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.

Solution:

1. **Subproblems**
   - Let $t(i)$ be the temperature on day i
   - $x(i)$: the maximum happiness increase possible during days i to n
2. **Relate**
   - Guess whether studying or playing on day i will yield more happiness
   - If study, no change in happiness, but may study or play on the next day, $x(i+1)$
   - If play, change happiness by $t(i)$ and either
     - study on day $i+1$, $t(i) + x(i+2)$, or
     - play on day $i+1$ and study on $i+2$, $t(i) + t(i+1) + x(i+3)$
   - $x(i) = \max\{x(i+1), t(i) + x(i+2), t(i) + t(i+1) + x(i+3)\}$ for $i \in \{1, \ldots, n-1\}$
3. **Topo. Order**
   - Subproblem $x(i)$ only depends on subproblems with strictly larger i, so acyclic
4. **Base**
   - $x(n) = \max\{0, t(n)\}$, if only one day, either play or study
   - $x(n+1) = x(n+2) = 0$, if no more days, can't increase happiness
5. **Original**
   - $x(1)$ is the maximum happiness increase possible from day 1 to day n
   - Store parent pointers to reconstruct days studied
6. **Time**
   - # subproblems: $x(i)$ for $i \in \{1, \ldots, n+2\}$, $n+2 = O(n)$
   - work per subproblem: $O(1)$ (constant branching factor)
   - $O(n)$ running time

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.

Solution:

1. **Subproblems**
   - First, use a hash table to assign each unique line a number in $O(kn)$ time
   - Now each line can be compared to others in $O(1)$ time
   - $x(i, j)$: the minimum non-swap changes to transform $A[:i]$ into $B[:j]$
2. **Relate**
   - If $A[i] = B[j]$, then recurse on remainder
   - Otherwise maximize a last change applied:
     - $A[i]$ is deleted
     - an insertion matches with $B[j]$
     - the last two in $A[:i]$ are swapped to match the last two in $B[:j]$
   - if $A[i] = A[j]$, $x(i, j) = x(i - 1, j - 1)$
   - otherwise, $x(i, j) = \min \left\{ \begin{array}{ll} 1 + x(i-1, j) & \text{delete} \\ 1 + x(i, j-1) & \text{insert} \\ x(i-2, j-2) & \text{swap if } A[i] = B[j-1] \text{ and } A[i-1] = B[j] \end{array} \right\}$
3. **Topo. Order**
   - Subproblem $x(i, j)$ only depends on subproblems with strictly smaller $i + j$, so acyclic
4. **Base**
   - $x(0, 0) = 0$, all lines converted
   - $x(i, 0) = i$, must delete remainder
   - $x(0, j) = j$, must insert remainder
5. **Original**
   - $x(n, n, 0)$ by definition
   - Store parent pointers to reconstruct which changes were made (for cases where $A[i] \neq A[j]$, remember whether a deletion, insertion, or swap occurred)
6. **Time**
   - pre-processing: $O(kn)$
   - # subproblems: $(n + 1)^2 = O(n^2)$, $x(i, j)$ for $i, j \in \{0, 1, \ldots, n\}$
   - work per subproblem: $O(1)$

- $O(n^2)$ running time

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.

Solution:

1. **Subproblems**
   - Each block may be used in one of three vertical orientations
   - (without loss of generality, block bases can always be stacked with the base's shorter side pointed in one direction)
   - Because the stacking requirement requires base dimensions to strictly decrease, any optimal tower may use any block type at most three times (once in each orientation)
   - Sort dimensions of each block and remove duplicates (e.g., in $O(n)$ time with hash table)
   - For each block type with sorted dimensions $a \leq b \leq c$, construct three block orientations (a, b, c), (a, c, b), (b, c, a) (last dimension corresponding to height), and add them to an oriented block list in $O(n)$ time
   - (triplicating block types allowable since there are at least three of each type)
   - Sort lexicographically (first dimension most significant) in $O(n \log n)$ time
   - Re-number sorted list, where block i has oriented dimensions (w_i, ℓ_i, h_i) with $w_i \leq \ell_i$
   - Any stackable tower of oriented blocks must be a subsequence of this sorted list since the w_i are sorted, though not every subsequence of this list comprises a valid tower, since the ℓ_i are not necessarily sorted
   - $x(i)$: the maximum tower height of any tower that uses block i and any subset of the remaining blocks 1 through $i - 1$
2. **Relate**
   - Guess the next lower block in the tower, only from blocks with strictly smaller ℓ_i
   - $x(i) = h_i + \max\{0\} \cup \{x(j) \mid j \in \{1, \ldots, i-1\} \text{ s.t. } \ell_i < \ell_j\}$
3. **Topo. Order**
   - Subproblem $x(i)$ only depends on subproblems with strictly smaller i, so acyclic
4. **Base**
   - $x(1) = h_1$, since the maximum in the recurrence is trivially zero
5. **Original**

[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 $q < s$.

- Some block must be the top block, so compare all possible top blocks
- $\max\{x(i) \mid i \in \{1, \ldots, n\}\}$

6. **Time**
   - pre-processing: $O(n \log n)$
   - # subproblems: n, $x(i)$ for $i \in \{1, \ldots, n\}$
   - work per subproblem: $O(n)$
   - computing the solution: $O(n)$
   - $O(n^2)$ running time
   - Note that this problem can be solved in $O(n \log n)$ with a similar optimization as in Longest Increasing Subsequence from Recitation 15.

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.

(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.

Solution:

1. **Subproblems**
   - Let upper left grid square be $(1, 1)$ and bottom right grid square be $O(n, n)$
   - Let $F[i][j]$ denote the contents of grid square (i, j)
   - Define two types of subproblems: one for optimal mushrooms and one to count number of paths
   - $k(i, j)$: the maximum number of mushrooms to reach grid square (i, j) from grid square $(1, 1)$ on a path touching $i + j - 1$ squares
   - $x(i, j)$: the number of paths from $(1, 1)$ to (i, j) collecting $k(i, j)$ mushrooms and touching $i + j - 1$ squares
2. **Relate**
   - A path touching $i+j-1$ squares ending at (i, j) must extend a path touching $i+j-2$ squares to either its left or upper neighbor
   - If $F[i][j]$ contains a tree:
     - There are no paths to (i, j)
     - $k(i, j) = -\infty$ and $x(i, j) = 0$

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

- Otherwise:
  - Let $m(i,j)$ be 1 if $F[i][j]$ is a mushroom and 0 otherwise
  - $k(i,j) = m(i,j) + \max(k(i-1,j), k(i,j-1))$
  - $x(i,j) = \sum \left\{ \begin{array}{ll} 0 & \text{always} \\ x(i-1,j) & \text{if } k(i-1,j) + m(i,j) = k(i,j) \\ x(i,j-1) & \text{if } k(i,j-1) + m(i,j) = k(i,j) \end{array} \right\}$

3. **Topo. Order**
   - Subproblem $k(i,j)$ only depends on k subproblems with strictly smaller $i+j$, so acyclic
   - Subproblem $x(i,j)$ only depends on x subproblems with strictly smaller $i+j$ (and k subproblems which do not depend on x subproblems), so acyclic
4. **Base**
   - $k(1,1) = 0$, no mushrooms to start
   - $x(1,1) = 1$, one path at start
   - negative grid squares impossible
   - $k(0,i) = k(i,0) = -\infty$ for $i \in \{0,\ldots,n\}$
   - $x(0,i) = x(i,0) = 0$ for $i \in \{0,\ldots,n\}$
5. **Original**
   - $x(n,n)$ by definition
6. **Time**
   - # subproblems: $2(n+1)^2 = O(n^2)$, $k(i,j)$ and $x(i,j)$ for $i,j \in \{0,1,\ldots,n\}$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

(b) [25 points] Write a Python function `count_paths(F)` that implements your algorithm from (a). You can download a code template containing some test cases from the website. Submit your code online at `alg.mit.edu`.

Solution:

```
def count_paths(F):
    n = len(F)
    K = [[-float('inf')]*(n + 1) for _ in range(n + 1)]      # init K memo
    X = [[0]*(n + 1) for _ in range(n + 1)]                  # init X memo
    for i in range(1, n + 1):                 # bottom-up dynamic program
        for j in range(1, n + 1):
            if F[i - 1][j - 1] == 't':        # base case
                continue
            if i == 1 and j == 1:             # base case
                K[1][1], X[1][1] = 0, 1
                continue
            if F[i - 1][j - 1] == 'm':        m = 1
            else:                             m = 0
            K[i][j] = m + max(K[i - 1][j], K[i][j - 1])
            if K[i - 1][j] + m == K[i][j]:  X[i][j] += X[i - 1][j]
            if K[i][j - 1] + m == K[i][j]:  X[i][j] += X[i][j - 1]
    return X[n][n]
```