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

H7.Mark Xiong

- 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: The solution is as follows:

(a) The greedy algorithm is as follows: Start by sorting the jobs by their end time. Initialize a priority queue of machines, which holds the end time of the last job on the machines. For each job in the sorted list, if the priority queue is empty, assign the job to a new machine. Otherwise, assign the job to the machine with the earliest end time with the exception that if the job's start time is before the end time of the machine, assign the job to a new machine. Update the end time of the machine with the end time of the job. Repeat until all jobs are assigned.

(b) We will prove the correctness of this algortihm using "greedy stays ahead" and by induction.

Base Case: For the base case, we have $N = 1$ job. The algorithm will assign the job to a new machine, which is the optimal solution.

Inductive Hypothesis: Assume that the algorithm is correct for $N = k$ jobs. Consider the case for $N = k + 1$ jobs. If an existing machine is available, the algorithm will assign the job to the machine with the earliest end time, which will not increase the count of machines. If the job's start time is before the end time of the machine, the algorithm will assign the job to a new machine. This is the optimal solution because the new job cannot overlap with any of the previous jobs on the same machine due to the overlap in timing, and therefore needs a new machine. Therefore, the $N = k + 1$ case also finds the optimal solution, which concludes the proof.

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: The solution is as follows:

(a) Define $dp[i][j]$ as the minimum cost of clustering the first i points into j clusters.
The recurrence relation is as follows: $dp[i][j] = \min_{1 \leq l \leq i}\{dp[l][j-1] + \text{cost}(l+1, i)\}$, where $\text{cost}(l+1, i)$ is the cost of clustering the points from $l+1$ to i into a single cluster. The cost is calculated as the sum of the squared distances from each point to the mean of the cluster.

To prove the correctness of the recurrence relation, we will use induction.

Base Case: For the base case, we have $j = 1$ cluster. $dp[i][1]$ is the cost of clustering the first i points into a single cluster, which is the sum of the squared distances from each point to the mean

of the cluster. This is the optimal solution. When $i = 0$, $dp[0][j] = 0$. This is also valid because clustering 0 points into 1 cluster has a cost of 0.

Inductive Step: Assume that the algorithm is correct for $j-1$ clusters and for all points up to i. Consider the case for j clusters. $cost(l+1, i)$ will accurately calculate the sum of the squared distances for points from $l+1$ to i to the mean of the cluster. For $dp[l][j-1]$, by our inductive hypothesis, it is the optimal way to cluster the first l points into $j-1$ clusters. To form j clusters optimally up to the ith point, we can add a new cluster starting after some point l and ending at the ith point. The optimality of the clusters from 1 to l is guaranteed by the inductive hypothesis combined with the optimal calculation of the cost from $l+1$ to i makes sure that we have the optimal result. The use of minimum operation over all possible values of l ensures that we are selecting the best possible place to split the sequence into a new cluster, thus ensuring the least cost. This concludes the proof.

(b) The algorithm is as follows:

Data Structure: We will have a DP table dp of size $n \times k$.

Base Cases: $dp[0][j] = 0$ for all j and $dp[i][1] = \text{cost}(1, i)$ for all i. These are optimal solutions as 1) we don't have clusters to calculate 2) clustring into one cluster only has 1 option.

Recurrence relation: Fill out the DP table by considering each number of clusters from 2 to k, and for each cluster count, calculate the cost for up to all points. Therefore, we have:
$dp[i][j] = \min_{1 \le l \le i}\{dp[l][j-1] + \text{cost}(l+1, i)\}$, where $cost(l+1, i)$ calculates the mean of points from $l+1$ to i and sum up the squared differences from this mean.

Ordering: We have the following ordering:

(1) iterate over j from 1 to k

(2) for each j, iterate over i from 1 to n

(3) for each i, iterate over possible l from 1 to i to find the minimal cost.

Extract Answer: the minimum cost of clustering all n points into k clusters is found at $dp[n][k]$.

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: The solution is as follows:

(a) The subproblem can be defined as follows: $dp[i][h]$ is the maximum value of any choice of talks to attend up to the ith talk at hotel h. The recurrence relation is as follows: $dp[i][h] = v_{i,h} + \max_{1 \le k \le 3}\{dp[i-1][h], max_{k \neq h}(dp[i-1][k] - c_{k,h})\}$. This relation calculates the maximum value of any choice of talks to attend up to the ith talk at hotel h by considering the maximum value of any choice of talks to attend up to the $i-1$th talk at any hotel k and subtracting the cost of the Uber from hotel k to hotel h.

To prove the correctness of the recurrence relation, we will use induction.
Base Case: For the base case, we have $i = 1$ talk. This is correct because the best value at the first time slot is simply attending the talk at that hotel as there are no prior talks or transitions that can influence the value.
Inductive Step: Assume that the algorithm is correct for $i-1$ talks. Consider the case for i talks. By our inductive hypothesis, $dp[i-1][h]$ is the maximum value of any choice of talks to attend up to the $i-1$th talk at hotel h. The maximum value of any choice of talks to attend up to the $i-1$th talk at any hotel k is also correct, hence subtracting the Uber cost $c_{h,k}$ computes the optimal value for making a switch to h. Since adding $v_{i,h}$ does not affect the max value of previous attendence and even

with potential Uber cost, the addition gives the highest value between staying or switchingh to hotel h. Therefore $dp[i][h]$ is the maximum achievable at time i when the last talk atteneded is at hotel h. This concludes the proof.

Algorithm 1 `Optimal_Value_from_Conference`$(\{v_{i,h}\}, \{c_{h,k}\})$.

(b)
Input: $\{v_{i,h}\}$, $\{c_{h,k}\}$ for $i = 1, \ldots, n$ and $h, k = 1, 2, 3$.
output: The maximum value of any choice of talks to attend.
Initialize dp as a matrix of size $n \times 3$.
for $i = 1$ to n do
  for $h = 1$ to 3 do
    $dp[i][h] = v_{i,h} + \max_{1 \le k \le 3}\{dp[i-1][h], max_{k \neq h}(dp[i-1][k] - c_{k,h})\}$
  end for
end for
return $\max_{1 \le h \le 3} dp[n][h]$

The initialization of the dp matrix takes $O(n)$ time as it has 3 rows with each taking n operations. Similarly, to fill out the matrix, the nested loops iterate over all n talks and all 3 hotels, which takes $O(n)$ time. Traversing the matrix to find the maximum value also takes $O(n)$ time. Therefore, the total runtime of the algorithm is $O(n)$.

The space complexity of the algorithm is $O(n)$ as the dp matrix has n rows and 3 columns, which costs space complexity of $O(3n) = O(n)$.

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: The soltion is as follows:

(a) We could define the subproblem as follows: $dp[i]$ is the minimum cost of gas needed to reach g_i. The recurrence relation is as follows: $dp[i] = \min_{1 \le j < i}\{dp[j] + c_j \cdot \frac{d_i - d_j}{m}\}$, where $\frac{d_i - d_j}{m} \le C$.

The correctness of the recurrence relation can be proven by the following: Since we start at g_1 with no gas, $dp[1] = 0$. Then, for each gas station g_i, we consider all feasible gas stations g_j that we could have come from and calculate the cost of reaching g_i from g_j and add the cost of gas at g_j. This ensures that $dp[i]$ is the minimum cost to reach g_i considering all valid paths from g_j to g_i. As the recurrence captures the decision process at each station, we ensure the global optimum by locally optimal steps.

(b) The algorithm is as follows:
We will have an array dp of size n. The algorithm is established based on the reccurance relation: $dp[i] = \min_{1 \le j < i}\{dp[j] + c_j \cdot \frac{d_i - d_j}{m}\}$, where $\frac{d_i - d_j}{m} \le C$. To start at the first station, we have $dp[1] = 0$ as the base case. This implies Lorenzo starts with the first gas station with zero cost because he starts with no gas and does not need to buy any gas to remain at g_1. We then iterate over all stations g_i and for each station, we iterate over all stations g_j such that $1 \le j < i$ and calculate the minimum cost to reach g_i from g_j. Update $dp[i]$ with the minimum cost. The final answer is the minimum cost to reach g_n, which is $dp[n]$.

The runtime of the algorithm is $O(n^2)$ because of the following: the initialization of the dp array takes $O(n)$ time. The nested loops iterate over all n stations and all n stations, which takes $O(n^2)$ time. Therefore, the total runtime of the algorithm is $O(n^2)$.

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: The solution is as follows:

(a) We could define the subproblem as follows: $dp[i][j]$ is the length of the longest palindromic subsequence of the substring $S[i,\ldots,j]$. The recurrence relation is as follows: $dp[i][j] = \begin{cases} 2+dp[i+1][j-1] & \text{if } S[i]=S[j] \\ \max(dp[i+1][j], dp[i][j-1]) & \text{if } S[i]\neq S[j] \end{cases}$

We could prove the correctness of the recurrence relation by induction.
Base Case: For the base case, we have $j=i$. The length of the longest palindromic subsequence of a single character is 1. Therefore, $dp[i][j]=1$ if $i=j$. Moreover, if $i>j$, $dp[i][j]=0$ because a substring with a negative length does not have a palindrome.

Inductive Step: Assume that the algorithm is correct for $j-i$ length substrings. Consider the case for $j-i+1$ length substrings. If $S[i]=S[j]$, then the length of the longest palindromic subsequence of the substring $S[i,\ldots,j]$ is 2 plus the length of the longest palindromic subsequence of the substring $S[i+1,\ldots,j-1]$, which is correct by our hypothesis. If $S[i]\neq S[j]$, then we have to ignore either $S[i]$ or $S[j]$. The length of the longest palindromic subsequence of the substring $S[i,\ldots,j]$ is the maximum of the length of the longest palindromic subsequence of the substring $S[i+1,\ldots,j]$ and the length of the longest palindromic subsequence of the substring $S[i,\ldots,j-1]$, which is also true by out hypothesis. This ensures that $dp[i][j]$ is the length of the longest palindromic subsequence of the substring $S[i,\ldots,j]$ for all i and j, and therefore concludes the proof.

(b) The algorithm is as follows:

We will have a 2D array dp of size $n\times n$ to store the length of the longest palindromic subsequence of the substring $S[i,\ldots,j]$. The algorithm is established based on the recurrence relation: $dp[i][j] = \begin{cases} 2+dp[i+1][j-1] & \text{if } S[i]=S[j] \\ \max(dp[i+1][j], dp[i][j-1]) & \text{if } S[i]\neq S[j] \end{cases}$. To start at the first station, we have $dp[i][j]=1$ for $i=j$ and $dp[i][j]$ for $i>j$ as the base cases. We then iterate over substrings with length 2, fill out the dp table according to our reccurence relation, and then procced to substrings with length 3 to fill out

the relevant entries, and so on until we fill out the entry where reaches the length of the string. The final answer is the length of the longest palindromic subsequence of the substring $S[1, \ldots, n]$, which is $dp[0][n-1]$.

The runtime of the algorithm is $O(n^2)$ because of the following: the initialization of the dp array takes $O(n^2)$ time as we have n row and n columns. The nested loops iterate over all n substrings of all n lengths, which also takes $O(n^2)$ time. Therefore, the total runtime of the algorithm is $O(n^2)$.