Recitation 16

Dynamic Programming Exercises

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.

Edit Distance

A plagiarism detector needs to detect the similarity between two texts, string A and string B. One measure of similarity is called **edit distance**, the minimum number of **edits** that will transform string A into string B. An edit may be one of three operations: delete a character of A, replace a character of A with another letter, and insert a character between two characters of A. Describe a $O(|A||B|)$ time algorithm to compute the edit distance between A and B.

Solution:

1. **Subproblems**
   - Approach will be to modify A until its last character matches B
   - $\boxed{x(i,j)\text{: minimum number of edits to transform prefix up to } A(i) \text{ to prefix up to } B(j)}$
2. **Relate**
   - If $A(i) = B(j)$, then match!
   - Otherwise, need to edit to make last element of A equal to $B(j)$
   - Edit is either an insertion, replace, or deletion **(Guess!)**
   - Deletion removes $A(i)$
   - Insertion adds $B(j)$ to end of A, then removes it and $B(j)$
   - Replace changes $A(i)$ to $B(j)$ and removes both $A(i)$ and $B(j)$
   - $x(i,j) = \begin{cases} x(i-1,j-1) & \text{if } A(i) = B(i) \\ 1 + \min(x(i-1,j), x(i,j-1), x(i-1,j-1)) & \text{otherwise} \end{cases}$
   - Subproblems $x(i,j)$ only depend on strictly smaller i and j, so acyclic
3. **Base**
   - $x(i,0) = i$, $x(0,j) = j$ (need many insertions or deletions)
4. **Solution**
   - Solve subproblems via recursive top down or iterative bottom up
   - Solution to original problem is $x(|A|,|B|)$
   - (Can store parent pointers to reconstruct edits transforming A to B)
5. **Time**
   - # subproblems: $O(n^2)$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

```
def edit_distance(A, B):
    x = [[None] * len(A) for _ in range(len(B))]  # memo
    x[0][0] = 0                                   # base cases
    for i in range(1, len(A)):
        x[i][0] = x[i - 1][0] + 1                 # delete A[i]
    for j in range(1, len(B)):
        x[0][j] = x[0][j - 1] + 1                 # insert B[j] into A
    for i in range(1, len(A)):                    # dynamic program
        for j in range(1, len(B)):
            if A[i] == B[j]:
                x[i][j] = x[i - 1][j - 1]         # matched! no edit needed
            else:                                 # edit needed!
                ed_del = 1 + x[i - 1][j]          # delete A[i]
                ed_ins = 1 + x[i][j - 1]          # insert B[j] after A[i]
                ed_rep = 1 + x[i - 1][j - 1]      # replace A[i] with B[j]
                x[i][j] = min(ed_del, ed_ins, ed_rep)
    return x[len(A) - 1][len(B) - 1]
```

Exercise: Modify the code above to return a minimal sequence of edits to transform string A into string B. (Note, the base cases in the above code are computed individually to make reconstructing a solution easier.)