```
GIVE-LINES(p, j)
i ← p[j]
if i = 1
  then 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.

Solution to Problem 15-3

a. Dynamic programming is the ticket. This problem is slightly similar to the longest-common-subsequence problem. In fact, we'll define the notational conveniences X_i and Y_j in the similar manner as we did for the LCS problem: $X_i = x[1 \,..\, i]$ and $Y_j = y[1 \,..\, j]$.

Our subproblems will be determining an optimal sequence of operations that converts X_i to Y_j, for $0 \le i \le m$ and $0 \le j \le n$. We'll call this the "$X_i \to Y_j$ problem." The original problem is the $X_m \to Y_n$ problem.

Let's suppose for the moment that we know what was the last operation used to convert X_i to Y_j. There are six possibilities. We denote by $c[i, j]$ the cost of an optimal solution to the $X_i \to Y_j$ problem.

- If the last operation was a copy, then we must have had $x[i] = y[j]$. The subproblem that remains is converting X_{i-1} to Y_{j-1}. And an optimal solution to the $X_i \to Y_j$ problem must include an optimal solution to the $X_{i-1} \to Y_{j-1}$ problem. The cut-and-paste argument applies. Thus, assuming that the last operation was a copy, we have $c[i, j] = c[i-1, j-1] + \text{cost(copy)}$.
- If it was a replace, then we must have had $x[i] \ne y[j]$. (Here, we assume that we cannot replace a character with itself. It is a straightforward modification if we allow replacement of a character with itself.) We have the same optimal substructure argument as for copy, and assuming that the last operation was a replace, we have $c[i, j] = c[i-1, j-1] + \text{cost(replace)}$.
- If it was a twiddle, then we must have had $x[i] = y[j-1]$ and $x[i-1] = y[j]$, along with the implicit assumption that $i, j \ge 2$. Now our subproblem is $X_{i-2} \to Y_{j-2}$ and, assuming that the last operation was a twiddle, we have $c[i, j] = c[i-2, j-2] + \text{cost(twiddle)}$.
- If it was a delete, then we have no restrictions on x or y. Since we can view delete as removing a character from X_i and leaving Y_j alone, our subproblem is $X_{i-1} \to Y_j$. Assuming that the last operation was a delete, we have $c[i, j] = c[i-1, j] + \text{cost(delete)}$.
- If it was an insert, then we have no restrictions on x or y. Our subproblem is $X_i \to Y_{j-1}$. Assuming that the last operation was an insert, we have $c[i, j] = c[i, j-1] + \text{cost(insert)}$.

- If it was a kill, then we had to have completed converting X_m to Y_n, so that the current problem must be the $X_m \to Y_n$ problem. In other words, we must have $i = m$ and $j = n$. If we think of a kill as a multiple delete, we can get any $X_i \to Y_n$, where $0 \le i < m$, as a subproblem. We pick the best one, and so assuming that the last operation was a kill, we have

  $$c[m, n] = \min_{0 \le i < m} \{c[i, n]\} + \text{cost(kill)} .$$

We have not handled the base cases, in which $i = 0$ or $j = 0$. These are easy. X_0 and Y_0 are the empty strings. We convert an empty string into Y_j by a sequence of j inserts, so that $c[0, j] = j \cdot \text{cost(insert)}$. Similarly, we convert X_i into Y_0 by a sequence of i deletes, so that $c[i, 0] = i \cdot \text{cost(delete)}$. When $i = j = 0$, either formula gives us $c[0, 0] = 0$, which makes sense, since there's no cost to convert the empty string to the empty string.

For $i, j > 0$, our recursive formulation for $c[i, j]$ applies the above formulas in the situations in which they hold:

$$c[i, j] = \min \begin{cases} c[i-1, j-1] + \text{cost(copy)} & \text{if } x[i] = y[j] , \\ c[i-1, j-1] + \text{cost(replace)} & \text{if } x[i] \ne y[j] , \\ c[i-2, j-2] + \text{cost(twiddle)} & \text{if } i, j \ge 2, \\ & x[i] = y[j-1], \\ & \text{and } x[i-1] = y[j] , \\ c[i-1, j] + \text{cost(delete)} & \text{always} , \\ c[i, j] = c[i, j-1] + \text{cost(insert)} & \text{always} , \\ \min\limits_{0 \le i < m} \{c[i, n]\} + \text{cost(kill)} & \text{if } i = m \text{ and } j = n . \end{cases}$$

Like we did for LCS, our pseudocode fills in the table in row-major order, i.e., row-by-row from top to bottom, and left to right within each row. Column-major order (column-by-column from left to right, and top to bottom within each column) would also work. Along with the $c[i, j]$ table, we fill in the table $op[i, j]$, holding which operation was used.

```
EDIT-DISTANCE(x, y, m, n)
for i ← 0 to m
    do c[i, 0] ← i · cost(delete)
       op[i, 0] ← DELETE
for j ← 0 to n
    do c[0, j] ← j · cost(insert)
       op[0, j] ← INSERT
for i ← 1 to m
    do for j ← 1 to n
           do c[i, j] ← ∞
              if x[i] = y[j]
                then c[i, j] ← c[i − 1, j − 1] + cost(copy)
                     op[i, j] ← COPY
              if x[i] ≠ y[j] and c[i − 1, j − 1] + cost(replace) < c[i, j]
                then c[i, j] ← c[i − 1, j − 1] + cost(replace)
                     op[i, j] ← REPLACE(by y[j])
              if i ≥ 2 and j ≥ 2 and x[i] = y[j − 1] and
                     x[i − 1] = y[j] and
                     c[i − 2, j − 2] + cost(twiddle) < c[i, j]
                then c[i, j] ← c[i − 2, j − 2] + cost(twiddle)
                     op[i, j] ← TWIDDLE
              if c[i − 1, j] + cost(delete) < c[i, j]
                then c[i, j] ← c[i − 1, j] + cost(delete)
                     op[i, j] ← DELETE
              if c[i, j − 1] + cost(insert) < c[i, j]
                then c[i, j] ← c[i, j − 1] + cost(insert)
                     op[i, j] ← INSERT(y[j])
for i ← 0 to m − 1
    do if c[i, n] + cost(kill) < c[m, n]
         then c[m, n] ← c[i, n] + cost(kill)
              op[m, n] ← KILL i
return c and op
```

The time and space are both $\Theta(mn)$. If we store a KILL operation in $op[m, n]$, we also include the index i after which we killed, to help us reconstruct the optimal sequence of operations. (We don't need to store $y[i]$ in the op table for replace or insert operations.)

To reconstruct this sequence, we use the op table returned by EDIT-DISTANCE. The procedure OP-SEQUENCE(op, i, j) reconstructs the optimal operation sequence that we found to transform X_i into Y_j. The base case is when $i = j = 0$. The first call is OP-SEQUENCE(op, m, n).

```
OP-SEQUENCE(op, i, j)
if i = 0 and j = 0
  then return
if op[i, j] = COPY or op[i, j] = REPLACE
  then i' ← i − 1
       j' ← j − 1
elseif op[i, j] = TWIDDLE
  then i' ← i − 2
       j' ← j − 2
elseif op[i, j] = DELETE
  then i' ← i − 1
       j' ← j
elseif op[i, j] = INSERT        ▷ Don't care yet what character is inserted.
  then i' ← i
       j' ← j − 1
else            ▷ Must be KILL, and must have i = m and j = n.
       let op[i, j] = KILLk
       i' ← k
       j' ← j
OP-SEQUENCE(op, i', j')
print op[i, j]
```

This procedure determines which subproblem we used, recurses on it, and then prints its own last operation.

b. The DNA-alignment problem is just the edit-distance problem, with

$$
\begin{aligned}
\text{cost(copy)} &= -1 , \\
\text{cost(replace)} &= +1 , \\
\text{cost(delete)} &= +2 , \\
\text{cost(insert)} &= +2 ,
\end{aligned}
$$

and the twiddle and kill operations are not permitted.

The score that we are trying to maximize in the DNA-alignment problem is precisely the negative of the cost we are trying to minimize in the edit-distance problem. The negative cost of copy is not an impediment, since we can only apply the copy operation when the characters are equal.

Solution to Problem 15-6

Denote each square by the pair (i, j), where i is the row number, j is the column number, and $1 \le i, j \le n$. Our goal is to find a most profitable way from any square in row 1 to any square in row n. Once we do so, we can look up all the most profitable ways to get to any square in row n and pick the best one.

A subproblem is the most profitable way to get from some square in row 1 to a particular square (i, j). We have optimal substructure as follows. Consider a subproblem for (i, j), where $i > 1$, and consider the most profitable way to (i, j).