HW 7 Solution

CS/ECE 374 B, Spring 2020 Version: **1.0**

Unless stated otherwise, for all questions in this homework involving dynamic programming, you need to provide a solution with explicit memoization.

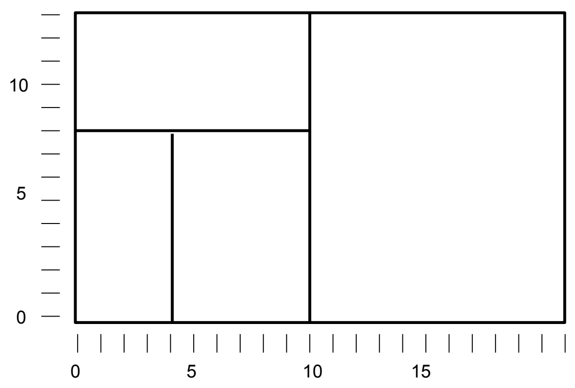
19 (100 PTS.) **Subdividing a rectangle slab.**

You have mined a large slab of marble from a quarry. For simplicity, suppose the marble slab is a rectangle measuring n inches in height and m inches in width. You want to cut the slab into smaller rectangles of various sizes - some for kitchen counter tops, some for large sculpture projects, others for memorial headstones. You have a marble saw that can make either horizontal or vertical cuts across any rectangular slab. At any time, you can query the spot price $P[x, y]$ of an x-inch by y-inch marble rectangle, for any positive integers x and y. These prices depend on customer demand, and people who buy marble counter tops are weird, so don't make any assumptions about them; in particular, larger rectangles may have significantly smaller spot prices. Given the array of spot prices and the integers m and n as input, design a dynamic programming algorithm to compute how to subdivide an $n \times m$ marble slab to maximize your profit.

Your solution must output *both* the maximum profit *as well* as the sequence of cuts necessary to obtain that profit. A sequence of cuts can be described as a sequence of tuples (V, x, y_{min}, y_{max}) for vertical cuts, (H, y, x_{min}, x_{max}) of horizontal cuts.

For example, the rectangle depicted below has been subdivided via the sequence of cuts

$$[(V, 10, 0, 13), (H, 8, 0, 10), (V, 4, 0, 8)]$$



Make your solution as asymptotically efficient as you can; we are not providing any particular target.

Solution:

Given a rectangle, we can make a horizontal cut at any point $1..h-1$, or a vertical cut at $1..w-1$. We could also make no cut, in which case the price could be read directly from the price function.

Recurrence relation:

$$\mathsf{OptCuts}(w,h) = \mathsf{min}\begin{cases} P[w,h] \\ \mathsf{max}_{x\in[1..w-1]}\mathsf{OptCuts}(x,h) + \mathsf{OptCuts}(w-x,h) \\ \mathsf{max}_{y\in[1..h-1]}\mathsf{OptCuts}(w,y) + \mathsf{OptCuts}(w,h-y) \end{cases}$$

Memoizing data structure: There are $h \times w$ distinct subproblems, corresponding to different (smaller) sizes of rectangles. To facilitate reconstructing the sequence of cuts, we'll store the location corresponding to the max in the table alongside the best price.

$\mathsf{OC}[]$ will be a $w \times h$ array that records the optimal decomposition, along with the location of the next cut.

The subproblem i, j depends on subproblems with smaller values of i or j. Any evaluation order from $(0,0)$ to (w,h) increasing will work. We will go row by row, from bottom to top, left to right.

Algorithm:

```
$OC[,] \leftarrow w \times h$ array
for $i = 1, ..., w$ do
for $j = 1, ..., h$ do
    $m \leftarrow P[w,h]$
    $s \leftarrow \perp$
    for $x = 1, ..., w-1$ do
        $(m_L, \_) \leftarrow OC[x,h]$
        $(m_R, \_) \leftarrow OC[w-x,h]$
        if $m_L + m_R > m$
            $m \leftarrow m_L + m_R$
            $s \leftarrow (V,x)$
    for $y = 1, ..., h-1$ do
        $(m_L, \_) \leftarrow OC[w,y]$
        $(m_R, \_) \leftarrow OC[w,h-y]$
        if $m_L + m_R > m$
            $m \leftarrow m_L + m_R$
            $s \leftarrow (H,y)$
    $OC[i,j] \leftarrow (m,s)$
return $OC[w,h]$
```

Analysis: Space is $O(hw)$. Time is $O(hw(h+w))$ based on the for loops in the algorithm.

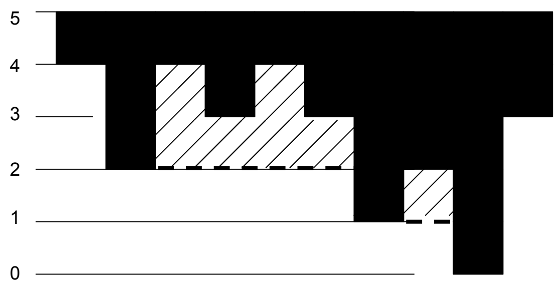
Extracting the solution: The local information at each table is (V,x) or (H,y), which tells us where else in the table we should look to read the next cut. However, to return the cuts in the form required, with absolute coordinates, it's necessary to keep track of "where" in the original slab we started from. The most convenient way to do this is with a recursive algorithm, where we keep track of the $(left, top, right, bottom)$ of the rectangle.

```
$cuts(l = 0, t = 0, r = w, b = h):$
$w' \leftarrow r - l$
$h' = b - t$
$\_, cut = OPT[w'][h']$
**if** $cut = \perp$: **do** return []
**if** $cut = (H, i)$
   $return[(H, t + i, l, r)] + cuts(l, t, r, t + i) + cuts(l, t + i, r, b)$
**if** $cut = (V, j)$
   $return[(V, l + j, t, b)] + cuts(l, t, l + j, b) + cuts(l + j, t, r, b)$
```

20 (100 PTS.) **How much air?**

An underground cavern near the town has flooded during a storm, trapping several unprepared hikers who were exploring at the time. An engineer at Daring Tech Solutions (DTS) comes up with a bold, though ultimately misguided plan: Without going into all the details, it involves breathing the air bubbles trapped in the rock formations making up the cavern's ceiling. To help work out if the plan is viable, DTS sends a robot to take measurements. Your goal is to estimate how much air could be trapped underneath the rocks.

The following figure illustrates the rock formation (shaded region) and the volume of air it can trap in a bubble (hatched region).



The robot is outfitted with a laser range finder, such that it can measure the height of the rocky ceiling (assume the laser passes through air and water exactly the same way, so the robot cannot measure the bubble directly).

Give a dynamic programming algorithm to compute the amount of air that can be trapped under the rocks.

To be clear about the model: Rocks and air are measured in integer units, corresponding to squares in the illustration. Range measurements are non-negative. A bubble square exists at height h if and only if there exists some rock at equal height (or lower) both to the left and to the right. You can assume the left and right boundaries are of infinite height (they're holes that go up to the surface, so they don't trap air).

As an example, the figure above corresponds to the input

$$H = [4, 2, 4, 3, 4, 3, 1, 2, 0, 3]$$

for which your algorithm should output 7. You should aim for an algorithm that takes $O(n)$ time and $O(n)$ space, where n is the length of the input array.

Solution:

Recurrence relation:

To understand the height of the bubble at each position, we need to know the minimum height to the right and the minimum height to the left. Each of these can be described as a dynamic programming problem, starting with the following recurrence relation:

$$LowestToRight(x) = \begin{cases} \infty & x \geq n-1 \\ \min(H[x+1], LowestToRight(x+1)) & otherwise \end{cases}$$

$$LowestToLeft(x) = \begin{cases} \infty & x \leq 0 \\ \min(H[x-1], LowestToLeft(x-1)) & otherwise \end{cases}$$

Then the final bubble height for each position can be read in $O(1)$ time from these tables

$$BubbleHeight(x) = \max(0, P[x] - \min(LowestToLeft(x), LowestToRight(x)))$$

It requires some care to define the boundaries here. As stated, $LowestToRight(x)$ defines the lowest height to the right, not including x. Other choices here are possible too.

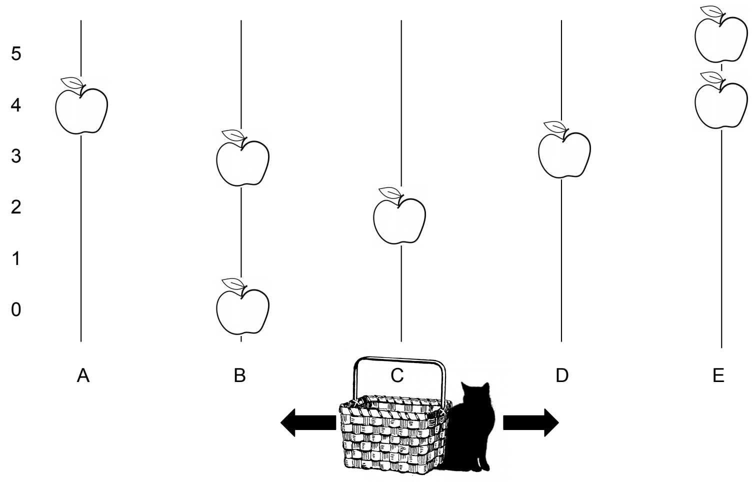
Memoizing data structure: To store *LowestToRight* and $LowestToLeft$ we will need two $O(n)$ data structures. The $LowestToRight$ be computed right to left, while $LowestToLeft$ is computed left to right.

Pseudocode: Pseudocode omitted from solution - straightforward given $LowestToLeft$ and $LowestToRight$.

Analysis: This requires $O(n)$ time and $O(n)$ space.

21 (100 PTS.) **Basket Game.**

The hottest video game this month is BasketCat. It's a game where a cat moves a basket to catch apples that fall from the sky. Honestly I don't know how a game like this can still become popular. Anyway, it looks like this:



A "stage" of BasketCat is encoded as five sequences A, B, C, D, E, each containing an ordered sequence describing when apples in that position reach the ground. The basket starts out at position C.

The cat has three moves: "Left", "Right", and "Stay" (as if "stay" is a command that works on cats...). In each game step:

1. First the cat's move is applied to the basket. (If the basket is at position A, then "Left" isn't a valid move, similar with E).
2. After making a move, if the basket is in the position of an apple, then the cat earns 1 point.
3. Finally, all of the apples move down one cell.

Use dynamic programming to design an algorithm for computing the optimal strategy for a given level of BasketCat. Your solution should run in $O(n^2)$ time (no restriction on space), where n is the length of the stage, the maximum value in any of A, B, C, D, E. The basket starts at position C.

For example, the level from the illustration corresponds to input:

$$A = [4], \quad B = [0, 3], \quad C = [2], \quad D = [3], \quad E = [4, 5]$$

for which your algorithm should output 5.

Solution:

It is helpful to translate the input to a better representation, a bit map indicating whether an apple is present at the given height. First reindex the columns with numbers,

$$L_0 = A, L_1 = B, L_2 = C, L_3 = D, L_4 = E.$$

Then the bitmap will be

$$S[i,j] = 1 \mathsf{if} j \in L_i$$

. This can be computed directly in $O(n)$ time, no dynamic programming needed.

Recurrence relation: The "state" of the game can be described by the progress j through the stage ($j \leq n$), and $p \in 0..4$ is the current position of the cat. This is one (verbose) way to write the recurrence relation, where the boundary checks are explicit.

$$OptScore(p,j) = \begin{cases} 0 & j > n \\ \max \begin{cases} S[i-1,j-1] + OptScore(p-1,j+1) \\ S[i-1,j] + OptScore(p,j+1) \\ S[i-1,j+1] + OptScore(p+1,j+1) \end{cases} & j < n \wedge i > 0 \wedge i < 4 \\ \max \begin{cases} S[i-1,j] + OptScore(p,j+1) \\ S[i-1,j+1] + OptScore(p+1,j+1) \end{cases} & j < n \wedge i = 0 \\ \max \begin{cases} S[i-1,j-1] + OptScore(p-1,j+1) \\ S[i-1,j] + OptScore(p,j+1) \end{cases} & j < n \wedge i = 4 \end{cases}$$

Memoizing data structure: The subproblems are defined by the position $p \in 1..5$ and the time since the start of the stage, $j \leq n$. The overall table will be $O(n)$ size. This can be computed in any order from n backward.

Pseudocode Omitted in solution

Analysis Computing *OptScore* takes $O(n)$ time and $O(n)$ space. The space could be compressed to $O(1)$, since only the previous row is accessed.