CS4102 - ALGORITHMS - SPRING 2022

DUE FRIDAY, APRIL 15, 2022 AT **11:30 pm** VIA GRADESCOPE

C1BASIC(MST3K)

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: 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: Surprisingly enough, this problem is more similar to the log-cutting problem than any other. First thing we do is sort the boxes by a single dimension (length, width, or height). Then we recursively call a function on each compatible box "behind" the selected box, adding 1 to the total number of boxes that the compatible box could hold. Finally, we can scan through the resulting array and return the largest value in the array.

```
mainMethod(boxes)
    boxes.order // all boxes are rotated so that their first dimension is their largest
    // length and last dimension is smallest length
    boxes.sort(key = 0) // sorting the boxes

    nests = array(boxes.length)
    nests.all = -infin
    dynamicProgramming(boxes, nests, boxes.length - 1)

    i = 0
    for j in nests
        if ( i > j )
            i = j

    return j

dynamicProgramming(boxes, nests, index)
    nests[0] = 0 // by definition, the first element can fit no boxes
    for i = 2 to boxes.length // loop from the beginning to the end
        best = 0
        for j = 1 to i
```

```
if ( boxes[j].w < boxes[i].w && boxes[j].h < boxes[i].h ) // only if the "prior"
    // checking is valid do we bother updating nests
    // note that we only have to compare width and height because
    // we're ordered by length from the initial sort
    best = max( best, nests[j] + 1 )
    nests[i] = max( best, nests[j] + 1 )
else
    continue
```

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 the same thing as the matrix multiplication problem from lecture with the only change being the distribution of the subtraction operator over a number of inputs. But the subtraction operator doesn't "do" anything because we only select the order of operations. So, just like the Matrix Chaining problem, we can just compute the results of the order of pairwise operations.

Note that ops is stored as a nums.length binary vector with a 1 wherever a "-" is located. Note that the first position is always 0 because the first number cannot have a "-" attached to it.

nums = [1, 2, 3, 4, 5, 6] ops = [-1, -1, 1, 1, 1, -1]

```
ArithmeticOrderOfOperations(ops, nums) {
    values = nums.length
    let sum[1..values, 1...values] and order[1...values - 1, 2...values] be new tables
    for i = 1 to values
        sum[i, i] = nums[i]
    for l = 2 to values
        for i = 1 to values - l + 1
            j = i + l - 1
                sum[i, j] = infin
                for k = i to j - 1
                    q = ops[i]*sum[i, k] + ops[k+1]*sum[k + 1, j]
                    if q > sum[i, j]
                        sum[i, j] = q
                        order[i, j] = k
```

```
        return sum and order
    }
```

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: The Optimal Substructrue is the idea that larger problems contain the optimal solution to smaller ones or that to solve larger problems, we need to optimally solve smaller problems first. The way this is used to solve the Seam Carving problem is that to compute the column containing the total least energy seam, we need to know the lowest cumulative seam values adjacent to the given column in the prior row.

   So, to evaluate the least energy seam for the current row, we need to know the least energy seams for each member of the prior row. For the prior row, we need to know the least energy seam for each member of the prior row and so on and so forth.

   Therefore, the Seam Carving algorithm uses an optimal substructure because each successive row requires knowing the optimal seams in the row prior.

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

   Solution: No, we do not need the optimal substructure for Divide and Conquer solutions. Consider the various DC solutions to sorting algorithms, specifically quicksort. Quicksort of all DC solutions is looking for the solution S(A). But, the array A consists of subarrays L and R. So, S(A) = S(L) + S(R). But these subarrays don't overlap, and don't demonstrate the optimal substructure for DC. In DP we can save a lot of computing power by using previously known solutions but this is not the case for DC problems where each subproblem needs to be manually computed (no memoization).

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: In this case, dynamic programming would not be a good design strategy. DP is a good strategy when there is much overlap between subproblems and memoization is strong. But if its subproblems do not overlap much (sorting, multiplying matrices [not order of multiplication]) then it is more efficient to use a DC approach.

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: The differences between top-down and bottom-up is that top-down is reminiscient of the traditional recursive substructure present in DC. But instead of just recalculating problems that we already have answers to, we simply store the answer when

we previously solved the problem and access it when needed. But all of these accesses come from recursive calls taking the larger, original problem in the original call.

Bottom-up is similar, but instead we continually solve smaller problems, building up to the larger problem until we find the answer for the larger problem.

I believe that both approaches have the same runtime because regardless of the approach taken, the same number of problems need to be solved.

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.