Selected Solutions for Chapter 15: Dynamic Programming

Solution to Exercise 15.2-5

Each time the l-loop executes, the i-loop executes $n - l + 1$ times. Each time the i-loop executes, the k-loop executes $j - i = l - 1$ times, each time referencing m twice. Thus the total number of times that an entry of m is referenced while computing other entries is $\sum_{l=2}^{n}(n - l + 1)(l - 1)2$. Thus,

$$\begin{aligned}
\sum_{i=1}^{n}\sum_{j=i}^{n} R(i, j) &= \sum_{l=2}^{n}(n - l + 1)(l - 1)2 \\
&= 2\sum_{l=1}^{n-1}(n - l)l \\
&= 2\sum_{l=1}^{n-1} nl - 2\sum_{l=1}^{n-1} l^2 \\
&= 2\frac{n(n-1)n}{2} - 2\frac{(n-1)n(2n-1)}{6} \\
&= n^3 - n^2 - \frac{2n^3 - 3n^2 + n}{3} \\
&= \frac{n^3 - n}{3} .
\end{aligned}$$

Solution to Exercise 15.3-1

Running RECURSIVE-MATRIX-CHAIN is asymptotically more efficient than enumerating all the ways of parenthesizing the product and computing the number of multiplications for each.

Consider the treatment of subproblems by the two approaches.

- For each possible place to split the matrix chain, the enumeration approach finds all ways to parenthesize the left half, finds all ways to parenthesize the right half, and looks at all possible combinations of the left half with the right half. The amount of work to look at each combination of left- and right-half subproblem results is thus the product of the number of ways to do the left half and the number of ways to do the right half.

- For each possible place to split the matrix chain, RECURSIVE-MATRIX-CHAIN finds the best way to parenthesize the left half, finds the best way to parenthesize the right half, and combines just those two results. Thus the amount of work to combine the left- and right-half subproblem results is $O(1)$.

Section 15.2 argued that the running time for enumeration is $\Omega(4^n/n^{3/2})$. We will show that the running time for RECURSIVE-MATRIX-CHAIN is $O(n3^{n-1})$.

To get an upper bound on the running time of RECURSIVE-MATRIX-CHAIN, we'll use the same approach used in Section 15.2 to get a lower bound: Derive a recurrence of the form $T(n) \le \ldots$ and solve it by substitution. For the lower-bound recurrence, the book assumed that the execution of lines 1–2 and 6–7 each take at least unit time. For the upper-bound recurrence, we'll assume those pairs of lines each take at most constant time c. Thus, we have the recurrence

$$T(n) \le \begin{cases} c & \text{if } n = 1 , \\ c + \displaystyle\sum_{k=1}^{n-1}(T(k) + T(n-k) + c) & \text{if } n \ge 2 . \end{cases}$$

This is just like the book's $\ge$ recurrence except that it has c instead of 1, and so we can be rewrite it as

$$T(n) \le 2\sum_{i=1}^{n-1} T(i) + cn .$$

We shall prove that $T(n) = O(n3^{n-1})$ using the substitution method. (Note: Any upper bound on $T(n)$ that is $o(4^n/n^{3/2})$ will suffice. You might prefer to prove one that is easier to think up, such as $T(n) = O(3.5^n)$.) Specifically, we shall show that $T(n) \le cn3^{n-1}$ for all $n \ge 1$. The basis is easy, since $T(1) \le c = c \cdot 1 \cdot 3^{1-1}$. Inductively, for $n \ge 2$ we have

$$\begin{aligned}
T(n) &\le 2\sum_{i=1}^{n-1} T(i) + cn \\
&\le 2\sum_{i=1}^{n-1} ci3^{i-1} + cn \\
&\le c \cdot \left(2\sum_{i=1}^{n-1} i3^{i-1} + n\right) \\
&= c \cdot \left(2 \cdot \left(\frac{n3^{n-1}}{3-1} + \frac{1-3^n}{(3-1)^2}\right) + n\right) \qquad \text{(see below)} \\
&= cn3^{n-1} + c \cdot \left(\frac{1-3^n}{2} + n\right) \\
&= cn3^{n-1} + \frac{c}{2}(2n + 1 - 3^n) \\
&\le cn3^{n-1} \text{ for all } c > 0, n \ge 1 .
\end{aligned}$$

Running RECURSIVE-MATRIX-CHAIN takes $O(n3^{n-1})$ time, and enumerating all parenthesizations takes $\Omega(4^n/n^{3/2})$ time, and so RECURSIVE-MATRIX-CHAIN is more efficient than enumeration.

Note: The above substitution uses the following fact:

$$\sum_{i=1}^{n-1} ix^{i-1} = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2} .$$

This equation can be derived from equation (A.5) by taking the derivative. Let

$$f(x) = \sum_{i=1}^{n-1} x^i = \frac{x^n - 1}{x-1} - 1 .$$

Then

$$\sum_{i=1}^{n-1} ix^{i-1} = f'(x) = \frac{nx^{n-1}}{x-1} + \frac{1-x^n}{(x-1)^2} .$$

Solution to Exercise 15.4-4

When computing a particular row of the c table, no rows before the previous row are needed. Thus only two rows—$2 \cdot Y.\mathit{length}$ entries—need to be kept in memory at a time. (Note: Each row of c actually has $Y.\mathit{length} + 1$ entries, but we don't need to store the column of 0's—instead we can make the program "know" that those entries are 0.) With this idea, we need only $2 \cdot \min(m, n)$ entries if we always call LCS-LENGTH with the shorter sequence as the Y argument.

We can thus do away with the c table as follows:

- Use two arrays of length $\min(m, n)$, *previous-row* and *current-row*, to hold the appropriate rows of c.
- Initialize *previous-row* to all 0 and compute *current-row* from left to right.
- When *current-row* is filled, if there are still more rows to compute, copy *current-row* into *previous-row* and compute the new *current-row*.

Actually only a little more than one row's worth of c entries—$\min(m, n) + 1$ entries—are needed during the computation. The only entries needed in the table when it is time to compute $c[i, j]$ are $c[i, k]$ for $k \leq j - 1$ (i.e., earlier entries in the current row, which will be needed to compute the next row); and $c[i-1, k]$ for $k \geq j - 1$ (i.e., entries in the previous row that are still needed to compute the rest of the current row). This is one entry for each k from 1 to $\min(m, n)$ except that there are two entries with $k = j - 1$, hence the additional entry needed besides the one row's worth of entries.

We can thus do away with the c table as follows:

- Use an array a of length $\min(m, n) + 1$ to hold the appropriate entries of c. At the time $c[i, j]$ is to be computed, a will hold the following entries:
  - $a[k] = c[i, k]$ for $1 \leq k < j - 1$ (i.e., earlier entries in the current "row"),
  - $a[k] = c[i-1, k]$ for $k \geq j - 1$ (i.e., entries in the previous "row"),

- $a[0] = c[i, j-1]$ (i.e., the previous entry computed, which couldn't be put into the "right" place in a without erasing the still-needed $c[i-1, j-1]$).
- Initialize a to all 0 and compute the entries from left to right.
  - Note that the 3 values needed to compute $c[i, j]$ for $j > 1$ are in $a[0] = c[i, j-1]$, $a[j-1] = c[i-1, j-1]$, and $a[j] = c[i-1, j]$.
  - When $c[i, j]$ has been computed, move $a[0]$ ($c[i, j-1]$) to its "correct" place, $a[j-1]$, and put $c[i, j]$ in $a[0]$.

Solution to Problem 15-4

Note: We assume that no word is longer than will fit into a line, i.e., $l_i \le M$ for all i.

First, we'll make some definitions so that we can state the problem more uniformly. Special cases about the last line and worries about whether a sequence of words fits in a line will be handled in these definitions, so that we can forget about them when framing our overall strategy.

- Define $\mathit{extras}[i, j] = M - j + i - \sum_{k=i}^{j} l_k$ to be the number of extra spaces at the end of a line containing words i through j. Note that *extras* may be negative.
- Now define the cost of including a line containing words i through j in the sum we want to minimize:

$$\mathit{lc}[i, j] = \begin{cases} \infty & \text{if } \mathit{extras}[i, j] < 0 \text{ (i.e., words } i, \ldots, j \text{ don't fit)} , \\ 0 & \text{if } j = n \text{ and } \mathit{extras}[i, j] \ge 0 \text{ (last line costs 0)} , \\ (\mathit{extras}[i, j])^3 & \text{otherwise} . \end{cases}$$

  By making the line cost infinite when the words don't fit on it, we prevent such an arrangement from being part of a minimal sum, and by making the cost 0 for the last line (if the words fit), we prevent the arrangement of the last line from influencing the sum being minimized.

We want to minimize the sum of *lc* over all lines of the paragraph.

Our subproblems are how to optimally arrange words $1, \ldots, j$, where $j = 1, \ldots, n$.

Consider an optimal arrangement of words $1, \ldots, j$. Suppose we know that the last line, which ends in word j, begins with word i. The preceding lines, therefore, contain words $1, \ldots, i-1$. In fact, they must contain an optimal arrangement of words $1, \ldots, i-1$. (The usual type of cut-and-paste argument applies.)

Let $c[j]$ be the cost of an optimal arrangement of words $1, \ldots, j$. If we know that the last line contains words $i, \ldots, j$, then $c[j] = c[i-1] + \mathit{lc}[i, j]$. As a base case, when we're computing $c[1]$, we need $c[0]$. If we set $c[0] = 0$, then $c[1] = \mathit{lc}[1, 1]$, which is what we want.

But of course we have to figure out which word begins the last line for the subproblem of words $1, \ldots, j$. So we try all possibilities for word i, and we pick the one that gives the lowest cost. Here, i ranges from 1 to j. Thus, we can define $c[j]$ recursively by

$$c[j] = \begin{cases} 0 & \text{if } j = 0 , \\ \min\limits_{1 \le i \le j} (c[i-1] + lc[i,j]) & \text{if } j > 0 . \end{cases}$$

Note that the way we defined lc ensures that

- all choices made will fit on the line (since an arrangement with $lc = \infty$ cannot be chosen as the minimum), and
- the cost of putting words $i, \ldots, j$ on the last line will not be 0 unless this really is the last line of the paragraph ($j = n$) or words $i \ldots j$ fill the entire line.

We can compute a table of c values from left to right, since each value depends only on earlier values.

To keep track of what words go on what lines, we can keep a parallel p table that points to where each c value came from. When $c[j]$ is computed, if $c[j]$ is based on the value of $c[k-1]$, set $p[j] = k$. Then after $c[n]$ is computed, we can trace the pointers to see where to break the lines. The last line starts at word $p[n]$ and goes through word n. The previous line starts at word $p[p[n]]$ and goes through word $p[n] - 1$, etc.

In pseudocode, here's how we construct the tables:

```
PRINT-NEATLY($l, n, M$)
let $extras[1..n, 1..n]$, $lc[1..n, 1..n]$, and $c[0..n]$ be new arrays
// Compute $extras[i,j]$ for $1 \le i \le j \le n$.
for $i = 1$ to $n$
    $extras[i,i] = M - l_i$
    for $j = i + 1$ to $n$
        $extras[i,j] = extras[i,j-1] - l_j - 1$
// Compute $lc[i,j]$ for $1 \le i \le j \le n$.
for $i = 1$ to $n$
    for $j = i$ to $n$
        if $extras[i,j] < 0$
            $lc[i,j] = \infty$
        elseif $j == n$ and $extras[i,j] \ge 0$
            $lc[i,j] = 0$
        else $lc[i,j] = (extras[i,j])^3$
// Compute $c[j]$ and $p[j]$ for $1 \le j \le n$.
$c[0] = 0$
for $j = 1$ to $n$
    $c[j] = \infty$
    for $i = 1$ to $j$
        if $c[i-1] + lc[i,j] < c[j]$
            $c[j] = c[i-1] + lc[i,j]$
            $p[j] = i$
return $c$ and $p$
```

Quite clearly, both the time and space are $\Theta(n^2)$.

In fact, we can do a bit better: we can get both the time and space down to $\Theta(nM)$. The key observation is that at most $\lceil M/2 \rceil$ words can fit on a line. (Each word is at least one character long, and there's a space between words.) Since a line with words $i, \ldots, j$ contains $j - i + 1$ words, if $j - i + 1 > \lceil M/2 \rceil$ then we know that $lc[i, j] = \infty$. We need only compute and store $extras[i, j]$ and $lc[i, j]$ for $j - i + 1 \le \lceil M/2 \rceil$. And the inner **for** loop header in the computation of $c[j]$ and $p[j]$ can run from $\max(1, j - \lceil M/2 \rceil + 1)$ to j.

We can reduce the space even further to $\Theta(n)$. We do so by not storing the *lc* and *extras* tables, and instead computing the value of $lc[i, j]$ as needed in the last loop. The idea is that we could compute $lc[i, j]$ in $O(1)$ time if we knew the value of $extras[i, j]$. And if we scan for the minimum value in *descending* order of i, we can compute that as $extras[i, j] = extras[i + 1, j] - l_i - 1$. (Initially, $extras[j, j] = M - l_j$.) This improvement reduces the space to $\Theta(n)$, since now the only tables we store are c and p.

Here's how we print which words are on which line. The printed output of GIVE-LINES(p, j) is a sequence of triples (k, i, j), indicating that words $i, \ldots, j$ are printed on line k. The return value is the line number k.

```
GIVE-LINES$(p, j)$
$i = p[j]$
if $i == 1$
    $k = 1$
else $k =$ GIVE-LINES$(p, i - 1) + 1$
print $(k, i, j)$
return $k$
```

The initial call is GIVE-LINES(p, n). Since the value of j decreases in each recursive call, GIVE-LINES takes a total of $O(n)$ time.