CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, APRIL 15, 2022 AT **11:30 pm** VIA GRADESCOPE C1BASIC(WYZ5RGE)

Collaboration Policy: You are encouraged to collaborate with up to 3 other students, but all work submitted must be your own *independently* written solution. List the computing ids of all of your collaborators in the `collabs` command at the top of the tex file. Do not share written notes, documents (including Google docs, Overleaf docs, discussion notes, PDFs), or code. Do not seek published or online solutions for any assignments. If you use any published or online resources (which may not include solutions) when completing this assignment, be sure to cite by naming the book etc. or listing a website's URL. Do not submit a solution that you are unable to explain orally to a member of the course staff. Any solutions that share similar text/code will be considered in breach of this policy. Please refer to the syllabus for a complete description of the collaboration policy.

Collaborators: jhs8cue, spj6s

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: $nestedBoxes(l, w, h) = max_{k=1}^{n}\{nestedBoxes(l_k, w_k, h_k)\} + 1$, where k represents the largest box that can fit into the current box.

This algorithm is a top-down approach that chooses the next box based on whether it can fit the most boxes inside of it after being placed in the first box, i.e., the next largest box. The length, width, and height of the first box are passed into the algorithm so the algorithm can determine if other boxes have smaller dimensions and can thus fit inside. The recursive call passes in the dimensions of the box chosen as that is the new volume of space that any subsequent boxes must fit into. 1 is added onto the end of the recursive call, as this counts the box that the smaller box was just put into. The algorithm will divide the problem into subproblems of all combinations of nested boxes that can fit into each larger box, choosing the optimal solution for each subproblem until it solves the overall problem. The algorithm keeps track of the solutions to subproblems through memoization through use of a 3D array, storing values at indices corresponding to a specific box's dimensions, and using those data points as inputs to larger subproblems.

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: This is similar to the log cutting problem we have seen in class. We can use dynamic programming to choose which number/operation to "cut", or exclude from a pair of parentheses in the best way so that the total value of the expression is maximized. Starting with the entire expression, the first number is positive, so compare it against the last number. Any positive number will contribute to the max value, so it can be excluded. Exclude the first or last number based on which one is bigger, leaving us with $maxVal(i,j) = maxmaxVal(i+1,j) + nums[i], maxVal(i,j-1) + nums[j]$. Once we encounter a minus sign at the beginning of a particular subproblem, then we want to find the minimum, as subtracting the most negative value is the same as adding the maximum value. To find the minimum value of an expression, we use the same approach, but excluding the first or last number based on which is more negative. Overall, our algorithm is then described as:

$maxVal(i,j)_{k=0}^{k=n} = max\{maxVal(i+1,j) + nums[i], maxVal(i,j-1) + nums[j]\}$
if ops[k] = "+";
$min\{minVal(i+1,j) + nums[i], minVal(i,j-1) + nums[j]\}$
if ops[k] = "-"

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: Optimal substructure is used in the seam carving algorithm, as the algorithm must choose the lowest energy seam from top to bottom, so it creates subproblems of the energy level of the current pixel added to the seam energy of one three adjacent pixels directly below it, and chooses the minimum.

   $S(n,k) = min\{S(n-1,k-1), S(n-1,k), S(n-1,k+1)\}$

   The values of these subproblems are stored via memoization with an adjacency matrix and then used in larger subproblems.

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

   Solution: Yes - divide and conquer solutions also need an optimal substructure, as they break a larger problem into smaller subproblems, whose solutions are combined to determine the solution to the main problem.

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: If no overlap or repetition exists in the subproblems, then dynamic programming would be a waste of time. There is no decision to be made about how a problem should be divided into subproblems, so memoization would be wasteful - a divide and conquer approach would be better.

2. As part of our process for creating a dynamic programming solution, we searched for a good order for solving the subproblems. 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 dynamic programming approach is recursive in nature, whereas bottom-up is iterative. Top-down begins with a main problem, whose solution depends on breaking it down into subproblems, until the subproblems are small enough to be saved to memory and their values/solutions can then be used to solve larger subproblems going back up the "tree" until an overall solution is computed. Bottom-up involves starting with simple problems and using memoization to use their solutions to build up to larger problems. Both approaches use a lookup table for memoization, but top-down uses recursive calls, giving it slightly more overhead than bottom-up, which only uses the table.

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.