, then we could get a better schedule to $T_h^{(i)}$ by re-scheduling the sub-schedule to $T_g^{(i-1)}$. From $T_g^{(i-1)}$, we can compute the schedule value by subtracting $c_{g,h}$ and adding $v_{i,h}$ to $\mathcal{L}_g^{(i-1)}$, which we know by our induction. $\mathcal{L}_h^{(i)}$ follows from choosing from the $|g|$ degrees of freedom, to get the maximum schedule value.

Now, once we have all the optimal schedule values, we can find the most valuable schedule for the whole day by choosing the maximum from among all $\mathcal{L}_h^{(i)}$ computed. This maximum will necessarily be found at the last slot.

(b) **Algorithm**

Algorithm 2 Dynamic Conference Scheduling Algorithm

Input: $\{c_{g,h}\}, \{v_{i,h}\}, H \equiv$ no. of hotels, $N \equiv$ no. of slots
Output: $\mathcal{L}$

$L \leftarrow arr[N, H]$
for $h \in \{1, \ldots, H\}$ **do**
    $L[1, h] \leftarrow v_{1,h}$
end for
for $i \in \{2, \ldots, N\}$ **do**
    for $h \in 1, \ldots, H$ **do**
        $L[i, h] \leftarrow \max_{1 \le g \le H} \left[L[i-1, g] - c_{g,h} \right]$
    end for
end for
return $\max_{1 \le h \le H} [L[N, h]]$

Runtime

The algorithm iterates to every element of L, which is $N \times H$. To compute the value of each element, it maximizes over all H elements in the previous slot i.

Thus, the algorithm is $O(NH^2) \equiv O(N)$ since H is constant.

It takes $O(NH) \equiv O(N)$ space.

Problem 4 (Lorenzo's Trip: Revenge of the Gas Engines) *Lorenzo is going on another trip! This time, he's back to his gasoline-powered car. He has already figured out the path he is going to take, and knows that there are gas stations $g_1, \ldots, g_n$ along the path (in that order). Lorenzo may buy a whole number of gallons of gas at each station, and pays c_i per gallon at station g_i. Lorenzo starts at g_1 with 0 gallons and ends at g_n. His tank holds at most C gallons of gas at once, and he gets m miles per gallon. Station g_i is distance d_i away from g_1, and the distance $|d_i - d_{i-1}|$ is a multiple of m but less than Cm for every $i > 1$. The problem we are interested in is the following: given $g_1, ..., g_n$, $d_1, ..., d_n$, C and m, find how much gas Lorenzo would need to buy at each station to reach g_n while minimizing the amount he pays for gas.*

(a) Define a subproblem and give a recurrence relation that will help you solve this problem, and sketch a proof that it is correct.

(b) Describe an algorithm that solves this problem based on your answer to (a), and state its runtime (with sketched proof). Note that a description of any Dynamic Programming algorithm entails:

- *The recurrence relation,*
- *The base cases,*
- *An explicit ordering in which the algorithm should evaluate the subproblems,*
- *How to extract the final answer from the completed table.*

Collaborators:

Solution:

(a) The **subproblem** is the cheapest sequence of gas purchases to get to g_i with n gallons of gas in the tank.

$$\textbf{Notation: } G[i, x] \equiv \text{Cost to get to } g_i \text{ with } x < C \text{ gallons left}$$
$$\Delta_i = \frac{1}{m}|d_i - d_{i-1}|$$

The **recurrence** is:

$$G[i, x] = \begin{cases} c_1 x & i = 1 \\ \min\limits_{0 \le y \le \Delta_{i-1}} \Big[G[i-1, y] \; + \; c_{i-1}(y - \Delta_i + x) \Big] & \text{otherwise} \end{cases}$$

Correctness can be shown using induction.

In the base case, Lorenzo is at g_1 with 0 gallons of gas, thus he just needs to add as much gas as he desires.

In the inductive case, assume we know all cheapest purchase sequences up to gas station g_{i-1} for any amount of gas. The cheapest $G[i, x]$ necessarily involves some cheapest $G[i-1, y]$, as otherwise $G[i, x]$ would not be the cheapest sequence. We thus need to search all y's and use basic arithmetic to determine the final cost for each. The inductive hypothesis gives us all $G[i-1, y]$'s.

(b) The **recurrence** is:

$$G[i, x] = \begin{cases} c_1 x & i = 1 \\ \min\limits_{0 \le y \le \Delta_{i-1}} \Big[G[i-1, y] \; + \; c_{i-1}(y - \Delta_i + x) \Big] & \text{otherwise} \end{cases}$$

The **base case** is $G[1, x]$, at which point we need to add $c_1 x$ worth of gas.

The **ordering** is determined by the dependence of $G[i, x]$ on all $G[i-1, y]$.

Suppose we have a table T such that $T[i, j] = G[i, j]$. Then to compute the value at some $T[i, j]$, we need all $T[i-1, y]$ for $0 \leq y \leq \Delta_{i-1}$. This corresponds to part of the $i-1$ row in T.

First, we fill row 1 using the base case. We then proceed row-by-row down the table.

The **final answer** follows from $G[n, 0]$, as any leftover gas would be a waste. Given T is completed, we can read $T[n, 0]$ to get the final cost. If we want the specific sequence, we can maintain another table to keep track of which y minimized $G[i-1, y] + c_{i-1}(y - \Delta_i + x)$. Alternatively we can run the algorithm in reverse to find the y.

The **runtime** is $O(nC^2)$. The table is $n \times C$. For each element in the table, the algorithm searches all $0 \leq y \leq \Delta_{i-1}$. Since Δ_{i-1} is bounded by C, each element takes $O(C)$ time to compute. Thus, the total runtime is $O(nC^2)$.

Problem 5 (Palindrome Detection) *A palindrome is a non-empty string that reads the same forward and backward. Examples of palindromes are "civic", "racecar", and "aibohphobia" (fear of palindromes). You are given a string as an array $S[1,\ldots,n]$ of length n. Your goal is to find the length of the longest palindromic subsequence of a string. Note that, by definition, a subsequence (unlike a subarray) can consist of* ***non-consecutive*** *elements. For example, for the string "character", the answer is* 5*, which corresponds to the palindromic subsequence "carac".*

(a) Define a subproblem and give a recurrence relation that will help you solve this problem. Formally prove the correctness of your recurrence.

(b) Describe an algorithm that solves this problem based on your answer to (a), and state its runtime (with sketched proof). Note that a description of any Dynamic Programming algorithm entails:

- *The recurrence relation,*
- *The base cases,*
- *An explicit ordering in which the algorithm should evaluate the subproblems,*
- *How to extract the final answer from the completed table.*

Collaborators:

Solution:

(a) Let $S = \{c_1,\ldots,c_n\}$ be a string of length n. We want to find the longest palindromic subsequence (PS) in S.

$$\begin{aligned}\textbf{Notation: } S[i,j] &= \{c_i,\ldots,c_{j-1}\}\\ P[i,j] &\equiv \text{Longest PS in } S[i,j]\\ \mathcal{L}[i,j] &\equiv \text{Length of } P[i,j]\end{aligned}$$

The **subproblem** is to find the longest PS in some slice of S.

The **recurrence** is:

$$\mathcal{L}[i,j] = \begin{cases} 0 & i > j-1\\ 1 & i = j-1\\ \mathcal{L}[i+1,j-1]+2 & c_i = c_{j-1}\\ \max\Big[\mathcal{L}[i+1,j],\mathcal{L}[i,j-1]\Big] & \text{otherwise}\end{cases}$$

Correctness follows from induction on the slice length.

In the base case, if $S[i,j]$ invalid, no palindrome can exist, so $\mathcal{L}[i,j] = 0$ is correct. If $S[i,j]$ is one character, then trivially $\mathcal{L}[i,j] = 1$.

In the inductive case, assume that we already know the correct $\mathcal{L}$ for all subsequences of length $< \ell - 1$. Now we want some $\mathcal{L}[i,j]$ where $j - i = \ell$.

If $c_i = c_{j-1}$, necessarily c_i and c_{j-1} must be apart of $P[i,j]$, otherwise we could make a longer $P[i,j]$ by concatenating them. Then $P[i,j] = c_i \cup P[i+1,j-1] \cup c_{j-1}$, so $\mathcal{L}[i,j] = \mathcal{L}[i+1,j-1]+2$. By the induction, we already know $\mathcal{L}[i+1,j-1]$ so this value can be computed correctly. Note that pruning c_i and c_{j-1} in the recursion solves the problem of one of c_i and c_{j-1} being 'paired' with some other character.

If $c_i \neq c_{j-1}$, we can use our inductive hypothesis by computing the two slices $\mathcal{L}[i+1,j]$ and $\mathcal{L}[i,j-1]$. One of them must be equal to $\mathcal{L}[i,j]$ since one of c_i and c_{j-1} might still be in $P[i,j]$, but the other will not be.

(b) The **recurrence** is:

$$\mathcal{L}[i,j] = \begin{cases} 0 & i > j-1 \\ 1 & i = j-1 \\ \mathcal{L}[i+1,j-1]+2 & c_i = c_{j-1} \\ \max\Big[\mathcal{L}[i+1,j], \mathcal{L}[i,j-1]\Big] & \text{otherwise} \end{cases}$$

The **base cases** are $i > j-1$ and $i = j-1$. We can ignore the former if we are smart about our ordering.

The **ordering** follows from the dependence of $\mathcal{L}[i,j]$ on $\mathcal{L}[i+1,j-1]$ if $c_i = c_{j-1}$, and both $\mathcal{L}[i+1,j]$ and $\mathcal{L}[i,j-1]$ otherwise. Suppose our subproblem table L is organized such that $L[i,j] = \mathcal{L}[i,j]$. The base cases give the values on the offset diagonal, and restrict the table to be upper triangular. The dependencies, most generally, require the cells below, to the left, and to the bottom left. Thus, we can compute all entries by traversing the diagonals, starting at the main diagonal filled with the base cases, and finishing at the upper right corner of the table.

The **final answer** is $\mathcal{L}[1,n]$, which is just the entry at the 1st row and nth column of the table.

The **runtime** is $O(n^2)$. There are $O(n^2)$ table entries to fill, and each entry takes a constant amount of computation.