Esercizi su Dynamic Programming

Exercise 1.1 Write an algorithm to find the maximum value that can be obtained with a full parenthesization of the expression

$$x_1/x_2/x_3/\ldots x_{n-1}/x_n,$$

where $x_1, x_2, \ldots, x_n$ are positive rational numbers and "/" denotes division.

Answer: For $1 \leq i \leq j \leq n$, denote by $X_{i\ldots j}$ the subexpression $x_i/x_{i+1}/\ldots/x_j$. Given a full parenthesization $\mathcal{P}_{1\ldots n}$ of $X_{1\ldots n}$, let its cost $c(\mathcal{P}_{1\ldots n})$ be the value obtained by performing the division according to the order dictated by the parentheses. An optimal parenthesization of $X_{1\ldots n}$ is one that maximizes the above cost function.

Any full parenthesization $\mathcal{P}_{1\ldots n}$ of $X_{1\ldots n}$ contains, at the outer level, parenthesizations $\mathcal{P}_{1\ldots k}$ and $\mathcal{P}_{k+1\ldots n}$ of the subsequences $X_{1\ldots k}$ and $X_{k+1\ldots n}$ for a given value k, $1 \leq k \leq n-1$. Moreover, this property holds recursively for $\mathcal{P}_{1\ldots k}$ and $\mathcal{P}_{k+1\ldots n}$. The relation among the costs of the above parenthesizations is the following:

$$c(\mathcal{P}_{1\ldots n}) = \frac{c(\mathcal{P}_{1\ldots k})}{c(\mathcal{P}_{k+1\ldots n})}.$$

The key observation upon which we will base our algorithm is that any maximizing (resp., minimizing) parenthesization $\overline{\mathcal{P}}_{1\ldots n}$ must be formed by a parenthesization $\overline{\mathcal{P}}_{1\ldots k}$ that *maximizes* (resp., *minimizes*) c for the string $X_{1\ldots k}$, and a parenthesization $\overline{\mathcal{P}}_{k+1\ldots n}$ that *minimizes* (resp., *maximizes*) c for $X_{k+1\ldots n}$, for some value k, $1 \leq k \leq n-1$. Indeed, if it were not so, a better $\overline{\mathcal{P}}_{1\ldots k}$ or $\overline{\mathcal{P}}_{k+1\ldots n}$ would immediately yield a better $\overline{\mathcal{P}}_{1\ldots n}$.

Let $M[i,j]$ denote the cost of a maximizing parenthesization of $X_{i\ldots j}$, $1 \leq i \leq j \leq n$, and let $m[i,j]$ denote the cost of a minimizing parenthesization of $X_{i\ldots j}$. Based on the above observations, we can write the following recurrence for $m[i,j]$ and $M[i,j]$:

$$M[i,j] \;=\; m[i,j] = x_i \qquad \text{if } i = j$$

$$M[i,j] = \max\left\{\frac{M[i,k]}{m[k+1,j]} : i \le k < j\right\} \text{ if } i < j$$
$$m[i,j] = \min\left\{\frac{m[i,k]}{M[k+1,j]} : i \le k < j\right\} \text{ if } i < j$$

The algorithm follows immediately from the above recurrence.

```
CHAIN_DIVISION(x_1, x_2, ..., x_n)
for i ← 1 to n do
    M[i,i] ← m[i,i] ← x_i
for ℓ ← 2 to n do {compute the values of M and m for substrings of length ℓ}
    for i ← 1 to n − ℓ + 1 do
        j ← i + l − 1
        M[i,j] ← 0
        m[i,j] ← ∞
        for k ← i to j − 1 do
            t_1 ← M[i,k]/m[k+1,j]
            t_2 ← m[i,k]/M[k+1,j]
            {M[i,k], m[i,k], M[k+1,j] and m[k+1,j] already
            available at this point}
            if M[i,j] < t_1 then M[i,j] ← t_1
            if m[i,j] > t_2 then m[i,j] ← t_2
return M[1,n]
```

The above algorithm computes the cost of an optimal parenthesization in $O(n^3)$ time. If we are interested in actually determining the structure of the parenthesization, it is sufficient to compute two additional tables, $s_M[1\ldots n, 1\ldots n]$ and $s_m[1\ldots n, 1\ldots n]$, with $s_M[i,j]$ (resp., $s_m[i,j]$) recording at which index k the maximizing (resp., minimizing) parenthesization of $X_{i\ldots j}$ is split into optimal parenthesizations for $X_{i\ldots k}$ and $X_{k+1\ldots j}$. Note that s_M and s_m can be computed without increasing the running time of the algorithm. □

Exercise 1.2 Give an algorithm that uses the table of additional information $S[\cdot,\cdot]$ (computed by the Matrix-Chain Multiplication dynamic programming algorithm seen in class) to print the optimal parenthesization for the matrix chain.

Answer: Let s be the array computed by MATRIX_CHAIN_ORDER. Recall that $s[i,j]$ stores the splitting index k of the optimal parenthesization of the subchain $A_{i..j}$ of matrices $A_i, A_{i+1}, \ldots, A_j$, with $1 \le i \le k < j \le n$. We can write the following recursive algorithm.

```
PRINT_OPTIMAL_PARENS(i, j)
if i = j
  then print('A_i')
       return
print('(')
k ← s[i, j]
PRINT_OPTIMAL_PARENS(i, k)
PRINT_OPTIMAL_PARENS(k + 1, j)
print(')')
return
```

Let us charge one time unit for any **print** statement, and let $T(n)$ be the running time of PRINT_OPTIMAL_PARENS$(1, n)$. When $n = 1$, the above procedure simply prints A_1. When $n > 1$, the number of **print** statements is the number of **print** statements performed by the two recursive calls plus 2. The size of the subinstances is $s[1, n]$ and $n - s[1, n]$, respectively. We obtain the following recurrence

$$\begin{cases} T(n) = T(s[1,n]) + T(n - s[1,n]) + 2, & n > 1, \\ T(1) = 1. \end{cases}$$

Let us prove by induction that $T(n) = 3n - 2$ (which is exactly the number of symbols of a full parenthesization of $A_1 A_2 \dots A_n$). The base case trivially holds. Assuming that $T(k) = 3k - 2$, for $1 \leq k < n$, we obtain

$$\begin{aligned} T(n) &= T(s[1,n]) + T(n - s[1,n]) + 2 \\ &= 3s[1,n] - 2 + 3(n - s[1,n]) - 2 + 2 \\ &= 3n - 2, \end{aligned}$$

and the inductive thesis follows. Therefore, PRINT_OPTIMAL_PARENS runs in linear time. □

Exercise 1.3 Given the string $A = \langle a_1, a_2, \dots, a_n \rangle$, we say that $A_{i..j} = \langle a_i, a_{i+1}, \dots, a_j \rangle$ is a *palindrome substring* of A if $a_{i+h} = a_{j-h}$, for $0 \leq h \leq j - i$. (Intuitively, a palindrome substring is one which is identical to its "mirror" image. For example, if $A = accaba$, then both $A_{1..4} = acca$ and $A_{4..6} = aba$ are palindrome substrings of A.)

(a) Design a dynamic programming algorithm that determines the length of a longest palindrome substring of a string A in $O(n^2)$ time and $O(n^2)$ space.

(b) Modify your algorithm so that it uses only $O(n)$ space, while the running time remains unaffected.

Answer:

(a) It is worth noting that there are no more than $O(n^2)$ *substrings* in a string of length n (while there are exactly 2^n *subsequences*). Therefore, we could scan each substring, check for palindromicity and update the length of the longest palindrome substring discovered so far. Since the palindromicity test takes time linear in the length of the substring, this simple idea yields a $\Theta(n^3)$ algorithm. However, we can use dynamic programming to devise a much better algorithm. For $1 \leq i \leq j \leq n$, define

$$P[i,j] = \begin{cases} \textbf{true} & \text{if } A_{i..j} \text{ is a palindrome substring,} \\ \textbf{false} & \text{otherwise.} \end{cases}$$

Clearly, $P[i,i] = \textbf{true}$, while $P[i,i+1] \Leftrightarrow a_i = a_{i+1}$, for $1 \leq i \leq n-1$. It is also immediate to see that for $j - i + 1 \geq 3$ (i.e., for strings of length at least 3), we have

$$P[i,j] \Leftrightarrow (P[i+1, j-1] \textbf{ and } a_i = a_j)\,. \tag{1.1}$$

Note that in order to obtain a well defined recurrence, we need to explicitly initialize *two* distinct diagonals of the boolean array $P[i,j]$, since the recurrence for entry $[i,j]$ uses the value $[i-1, j-1]$, which is two diagonals away from $[i,j]$ (in other words, for a substring of length ℓ, we need to know the status of a substring of length $\ell - 2$).

The following algorithm is immediately obtained from the above considerations.

```
LONGEST_PALINDROME_SUBSTRING(A)
n ← length(A)
max ← 1
for i ← 1 to n − 1 do
    P[i, i] ← true
    { note that P[n, n] will be never used below }
    if a_i = a_{i+1}
      then P[i, i + 1] ← true;
           max ← 2
      else P[i, i + 1] ← false
```

(the algorithm continues on next page ...)

```
for ℓ ← 3 to n do
    { check the substrings of length ℓ }
        for i ← 1 to n − ℓ + 1 do
            j ← i + ℓ − 1
            if (P[i + 1, j − 1] and a_i = a_j)
               { P[i + 1, j − 1] already available at this point }
               then P[i, j] ← true
                        max ← ℓ
               else P[i, j] ← false
return max
```

Since the algorithm performs a constant number of operations for each of the $\Theta(n^2)$ substrings of A, it takes $O(n^2)$ time, while the space needed to store the table $P[i, j]$ is clearly $O(n^2)$.

(b) Note that by the ℓ-th iteration of the outer **for** loop, we only need values $P[i, j]$ with $j - i + 1 = \ell - 2$ (needed for iteration ℓ), $\ell - 1$ (needed for iteration $\ell + 1$), or ℓ, (the ones that we are computing). These are the values of P on diagonals $\ell - 2$ and $\ell - 1$ and ℓ. Therefore, at any time in the algorithm, it is sufficent to store no more than $3n$ entries of P. The algorithm above can be easily modified as follows.

```
LINEAR_SPACE_LPS(A)
n ← length(A)
max ← 1
for i ← 1 to n − 1 do
    P[i, 1] ← true
    { P is an array with only 3 columns }
    if a_i = a_{i+1}
       then P[i, 2] ← true
                max ← 2
       else P[i, 2] ← false
```

```
for ℓ ← 3 to n do
    { check the substrings of length ℓ }
        for i ← 1 to n − ℓ + 1 do
            if (P[i + 1, 1] and a_i = a_{i+ℓ−1})
              then P[i, 3] ← true
                   max ← ℓ
              else P[i, 3] ← false
            P[i, 1] ← P[i, 2]
            P[i, 2] ← P[i, 3]
            { shift relevant entries one column left }
return max
```

We can further improve the above algorithm so that it uses only two column vectors. In fact, after we check for palindromicity of the substring of length ℓ starting at i, we could first save $P[i, 2]$ into $P[i, 1]$ (which is not needed anymore) and then store the newly computed value directly in $P[i, 2]$, rather than $P[i, 3]$. However, the given algorithm is sufficient to achieve linear space, with a running time which is is no more than three times the running time of the algorithm of Part (a), whose space requirement was $\Theta(n^2)$. □

Exercise 1.4 Design and analyze a dynamic programming algorithm which, on input a string X, determines the minimum number p of palindrome substrings of X, $Y_1, Y_2, \ldots, Y_p$ such that $X = \langle Y_1, Y_2, \ldots, Y_p \rangle$.

Answer: Recall that a palindrome is a string $Z = \langle z_1, z_2, \ldots, z_k \rangle$ such that $z_{1+h} = z_{k-h}$, with $0 \leq h \leq k - 1$. We set up an optimal substructure property on the space of all prefixes of $X = \langle x_1, x_2, \ldots, x_n \rangle$. For the base case, consider the empty prefix $X_0 = \epsilon$ and let us conventionally set the length of its (unique) palindrome decomposition to 0. For $0 < i \leq n$, let $\langle Y_1, Y_2, \ldots, Y_p \rangle$ be an optimal decomposition of X_i into the smallest number $p \geq 1$ of palindromes. Then, clearly, $Y_p = \langle x_{s^\star}, \ldots, x_i \rangle$ is a palindrome, where $s^\star$ is a certain value $1 \leq s^\star \leq i$, and it can be easily argued that $\langle Y_1, Y_2, \ldots, Y_{p-1} \rangle$ must be an optimal decomposition of $X_{s^\star - 1}$ into the smallest number of palindromes. Indeed, if $X_{s^\star - 1}$ admitted a decomposition into less than $p - 1$ palindromes, then we could immediately obtain a decomposition of X_i into a number of palindromes smaller than p, a contradiction. Also, the value $p - 1$ must be the smallest among the lengths of all decompositions of X_{s-1}, for *all* palindrome substrings $X_{s..i}$, with $1 \leq s \leq i$, or again we could obtain a better decomposition of X_i.

Let $p(i)$ be the length of an optimal palindrome decomposition of X_i. Based on the above substructure property, we can write the following recurrence.

$$p(i) = \begin{cases} 0 & i = 0 \\ 1 + \min\{p(s-1) : (1 \leq s \leq i) \wedge (X_{s..i} \text{ is a palidrome})\} & i > 0 \end{cases}$$

We can immediately write a bottom-up code based on the above recurrence. Clearly, we have to compute $p(i)$ for increasing values of i, storing them into a table $P[\cdot]$. Also, in the code, we let PALINDROME(Z) be the straightforward subroutine (based on the definition) that returns **true** if and only of Z is a palindrome, by performing $|Z|$ character comparisons.

```
MIN_DECOMPOSITION(X)
n ← |X|
P[0] = 0
for i ← 1 to n do
    P[i] ← +∞
    for s ← 1 to i do
        if (PALINDROME(X_{s..i}))
          then P[i] ← MIN(P[i], 1 + P[s − 1])
return P[n]
```

The correctness of the above algorithm follows from the optimal substructure property proved above and from the fact that values of table P are always computed prior to their use. Its running time (in terms of character comparisons) $T_{\text{MD}}(n)$ depends on the invocations of the PALINDROME subroutine. We have:

$$\begin{aligned} T_{\text{MD}}(n) &= \sum_{i=1}^{n}\sum_{s=1}^{i}(i-s+1) \\ &= \sum_{i=1}^{n}\sum_{s'=1}^{i} s' \text{ (letting } s' = i-s+1) \\ &= \sum_{i=1}^{n} i(i+1)/2 \\ &= (1/2)(\sum_{i=1}^{n} i^2 + \sum_{i=1}^{n} i) \\ &= (1/12)n(n+1)(2n+1) + (1/4)n(n+1) \\ &= (1/4)n(n+1)((2n+1)/3+1) \end{aligned}$$

$$
\begin{aligned}
&= (1/6)n(n+1)(n+2) \\
&= \Theta(n^3).
\end{aligned}
$$

In the above algorthm, we can trade space for time if extend the code so that it precomputes a boolean table $PAL[r,t]$, with $PAL[r,t] = \textbf{true}$ if $X_{r..t}$ is a palindrome, for $1 \leq r \leq t \leq n$. As shown in Point **a** of the previous exercise, this table can be obtained in time $\Theta(n^2)$. Once the table is computed, a call to PALIDROME($X_{s..i}$) can then be substituted with a lookup to $PAL[s,i]$. (The required code modification is left as a simple exercise.) The running time $T'_{\mathrm{MD}}(n)$ of this modified algorithm in terms of comparisons is then simply the time required to precompute the table, hence $T'_{\mathrm{MD}}(n) = \Theta(n^2)$. □

Exercise 1.5 Design and analyze a dynamic programming algorithm that, given in input a string X, returns the maximum length of a palindrome *subsequence* of X. The algorithm must run in time and space $O(n^2)$.

Answer: We select the set $\mathcal{S}_X = \{X_{i..j} : 1 \leq i \leq j \leq n\}$ of all nonempty substrings of X as a subproblem space. We will prove the following optimal substructure property. Let $Z = \langle z_1, z_2, \ldots, z_m \rangle$ be a Longest Palindrome Subsequence (LPS) of $X_{i..j}$.
1. If $i = j$, then $Z = \langle x_i \rangle = X_{i..j}$;
2. If $j = i+1$ and $x_i = x_j$, then $Z = \langle x_i, x_j \rangle = X_{i..j}$;
3. If $j = i+1$ and $x_i \neq x_j$, then either $Z = \langle x_i \rangle$ or $Z = \langle x_j \rangle$;
4. If $j > i+1$ and $x_i = x_j$, then $z_1 = z_m = x_i$ and $Z_{2..m-1}$ is a LPS of $X_{i+1..j-1}$;
5. If $j > i+1$ and $x_i \neq x_j$, then either Z is a LPS of $X_{i+1..j}$ or a LPS of $X_{i..j-1}$.

Let us prove the above stated property. The three base cases (Points 1, 2, and 3) relative to substrings of length 1 and 2 are trivial. Consider now the case $j > i+1$ (i.e., substrings of length at least 3), and let $Z = \langle x_{i_1}, x_{i_2}, \ldots x_{i_m} \rangle$, for $i \leq i_1 < i_2 \ldots < i_m \leq j$. For Point 4, let $x_i = x_j$ and assume that $z_1, z_m \neq x_i$. Then $i_1 > i$ and $i_m < j$, hence the string $Z' = \langle x_i, Z, x_j \rangle$ is still a palindrome subsequence of $X_{i..j}$ (realized by the sequence of indices $i'_1 = i$, $i'_{j+1} = i_j$ for $1 \leq j \leq m$, and $i'_{m+2} = j$) which is two characters longer than Z, which contradicts the fact that Z is a LPS of $X_{i..j}$. Also, $Z_{2..m-1}$ is a palindrome subsequence of $X_{i+1..j-1}$ and it must be a LPS, or otherwise there would exist a longer LPS than Z for $X_{i..j}$. For Point 5, let $x_i \neq x_j$. Since Z is a palindrome, we have $z_1 = z_m$ hence either $i_1 > i$ or $i_m < j$. In the first case Z is a palindrome subsequence of $X_{i+1..j}$, while in the second case Z is a palindrome subsequence of $X_{i..j-1}$. In both cases, Z must be the respective LPS or otherwise one could obtain a longer LPS than Z for $X_{i..j}$.

Let $p(i,j) = |LPS(X_{i..j})|$. The optimal substructure property proved in Point 2 yields

the following recurrence on $p(i,j)$, for $1 \leq i \leq j \leq n$:

$$p(i,j) = \begin{cases} 1 & i = j, \\ 2 & j = i+1 \text{ and } x_i = x_j, \\ 1 & j = i+1 \text{ and } x_i \neq x_j, \\ 2 + p(i+1, j-1) & j > i+1 \text{ and } x_i = x_j, \\ \max\{p(i+1,j), p(i,j-1)\} & j > i+1 \text{ and } x_i \neq x_j. \end{cases}$$

Let $\ell = j - i + 1$ be the length of the substring $X_{i..j}$. Observe that a bottom-up computation of our solution $p(1,n)$ requires computing the subproblems for increasing values of ℓ. Indeed, the base cases concern the values $\ell = 1$ and $\ell = 2$, while for $\ell \geq 3$, a subproblem $X_{i..j}$ of size ℓ requires the solution to subproblems $X_{i+1..j-1}$, $X_{i+1..j}$, and $X_{i..j-1}$ whose sizes are $\ell - 2$, $\ell - 1$ and $\ell - 1$, respectively. Therefore a diagonal scan of the upper triangle of a table $P[i,j]$ suffices to correctly compute all relevant values. The code follows.

```
COMPUTE_P(X)
n ← length(X)
for i ← 1 to n − 1 do
    P[i, i] ← 1
    if (x_i = x_{i+1})
        then P[i, i + 1] ← 2
        else P[i, i + 1] ← 1
P[n, n] ← 1
{ initialize the first two diagonals with the base cases ℓ = 1 and ℓ = 2}
for ℓ ← 3 to n do
    for i ← 1 to n − ℓ + 1 do
        j ← i + ℓ − 1
        if (x_i = x_j)
            then P[i, j] ← P[i + 1, j − 1] + 2
            else P[i, j] ← MAX(P[i + 1, j], P[i, j − 1])
return P[1, n]
```

The correctness of the above algorithm follows from the previous discussion. The algorithm makes use of $\Theta(n^2)$ space (the $n \times n$ table P) and performs exactly one character comparison for each substring of length $\ell \geq 2$. Its running time is therefore

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1 = \frac{n(n-1)}{2} = \Theta(n^2).$$

It must be remarked that the above code can be made more space-efficient since we only need maintain three diagonals (the one being currently computed and the two immediately preceeding). Thus we can use three vectors rather than a full $n \times n$ table (see Exercise 1.3 for details). □

Exercise 1.6 Given a string of *arbitrary* integers $Z = \langle z_1, z_2, \ldots, z_k \rangle$ let weight$(Z) = \sum_{i=1}^{k} z_i$ (note that weigth$(\epsilon) = 0$). Given two integer strings $X = \langle x_1, x_2, \ldots, x_m \rangle$ e $Y = \langle y_1, y_2, \ldots, y_n \rangle$, design a dynamic programming algorithm to determine a Maximum-Weight Common Subsequence (MWCS) Z of X and Y.

Answer: Let $Z = \langle z_1, z_2, \ldots z_k \rangle$ be a MWCS of X and Y, with $Z = \langle x_{j_1}, x_{j_2}, \ldots x_{j_k} \rangle = \langle y_{h_1}, y_{h_2}, \ldots y_{h_k} \rangle$, with $1 \leq j_1 < j_2 < \ldots < j_k \leq m$ and $1 \leq h_1 < h_2 < \ldots < h_k \leq n$. Note that for $1 \leq i \leq k$, it must be $z_i \geq 0$, or otherwise $Z' = \langle z_1, \ldots z_{i-1}, z_{i+1}, \ldots z_k \rangle$ (which is a CS of X and Y) would have a higher weight than Z. Furthermore, we can assume without loss of generality that $z_i \neq 0$, since otherwise we can simply remove z_i, so that Z' remains an MWCS of X and Y. We can prove the following optimal substructure property:

1. *If $x_m = y_n \leq 0$, then Z is an MWCS of X_{m-1} and Y_{n-1}.*
   Proof: Since Z contains only numbers greater than zero, it must be $j_k < m$ and $h_k < n$. Hence Z is a CS of X_{m-1} and Y_{n-1}. Clearly, it must be an MWCS of X_{m-1} and Y_{n-1}, or otherwise an MWCS of X_{m-1} and Y_{n-1} would also be a CS of X and Y heavier than Z.

2. *If $x_m = y_n > 0$, then $z_k = x_m$ and Z_{k-1} is an MWCS of X_{m-1} and Y_{n-1}.*
   Proof: If $z_k \neq x_m = y_n$, it must be $j_k < m$ and $h_k < n$. But then $Z' = \langle Z, x_m \rangle$ is still a CS of X and Y with weight $x_m + \text{weight}(Z) > \text{weight}(Z)$, a contradiction. Also, Z_{k-1} is a CS of X_{m-1} and Y_{n-1}. By arguing as above we claim that it must be the one of maximum weight.

3. *If $x_m \neq y_n$ then Z is the sequence of maximum weight between an MWCS of X_m and Y_{n-1} and an MWCS of X_{m-1} and Y_n.*
   Proof: Since $x_m \neq y_n$, either $j_k < m$ or $h_k < n$. Hence Z must be either an MWCS of X_{m-1} and Y (first case) or an MWCS of X and Y_{n-1} (second case). Clearly, Z must be the sequence of largest weight among the two.

Let $W[i, j]$ be the weight of an MWCS of X_i and Y_j, with $0 \leq i \leq m$ and $0 \leq j \leq n$. The

above property implies the following recurrence for $W[i,j]$.

$$W[i,j] = \begin{cases} 0 & i = 0 \text{ or } j = 0, \\ W[i-1,j-1] & \text{if } x_i = y_j \leq 0, \\ x_i + W[i-1,j-1] & \text{if } x_i = y_j > 0, \\ \max\{W[i-1,j], W[i,j-1]\} & \text{otherwise.} \end{cases}$$

The bottom-up computation of the above recurrence can be performed as follows:

```
MWCS(X, Y)
    m ← length(X)
    n ← length(Y)
    for i ← 0 to m do W[i, 0] ← 0
    for j ← 1 to n do W[0, j] ← 0
    for i ← 1 to m do
        for j ← 1 to n do
            if (x_i = y_j)
                then if (x_i ≤ 0)
                        then W[i, j] ← W[i − 1, j − 1]
                        else W[i, j] ← x_i + W[i − 1, j − 1]
                else W[i, j] ← MAX(W[i, j − 1], W[i − 1, j])
    return W[m, n]
```

The correctness of the above algorithm follows from the optimal substructure property and from the fact that the values $W[i-1,j-1], W[i,j-1]$ and $W[i-1,j]$ have already been computed when $W[i,j]$ is being computed. The algorithm's running time is clearly $\Theta(mn)$. □

Exercise 1.7 Design and analyze a dynamic programming algorithm which, on input two nonnegative integers n and k, with $n > 0$ and $0 \leq k \leq n$, outputs $\binom{n}{k}$ by performing $\Theta(nk)$ sums. (*Hint:* Prove that for $0 < k < n$, $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.)

Answer: Recall that $\binom{n}{k}$ is defined as the number of distinct subsets of cardinality k that can be extracted from a set of cardinality n. Thus $\binom{n}{k}\Big|_{(k=0)\vee(k=n)} = 1$. To prove the relation in the hint, recall that for $0 < k < n$ the number of subsets of cardinality k can be partitioned into the family of subsets which do not contain the n-th element of the set and the family of subsets which do contain such an element. Clearly, the former family has

cardinality $\binom{n-1}{k}$, while each subset in the latter family is in one-to-one correspondence with a distinct subset of cardinality $k-1$ extracted from the set of the first $n-1$ elements, thus its cardinality is $\binom{n-1}{k-1}$.

For $0 < k < n$, to compute $\binom{n}{k}$ in a bottom-up fashion, we use a (rectangular) table $C[\cdot,\cdot]$ of $n \times (k+1)$ entries, where entry $C[i,j]$ will eventually store the value $\binom{i}{j}$, for $i > 0$ and $0 \leq j \leq i$. The first column of the table ($j = 0$) and the k elements $C[i,i]$, for $1 \leq i \leq k$, correspond to the base cases $\binom{i}{0} = \binom{i}{i}$ and must therefore be initialized to 1. A row-major scan guarantees that each table entry has been computed prior to being used.

```
BIN_COEFF(n, k)
if ((k = 0) ∨ (k = n)) then return 1
for i ← 1 to k do C[i, 0] ← C[i, i] ← 1
for i ← k + 1 to n do C[i, 0] ← 1
for i ← 2 to n do
    for j ← 1 to min{i − 1, k} do
        C[i, j] ← C[i − 1, j] + C[i − 1, j − 1]
return C[n, k]
```

Observe that the upper bound on the inner **for** loop makes sure that we do not compute table entries corresponding to indices (i,j) with $j > i$. The running time of the above algorithm, in terms of number of additions performed is

$$\begin{aligned}
T_{\mathrm{BC}}(n,k) &= \sum_{i=2}^{n} \sum_{j=1}^{\min\{i-1,k\}} 1 \\
&= \sum_{i=2}^{k+1} \sum_{j=1}^{i-1} 1 + \sum_{i=k+2}^{n} \sum_{j=1}^{k} 1 \\
&= \sum_{i=2}^{k+1} (i-1) + \sum_{i=k+2}^{n} k \\
&= k(k+1)/2 + (n-(k+1))k \\
&= nk - k(k+1)/2 \\
&= \Theta(nk).
\end{aligned}$$

□

Exercise 1.8 Given two strings X and Y, a third string Z is a *common superstring* of X and Y, if X and Y are both subsequences of Z. (*Example:* if $X =$ `sos` and $Y =$ `soia`,

then $Z =$ `sosia` is a common superstring of X and Y.) Design and analyze a dynamic programming algorithm which, given as input two strings X and Y, returns the length of the *Shortest Common Superstring* (SCS) of X and Y and additional information needed to print the SCS. The algorithm must run in time $\Theta(|X||Y|)$. (*Hint:* Use an approach similar to the one used to compute the LCS of two strings.)

Answer: For $0 \leq i \leq m$, $0 \leq j \leq n$, let X_i and Y_j denote the prefixes of X and Y up to and including the i-th and j-th charachter, respectively (the index 0 is associated to the empty prefix ε). Let us also denote by $L[i,j]$ the length of a shortest common superstring of X_i and Y_j. Our dynamic programming algorithm is based on the following recurrence:

$$L[i,j] = \begin{cases} i & \text{if } j = 0, \\ j & \text{if } i = 0 \text{ and } j > 0, \\ 1 + L[i-1, j-1] & \text{if } i,j > 0 \text{ and } x_i = y_j, \\ 1 + \min\{L[i, j-1], L[i-1, j]\} & \text{if } i,j > 0 \text{ and } x_i \neq y_j. \end{cases}$$

In order to prove the above recurrence, let $Z = z_1 z_2 \ldots z_k = \text{SCS}(X_i, Y_j)$. First note that $j = 0 \Rightarrow Z = X_i$ and $i = 0 \Rightarrow Z = Y_j$, which suffices to prove the first two cases.

Assume now $i, j > 0$ and $x_i = y_j$. Clearly, it must be $z_k = x_i = y_j$, or otherwise z_k could be eliminated and Z_{k-1} would be an CS of X_i and Y_j shorter than Z. Therefore, Z_{k-1} must be a common superstring of X_{i-1} and Y_{j-1}. Indeed, $Z_{k-1} = \text{SCS}(X_{i-1}, Y_{j-1})$ or otherwise we could obtain a supertring shorter than Z for X_i and Y_j. This proves the third case.

Finally, let $i, j > 0$ and $x_i \neq y_j$. Then, either $z_k = x_i$ or $z_k = y_j$, or otherwise z_k could be eliminated and Z_{k-1} would be a CS of X_i and Y_j shorter than Z. If $z_k = x_i$ then Z_{k-1} must be a common superstring of X_{i-1} and Y_j, otherwise Z_{k-1} must be a common superstring of X_i and Y_{j-1}, and in both cases Z_{k-1} must be the shortest. Which case occurs depends on whether $|\text{SCS}(X_{i-1}, Y_j)| \leq |\text{SCS}(X_i, Y_{j-1})|$ or vice versa. This proves the fourth case.

The algorithm follows:

```
SCS_LENGTH(X, Y)
m ← length(X)
n ← length(Y)
for i ← 0 to m do L[i, 0] ← i
for j ← 1 to n do L[0, j] ← j
for i ← 1 to m do
    for j ← 1 to n do
        if x_i = y_j
          then L[i, j] ← 1 + L[i − 1, j − 1]
          else L[i, j] ← 1+MIN(L[i, j − 1], L[i − 1, j])
        { L[i − 1, j − 1], L[i, j − 1], L[i − 1, j] available at this point }
return L[m, n]
```

The running time $T_{\text{SCS}}(m, n)$ of SCS_LENGTH(X, Y) is dominated by the execution of two nested loops of m and n constant-time iterations, hence $T_{\text{SCS}}(m, n) = O(mn)$. □

Exercise 1.9 Given two strings of integers Z^1 and Z^2, with $|Z^1| = |Z^2| = k$, we define their *discrepancy* as $d(Z^1, Z^2) = \sum_{i=1}^{k} |Z_i^1 - Z_i^2|$. Design and analyze a dynamic programming algorithm which, on input two (arbitrary) strings of integers X e Y, computes the maximum discrepancy obtainable by a subsequence of X and a subsequence of Y of equal length by performing $\Theta(|X||Y|)$ comparisons and sums between integers.

Answer: Let us choose as our subproblem space the cartesian product of all prefixes of strings X and Y, that is, the set $\{(X_i, Y_j) : 0 \leq i \leq |X|, 0 \leq j \leq |Y|\}$. Our dynamic programming algorithm will be based on the following optimal substructure property. Clearly, if either $i = 0$ or $j = 0$ then the only pair of subsequences of equal length that can be extracted from X_i and Y_j is (ϵ, ϵ), yielding null discrepancy. Otherwise, let $Z^1 = \langle z_1^1, z_2^1, \ldots, z_k^1 \rangle$ and $Z^2 = \langle z_1^2, z_2^2, \ldots, z_k^2 \rangle$ be the two equal-length subsequences of X_i and Y_j which achieve maximum discrepancy. We can have three cases. If $z_k^1 = x_i$ and $z_k^2 = y_j$, then Z_{k-1}^1 and Z_{k-1}^2 must be subsequences of X_{i-1} and Y_{j-1}, respectively, achieving maximum discrepancy or we could obtain a larger discrepancy than the one realized by Z for X_i and Y_j. Using a similar argument, it is easy to see that if $z_k^1 \neq x_i$, then Z^1 is a subsequence of X_{i-1} and it must achieve maximum discrepancy with Z^2, a subsequence of Y_j. Analogously, if $z_k^2 \neq y_j$, then Z^2 is a subsequence of Y_{j-1} yielding maximum discrepancy with Z^1, a subsequence of X_i. Out of the three cases above, the one maximizing the discrepancy will occur.

Let $D(i,j)$ be the maximum discrepancy $d(Z^1, Z^2)$ achievable by a pair of equal-length subsequences Z^1 of X_i and Z^2 of Y_j. The above substructure property immediately yields the following recurrence for $D(i,j)$:

$$D(i,j) = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ \max\{|x_i - y_j| + D(i-1, j-1), D(i-1, j), D(i, j-1)\} & \text{otherwise.} \end{cases}$$

From the recurrence, we immediately obtain the desired algorithm.

```
DISCREPANCY(X, Y)
m ← length(X)
n ← length(Y)
for i ← 0 to m do
    D[i, 0] ← 0
for j ← 0 to n do
    D[0, j] ← 0
for i ← 1 to m do
    for j ← 1 to n do
        D[i, j] ← MAX(|x_i − y_j| + D[i − 1, j − 1], D[i − i, j], D[i, j − 1])
return D[m, n]
```

The correctness of the above algorithm easily follows from the correctness of the recurrence and from the fact that the row-major scan computes every table entry prior to any of its uses. The running time of the algorithm is clearly $\Theta(mn) = \Theta(|X||Y|)$. □

Exercise 1.10 Let $n > 0$. Given a string of n integers $A = \langle a_1, a_2, \ldots, a_n \rangle$, consider the following recurrence, defined for all pairs (i,j), with $1 \leq i \leq j \leq n$:

$$B(i,j) = \begin{cases} a_i & 1 \leq i = j \leq n, \\ \max\{B(i,k) \cdot B(k+1, j) : i \leq k \leq j-1\} & 1 \leq i < j \leq n. \end{cases}$$

Design and analyze an iterative bottom-up algorithm that, on input A, returns $B(1,n)$ by performing $O(n^3)$ sums.

Answer: In order to organize a bottom-up computation, let $\ell_{i,j} = j - i + 1$ and note that the computation of $B(i,j)$ depends on elements $B(i,k)$ and $B(k+1,j)$, with $i \leq k < j$. For such range of values of k, we have $\ell_{i,k} = k - i + 1 < j - i + 1 = \ell_{i,j}$ and $\ell_{k+1,j} = j - k \leq j - i < j - i + 1 = \ell_{i,j}$. Hence, we can organize the computation of

$B(1,n)$ in a bottom-up fashion by computing the table entries by increasing values of ℓ. This corresponds to scanning the upper triangle of the table storing the values $B(i,j)$, for $1 \leq i \leq j \leq n$, by diagonals parallel to the principal one. The algorithm follows.

```
COMPUTE_B(a)
n ← length(a)
for i ← 1 to n do
    B[i,i] ← a_i
for ℓ ← 2 to n do
    for i ← 1 to n − ℓ + 1 do
        j ← i + ℓ − 1
        B[i,j] ← −∞
        for k ← i to j − 1 do
            B[i,j] ← MAX(B[i,j], B[i,k] * B[k+1,j])
return B[1,n]
```

The correctness of the above algorithm follows from the fact that it correctly implements a bottom-up computation of $B[1,n]$, as argued above. Its running time is dominated by the overall number of iterations of the three nested loops, which yield a time complexity of

$$\Theta\left(\sum_{\ell=2}^{n}\sum_{i=1}^{n-\ell+1}\sum_{k=i}^{i+\ell-2}1\right) = \Theta\left(n^3\right).$$

(This is the same analysis of the matrix-chain multiplication algorithm seen in class.) □

Exercise 1.11 Let $n > 0$. Assume that a given dynamic programming strategy leads to the following recurrence, defined for all values of i and j with $1 \leq i \leq j \leq n$:

$$C(i,j) = \begin{cases} 1 & (i=1) \textbf{ and } (j=n), \\ \sum_{r=1}^{i-1} C(r,j) + \sum_{s=j+1}^{n} C(i,s) & \text{altrimenti.} \end{cases}$$

Design and analyze an iterative bottom-up algorithm that computes all values $C(i,j)$, $1 \leq i \leq j \leq n$.

Answer: Let us first begin with a simple $O(n^3)$ algorithm. Consider a table $C[i,j]$ which stores the value of $C(i,j)$, for $1 \leq i \leq j \leq n$. From the definition of T it is clear that we can compute $C[i,j]$ if the table entries $C[r,j]$, for $1 \leq r < i$ and $C[i,s]$, for $j < s \leq n$ have already been computed. Let $\ell = j - i + 1$ (ℓ can be seen as the "size" of "subproblem" (i,j)). We have that $j - r + 1 > j - i + 1 = \ell$ and $s - i + 1 > j - i + 1 = \ell$, hence we can

organize the iterations by decreasing size. The code immediately follows.

```
COMPUTE_T(n)
C[1, n] ← 1
for ℓ ← n − 1 downto 1 do
    { process subproblems of decreasing length ...}
    for i ← 1 to n − ℓ + 1 do
        j ← i + ℓ − 1
        { ready to compute C[i, j] }
        C[i, j] ← 0
        for r ← 1 to i − 1 do
            C[i, j] ← C[i, j] + C[r, j]
        for s ← j + 1 to n do
            C[i, j] ← C[i, j] + C[i, s]
return C
```

The correctness of the code follows from the fact that the values $C[r, j]$ and $C[i, s]$ have already been (correctly) computed when used. The running time of the above algorithm is evaluated as follows:

$$\begin{aligned} T_{\text{COMP}}(n) &= \sum_{\ell=1}^{n-1} \sum_{i=1}^{n-\ell+1} [(i-1) + (n-i-\ell+1)] \\ &= \sum_{\ell=1}^{n-1} \sum_{i=1}^{n-\ell+1} (n-l) = \sum_{\ell=1}^{n-1} (n-\ell+1)(n-\ell) \\ &= \sum_{\ell=1}^{n-1} (\ell+1)\ell = \sum_{\ell=1}^{n-1} \ell^2 + \sum_{\ell=1}^{n-1} \ell = \Theta(n^3) \end{aligned}$$

A much more efficient algorithm can be obtained by observing that each $C(i, j)$ is the sum of two terms. By keeping the two terms separate, storing them into two vectors, we can reduce the time to compute a single entry of C to a constant. Specifically, define the following two vectors as a function of vector C:

$$Up[i, j] = \begin{cases} 1 & (i = 1) \textbf{ and } (j = n), \\ \sum_{r=1}^{i-1} C[r, j] & \text{otherwise;} \end{cases}$$

and

$$Right[i, j] = \sum_{s=j+1}^{n} C[i, s].$$

Note that from the definition of $C(i,j)$, it immediately follows that $C[i,j] = Up[i,j] + Right[i,j]$. Moreover, the following recurrences hold:

$$Up[i,j] = \begin{cases} 1 & i = 1, j = n, \\ 0 & i = 1, j \neq n \\ Up[i-1,j] + C[i-1,j] & \text{otherwise;} \end{cases}$$

and

$$Right[i,j] = \begin{cases} 0 & j = n \\ Right[i,j+1] + C[i,j+1] & \text{otherwise;} \end{cases}$$

Note that in order to compute the values at coordinates (i,j) for vectors C, Up and $Right$, we only need values at coordinates $(i-1,j)$ and $(i,j+1)$. Hence we can keep the same type of scan used in the previous algorithm. We obtain the following code:

```
COMPUTE2_T(n)
Up[1,n] ← 1
Right[1,n] ← 0
C[1,n] ← 1
for ℓ ← n − 1 downto 1 do
    Up[1,ℓ] ← 0
    Right[1,ℓ] ← Right[1,ℓ+1] + C[1,ℓ+1]
    C[1,ℓ] ← Right[1,ℓ]
    {Element of length ℓ on the first row}
    for i ← 2 to n − ℓ do
        j ← i + ℓ − 1
        Up[i,j] ← Up[i−1,j] + C[i−1,j]
        Right[i,j] ← Right[i,j+1] + C[i,j+1]
        C[i,j] ← Up[i,j] + Right[i,j]
    Right[n−ℓ+1,n] ← 0
    Up[n−ℓ+1,n] ← Up[n−ℓ,n] + C[n−ℓ,n]
    C[n−ℓ+1,n] ← Up[n−ℓ+1,n]
    {Element of length ℓ on the last column}
return C
```

As before, all values needed in an iteration have been computed previously, hence the code is correct. The new algorithm performs a constant number of operations for each entry of C to be computed, hence its running time is $\Theta(n^2)$. □

Exercise 1.12 Given the following bottom-up code:

```
DP_SUM(n)
for i ← 1 to n do A[i, i] ← i
for ℓ ← 2 to n do
    for i ← 1 to n − ℓ + 1 do
        j ← i + ℓ − 1
        A[i, j] ← A[i, j − 1] + A[i + 1, j]
return A[1, n]
```

write an equivalent memoized code and analyze its running time in terms of sums between integers.

Answer: The given code computes the following recurrence:

$$A(i,j) = \begin{cases} i & 1 \le i = j \le n \\ A(i,j-1) + A(i+1,j) & 1 \le i < j \le n \end{cases}$$

The memoized code to compute this recurrence is given by the following pair of routines:

```
INIT_A(n)
for i ← 1 to n do
    A[i, i] ← i
    for j ← i + 1 to n do
        A[i, j] ← 0 { note that A(i, j) is always positive}
return REC_A(1, n)

REC_A(i, j)
if A[i, j] = 0
    then A[i, j] ← REC_A(i, j − 1) + REC_A(i + 1, j)
return A[i, j]
```

To obtain $A(1,n)$, we call INIT_A(n), which in turn invokes REC_A$(1,n)$. This latter routine executes exactly one sum for each pair (i,j), with $1 \le i < j \le n$, for a total running time

$$T_A(n) \;=\; \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} 1$$

$$
\begin{aligned}
&= \sum_{i=1}^{n-1} (n-i) \\
&= \sum_{i'=1}^{n-1} i' \text{ (letting } i' = n-i) \\
&= n(n-1)/2 = \Theta(n^2),
\end{aligned}
$$

which is also the running time of the corresponding dynamic programming code. □

Exercise 1.13 Given a string $X = \langle x_1, x_2, \ldots, x_n \rangle$, consider the following recurrence $\ell(i,j)$, defined for $1 \le i \le j \le n$:

$$
\ell(i,j) = \begin{cases}
1 & i = j, \\
2 & i = j-1 \\
2 + \ell(i+1, j-1) & (i < j-1) \wedge (x_i = x_j) \\
\sum_{k=i}^{j-1} (\ell(i,k) + \ell(k+1,j)) & (i < j-1) \wedge (x_i \neq x_j).
\end{cases}
$$

Design memoized code to return the value $\ell(1,n)$ and analyze the code both in the worst case and in the best case, assuming that the only unit-cost operations are character comparisons.

Answer: Our memoized code consists of a pair of subroutines, INIT_L(X) and REC_L(X,i,j). INIT_L(X) returns directly on base cases $n = 1$ or $n = 2$. Otherwise it initializes a look-up table with base and default values and then invokes REC_L($X,1,n$), which computes recursively all nonbase values, storing them in the table to avoid repeated calls. Observe that 0 is a suitable default value, since $\ell(i,j) > 0$ for $1 \le i \le j \le n$. The code follows.

```
INIT_L(X, n)
if n = 1 then return 1
    {ℓ(1,1) = 1, base case}
if n = 2 then return 2
    {ℓ(1,2) = 2, base case}
for i ← 1 to n − 1 do
    L[i, i] ← 1
    L[i, i + 1] ← 2
L[n, n] ← 1
for i ← 1 to n − 2 do
    forj ← i + 2 to n do
        L[i, j] ← 0
return REC_L(X, 1, n)
```

```
REC_L(X, i, j)
if (L[i, j] = 0)
    then
        if (x_i = x_j)
            then
            L[i, j] ← 2 + REC_L(X, i + 1, j − 1)
            else
            for k ← i to j − 1 do
                L[i, j] ←L[i, j]+
                        REC_L(X, i, k)+
                        REC_L(X, k + 1, j)
return L[i, j]
```

Let us analyze the above algorithm. Observe that INIT_L(X) does not perform any character comparisons. Let us now analyze the recursion tree associated to the call REC_L($X, 1, n$). The best case is clearly the one where all characters are the same, since the recursion tree in that case is unary and its internal nodes correspond to the calls with indices $(1, n), (2, n-1), \ldots (k, n-k+1)$, each associated to a cost of 1 (one comparison). There is one such call for every $1 \leq k \leq n-k-1$ (nonbase cases), whence $1 \leq k \leq (n-1)/2$, for a running time $T_B(n) \leq \lfloor n/2 \rfloor = O(n)$.

Let us now concentrate on the worst case. If $x_i \neq x_j$ for $1 \leq i < j \leq n$ all nonbase subproblems are eventually solved. Thus, in this case, thanks to memoization, there is exactly one internal node for each call with indices (i, j), with $1 \leq i < j - 1$, each associated to a cost of 1 (again, one comparison only), for a total time

$$\begin{aligned} T_W(n) &= \sum_{i=1}^{n-2} \sum_{j=i+2}^{n} 1 \\ &= \sum_{i=1}^{n-2} (n-1-i) \\ &= \sum_{k=1}^{n-2} k \\ &= (n-2)(n-1)/2 = \Theta(n^2). \end{aligned}$$

Finally, note that if we considered additions as operations also requiring unit cost, the running time in the best case would stay linear in n, while the running time in the worst

case would become cubic in n! □