Recitation 16

Dynamic Programming Exercises

Max Subarray Sum

Given an array A of n integers, what is the largest sum of any **nonempty** subarray?
(in this class, **subarray** always means a contiguous sequence of elements)
Example: `A = [-9, 1, -5, 4, 3, -6, 7, 8, -2]`, largest `subsum` is 16.

Solution: We could brute force in $O(n^3)$ by computing the sum of each of the $O(n^2)$ subarrays in $O(n)$ time. We can get a faster algorithm by noticing that the subarray with maximum sum must end somewhere. Finding the maximum subarray ending at a particular location k can be computed in $O(n)$ time by scanning to the left from k, keeping track of a rolling sum, and remembering the maximum along the way; since there are n ending locations, this algorithm runs in $O(n^2)$ time. We can do even faster by recognizing that each successive scan to the left is redoing work that has already been done in earlier scans. Let's use dynamic programming to reuse this work!

1. **Subproblems**
   - $\boxed{x(k)\text{: the max subarray sum ending at } A[k]}$
   - Prefix subproblem, but with condition, like Longest Increase Subsequence
   - (Exercise: reformulate in terms of suffixes instead of prefixes)
2. **Relate**
   - Maximizing subarray ending at k either uses item $k-1$ or it doesn't
   - If it doesn't, then subarray is just $A[k]$
   - Otherwise, $k-1$ is used, and we should include the maximum subarray ending at $k-1$
   - $x(k) = \max\{A[k], A[k] + x(k-1)\}$
3. **Topo. Order**
   - Subproblems $x(k)$ only depend on strictly smaller k, so acyclic
4. **Base**
   - $x(0) = A[0]$ (since subarray must be nonempty)
5. **Original**
   - Solve subproblems via recursive top down or iterative bottom up

- Solution to original is max of all subproblems, i.e., $\max\{x(k) \mid k \in \{0, \ldots, n-1\}\}$
- Subproblems are used twice: when computing the next larger, and in the final max

6. **Time**
   - # subproblems: $O(n)$
   - work per subproblem: $O(1)$
   - time to solve original problem: $O(n)$
   - $O(n)$ time in total

```python
# bottom up implementation
def max_subarray_sum(A):
    x = [None for _ in A]                   # memo
    x[0] = A[0]                             # base case
    for k in range(1, len(A)):              # iteration
        x[k] = max(A[k], A[k] + x[k - 1])   # relation
    return max(x)                           # original
```

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
   - $\boxed{x(i, j)\text{: minimum number of edits to transform prefix up to } A(i) \text{ to prefix up to } B(j)}$
   - (Exercise: reformulate in terms of suffixes instead of prefixes)
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 $B(j)$
- $x(i,j) = \begin{cases} x(i-1, j-1) & \text{if } A(i) = B(i) \\ 1 + \min(x(i-1,j), x(i,j-1), x(i-1,j-1)) & \text{otherwise} \end{cases}$

3. **Topo. Order**
   - Subproblems $x(i,j)$ only depend on strictly smaller i and j, so acyclic

4. **Base**
   - $x(i,0) = i$, $x(0,j) = j$ (need many insertions or deletions)

5. **Original**
   - 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)

6. **Time**
   - # subproblems: $O(n^2)$
   - work per subproblem: $O(1)$
   - $O(n^2)$ running time

```python
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.)