Ravi Dayabhai

* correct after re-attempt (noticing undefined detail)

Problem Session 8

Problem 8-1. Sunny Studies

Tim the Beaver needs to study for exams, but it's getting warmer, and Tim wants to spend more time outside. Tim enjoys being outside more when the weather is warmer: specifically, if the temperature outside is t integer units above zero, Tim's happiness will increase by t after spending the day outside (with a decrease in happiness when t is negative). On each of the n days until finals, Tim will either study or play outside (never both on the same day). In order to stay on top of coursework, Tim resolves never to play outside more than two days in a row. Given a weather forecast estimating temperature for the next n days, describe an $O(n)$-time dynamic programming algorithm to determine which days Tim should study in order to increase happiness the most.

* similar to solution, but should've been explicit of how to make line comparisons $O(1)$.

Problem 8-2. Diffing Data

Operating system Menix has a `diff` utility that can compare files. A **file** is an ordered sequence of strings, where the i^{th} string is called the i^{th} **line** of the file. A single **change** to a file is either:

- the insertion of a single new line into the file;
- the removal of a single line from the file; or
- swapping two adjacent lines in the file.

In Menix, swapping two lines is cheap, as they are already in the file, but inserting or deleting a line is expensive. A **diff** from a file A to a file B is any sequence of changes that, when applied in sequence to A will transform it into B, under the conditions that any line may be swapped at most once and any pair of swapped lines appear adjacent in A and adjacent in B. Given two files A and B, each containing exactly n lines, describe an $O(kn + n^2)$-time algorithm to return a diff from A to B that minimizes the number of changes that are **not swaps**, assuming that any line from either file is at most k ASCII characters long.

Problem 8-3. Building Blocks

Saggie Mimpson is a toddler who likes to build block towers. Each of her blocks is a 3D rectangular prism, where each block b_i has a positive integer width w_i, height h_i, and length ℓ_i, and she has at least three of each type of block. Each block may be **oriented** so that any opposite pair of its rectangular faces may serve as its **top** and **bottom** faces, and the **height** of the block in that orientation is the distance between those faces. Saggie wants to construct a tower by stacking her blocks as high as possible, but she can only stack an oriented block b_i on top of another oriented block b_j if the dimensions of the bottom of block b_i are strictly smaller[1] than the dimensions of the top of block b_j. Given the dimensions of each of her n blocks, describe an $O(n^2)$-time algorithm to determine the height of the tallest tower Saggie can build from her blocks.

Nice work seeing isomorphism, but you should probably have used a dynamic programming approach first :)

[1] If the bottom of block b_i has dimensions $p \times q$ and the top of block b_j has dimensions $s \times t$, then b_i can be stacked on b_j in this orientation if either: $p < s$ and $q < t$; or $p < t$ and $q < s$.

(i) This sounds a lot like the max-WIS / House Robber problem except elements in the n-length array can be ⊖. Didn't read carefully enough! Tim can go outside 2 days in a row!

We can determine which days Tim goes outside, and taking the complement gives Tim's study days inside.

Using "SRTBOT": (re-attempted after noticing the above):

- $\boxed{S}$ Let W_k be the maximum happiness up to day k (including day k)
- $\boxed{R}$ $W_k = \max\{ \underbrace{W_{k-1}}_{1}, \underbrace{W_{k-2} + A[k]}_{2}, \underbrace{W_{k-3} + A[k-1] + A[k]}_{3} \}$
- $\boxed{T}$ increasing k
- $\boxed{B}$ $W_0 = \max\{0, A[0]\}$, 0 if $k < 0$.
- $\boxed{O}$ W_{n-1}
- $\boxed{T}$ Exponential without memoization; with memo, $\Theta(n)$

To recover the actual indices themselves, we can traverse the memo table backwards and see which case computed W_k; add the element at k (if case 2) or the elements at $k-1, k$ (if case 3); and jump to W_{k-2} (if case 2), W_{k-3} (if case 3), or W_{k-1} (if case 1) and repeat until all elements have been recovered.

2 We can precompute the swappable lines between A & B and make the swaps to get A', B'. Then we can proceed with an edit distance-like dynamic program:

$\boxed{S}$ Let $x(i,j)$ be the number of inserts or deletions required to transform $A'[:i]$ to $B'[:j]$ (prefixes)

$\boxed{R}$ $$x(i,j) = \begin{cases} \text{if } A'[i-1] == B'[i-1]: & x(i-1, j-1) \\ \text{else}: & 1 + \min\{x(i-1,j),\ x(i, j-1)\} \end{cases}$$

$\boxed{T}$ increasing i, j

$\boxed{B}$ $x(i,0) = i$; $x(0,j) = j$; $x(0,0) = 0$

$\boxed{O}$ $x(\text{len}(A), \text{len}(B))$

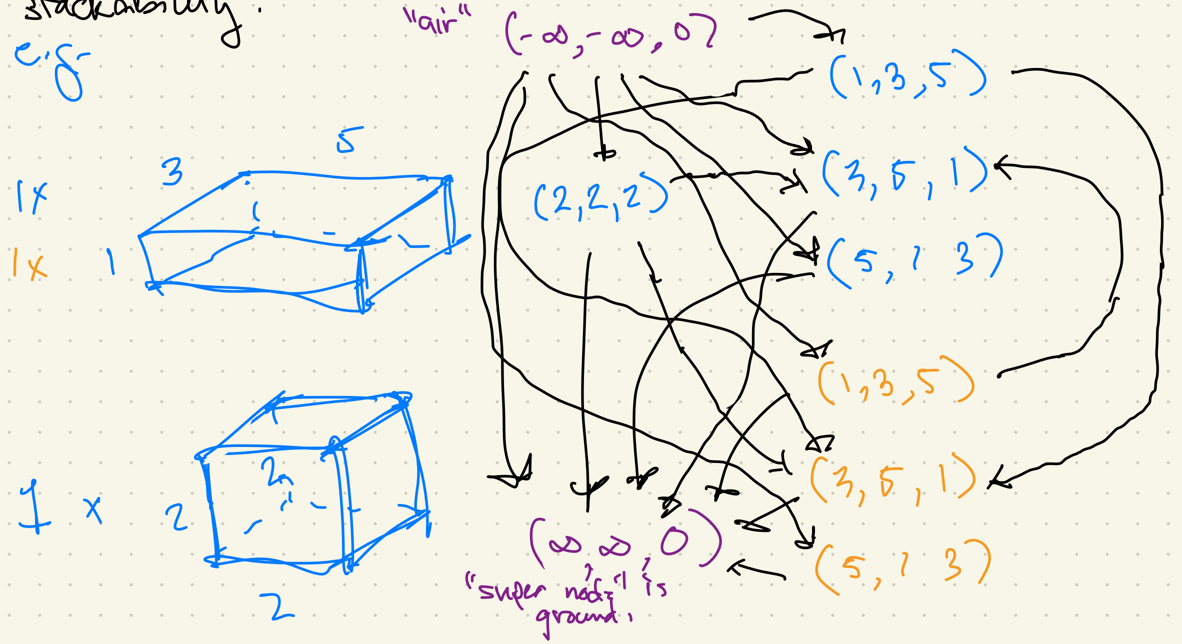
$\boxed{T}$ $\Theta(n^2)$ subproblems, $\Theta(1)$ work per subproblem $\Rightarrow \Theta(n^2)$; additional $O(nk)$ work for precomputing swappable lines.

Use a hash table mapping lines to numbers to enable $O(1)$ comparisons?

Recovering the diff requires storing parent pointers for deletions, insertions, no op + remembering which lines were swapped in precomputation.

③ Notice each block can be oriented 3 different ways. We can construct a DAG with each block orientation as nodes and directed edges representing "stackability":

e.g.



The weights of edges are the heights of the vertices wherein the edge is outgoing, so paths from any vertex to the "super node" (i.e., the ground) represent possible tower heights — we care about the longest such path.

Since we have a DAG, we can negate all edge weights and run DAG relaxation from the "air" node that is connected to each vertex by a zero-weight edge; this SSSP to the "ground" to the "air" negated weight is the tallest tower that can be constructed.

(③, CONT.)

Constructing the DAG on block orientations takes $\Theta(n^2)$ time, and running DAG relaxation takes $\Theta(n^2)$ time. Overall, this takes $\Theta(n^2)$ time.

④ First, we need to determine k. To do this we use the following algorithm:

$\boxed{S}$ Let $x(i,j)$ be the maximum # of mushrooms collected through some path to coordinate (i,j).

$\boxed{R}$ $x(i,j) :=$ if $A[i][j]$ is mushroom:

$$1 + \max\begin{Bmatrix} x(i-1,j) \\ x(i,j-1) \end{Bmatrix}$$

elif $A[i][j]$ is empty:

$$0 + \max\begin{Bmatrix} x(i-1,j) \\ x(i,j-1) \end{Bmatrix}$$

else: $-\infty$

This is because quick paths must be some permutation of (n-1) DOWN & (n-1) RIGHT movements, if a given permutation doesn't result in running into a tree (i.e. if such path exists).

$\boxed{T}$ increasing i, j

$\boxed{B}$ $x(0,0) = 0$

$\boxed{O}$ $x(\text{len}(A)-1, \text{len}(A)-1) = k$

$\boxed{T}$ $\Theta(n^2)$ time to construct memo table = $\Theta(n^2)$ subproblems $\times$ $\Theta(1)$ nonrecursive work per subproblem.

(4, CONT.)

with k and memo table x, we can construct another memo table y, solving another dynamic programming problem:

$\boxed{S}$ Let $y(i,j)$ be the number of distinct optimal quick paths to coordinate (i,j)

$\boxed{R}$ Let $t(i,j) := 1$ if $A[i][j]$ is mushroom else 0

$y(i,j) :=$ if $A[i][j]$ is tree: $t(i,j)$
elif $(i+1,j)$ and $(i,j+1)$ exist:
$y(i+1,j) + y(i,j+1)$
elif $(i+1,j)$ exists:
1 if $x(i+1,j) = x(i,j) + t(i,j)$ else 0
elif $(i,j+1)$ exists):
1 if $x(i,j+1) = x(i,j) + t(i,j)$ else 0

(SEE NEXT PAGE FOR EXAMPLE)

$\boxed{T}$ decreasing i, j

$\boxed{B}$ $y(\text{len}(A)-1, \text{len}(A)-1)$

$\boxed{O}$ $y(0,0)$

$\boxed{T}$ $\Theta(n^2)$ (similar reasoning as computing k)

Problem 8-4. Princess Plum

Princess Plum is a video game character collecting mushrooms in a digital haunted forest. The forest is an $n \times n$ square grid where each grid square contains either a tree, mushroom, or is empty. Princess Plum can move from one grid square to another if the two squares share an edge, but she cannot enter a grid square containing a tree. Princess Plum starts in the upper left grid square and wants to reach her home in the bottom right grid square[2]. The haunted forest is scary, so she wants to reach home via a **quick path**: a route from start to home that goes through at most $2n - 1$ grid squares (including start and home). If Princess Plum enters a square with a mushroom, she will pick it up. Let k be the maximum number of mushrooms she could pick up along any quick path, and let a quick path be **optimal** if she could pick up k mushrooms along that path.

(SEE PREV. PAGE)

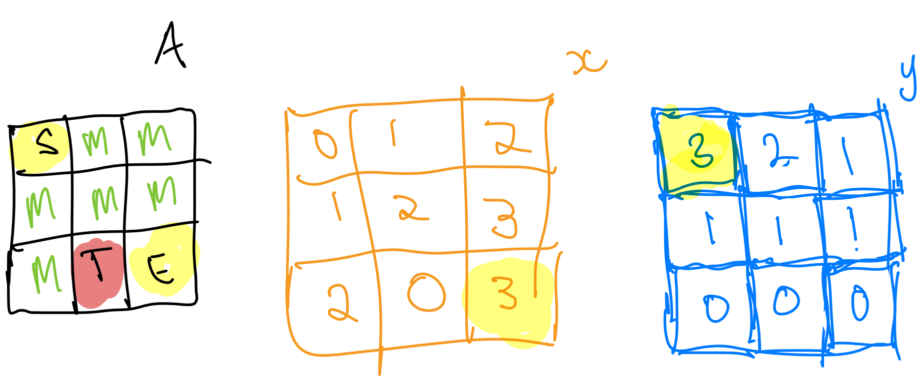
(a) [15 points] Given a map of the forest grid, describe an $O(n^2)$-time algorithm to return the number of distinct optimal quick paths through the forest, assuming that some quick path exists.

(b) [25 points] Write a Python function `count_paths(F)` that implements your algorithm from (a).

(SKIPPED FOR TIME)

```
def count_paths(F):
    '''
    Input:  F | size-n direct access array of size-n direct access arrays
              | each F[i][j] is either 't', 'm', or 'x'
              | for tree, mushroom, empty respectively
    Output: p | the number of distinct optimal paths in F
              | starting from (0,0) and ending at (n-1,n-1)
    '''
    p = 0
    ##################
    # YOUR CODE HERE #
    ##################
    return p
```

NICE.



[2] Assume that both the start and home grid squares are empty.