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:** Let us arbitrarily fix the ordering of our boxes and label them $b_1, b_2, ..., b_n$. Define the function $fits(b_i, b_j)$ that defines when box $b_i$ fits inside of $b_j$. We can then define our function $Best(b_i)$, the best number of boxes to nest inside of box $b_i$, as

$$Best(b_i) = \begin{cases} \max_j Best(b_j) + 1 & \text{if } fits(b_j, b_i) \\ 1 & \text{otherwise} \end{cases}.$$

Even if you were to sort boxes, the box $b_i$ that's "right below" may not have the best score of other boxes below $b_i$. So we have to find the max score of among all the boxes that could fit inside our current box, and for that reason some kind of ordering isn't necessary. If none of the other boxes fit in $b_i$, then its score is 1.

This top-down solution, using a memory, will then provide the best way to nest all boxes inside of any $b_i$. We don't know which of the boxes had the best score, so we can find the solution by finding

$$\max_i Best(b_i).$$

(It's not necessarily the box with the biggest volume, for example.)

## 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:** Let `max[i][j]` denote the maximum value of the expression starting from the $i^{\text{th}}$ value and ending at the $j^{\text{th}}$ value. Define `min[i][j]` accordingly for the minimum value. Then, the recurrence is

$$\texttt{max[i][j]} = \max_{i<k\leq j} \begin{cases} \texttt{max[i][k - 1]} + \texttt{max[k][j]} & \text{if } \texttt{ops[k - 1]} = \texttt{'+'} \\ \texttt{max[i][k - 1]} - \texttt{min[k][j]} & \text{if } \texttt{ops[k - 1]} = \texttt{'-'} \end{cases}$$

$$\texttt{min[i][j]} = \min_{i<k\leq j} \begin{cases} \texttt{min[i][k - 1]} + \texttt{min[k][j]} & \text{if } \texttt{ops[k - 1]} = \texttt{'+'} \\ \texttt{min[i][k - 1]} - \texttt{max[k][j]} & \text{if } \texttt{ops[k - 1]} = \texttt{'-'} \end{cases}$$

Finally, `max[i][i]` = `nums[i]` = `min[i][i]` for all $i = 1, \ldots, n$. We compute `max[i][j]` and `min[i][j]` along the "diagonal" (same as in matrix chain multiplication). The answer is `max[1][n]`.

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 computed best seam ending at pixel $(i, j)$. When looking at a pixel $p_{ij}$ to find the "best seam ending here" $S(i, j)$, we used optimal substructure to look at the best seams ending at the three pixels below this one ($S(i-1, j-1)$, $S(i-1, j)$, $S(i-1, j+1)$).

2. Do we need optimal substructure for Divide and Conquer solutions? Why or why not?
   **Solution:** No. For D&C, we could combine the solutions to the smaller subproblems in different ways. For example, with Mergesort, we used *merge*() to combine the subsolutions. In DP, we need to incorporate the subsolutions directly.
   In addition, optimal substructure is defined as a property of *optimization problems*, and the problems we've solved with divide and conquer were not this type of problem. We know that this property means that an optimal solution for an optimization problem can be constructed from optimal solutions of its subproblems, so one could argue that there's no concept of an optimal solution to, say, the sorting problem (because a sequence is sorted or not). But if we think of "optimal" as meaning "correct" then what's said earlier about D&C makes sense and is a true difference between how these approaches work and when they're appropriate.

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, DP's strength is that is addresses those two issues. Divide and Conquer is a better thing to try here, since your can recursively calculate each subproblem one time, and combine solutions to independent, non-overlapping subproblems.

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:** Top-down is usually the recursive solution, which looks at the larger problem and breaks it down into smaller components to solve. Bottom-up solves the smallest subproblems first, building up to the solution for the larger problem.

In terms of runtime, it's complicated. While both are valid solutions, the bottom-up approach will result in less recursive overhead and likely fewer accesses to the memory.

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.