C1 BASIC (YLW4SJ)

Collaborators: list collaborators's computing IDs

Sources: Cormen, et al, Introduction to Algorithms. *(add others here)*

PROBLEM 1 *Birthday Prank*

Prof Hott's brother-in-law loves pranks, and in the past he's played the nested-present-boxes prank. I want to repeat this prank on his birthday this year by putting his tiny gift in a bunch of progressively larger boxes, so that when he opens the large box there's a smaller box inside, which contains a smaller box, etc., until he's finally gotten to the tiny gift inside. The problem is that I have a set of n boxes after our recent move and I need to find the best way to nest them inside of each other. Write a **dynamic programming** algorithm which, given a list of dimensions (length, width, and height) of the n boxes, returns the maximum number of boxes I can nest (i.e. gives the count of the maximum number of boxes my brother-in-law must open).

Solution: First, sort the boxes in decreasing order based on volume and put them in an array called *boxes*. Next, create an array *max* that keeps track of the maximum number of nested boxes for every box i with i as the last box to be opened. Fill *max* with 1's. Next, use a for loop: i for $i < n$, and inside that another for loop: j for $j < i$. If the width, depth, and height of $boxes[i]$ is less than the width, depth, and height of $boxes[j]$ and max[i] < max[j] + 1, then max[i] = max[j] + 1. When the for loops are finished, return the maximum value in max[].

PROBLEM 2 *Arithmetic Optimization*

You are given an arithmetic expression containing n integers and the only operations are additions $(+)$ and subtractions $(-)$. There are no parenthesis in the expression. For example, the expression might be: $1 + 2 - 3 - 4 - 5 + 6$.

You can change the value of the expression by choosing the best order of operations:

$$((((1 + 2) - 3) - 4) - 5) + 6 = -3$$
$$(((1 + 2) - 3) - 4) - (5 + 6) = -15$$
$$((1 + 2) - ((3 - 4) - 5)) + 6 = 15$$

Give a **dynamic programming** algorithm that computes the maximum possible value of the expression. You may assume that the input consists of two arrays: `nums` which is the list of n integers and `ops` which is the list of operations (each entry in `ops` is either `'+'` or `'-'`), where `ops[0]` is the operation between `nums[0]` and `nums[1]`. *Hint: consider a similar strategy to our algorithm for matrix chaining.*

Solution:
First, create an n by n grid called *maxValue*. Fill the diagonal (where $i = j$) with $nums[i]$. For example, at $i = 3$ and $j = 3$, fill that index with $nums[3]$. Keep the rest of the grid blank.

Next, start filling in the grid diagonally, moving up from the diagonal you just filled in. For every location $maxValue[i_{curr}][j_{curr}]$, do the following:

for k in range(j_{curr}):
  operator = ops[k]
  if (operator == "+"):
    tempMax = maxValue[i_{curr}][k] + maxValue[k+1][j_{curr}]
  if (operator == "-"):
    tempMax = maxValue[i_{curr}][k] - maxValue[k+1][j_{curr}]
  if (maxValue[i_{curr}][j_{curr}] is blank or maxValue[i_{curr}][j_{curr}] $<$ tempMax):
    $maxValue[i_{curr}][j_{curr}]$ = tempMax

This essentially breaks the problem into smaller sub-problems and finds the maximum possible value of the expression at each step and uses those values to solve larger sub-problems. The answer will be at $maxValue[0][n-1]$ (the top right corner of the grid).

PROBLEM 3 *Optimal Substructure*

Please answer the following questions related to *Optimal Substructure*.

1. Briefly describe how you used *optimal substructure* for the Seam Carving algorithm.

   Solution: We used the optimal substructure by picking the least-energy seam connected to a pixel at each step. The energy of each pixel represents the least-energy path from the bottom of the image to that pixel. Therefore, larger sub-problems can be solved using the solutions to smaller sub-problems by using previously calculated optimal paths.

   (a) M(x, y) = lowest energy seam that starts at the bottom and ends at pixel x, y
   (b) M(x, y) = e(x, y) + min(M(x-1, y-1), M(x, y-1), M(x+1, y-1)
   (c) If y=0, M(x, y) = e(x, 0)

2. Do we need optimal substructure for Divide and Conquer solutions? Why or why not?

   Solution: No, we do not need optimal substructure. This is because each sub-problem in divide and conquer does not overlap, therefore an optimal solution does not contain optimal solutions to sub-problems.

PROBLEM 4 *Dynamic Programming*

1. If a problem can be defined recursively but its subproblems do not overlap and are not repeated, then is dynamic programming a good design strategy for this problem? If not, is there another design strategy that might be better?

   Solution: No, dynamic programming is not a good design strategy since it considers several choices for a sub problem. Since the sub problems do not overlap, a greedy algorithm would be best for this problem.

2. As part of our process for creating a dynamic programming solution, we searched for a good order for solving the sub-problems. Briefly (and intuitively) describe the difference between a top-down and bottom-up approach. Do both approaches to the same problem produce the same runtime?

Solution: A top-down approach recursivley solves sub-problems and stores the results for further use so that recalculation is not needed. A bottom-up approach iteratively solves sub-problems smallest to largest (using the solutions to the smaller problems for the larger problems). These approaches produce the same run-time.

PROBLEM 5 *Gradescope Submission*

Submit a version of this `.tex` file to Gradescope with your solutions added, along with the compiled PDF. You should only submit your `.pdf` and `.tex` files.