problem 1 *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.

   Solution:

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.

   Solution:

   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.

problem 2 *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 $fits(b_i, b_j)$ function that determines if box b_i fits inside box b_j, 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:

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

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

This top-down solution, using a memory, will then provide the best way to nest all boxes inside of any b_i, so we can find the solution by asking

$$\max_i Best(b_i).$$

PROBLEM 3 *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:

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

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

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

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 4 *Stranger Things*

The town of Hawkins, Indiana is being overrun by interdimensional beings called Demogorgons. The Hawkins lab has developed a Demogorgon Defense Device (DDD) to help protect the town. The DDD continuously monitors the inter-dimensional ether to perfectly predict all future Demogorgon invasions.

The DDD allows Hawkins to predict that i days from now a_i Demogorgons will attack. The DDD has a laser gun that is able to eliminate Demogorgons, but the device takes a lot of time to charge. In general, charging the laser for j days will allow it to eliminate d_j Demogorgons.

Example: Suppose $(a_1, a_2, a_3, a_4) = (1, 10, 10, 1)$ and $(d_1, d_2, d_3, d_4) = (1, 2, 4, 8)$. The best solution is to fire the laser at times 3, 4 in order to eliminate 5 Demogorgons.

1. Construct an instance of the problem on which the following "greedy" algorithm returns the wrong answer:

   BADLASER$((a_1, a_2, a_3, \ldots, a_n), (d_1, d_2, d_3, , \ldots, d_n))$:

   Compute the smallest j such that $d_j \geq a_n$, Set $j = n$ if no such j exists

   Shoot the laser at time n

   if $n > j$ then BADLASER$((a_1, \ldots, a_{n-j}), (d1, \ldots, d_{n-j}))$

   Intuitively, the algorithm figures out how many days (j) are needed to kill all the Demogorgons in the last time slot. It shoots during that last time slot, and then accounts for the j days required to recharge for that last slot, and recursively considers the best solution for the smaller problem of size $n - j$.

   Solution:

   Solution: Consider the problem $(1, 10, 10, 2)$ and the same array $(1, 2, 4, 8)$ from above. The optimal solution is still $(3, 4)$, but BADLASER instead outputs times $(2, 4)$ which kills only $4 < 5$.

2. Given an array holding a_i and d_j, devise a dynamic programming algorithm that eliminates the maximum number of Demogorgons. Analyze the running time of your solution. *Hint: it is always optimal to fire during the last time slot.*

   Solution:

   Solution: It is always optimal to fire during the last time slot. The only question is how long to charge for the last shot. The optimal solution at time n maximizes the following:

   $$Best(n) = \max_{i=1}^{n}(\min(d_i, x_n) + Best(n - i))$$

   In other words, we must choose the number of seconds i that we wait to recharge the laser before we fire at time n. If we charge for i seconds, then the total score will be the last shot, which scores $\min(d_i, x_n)$ plus the score of the best solution that ends at time ni Note that it takes $\Theta(i)$ time to compute $Best(i)$. Therefore, this algorithm requires $\Theta(n^2)$ time.