CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, APRIL 15, 2022 AT **11:30 pm** VIA GRADESCOPE C1BASIC(RTG5XKH)

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: ra9ha

Sources: Cormen, et al, Introduction to Algorithms.

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, we will pre-process each box b, finding its shortest dimension (defined by b.sd), its middle-length dimension (defined by b.md) and its largest dimension (defined by b.ld). We will also pre-process each box by finding every other box in the list that can fit into this box. This will be represented as a list attribute of the box, b.canContain, and will be found by calling the function $fits(b_2, b_i)$ on each other box b2 in the group for the b_i that we are currently processing. If $fits(b_2, b_i)$ is True, we will add b_2 to b_i.canContain. fits can be defined like so:

def fits(b_1, b_2):
  if $b1.sd < b2.sd$ and $b1.md < b2.md$ and $b1.ld < b2.ld$:
    return True
  else:
    return False

We can find an optimal solution by finding the maximum number of boxes in solutions starting with each box, and returning the maximum of all of these results.
To find the most boxes that can be nested in a solution starting with box b, we will make use of mostBoxes(b), which is defined by:
mostBoxes(b) = 1 + max(mostBoxes(b_i) for each box b_i in b.canContain). The base case is when b.canContain has 0 boxes, in which case mostBoxes(b) will simply return 1 (to represent that box itself).
The memory for this dp problem will contain n spaces, one for each box, and it will store the maximum number of boxes that can be nested within the box represented by a certain index of the memory. Thus, mostBoxes(b_i) can obviously be retrieved in constant time straight from memory once it has been computed once for a given box b_i. This saves time by preventing repeat computations of mostBoxes() for the same box.
Once we have computed mostBoxes(b) for each box in the list of boxes, we can take the max of

these results to determine the final answer for the max number of boxes that can be nested.

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: For this problem, we will need to consider different cases for what the "best" solution to a subproblem means. To find the maximum value, if we are subtracting a number from another number, we will want the number we are subtracting to be as small as possible. On the other hand, if we are adding a number, we will want that number to be as large as possible. We will define a recursive max(i, j) function, which will compute the maximum possible value for the subproblem consisting of nums[i] through nums[j] and ops[i] through ops[j + 1]. However, it is not as simple as this because we will need to account for the cases mentioned above. We will also need to compute and store min(i, j), because when we encounter a minus sign, it will be the best solution to subtract the minimum value possible.
Thus, to calculate max we will need to consider the following possibilities:

max(i, j) = max(max(i, i+k) ops[i+k] max(i+k+1, j), max(i, i+k) ops[i+k] min(i+k+1, j))
for all k in range(1, j-i-1)

Similarly, we can define min as the following:
min(i, j) = min(min(i, i+k) ops[i+k] max(i+k+1, j), min(i, i+k) ops[i+k] min(i+k+1, j))
for k in range(1, j-i-1)

For the base cases, max(i, i) = nums[i], and min(i, i) = nums[i]

Now that we have accounted for the intricacies of the problem with a DP structure, we can simply call max(0, n-1) to compute the result of the entire problem. We will of course use memoization to take advantage of overlapping subproblems and save time with repeated computations. Memory will consist of not one, but 2 2D arrays. One for the maxes and one for the mins. The memory for the maxes at index (0, n-1) will contain the overall solutions to the problem.

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: For seam carving, the optimal substructure I made use of was the fact that at

each row r, an optimal seam from the start up to r must include one of the optimal seams from the start up to row $r-1$. Thus, I was able to use dynamic programming to store the answers to subproblems and work my way along the rows to the final row using these stored answers.

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

   Solution: No. Divide and conquer algorithms do not explicitly need optimal substructure because part of the solution to the overall problem is not explicitly contained within the subproblems. D&C needs a combine step to figure out the answer to the overall problems, so the answer cannot be explicitly constructed by its subproblems. Thus D&C does not need optimal substructure in the same way that DP or algorithms do.

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 effective for problems where the answers to subproblems are repeatedly needed. With non-overlapping subproblems, the idea behind DP that makes it efficient no longer applies. In this situation, a Divide and Conquer algorithm would be more useful than DP because D&C algorithms make use of the fact that the same operations are recursively applied to several subproblems, each of which are only looked at once (they do not overlap). This matches the problem type described in the question.

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: In the top down approach, we essentially start by assigning the solution to the overall problem (the "last" space in memory) in terms of its subproblems. The subproblems will of course all be solved recursively before this overall solution can be reached, however in code we essentially define the overall solution first, and work recursively from there.
   On the other hand, the bottom up approach's code explicitly begins by filling out memory for the smallest subproblem and works its way up by using those smaller subproblems to solve larger ones. It's last step is to fill out the "last" space in memory which is the overall solution.
   While they are defined slightly differently, both approaches will end up doing the same amount of work and filling out memory in the same manner, thus their time complexities will be the same (they might vary by constants, but wouldn't be asymptotically different).

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.