Assignment 2, COMP251, Winter 2019. Jan 22, 2019. Solutions.

Exercise 1. DYNAMIC PROGRAMMING. Let $N[n,k]$ be the number of ways of cutting an n-inch rod into pieces of integer-valued lengths, where the lengths are nonincreasing from left to right, and do not exceed k. Thus, $N[n,n]$ is the number of ways of cutting that rod into pieces of decreasing lengths.

(i) Derive a simple recursion for $N[n,k]$ in terms of some values of the N matrix with smaller arguments.

(ii) Use that recursion to set up an $O(n^2)$ dynamic programming solution for $N[n,n]$.

SOLUTION. Note that $N[n,0] = 0$ for $n > 0$, and that $N[0,k] = 1$ for $k \geq 0$. For $k > 0$, $n \geq 0$, the recursion is

$$N[n,k] = \begin{cases} N[n-k,k] + N[n,k-1] & \text{if } n \geq k, \\ N[n,k-1] & \text{if } n < k. \end{cases}$$

The dynamic program is

```
for i = 0 to n do
for j = 0 to k do
  case
      j = 0, i > 0:   N[i,0] = 0
      i = 0:   N[0,j] = 1
      ij > 0:  if i < j then N[i,j] = N[i,j-1]
               else N[i,j] = N[i-j,j] + N[i,j-1]
```

Exercise 2. DYNAMIC PROGRAMMING. We have at our disposal an unlimited number of banknotes (currency notes) that are issued in $a_1 < a_2 < \cdots < a_n$ denominations, where $1 \leq a_1$. (For example, in Canada, $a_1 = 5, a_2 = 10, a_3 = 20, a_4 = 50, a_5 = 100$.) Give a dynamic programming algorithm for computing the number of different ways in which one can pay a bill of k dollars. Show that the RAM-model complexity of your algorithm is not more than a constant times $n \max(k,n)$ given that we know that $a_{j+1} \geq 2a_j$ for all j (as in the Canadian example).

SOLUTION. Let $N[i,j]$ be the number of ways to pay a bill of j dollars using only banknotes of denomination $a_1, \ldots, a_i$. We will compute $N[i,j]$ for all $0 \leq i \leq n, 0 \leq j \leq k$. We make a few key observations:

(i) $N[i,0] = 1$ for all $i \geq 0$, and $N[0,j] = 0$ for all $j > 0$.

(ii) To pay a bill of j dollars using notes up to a_i, we have a choice to include a_i up to ℓ times, where $\ell = \lfloor j/a_i \rfloor$.

Putting this together, we have the following algorithm:

```
for i = 0 to n do
for j = 0 to k do
  case
      j = 0:   N[i, 0] = 1
      i = 0, j > 0:   N[0, j] = 0
      ij > 0:   ℓ ← ⌊j/a_i⌋
                N[i, j] = Σ_{r=0}^{ℓ} N[i − 1, j − r a_i]
```

To study the complexity, note that every line costs one time unit in the RAM model, except the last one, which costs $1 + \lfloor j/a_i \rfloor$. Summing all yields an upper bound on the time of

$$n + k + 2 + \sum_{i=1}^{n} \sum_{j=1}^{k} (1 + \lfloor j/a_i \rfloor) .$$

This is not more than

$$O(nk) + \sum_{i=1}^{n} \sum_{j=1}^{k} \frac{j}{a_i}.$$

But because $a_i \geq 2^i$, the latter sum is not more than

$$O(nk) + \sum_{j=1}^{k} j \sum_{i=1}^{n} \frac{1}{2^i} < O(nk) + k(k+1) = O(nk + k^2).$$

Exercise 3. DYNAMIC PROGRAMMING. Write a dynamic programming solution for computing the length of the longest increasing subsequence in $(x_1, \ldots, x_n)$, where we assume that the x_i's form a permutation of $(1, 2, \ldots, n)$. Also output one of the longest increasing subsequences.

SOLUTION. We introduce $L[i, j]$, which is the length of the longest sequence among $(x_1, \ldots, x_i)$, if we only consider x_r's that are at most j. The value $L[n, n]$ is what we need, so we will compute the entire $n \times n$ matrix. Formally, we could allow i or j to be zero. In either case, the value of L is zero if this happens. For $i, j > 0$, if $x_i = j$, then we should include x_i in the longest increasing subsequence of length $L[i, j]$. The algorithm:

```
for i = 0 to n do
for j = 0 to n do
  case
      j = 0 or i = 0:   L[i,j] = 0
      ij > 0:   if x_i = j then L[i,j] = 1 + L[i-1,j-1]
                           else L[i,j] = max(L[i-1,j], L[i,j-1])
```

Having stored the L matrix, we can output a longest increasing subsequence by tracing back what we did, as follows:

```
set i = j = n
while i > 0 do:
      if x_i = j then output (i,j), i ← i-1, j ← j-1
                 else if L[i-1,j] ≥ L[i,j-1] then i ← i-1
                                             else j ← j-1
```

Exercise 4. DYNAMIC PROGRAMMING: MINIMUM EDIT DISTANCE BETWEEN STRINGS. This question involves a very practical question in "stringology": computing the minimum edit distance between two strings of symbols. Given are two strings $A = a(1) \ldots a(n)$ and $B = b(1) \ldots b(m)$. We would like to know how close A is to B, where closeness is measured by the minimum edit distance between A and B, that is, the minimum number of basic edit steps to change A into B. These edit steps are (1) `insert` a character into the string; (2) `delete` a character from the string; and (3) `replace` one character by another. As an example, consider the strings $A =$ `apple` and $B =$ `pear`. By means of the above edit steps, we can transform A into B as follows: `apple`, `pple`, `pele`, `peae`, `pear`. We needed four operations, so the minimum edit distance is ≤ 4.

There exists a dynamic programming solution to this problem that uses $O(nm)$ time. If we write $A[n]$ and $B[m]$ and let $A[i]$ and $B[j]$ denote the prefixes of $A[n]$ and $B[m]$ of lengths i and j respectively

(so that $A[i] = a(1) \dots a(i)$), then it is possible to write the minimum edit distance between $A[n]$ and $B[m]$ (which we denote by $M[n, m]$) as a function of $M[n-1, m]$, $M[n, m-1]$ and $M[n-1, m-1]$ and possibly the outcome of the comparison $(a(n) = b(m))$. This should allow you to construct the $n \times m$ matrix of minimum edit distances $M[i, j]$, with $1 \leq i \leq n$, $1 \leq j \leq m$. Work out the details of the dynamic programming solution.

SOLUTION. Consider all the possibilities when trying to change $A[n]$ into $B[m]$, from the right to the left. Without loss of generality, we assume that B never changes. For fixed n, m, we could have $a(n) = b(m)$. In that case, we obviously have $M[n, m] = M[n-1, m-1]$, as we can remove that end-word match from both $A[n]$ and $B[m]$. Otherwise, we could do three things: either change $a(n)$ into $b(m)$ (so, $M[n, m] \leq M[n-1, m-1] + 1$), or add $b(m)$ to $A[n]$ as a suffix (so, $M[n, m] \leq M[n, m-1] + 1$), or delete $a(n)$ (so, $M[n, m] \leq M[n-1, m] + 1$). Clearly, the best of these three operations should be used. We summarize:

```
for i = 0 to n do
for j = 0 to m do
  case
      j = 0 or i = 0:   M[i,j] = i + j
      ij > 0:   if a(i) = b(j) then M[i,j] = M[i-1,j-1]
                               else M[i,j] = 1 + min(M[i-1,j], M[i,j-1], M[i-1,j-1])
```