Longest Convex Subsequence

Call a sequence $x_1 \dots x_m$ of numbers convex if $2x_i < x_{i-1} + x_{i+1}$ for all i. Describe an efficient algorithm to compute the length of the longest convex subsequence of an arbitrary array $A[1 \dots n]$ of integers.

In the following *preprev* may get set to the dummy value -1, which removes the convexity requirement.

```
1: procedure LXS(preprev, prev, cur)
2:     if cur > n then
3:         return 0
4:     ignore = LXS(preprev, prev, cur + 1)
5:     best = ignore
6:     if (preprev = −1) ∨ (A[cur] > 2A[prev] − A[preprev]) then
7:         include = 1 + LXS(prev, cur, cur + 1)
8:         if include > ignore then
9:             best = include
10:    return best
```

$LXS(preprev, prev, cur)$ computes the length of the longest convex subsequence in the array $A[cur \dots n]$, but with the requirement that the sequence is still convex after appending $A[preprev]A[prev]$ to the front (unless $preprev = -1$, in which case there is no additional requirement involving $A[preprev]A[prev]$). Thus the longest convex subsequence of A is given by $LXS(-1, -1, 1)$. Note that the optimal such subsequence either "includes" or "ignores" the element $A[cur]$ and the algorithm just returns the best of these two options (where including is only allowed if convexity is satisfied).

Each parameter in the above algorithm ranges over $O(n)$ values, and hence the DP table has size $O(n^3)$. Each entry takes $O(1)$ to compute so the total run time is $O(n^3)$. Every subproblem has a strictly larger last parameter hence the table can be filled with a decreasing outermost for loop for this parameter, and two nested inner for loops with any order for the other parameters.

Shortest Common Supersequence

Let $A[1 \ldots m]$ and $B[1 \ldots n]$ be two arbitrary arrays. A common supersequence of A and B is another sequence that contains both A and B as subsequences. Describe an efficient algorithm to compute the length of the shortest common supersequence of A and B.

```
1: procedure SCS(curA, curB)
2:     if curA > m then
3:         return n − curB + 1
4:     if curB > n then
5:         return m − curA + 1
6:     different = min{1 + SCS(curA + 1, curB), 1 + SCS(curA, curB + 1)}
7:     best = different
8:     if A[curA] = B[curB] then
9:         match = 1 + SCS(curA + 1, curB + 1)
10:        if match < different then
11:            best = match
12:    return best
```

Assuming we argued you can always take a match, this simplifies to:

```
1: procedure SCS(curA, curB)
2:     if curA > m then
3:         return n − curB + 1
4:     if curB > n then
5:         return m − curA + 1
6:     if A[curA] = B[curB] then
7:         return 1 + SCS(curA + 1, curB + 1)
8:     return min{1 + SCS(curA + 1, curB), 1 + SCS(curA, curB + 1)}
```

$SCS(curA, curB)$ is the length of the shortest common supersequence of $A[curA \ldots m]$ and $B[curB \ldots n]$, hence the shortest common supersequence of $A[1 \ldots m]$ and $B[1 \ldots n]$ is $SCS(1, 1)$. Specifically the next element in the shortest common supersequence is either $A[curA]$ or $B[curB]$, and the algorithm tries both options (where if $A[curA] = B[curB]$ then the supersequence only needs one character to cover both $A[curA]$ and $B[curB]$).

Each of the two parameters range over $O(n)$ values so use a DP table of size $O(n^2)$. Each entry takes $O(1)$ time to compute, so the total time is $O(n^2)$. At least one of the two parameters is larger (and the other not smaller), hence we can fill in the table with two nested decreasing for loops going over the two parameters.