Judah Goldring
Professor Leff
Spring 2023
Design and Analysis

DamConstruction Write-Up:

Using a counting argument I would argue that with a brute force solution you would end up evaluating N! dams since you could evaluate all N dams first then N-1 dams second and so on.

Essentially the key insight in solving this problem is the fact that the dams being provided are sorted in ascending order. Meaning if you build a dam at index i, the stretch of river to the right and left of that dam are not affected by each other. For example, lets say the provided array was {2, 4. 6} creating a dam at 4, would leave you with a stretch of land from 0-4 and 4-riverend and with dams {2} and {6}. Since its sorted, creating a dam between 0-4 i.e. at {2} has no effect on the segment between 4 and riverend i.e. {6}. These two unrelated substructures mean you can recursively solve the right side and left side of the problem after splitting at a certain index. After having this important insight, a recursive solution arises. Essentially just loop through the array, creating a dam at index i and recursing down the left and right substructures of this created dam. Another key insight is the identification that to calculate cost of creating a specific dam, you would need to subtract the distance of the dam to the left from the distance of the dam to the right. Meaning a dam to the left and right of you needs to exist. That is why having a dam created for us at the riverend and start is so integral to solving this problem. To use this vital information, we need to add the dam at 0 and riverend to our array of dams, so that when evaluating, we'll always have a dam in front and behind. The evaluation of a specific river segment i,j would then just be arr[j+1] – arr[i-1]. And because well never be evaluating dams at 0 or riverend well never go out of bounds.

Now for the structure, when we loop through a river segment, we check the cost of creating a dam at index k and then recurse to the right and left of that index. We do that for every index of our dam array and compare their costs against each other. The minimum cost is then returned. Since were looping through our array and constantly recursing down left and right substructures we will encounter substructures that we have already calculated and recalculating them would be a big waste of time. Meaning let's say you have array {2, 4, 6, 8, 10} and we create a dam at {6}. Now we have river segment {2, 4} and {8,10}, but in the previous loop when we decided to create our first dam at {4} and then a dam at {6} and were left with segment {8,10}. It would be a big waste of time to keep calculating cost of substructures that we encounter again. So, this is where Dynamic Programming comes in. Dynamic Programming tells us to break solutions down into smaller problems, which we have done, and save the solutions to the smaller problems, which we haven't done yet, to save time and recalculation of a problem we already solved. Instead of recalculating the cost of substructure I,j we will save it in a 2d array we'll call dp. The value at dp[i][j] will tell us the minimum cost of constructing dams for the segment i j. Therefore, when we're recursing we can check if it's been calculated already and break right there!

Reccurence:

Base case:

i > j return 0

Else:

Cost(i, j, arr) = min(arr[j +1] – arr[i-1] + cost(i, k-1) + cost(k+1, j)) where i <= k <= j

And in our case we would store the values returned to cost in a 2d memorization table allowing us to save our work.

At the end of the day all dams still need to be evaluated to determine the minimum cost of construction, however with dynamic programming we don't need to recheck the cost of segments that were already figured out saving us time in calculation.

Also, even though you didn't ask for it, I figured I'll explain how I did cost, since I didn't use any information from solve() and I liked my solution. Essentially, I first created an array list and added 0 and riverend to list. I then looped through the evaluation sequence and using binary search found the index j of the array list that adding the value at index i of the sequence would maintain sorted order. I then added to a total cost variable the difference of the index in front and behind this added evaluation sequence element, similar to the (j+1) – (i-1) I did with solve(). Since the list always included a dam at 0 and riverend there was always going to be a dam in front and behind any dams to be constructed in the evaluation sequence.