Chapter 5

Dynamic Programming

Exercise 5.1 Write an algorithm to find the maximum value that can be obtained by an appropriate placement of parentheses in 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 $M[i,j]$:

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

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 5.2 In Algorithm MATRIX_CHAIN_ORDER (CLRS, page 336), determine the *exact* number of times that the following Line (Line 9) is executed:

$$\textbf{do } q \leftarrow m[i,k] + m[k+1,j] + p_{i-1}p_kp_i$$

Answer: Line 9 is executed once in each iteration of the innermost loop,

$$\textbf{for } k \leftarrow i \textbf{ to } j-1 \textbf{ do } \ldots$$

This loop is executed once in each iteration of the loop

$$\textbf{for } i \leftarrow 1 \textbf{ to } n - \ell + 1 \textbf{ do } ...,$$

which is in turn executed once in each iteration of the loop

$$\textbf{for } l \leftarrow 2 \textbf{ to } n \textbf{ do } ....$$

Recall that $j = i + \ell - 1$. Therefore, the total number of times that Line 9 is executed is

$$\begin{aligned}
T_9(n) &= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} \sum_{k=i}^{i+l-2} 1 \\
&= \sum_{l=2}^{n} \sum_{i=1}^{n-l+1} (l-1) \\
&= \sum_{l=2}^{n} (l-1)(n-l+1) \\
&= \sum_{l=2}^{n} l(n-l+1) - \sum_{l=2}^{n} (n-l+1) \\
&= (n+1)\sum_{l=2}^{n} l - \sum_{l=2}^{n} l^2 - \sum_{h=1}^{n-1} h \text{ (we set } h = n-l+1 \text{ in the second sum)} \\
&= (n+1)\left[\frac{n(n+1)}{2} - 1\right] - \frac{n(n+1)(2n+1)}{6} + 1 - \frac{(n-1)n}{2} \\
&= \frac{n^3+2n^2+n}{2} - n - 1 - \frac{2n^3+3n^2+n}{6} + 1 - \frac{n^2-n}{2} \\
&= \frac{3n^3-2n^3}{6} + \frac{2n^2-n^2-n^2}{2} + \frac{3n-6n-n+3n}{6} - 1 + 1 \\
&= \frac{n^3-n}{6}.
\end{aligned}$$

Note that $T_9(n) = \Theta(n^3)$. □

Exercise 5.3 Give an algorithm that uses the vector s computed by Algorithm MATRIX_CHAIN_ORDER (CLR, page 306) 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 $1 \leq i \leq k < j \leq n$. 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 5.4 Given the string $A = \langle a_1, a_2, \dots, a_n$, 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

$$P[i,j] \Leftrightarrow (P[i+1, j-1] \textbf{ and } a_i = a_j)\,. \tag{5.1}$$

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 5.5 Given a string of *arbitrary* itegers $Z = \langle z_1, z_2, \ldots, z_k \rangle$ 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 5.6 Design and analyze a dynamic programming algorithm which, on input two nonnegative integers n and k, with $n \geq k$, outputs $\binom{n}{k}$ in $O(nk)$ time. (*Hint:* Prove that for $0 < k < n$, $\binom{n}{k} = \binom{n-1}{k} + \binom{n-1}{k-1}$.)

Exercise 5.7 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 $X = \langle Y_1, Y_2, \ldots, Y_p \rangle$.

Exercise 5.8 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.)

Exercise 5.9 Given a string $X = \langle x_1, x_2, \ldots, x_n \rangle$ of n integers, a *Spaced Monotonically Increasing Subsequence* (SMIS) of X is a subsequence $Z = \langle x_{j_1}, x_{j_2}, \ldots, x_{j_k} \rangle$, with $x_{j_i} < x_{j_{i+1}}$ **and** $j_{i+1} \geq j_i + 2$, for $1 \leq i < k$. Design and analyze a dynamic programming algorithm that returns a *Longest* SMIS (LSMIS) of the input string X in time $\Theta(n^2)$.