Recitation 17

Dynamic Programming Exercises

Longest Common Subsequence

Given two strings A and B, a **Longest Common Subsequence** is a longest (not necessarily contiguous) subsequence of A that is also a subsequence of B. For example, if $A =$ `akdfsjhlj`, and $B =$ `adfjlkhoqeipr`, a longest common subsequence contains 5 characters (e.g. `adfjl` or `adfjh`). Describe a dynamic program to compute the length of a longest common subsequence of two strings A and B.

Solution:

1. **Subproblems**
   - This is a maximization problem on the length of any subsequence
   - $\boxed{x(i,j) \text{ length of longest common subsequence of prefixes } A[:i] \text{ and } B[:j]}$
2. **Relate**
   - Either last characters match or they don't **(Guess!)**
   - If last characters match, some longest common subsequence will use them
   - (if no LCS uses last matched pair, using it will only improve solution)
   - (if an LCS uses last in $A[:i]$ and not last in $B[:j]$, matching $B[j-1]$ is also optimal)
   - If they do not match, they cannot both be in a longest common subsequence
   - $x(i,j) = \begin{cases} x(i-1,j-1)+1 & \text{if } A[i-1] = B[j-1] \\ \max\{x(i-1,j), x(i,j-1)\} & \text{otherwise} \end{cases}$
   - (draw subset of all rectangular grid dependencies)
   - Subproblems $x(i,j)$ only depend on strictly smaller $i+j$, so acyclic
3. **Base**
   - $x(i,0) = x(0,j) = 0$ (one string is empty)
4. **Solution**
   - Solve subproblems via recursive top down or iterative bottom up
   - Length of longest common subsequence of A and B is $x(|A|,|B|)$
   - Store parent pointers, add letter to subsequence if parent decreases both indices

5. Time

- # subproblems: $(|A|+1)(|B|+1)$
- work per subproblem: $O(1)$
- $O(|A||B|)$ running time

```
def lcs(A, B):
    a, b = len(A), len(B)
    x = [[0] * (a + 1) for _ in range(b + 1)]
    for i in range(1, a + 1):
        for j in range(1, b + 1):
            if A[i - 1] == B[j - 1]:
                x[i][j] = x[i - 1][j - 1] + 1
            else:
                if x[i][j - 1] < x[i - 1][j]:
                    x[i][j] = x[i - 1][j]
                else:
                    x[i][j] = x[i][j - 1]
    return x[a][b]
```

Exercise: Modify the code above to return a longest common subsequence.

Alternating Coin Game

A sequence of coins having different values are placed in a row, with coin i having value $v(i)$. You and a friend take turns removing a single coin from either end of the row. After all coins have been removed, the person possessing the largest coin value total wins. Assuming both players will play optimally, decide if you should play first or second.

Solution:

1. **Subproblems**
   - Choose subproblems that correspond to the state of the game
   - For every contiguous subsequence of coins from i to j
   - $\boxed{x(i,j)\text{: maximum value winnable for player moving when \# coins } j-i+1 \text{ is even}}$
2. **Relate**
   - Player must choose either coin i or coin j **(Guess!)**
   - Even (odd) player will seek to maximize (minimize) $x(i,j)$
   - $x(i,j) = \begin{cases} \max(v(i)+x(i+1,j), v(j)+x(i,j-1)) & j-i+1 \text{ even} \\ \min(x(i+1,j), x(i,j-1)) & j-i+1 \text{ odd} \end{cases}$
   - Subproblems $x(i,j)$ only depend on strictly smaller $j-i$, so acyclic
3. **Base**
   - $x(i,i) = 0$ (even player never gains value from last coin)
4. **Solution**
   - Solve subproblems via recursive top down or iterative bottom up
   - If $x(1,n) \geq$ half sum of all coin values, play on even turns, odd turns otherwise
   - (Can store parent pointers to reconstruct choices)
5. **Time**
   - \# subproblems: $O(n^2)$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

```
def play_first(v):
    # Return True iff you should play first, for sequential coin game values v
    n = len(v)
    x = [[0] * n for _ in range(n)]       # memo
    sum = 0
    for i in range(n - 1, -1, -1):        # dynamic program
        sum += v[i]
        for j in range(i + 1, n):
            if (j - i + 1) % 2 == 0:                # even number of coins
                if v[i] + x[i + 1][j] < v[j] + x[i][j - 1]:
                    x[i][j] = v[j] + x[i][j - 1]    # choose j
                else:
                    x[i][j] = v[i] + x[i + 1][j]    # choose i
            else:                                   # odd number of coins
                if x[i][j - 1] < x[i + 1][j]:
                    x[i][j] = x[i][j - 1]           # choose j
                else:
                    x[i][j] = x[i + 1][j]           # choose i
    if n % 2 == 0:                          # number of coins even
        return (sum / 2) <= x[0][n - 1]     # if first player gets more than half
    else:
        return x[0][n - 1] <= (sum / 2)     # if first player gets less than half
```

We've made a JavaScript visualizer for this dynamic program which you can find here:
`https://codepen.io/mit6006/pen/mQWxpb`

Exercise: Modify the code above to return a maximal set of coins chosen by the first player assuming both players play optimally.