15 Dynamic Programming

15.1 Rod cutting

15.1-1

> Show that equation (15.4) follows from equation (15.3) and the initial condition $T(0) = 1$.

- For $n = 0$, this holds since $2^0 = 1$.
- For $n > 0$, substituting into the recurrence, we have

$$\begin{aligned} T(n) &= 1 + \sum_{j=0}^{n-1} 2^j \\ &= 1 + (2^n - 1) \\ &= 2^n. \end{aligned}$$

15.1-2

> Show, by means of a counterexample, that the following "greedy" strategy does not always determine an optimal way to cut rods. Define the ***density*** of a rod of length i to be p_i/i, that is, its value per inch. The greedy strategy for a rod of length n cuts off a first piece of length i, where $1 \le i \le n$, having maximum density. It then continues by applying the greedy strategy to the remaining piece of length $n - i$.

The counterexample:

length i	1	2	3	4
price p_i	1	20	33	36
p_i/i	1	10	11	9

15.1-3

> Consider a modification of the rod-cutting problem in which, in addition to a price p_i for each rod, each cut incurs a fixed cost of c. The revenue associated with a solution is now the sum of the prices of the pieces minus the costs of making the cuts. Give a dynamic-programming algorithm to solve this modified problem.

We can modify BOTTOM-UP-CUT-ROD algorithm from section 15.1 as follows:

```
MODIFIED-CUT-ROD(p, n, c)
    let r[0..n] be a new array
    r[0] = 0
    for j = 1 to n
        q = p[j]
        for i = 1 to j - 1
            q = max(q, p[i] + r[j - i] - c)
        r[j] = q
    return r[n]
```

We need to account for cost c on every iteration of the loop in lines 5-6 but the last one, when $i = j$ (no cuts).

We make the loop run to $j - 1$ instead of j, make sure c is subtracted from the candidate revenue in line 6, then pick the greater of current best revenue q and $p[j]$ (no cuts) in line 7.

15.1-4

> Modify MEMOIZED-CUT-ROD to return not only the value but the actual solution, too.

```
MEMOIZED-CUT-ROD(p, n)
    let r[0..n] and s[0..n] be new arrays
    for i = 0 to n
        r[i] = -∞
    (val, s) = MEMOIZED-CUT-ROD-AUX(p, n, r, s)
    print "The optimal value is" val "and the cuts are at" s
    j = n
    while j > 0
        print s[j]
        j = j - s[j]
```

```
MEMOIZED-CUT-ROD-AUX(p, n, r, s)
    if r[n] ≥ 0
        return r[n]
    if n == 0
        q = 0
    else q = -∞
        for i = 1 to n
            (val, s) = MEMOIZED-CUT-ROD-AUX(p, n - i, r, s)
            if q < p[i] + val
                q = p[i] + val
                s[n] = i
    r[n] = q
    return (q, s)
```

15.1-5

> The Fibonacci numbers are defined by recurrence (3.22). Give an $O(n)$-time dynamic-programming algorithm to compute the nth Fibonacci number. Draw the subproblem graph. How many vertices and edges are in the graph?

```
FIBONACCI(n)
    let fib[0..n] be a new array
    fib[0] = 1
    fib[1] = 1
    for i = 2 to n
        fib[i] = fib[i - 1] + fib[i - 2]
    return fib[n]
```

There are $n + 1$ vertices in the subproblem graph, i.e., $v_0, v_1, \ldots, v_n$.

- For v_0, v_1, each has 0 leaving edge.
- For $v_2, v_3, \ldots, v_n$, each has 2 leaving edges.

Thus, there are $2n - 2$ edges in the subproblem graph.

15.2 Matrix-chain multiplication

15.2-1

> Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$.

$$((5 \times 10)(10 \times 3))(((3 \times 12)(12 \times 5))((5 \times 50)(50 \times 6))).$$

15.2-2

> Give a recursive algorithm $\text{MATRIX-CHAIN-MULTIPLY}(A, s, i, j)$ that actually performs the optimal matrix-chain multiplication, given the sequence of matrices $\langle A_1, A_2, \ldots, A_n \rangle$, the s table computed by MATRIX-CHAIN-ORDER, and the indices i and j. (The initial call would be $\text{MATRIX-CHAIN-MULTIPLY}(A, s, 1, n)$.)

```
MATRIX-CHAIN-MULTIPLY(A, s, i, j)
    if i == j
        return A[i]
    if i + 1 == j
        return A[i] * A[j]
    b = MATRIX-CHAIN-MULTIPLY(A, s, i, s[i, j])
    c = MATRIX-CHAIN-MULTIPLY(A, s, s[i, j] + 1, j)
    return b * c
```

15.2-3

> Use the substitution method to show that the solution to the recurrence (15.6) is $\Omega(2^n)$.

Suppose $P(n) \geq c2^n$,

$$\begin{aligned} P(n) &\geq \sum_{k=1}^{n-1} c2^k \cdot c2^{n-k} \\ &= \sum_{k=1}^{n-1} c^2 2^n \\ &= c^2(n-1)2^n \\ &\geq c^2 2^n && (n > 1) \\ &\geq c2^n. && (c \geq 1) \end{aligned}$$

15.2-4

> Describe the subproblem graph for matrix-chain multiplication with an input chain of length n. How many vertices does it have? How many edges does it have, and which edges are they?

The vertices of the subproblem graph are the ordered pair v_{ij}, where $i \leq j$.

- If $i = j$, the vertex v_{ij} has no output edge.
- If $i < j$, for each k, s.t. $i \leq k < j$, the subproblem graph contains edges (v_{ij}, v_{ik}) and $(v_{ij}, v_{k+1,j})$, and these edges indicate that to solve the subproblem of optimally parenthesizing the product $A_i \cdots A_j$, we need to solve subproblems of optimally parenthesizing the products $A_i \cdots A_k$ and $A_{k+1} \cdots A_j$.

The number of vertices is

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

The number of edges is

$$\sum_{i=1}^{n} \sum_{j=i}^{n} (j-i) = \frac{(n-1)n(n+1)}{6}.$$

15.2-5

> Let $R(i, j)$ be the number of times that table entry $m[i, j]$ is referenced while computing other table entries in a call of MATRIX-CHAIN-ORDER. Show that the total number of references for the entire table is

> $$\sum_{i=1}^{n}\sum_{j=i}^{n} R(i,j) = \frac{n^3 - n}{3}.$$
>
> (Hint: You may find equation (A.3) useful.)

We count the number of times that we reference a different entry in m than the one we are computing, that is, 2 times the number of times that line 10 runs.

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

15.2-6

> Show that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses.

We proceed by induction on the number of matrices. A single matrix has no pairs of parentheses. Assume that a full parenthesization of an n-element expression has exactly $n-1$ pairs of parentheses. Given a full parenthesization of an $(n+1)$-element expression, there must exist some k such that we first multiply $B = A_1 \cdots A_k$ in some way, then multiply $C = A_{k+1} \cdots A_{n+1}$ in some way, then multiply B and C. By our induction hypothesis, we have $k-1$ pairs of parentheses for the full parenthesization of B and $n+1-k-1$ pairs of parentheses for the full parenthesization of C. Adding these together, plus the pair of outer parentheses for the entire expression, yields $k-1+n+1-k-1+1 = (n+1)-1$ parentheses, as desired.

15.3 Elements of dynamic programming

15.3-1

> Which is a more efficient way to determine the optimal number of multiplications in a matrix-chain multiplication problem: enumerating all the ways of parenthesizing the product and computing the number of multiplications for each, or running RECURSIVE-MATRIX-CHAIN? Justify your answer.

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 each approach:

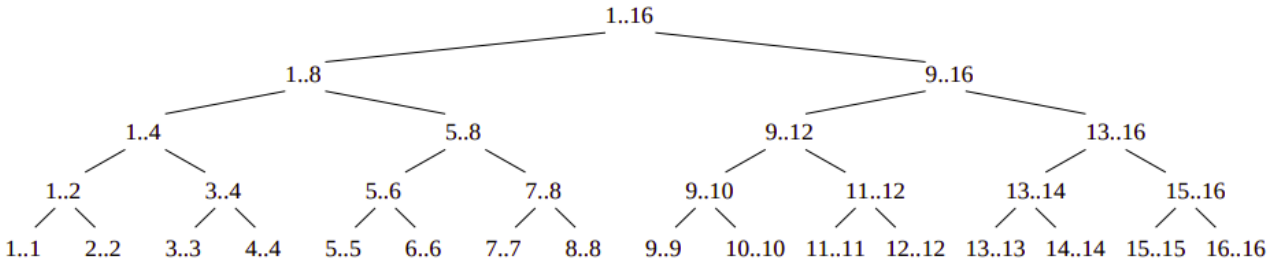
1. For each possible place to split the matrix chain, the enumeration approach finds all ways to parenthesize the left half, finds all ways to parenthesize the right half, and looks at all possible combinations of the left half with the right half. The amount of work to look at each combination of left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

2. 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 $O(1)$.

15.3-2

> Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MERGE-SORT.

Draw a recursion tree.



The MERGE-SORT procedure performs at most a single call to any pair of indices of the array that is being sorted. In other words, the subproblems do not overlap and therefore memoization will not improve the running time.

15.3-3

> Consider a variant of the matrix-chain multiplication problem in which the goal is to parenthesize the sequence of matrices so as to maximize, rather than minimize, the number of scalar multiplications. Does this problem exhibit optimal substructure?

Yes, this problem also exhibits optimal substructure. If we know that we need the subproduct $(A_l \cdot A_r)$, then we should still find the most expensive way to compute it — otherwise, we could do better by substituting in the most expensive way.

15.3-4

> As stated, in dynamic programming we first solve the subproblems and then choose which of them to use in an optimal solution to the problem. Professor Capulet claims that we do not always need to solve all the subproblems in order to find an optimal solution. She suggests that we can find an optimal solution to the matrix-chain multiplication problem by always choosing the matrix A_k at which to split the subproduct $A_i A_{i+1} \cdots A_j$ (by selecting k to minimize the quantity $p_{i-1} p_k p_j$) *before* solving the subproblems. Find an instance of the matrix-chain multiplication problem for which this greedy approach yields a suboptimal solution.

Suppose that we are given matrices A_1, A_2, A_3, and A_4 with dimensions such that

$$p_0, p_1, p_2, p_3, p_4 = 1000, 100, 20, 10, 1000.$$

Then $p_0 p_k p_4$ is minimized when $k = 3$, so we need to solve the subproblem of multiplying $A_1 A_2 A_3$, and also A_4 which is solved automatically. By her algorithm, this is solved by splitting at $k = 2$. Thus, the full parenthesization is $(((A_1 A_2) A_3) A_4)$.

This requires

$$1000 \cdot 100 \cdot 20 + 1000 \cdot 20 \cdot 10 + 1000 \cdot 10 \cdot 1000 = 12200000$$

scalar multiplications.

On the other hand, suppose we had fully parenthesized the matrices to multiply as $((A_1 (A_2 A_3)) A_4)$. Then we would only require

$$100 \cdot 20 \cdot 10 + 1000 \cdot 100 \cdot 10 + 1000 \cdot 10 \cdot 1000 = 11020000$$

scalar multiplications, which is fewer than Professor Capulet's method.

Therefore her greedy approach yields a suboptimal solution.

15.3-5

> Suppose that in the rod-cutting problem of Section 15.1, we also had limit l_i on the number of pieces of length i that we are allowed to produce, for $i = 1, 2, \ldots, n$. Show that the optimal-substructure property described in Section 15.1 no longer holds.

The optimal substructure property doesn't hold because the number of pieces of length i used on one side of the cut affects the number allowed on the other. That is, there is information about the particular solution on one side of the cut that changes what is allowed on the other.

To make this more concrete, suppose the rod was length 4, the values were $l_1 = 2, l_2 = l_3 = l_4 = 1$, and each piece has the same worth regardless of length. Then, if we make our first cut in the middle, we have that the optimal solution for the two rods left over is to cut it in the middle, which isn't allowed because it increases the total number of rods of length 1 to be too large.

15.3-6

> Imagine that you wish to exchange one currency for another. You realize that instead of directly exchanging one currency for another, you might be better off making a series of trades through other currencies, winding up with the currency you want. Suppose that you can trade n different currencies, numbered $1, 2, \ldots, n$, where you start with currency 1 and wish to wind up with currency n. You are given, for each pair of currencies i and j , an exchange rate r_{ij}, meaning that if you start with d units of currency i , you can trade for dr_{ij} units of currency j. A sequence of trades may entail a commission, which depends on the number of trades you make. Let c_k be the commission that you are charged when you make k trades. Show that, if $c_k = 0$ for all $k = 1, 2, \ldots, n$, then the problem of finding the best sequence of exchanges from currency 1 to currency n exhibits optimal substructure. Then show that if commissions c_k are arbitrary values, then the problem of finding the best sequence of exchanges from currency 1 to currency n does not necessarily exhibit optimal substructure.

First we assume that the commission is always zero. Let k denote a currency which appears in an optimal sequence s of trades to go from currency 1 to currency n. p_k denote the first part of this sequence which changes currencies from 1 to k and q_k denote the rest of the sequence. Then p_k and q_k are both optimal sequences for changing from 1 to k and k to n respectively. To see this, suppose that p_k wasn't optimal but that p'_k was. Then by changing currencies according to the sequence $p'_k q_k$ we would have a sequence of changes which is better than s, a contradiction since s was optimal. The same argument applies to q_k.

Now suppose that the commissions can take on arbitrary values. Suppose we have currencies 1 through 6, and $r_{12} = r_{23} = r_{34} = r_{45} = 2$, $r_{13} = r_{35} = 6$, and all other exchanges are such that $r_{ij} = 100$. Let $c_1 = 0$, $c_2 = 1$, and $c_k = 10$ for $k \geq 3$.

The optimal solution in this setup is to change 1 to 3, then 3 to 5, for a total cost of 13. An optimal solution for changing 1 to 3 involves changing 1 to 2 then 2 to 3, for a cost of 5, and an optimal solution for changing 3 to 5 is to change 3 to 4 then 4 to 5, for a total cost of 5. However, combining these optimal solutions to subproblems means making more exchanges overall, and the total cost of combining them is 18, which is not optimal.

15.4 Longest common subsequence

15.4-1

> Determine an LCS of $\langle 1, 0, 0, 1, 0, 1, 0, 1 \rangle$ and $\langle 0, 1, 0, 1, 1, 0, 1, 1, 0 \rangle$.

$\langle 1, 0, 0, 1, 1, 0 \rangle$ or $\langle 1, 0, 1, 0, 1, 0 \rangle$.

15.4-2

> Give pseudocode to reconstruct an LCS from the completed c table and the original sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ in $O(m + n)$ time, without using the b table.

```
PRINT-LCS(c, X, Y, i, j)
    if c[i, j] == 0
        return
    if X[i] == Y[j]
        PRINT-LCS(c, X, Y, i - 1, j - 1)
        print X[i]
    else if c[i - 1, j] > c[i, j - 1]
        PRINT-LCS(c, X, Y, i - 1, j)
    else
        PRINT-LCS(c, X, Y, i, j - 1)
```

15.4-3

> Give a memoized version of LCS-LENGTH that runs in $O(mn)$ time.

```
MEMOIZED-LCS-LENGTH(X, Y, i, j)
    if c[i, j] > -1
        return c[i, j]
    if i == 0 or j == 0
        return c[i, j] = 0
    if x[i] == y[j]
        return c[i, j] = LCS-LENGTH(X, Y, i - 1, j - 1) + 1
    return c[i, j] = max(LCS-LENGTH(X, Y, i - 1, j), LCS-LENGTH(X, Y, i, j - 1))
```

15.4-4

> Show how to compute the length of an LCS using only $2 \cdot \min(m, n)$ entries in the c table plus $O(1)$ additional space. Then show how to do the same thing, but using $\min(m, n)$ entries plus $O(1)$ additional space.

Since we only use the previous row of the c table to compute the current row, we compute as normal, but when we go to compute row k, we free row $k - 2$ since we will never need it again to compute the length. To use even less space, observe that to compute $c[i, j]$, all we need are the entries $c[i - 1, j]$, $c[i - 1, j - 1]$, and $c[i, j - 1]$. Thus, we can free up entry-by-entry those from the previous row which we will never need again, reducing the space requirement to $\min(m, n)$. Computing the next entry from the three that it depends on takes $O(1)$ time and space.

15.4-5

> Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

Given a list of numbers L, make a copy of L called L' and then sort L'.

```
PRINT-LCS(c, X, Y)
    n = c[X.length, Y.length]
    let s[1..n] be a new array
    i = X.length
    j = Y.length
    while i > 0 and j > 0
        if x[i] == y[j]
            s[n] = x[i]
            n = n - 1
            i = i - 1
            j = j - 1
        else if c[i - 1, j] ≥ c[i, j - 1]
            i = i - 1
        else j = j - 1
    for i = 1 to s.length
        print s[i]
```

```
MEMO-LCS-LENGTH-AUX(X, Y, c, b)
    m = |X|
    n = |Y|
    if c[m, n] ≠ 0 or m = 0 or n = 0
        return
    if x[m] = y[n]
        b[m, n] = ↖
        c[m, n] = MEMO-LCS-LENGTH-AUX(X[1..m - 1], Y[1..n - 1], c, b) + 1
    else if MEMO-LCS-LENGTH-AUX(X[1..m - 1], Y, c, b) ≥ MEMO-LCS-LENGTH-AUX(X, Y[1..n -
1], c, b)
        b[m, n] = ↑
        c[m, n] = MEMO-LCS-LENGTH-AUX(X[1..m - 1], Y, c, b)
    else
        b[m, n] = ←
        c[m, n] = MEMO-LCS-LENGTH-AUX(X, Y[1..n - 1], c, b)
```

```
MEMO-LCS-LENGTH(X, Y)
    let c[1..|X|, 1..|Y|] and b[1..|X|, 1..|Y|] be new tables
    MEMO-LCS-LENGTH-AUX(X, Y, c, b)
    return c and b
```

Then, just run the LCS algorithm on these two lists. The longest common subsequence must be monotone increasing because it is a subsequence of L' which is sorted. It is also the longest monotone increasing subsequence because being a subsequence of L' only adds the restriction that the subsequence must be monotone increasing. Since $|L| = |L'| = n$, and sorting L can be done in $o(n^2)$ time, the final running time will be $O(|L||L'|) = O(n^2)$.

15.4-6 ⋆

> Give an $O(n \lg n)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers. (Hint: Observe that the last element of a candidate subsequence of length i is at least as large as the last element of a candidate subsequence of length $i - 1$. Maintain candidate subsequences by linking them through the input sequence.)

The algorithm $\text{LONG-MONOTONIC}(A)$ returns the longest monotonically increasing subsequence of A, where A has length n.

The algorithm works as follows: a new array B will be created such that $B[i]$ contains the last value of a longest monotonically increasing subsequence of length i. A new array C will be such that $C[i]$ contains the monotonically increasing subsequence of length i with smallest last element seen so far.

To analyze the runtime, observe that the entries of B are in sorted order, so we can execute line 9 in $O(\lg n)$ time. Since every other line in the for-loop takes constant time, the total run-time is $O(n \lg n)$.

```
LONG-MONOTONIC(A)
    let B[1..n] be a new array where every value = ∞
    let C[1..n] be a new array
    L = 1
    for i = 1 to n
        if A[i] < B[1]
            B[1] = A[i]
            C[1].head.key = A[i]
        else
            let j be the largest index of B such that B[j] < A[i]
            B[j + 1] = A[i]
            C[j + 1] = C[j]
            INSERT(C[j + 1], A[i])
            if j + 1 > L
                L = L + 1
    print C[L]
```

15.5 Optimal binary search trees

15.5-1

Write pseudocode for the procedure CONSTRUCT-OPTIMAL-BST(root) which, given the table root, outputs the structure of an optimal binary search tree. For the example in Figure 15.10, your procedure should print out the structure

k_2 is the root
k_1 is the left child of k_2
d_0 is the left child of k_1
d_1 is the right child of k_1
k_5 is the right child of k_2
k_4 is the left child of k_5
k_3 is the left child of k_4
d_2 is the left child of k_3
d_3 is the right child of k_3
d_4 is the right child of k_4
d_5 is the right child of k_5

corresponding to the optimal binary search tree shown in Figure 15.9(b).

```
CONSTRUCT-OPTIMAL-BST(root, i, j, last)
    if i == j
        return
    if last == 0
        print root[i, j] + "is the root"
    else if j < last
        print root[i, j] + "is the left child of" + last
    else
        print root[i, j] + "is the right child of" + last
    CONSTRUCT-OPTIMAL-BST(root, i, root[i, j] - 1, root[i, j])
    CONSTRUCT-OPTIMAL-BST(root, root[i, j] + 1, j, root[i, j])
```

15.5-2

Determine the cost and structure of an optimal binary search tree for a set of $n = 7$ keys with the following probabilities

i	0	1	2	3	4	5	6	7
p_i		0.04	0.06	0.08	0.02	0.10	0.12	0.14
q_i	0.06	0.06	0.06	0.06	0.05	0.05	0.05	0.05

Cost is 3.12.

k_5 is the root
k_2 is the left child of k_5
k_1 is the left child of k_2
d_0 is the left child of k_1
d_1 is the right child of k_1
k_3 is the right child of k_2
d_2 is the left child of k_3
k_4 is the right child of k_3
d_3 is the left child of k_4
d_4 is the right child of k_4
k_7 is the right child of k_5
k_6 is the left child of k_7
d_5 is the left child of k_6
d_6 is the right child of k_6
d_7 is the right child of k_7

15.5-3

> Suppose that instead of maintaining the table $w[i,j]$, we computed the value of $w(i,j)$ directly from equation (15.12) in line 9 of OPTIMAL-BST and used this computed value in line 11. How would this change affect the asymptotic running time of OPTIMAL-BST?

Computing $w(i,j)$ from the equation is $\Theta(j-i)$, since the loop below on lines 10-14 is also $\Theta(j-i)$, it wouldn't affect the asymptotic running time of OPTIMAL-BST which would stay $\Theta(n^3)$.

15.5-4 ⋆

> Knuth [212] has shown that there are always roots of optimal subtrees such that $root[i,j-1] \le root[i,j] \le root[i+1,j]$ for all $1 \le i < j \le n$. Use this fact to modify the OPTIMAL-BST procedure to run in $\Theta(n^2)$ time.

Change the **for** loop of line 10 in OPTIMAL-BST to

```
for r = r[i, j - 1] to r[i + 1, j]
```

Knuth's result implies that it is sufficient to only check these values because optimal root found in this range is in fact the optimal root of some binary search tree. The time spent within the **for** loop of line 6 is now $\Theta(n)$. This is because the bounds on r in the new **for** loop of line 10 are nonoverlapping.

To see this, suppose we have fixed l and i. On one iteration of the **for** loop of line 6, the upper bound on r is

$$r[i+1,j] = r[i+1,i+l-1].$$

When we increment i by 1 we increase j by 1. However, the lower bound on r for the next iteration subtracts this, so the lower bound on the next iteration is

$$r[i+1,j+1-1] = r[i+1,j].$$

Thus, the total time spent in the **for** loop of line 6 is $\Theta(n)$. Since we iterate the outer **for** loop of line 5 n times, the total runtime is $\Theta(n^2)$.

Problem 15-1 Longest simple path in a directed acyclic graph

> Suppose that we are given a directed acyclic graph $G = (V, E)$ with real-valued edge weights and two distinguished vertices s and t. Describe a dynamic-programming approach for finding a longest weighted simple path from s to t. What does the subproblem graph look like? What is the efficiency of your algorithm?

Since any longest simple path must start by going through some edge out of s, and thereafter cannot pass through s because it must be simple, that is,

$$\text{LONGEST}(G, s, t) = 1 + \max_{s \sim s'} \{\text{LONGEST}(G|_{V \setminus \{s\}}, s', t)\},$$

with the base case that if $s = t$ then we have a length of 0.

A naive bound would be to say that since the graph we are considering is a subset of the vertices, and the other two arguments to the substructure are distinguished vertices, then, the runtime will be $O(|V|^2 2^{|V|})$. We can see that we will actually have to consider this many possible subproblems by taking $|G|$ to be the complete graph on $|V|$ vertices.

Problem 15-10 Planning an investment strategy

> Your knowledge of algorithms helps you obtain an exciting job with the Acme Computer Company, along with a $$10,000$ signing bonus. You decide to invest this money with the goal of maximizing your return at the end of 10 years. You decide to use the Amalgamated Investment Company to manage your investments. Amalgamated Investments requires you to observe the following rules. It offers n different investments, numbered 1 through n. In each year j, investment i provides a return rate of r_{ij} . In other words, if you invest d dollars in investment i in year j, then at the end of year j , you have dr_{ij} dollars. The return rates are guaranteed, that is, you are given all the return rates for the next 10 years for each investment. You make investment decisions only once per year. At the end of each year, you can leave the money made in the previous year in the same investments, or you can shift money to other investments, by either shifting money between existing investments or moving money to a new investement. If you do not move your money between two consecutive years, you pay a fee of f_1 dollars, whereas if you switch your money, you pay a fee of f_2 dollars, where $f_2 > f_1$.
>
> **a.** The problem, as stated, allows you to invest your money inmultiple investments in each year. Prove that there exists an optimal investment strategy that, in each year, puts all the money into a single investment. (Recall that an optimal investment strategy maximizes the amount of money after 10 years and is not concerned with any other objectives, such as minimizing risk.)
>
> **b.** Prove that the problem of planning your optimal investment strategy exhibits optimal substructure.
>
> **c.** Design an algorithm that plans your optimal investment strategy. What is the running time of your algorithm?
>
> **d.** Suppose that Amalgamated Investments imposed the additional restriction that, at any point, you can have no more than $$15,000$ in any one investment. Show that the problem of maximizing your income at the end of 10 years no longer exhibits optimal substructure.

a. Without loss of generality, suppose that there exists an optimal solution S which involves investing d_1 dollars into investment k and d_2 dollars into investement m in year 1. Further, suppose in this optimal solution, you don't move your money for the first j years. If $r_{k1} + r_{k2} + \ldots + r_{kj} > r_{m1} + r_{m2} + \ldots + r_{mj}$ then we can perform the usual cut-and-paste maneuver and instead invest $d_1 + d_2$ dollars into investment k for j years. Keeping all other investments the same, this results in a strategy which is at least as profitable as S, but has reduced the number of different investments in a given span of years by 1. Continuing in this way, we can reduce the optimal strategy to consist of only a single investment each year.

b. If a particular investment strategy is the year-one-plan for a optimal investment strategy, then we must solve two kinds of optimal suproblem: either we maintain the strategy for an additional year, not incurring the moneymoving fee, or we move the money, which amounts to solving the problem where we ignore all information from year 1. Thus, the problem exhibits optimal substructure.

c. The algorithm works as follows: We build tables I and R of size 10 such that $I[i]$ tells which investment should be made (with all money) in year i, and $R[i]$ gives the total return on the investment strategy in years i through 10.

```
INVEST(d, n)
    let I[1..10] and R[1..10] be new tables
    for k = 10 downto 1
        q = 1
        for i = 1 to n
            if r[i, k] > r[q, k]   // i now holds the investment which looks best for a
given year
                q = i
        if R[k + 1] + dr_{I[k + 1]k} - f[1] > R[k + 1] +  dr[q, k] - f[2]  // If revenue
is greater when money is not moved
            R[k] = R[k + 1] + dr_{I[k + 1]k} - f[1]
            I[k] = I[k + 1]
        else
            R[k] = R[k + 1] + dr[q, k] - f[2]
            I[k] = q
    return I as an optimal stategy with return R[1]
```

d. The previous investment strategy was independent of the amount of money you started with. When there is a cap on the amount you can invest, the amount you have to invest in the next year becomes relevant. If we know the year-one-strategy of an optimal investment, and we know that we need to move money after the first year, we're left with the problem of investing a different initial amount of money, so we'd have to solve a subproblem for every possible initial amount of money. Since there is no bound on the returns, there's also no bound on the number of subproblems we need to solve.

Problem 15-11 Inventory planning

> The Rinky Dink Company makes machines that resurface ice rinks. The demand for such products varies from month to month, and so the company needs to develop a strategy to plan its manufacturing given the fluctuating, but predictable, demand. The company wishes to design a plan for the next n months. For each month i, the company knows the demand d_i, that is, the number of machines that it will sell. Let $D = \sum_{i=1}^{n} d_i$ be the total demand over the next n months. The company keeps a full-time staff who provide labor to manufacture up to m machines per month. If the company needs to make more than m machines in a given month, it can hire additional, part-time labor, at a cost that works out to c dollars per machine. Furthermore, if, at the end of a month, the company is holding any unsold machines, it must pay inventory costs. The cost for holding j machines is given as a function $h(j)$ for $j = 1, 2, \ldots, D$, where $h(j) \geq 0$ for $1 \leq j \leq D$ and $h(j) \leq h(j+1)$ for $1 \leq j \leq D-1$.
>
> Give an algorithm that calculates a plan for the company that minimizes its costs while fulfilling all the demand. The running time should be polyomial in n and D.

Our subproblems will be indexed by and integer $i \in [n]$ and another integer $j \in [D]$. i will indicate how many months have passed, that is, we will restrict ourselves to only caring about $(d_i, \ldots, d_n)$. j will indicate how many machines we have in stock initially. Then, the recurrence we will use will try producing all possible numbers of machines from 1 to $[D]$. Since the index space has size $O(nD)$ and we are only running through and taking the minimum cost from D many options when computing a particular subproblem, the total runtime will be $O(nD^2)$.

Problem 15-12 Signing free-agent baseball players

> Suppose that you are the general manager for a major-league baseball team. During the off-season, you need to sign some free-agent players for your team. The team owner has given you a budget of
>
> $$X \$ to spend on free agents. You are allowed to spend less than$$
>
> X altogether, but the owner will fire you if you spend any more than X.
>
> You are considering N different positions, and for each position, P free-agent players who play that position are available. Because you do not want to overload your roster with too many players at any position, for each

> position you may sign at most one free agent who plays that position. (If you do not sign any players at a particular position, then you plan to stick with the players you already have at that position.)
>
> To determine how valuable a player is going to be, you decide to use a sabermetric statistic known as "VORP", or "value over replacement player". A player with a higher VORP is more valuable than a player with a lower VORP. A player with a higher VORP is not necessarily more expensive to sign than a player with a lower VORP, because factors other than a player's value determine how much it costs to sign him.
>
> For each available free-agent player, you have three pieces of information:
>
> - the player's position,
> - the amount of money it will cost to sign the player, and
> - the player's VORP.
>
> Devise an algorithm that maximizes the total VORP of the players you sign while spending no more than $$X$$ altogether. You may assume that each player signs for a multiple of $100,000$. Your algorithm should output the total VORP of the players you sign, the total amount of money you spend, and a list of which players you sign. Analyze the running time and space requirement of your algorithm.

We will make an $N+1$ by $X+1$ by $P+1$ table. The runtime of the algorithm is $O(NXP)$.

```
BASEBALL(N, X, P)
    initialize a table B of size (N + 1) by (X + 1)
    initialize an array P of length N
    for i = 0 to N
        B[i, 0] = 0
    for j = 1 to X
        B[0, j] = 0
    for i = 1 to N
        for j = 1 to X
            if j < i.cost
                B[i, j] = B[i - 1, j]
            q = B[i - 1, j]
            p = 0
            for k = 1 to P
                if j ⩾ i.cost
                    t = B[i - 1, j - i.cost] + i.value
                    if t > q
                        q = t
                        p = k
            B[i, j] = q
            P[i] = p
    print("The total VORP is", B[N, X], "and the players are:")
    i = N
    j = X
    C = 0
    for k = 1 to N // prints the players from the table
        if B[i, j] ≠ B[i - 1, j]
            print(P[i])
            j = j - i.cost
            C = C + i.cost
        i = i - 1
    print("The total cost is", C)
```

Problem 15-2 Longest palindrome subsequence

> A ***palindrome*** is a nonempty string over some alphabet that reads the same forward and backward. Examples of palindromes are all strings of length 1, civic, racecar, and aibohphobia (fear of palindromes).
>
> Give an efficient algorithm to find the longest palindrome that is a subsequence of a given input string. For example, given the input character, your algorithm should return carac. What is the running time of your

> algorithm?

Let $A[1..n]$ denote the array which contains the given word. First note that for a palindrome to be a subsequence we must be able to divide the input word at some position i, and then solve the longest common subsequence problem on $A[1..i]$ and $A[i+1..n]$, possibly adding in an extra letter to account for palindromes with a central letter. Since there are n places at which we could split the input word and the LCS problem takes time $O(n^2)$, we can solve the palindrome problem in time $O(n^3)$.

Problem 15-3 Bitonic euclidean

> In the ***euclidean traveling-salesman problem***, we are given a set of n points in the plane, and we wish to find the shortest closed tour that connects all n points. Figure 15.11(a) shows the solution to a 7-point problem. The general problem is NP-hard, and its solution is therefore believed to require more than polynomial time (see Chapter 34).
>
> J. L. Bentley has suggested that we simplify the problem by restricting our attention to ***bitonic tours***, that is, tours that start at the leftmost point, go strictly rightward to the rightmost point, and then go strictly leftward back to the starting point. Figure 15.11(b) shows the shortest bitonic tour of the same 7 points. In this case, a polynomial-time algorithm is possible.
>
> Describe an $O(n^2)$-time algorithm for determining an optimal bitonic tour. You may assume that no two points have the same x-coordinate and that all operations on real numbers take unit time. ($Hint$: Scan left to right, maintaining optimal possibilities for the two parts of the tour.)

First sort all the points based on their x coordinate. To index our subproblem, we will give the rightmost point for both the path going to the left and the path going to the right. Then, we have that the desired result will be the subproblem indexed by v, where v is the rightmost point.

Suppose by symmetry that we are further along on the left-going path, that the leftmost path is going to the ith one and the right going path is going until the jth one. Then, if we have that $i > j + 1$, then we have that the cost must be the distance from the $i-1$ st point to the ith plus the solution to the subproblem obtained where we replace i with $i-1$. There can be at most $O(n^2)$ of these subproblem, but solving them only requires considering a constant number of cases. The other possibility for a subproblem is that $j \leq i \leq j+1$. In this case, we consider for every k from 1 to j the subproblem where we replace i with k plus the cost from kth point to the ith point and take the minimum over all of them. This case requires considering $O(n)$ things, but there are only $O(n)$ such cases. So, the final runtime is $O(n^2)$.

Problem 15-4 Printing neatly

> Consider the problem of neatly printing a paragraph with a monospaced font (all characters having the same width) on a printer. The input text is a sequence of n words of lengths $l_1, l_2, \ldots, l_n$, measured in characters. We want to print this paragraph neatly on a number of lines that hold a maximum of M characters each. Our criterion of "neatness" is as follows. If a given line contains words i through j, where $i \leq j$, and we leave exactly one space between words, the number of extra space characters at the end of the line is $M - j + i - \sum_{k=i}^{j} l_k$, which must be nonnegative so that the words fit on the line. We wish to minimize the sum, over all lines except the last, of the cubes of the numbers of extra space characters at the ends of lines. Give a dynamic-programming algorithm to print a paragraph of n words neatly on a printer. Analyze the running time and space requirements of your algorithm.

First observe that the problem exhibits optimal substructure in the following way: Suppose we know that an optimal solution has k words on the first line. Then we must solve the subproblem of printing neatly words $l_{k+1}, \ldots, l_n$. We build a table of optimal solutions solutions to solve the problem using dynamic programming. If $n - 1 + \sum_{k=1}^{n} l_k < M$ then put all words on a single line for an optimal solution. In the following algorithm $PRINT\text{-}NEATLY(n)$, $C[k]$ contains the cost of printing neatly words l_k through l_n. We can determine the cost of an optimal solution upon termination by examining $C[1]$. The entry $P[k]$ contains the position of the last word which should appear on the first line of the optimal solution of words $l_k, \ldots, l_n$. Thus, to obtain the optimal way to place the words, we make $l_{P[1]}$ the last word on the first line, $l_{P[P[1]]}$ the last word on the second line, and so on.

```
PRINT-NEATLY(n)
    let P[1..n] and C[1..n] be new tables
    for k = n downto 1
        if sum_{i = k}^n l_i + n - k < M
            C[k] = 0
        q = ∞
        for j = 1 to n - k
            cost = sum_{m = 1}^j l_{k + m} + m - 1
            if cost < M and (M - cost)^3 + C[k + m + 1] < q
                q = (M - cost)^3 + C[k + m + 1]
                P[k] = k + j
        C[k] = q
```

Problem 15-5 Edit distance

In order to transform one source string of text $x[1..m]$ to a target string $y[1..n]$, we can perform various transformation operations. Our goal is, given x and y, to produce a series of transformations that change x to y. We use an array z—assumed to be large enough to hold all the characters it will need—to hold the intermediate results. Initially, z is empty, and at termination, we should have $z[j] = y[j]$ for $j = 1, 2, \ldots, n$. We maintain current indices i into x and j into z, and the operations are allowed to alter z and these indices. Initially, $i = j = 1$. We are required to examine every character in x during the transformation, which means that at the end of the sequence of transformation operations, we must have $i = m + 1$.

We may choose from among six transformation operations:

Copy a character from x to z by setting $z[j] = x[i]$ and then incrementing both i and j. This operation examines $x[i]$.

Replace a character from x by another character c, by setting $z[j] = c$, and then incrementing both i and j. This operation examines $x[i]$.

Delete a character from x by incrementing i but leaving j alone. This operation examines $x[i]$.

Insert the character c into z by setting $z[j] = c$ and then incrementing j, but leaving i alone. This operation examines no characters of x.

Twiddle (i.e., exchange) the next two characters by copying them from x to z but in the opposite order; we do so by setting $z[j] = x[i + 1]$ and $z[j + 1] = x[i]$ and then setting $i = i + 2$ and $j = j + 2$. This operation examines $x[i]$ and $x[i + 1]$.

Kill the remainder of x by setting $i = m + 1$. This operation examines all characters in x that have not yet been examined. This operation, if performed, must be the final operation.

As an example, one way to transform the source string algorithm to the target string altruistic is to use the following sequence of operations, where the underlined characters are $x[i]$ and $z[j]$ after the operation:

Operation	x	z
initial strings	algorithm	\textunderscore
copy	algorithm	a\textunderscore
copy	algorithm	al\textunderscore
replace by t	algorithm	alt\textunderscore
delete	algorithm	alt\textunderscore
copy	algorithm	altr\textunderscore
insert u	algorithm	altru\textunderscore
insert i	algorithm	altrui\textunderscore
insert s	algorithm	altruis\textunderscore
twiddle	algorithm	altruisti\textunderscore
insert c	algorithm	altruistic\textunderscore
kill	algorithm\textunderscore	altruistic\textunderscore

Note that there are several other sequences of transformation operations that transform algorithm to altruistic.

Each of the transformation operations has an associated cost. The cost of an operation depends on the specific application, but we assume that each operation's cost is a constant that is known to us. We also assume that the individual costs of the copy and replace operations are less than the combined costs of the delete and insert operations; otherwise, the copy and replace operations would not be used. The cost of a given sequence of transformation operations is the sum of the costs of the individual operations in the sequence. For the sequence above, the cost of transforming algorithm to altruistic is

$$(3 \cdot \text{cost(copy)}) + \text{cost(replace)} + \text{cost(delete)} + (4 \cdot \text{cost(insert)}) + \text{cost(twiddle)} + \text{cost(kill)}.$$

a. Given two sequences $x[1..m]$ and $y[1..n]$ and set of transformation-operation costs, the ***edit distance*** from x to y is the cost of the least expensive operatoin sequence that transforms x to y. Describe a dynamic-programming algorithm that finds the edit distance from $x[1..m]$ to $y[1..n]$ and prints an optimal opeartion sequence. Analyze the running time and space requirements of your algorithm.

The edit-distance problem generalizes the problem of aligning two DNA sequences (see, for example, Setubal and Meidanis [310, Section 3.2]). There are several methods for measuring the similarity of two DNA sequences by aligning them. One such method to align two sequences x and y consists of inserting spaces at arbitrary locations in the two sequences (including at either end) so that the resulting sequences x' and y' have the same length but do not have a space in the same position (i.e., for no position j are both $x'[j]$ and $y'[j]$ a space). Then we assign a "score" to each position. Position j receives a score as follows:

- $+1$ if $x'[j] = y'[j]$ and neither is a space,
- -1 if $x'[j] \neq y'[j]$ and neither is a space,
- -2 if either $x'[j]$ or $y'[j]$ is a space.

The score for the alignment is the sum of the scores of the individual positions. For example, given the sequences $x = \text{GATCGGCAT}$ and $y = \text{CAATGTGAATC}$, one alignment is

$$ \begin{array}{ccccccccccc} \text G & & \text A & \text T & \text C & \text G & & \text G & \text C & \text A & \text T & \\ \text C & \text A & \text A & \text T & & \text G & \text T & \text G & \text A & \text A & \text T & \text C \\

- ```
    & *       & +       & +       & *       & +       & *       & +       & -
  & +       & +       & *
  ```

\end{array} $$

A $+$ under a position indicates a score of $+1$ for that position, a $-$ indicates a score of -1, and a $*$ indicates a score of -2, so that this alignment has a total score of $6 \cdot -2 \cdot 1 - 4 \cdot 2 = -4$.
  ```

> **b.** Explain how to cast the problem of finding an optimal alignment as an edit distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

a. We will index our subproblems by two integers, $1 \le i \le m$ and $1 \le j \le n$. We will let i indicate the rightmost element of x we have not processed and j indicate the rightmost element of y we have not yet found matches for. For a solution, we call $\text{EDIT}(x, y, i, j)$.

b. We will set

$$\text{cost(delete)} = \text{cost(insert)} = 2,$$

$$\text{cost(copy)} = -1,$$

$$\text{cost(replace)} = 1,$$

and

$$\text{cost(twiddle)} = \text{cost(kill)} = \infty.$$

Then a minimum cost translation of the first string into the second corresponds to an alignment.

We view

- a copy or a replace as incrementing a pointer for both strings,
- a insert as putting a space at the current position of the pointer in the first string, and
- a delete operation means putting a space in the current position in the second string.

Since twiddles and kills have infinite costs, we will have neither of them in a minimal cost solution. The final value for the alignment will be the negative of the minimum cost sequence of edits.

```
EDIT(x, y, i, j)
    let m = x.length
    let n = y.length
    if i == m
        return (n - j)cost(insert)
    if j == n
        return min{(m - i)cost(delete), cost(kill)}
    initialize o1, ..., o5 to ∞
    if x[i] == y[j]
        o1 = cost(copy) + EDIT(x, y, i + 1, j + 1)
    o2 = cost(replace) + EDIT(x, y, i + 1, j + 1)
    o3 = cost(delete) + EDIT(x, y, i + 1, j)
    o4 = cost(insert) + EDIT(x, y, i, j + 1)
    if i < m - 1 and j < n - 1
        if x[i] == y[j + 1] and x[i + 1] == y[j]
            o5 = cost(twiddle) + EDIT(x, y, i + 2, j + 2)
    return min_{i ∈ [5]}{o_i}
```

Problem 15-6 Planning a company party

> Professor Stewart is consulting for the president of a corporation that is planning a company party. The company has a hierarchical structure; that is, the supervisor relation forms a tree rooted at the president. The personnel office has ranked each employee with a conviviality rating, which is a real number. In order to make the party fun for all attendees, the president does not want both an employee and his or her immediate supervisor to attend.
>
> Professor Stewart is given the tree that describes the structure of the corporation, using the left-child, right-sibling representation described in Section 10.4. Each node of the tree holds, in addition to the pointers, the name of an employee and that employee's conviviality ranking. Describe an algorithm to make up a guest list that maximizes the sum of the conviviality ratings of the guests. Analyze the running time of your algorithm.

The problem exhibits optimal substructure in the following way: If the root r is included in an optimal solution, then we must solve the optimal subproblems rooted at the grandchildren of r. If r is not included, then we must solve the optimal subproblems on trees rooted at the children of r. The dynamic programming algorithm to solve this problem works as follows: We make a table C indexed by vertices which tells us the optimal conviviality ranking of a guest list obtained from the subtree with root at that vertex. We also make a table G such that $G[i]$ tells us the guest list we would use when vertex i is at the root. Let T be the tree of guests. To solve the problem, we need to examine the guest list stored at $G[T.root]$. First solve the problem at each leaf L. If the conviviality ranking at L is positive, $G[L] = \{L\}$ and $C[L] = L.conviv$. Otherwise $G[L] = \emptyset$ and $C[L] = 0$. Iteratively solve the subproblems located at parents of nodes at which the subproblem has been solved. In general for a node x,

- [x] = \max(\sum_{y\text{ is a child of } x} C[y], x.conviv + \sum_{y\text{ is a grandchild of } x} C[y]).$$

The runtime is $O(n)$ since each node appears in at most two of the sums (because each node has at most 1 parent and 1 grandparent) and each node is solved once.

Problem 15-7 Viterbi algorithm

> We can use dynamic programming on a directed graph $G = (V, E)$ for speech recognition. Each edge $(u, v) \in E$ is labeled with a sound $\sigma(u, v)$ from a finite set Σ of sounds. The labeled graph is a formal model of a person speaking a restricted language. Each path in the graph starting from a distinguished vertex $v_0 \in V$ corresponds to a possible sequence of sounds producted by the model. We define the label of a directed path to be the concatenation of the labels of the edges on that path.
>
> **a.** Describe an efficient algorithm that, given an edge-labeled graph G with distinguished vertex v_0 and a sequence $s = \langle \sigma_1, \sigma_2, \ldots, \sigma_k \rangle$ of sounds from Σ, returns a path in G that begins at v_0 and has s as its label, if any such path exists. Otherwise, the algorithm should return NO-SUCH-PATH. Analyze the running time of your algorithm. (Hint: You may find concepts from Chapter 22 useful.)
>
> Now, suppose that every edge $(u, v) \in E$ has an associated nonnegatve probability $p(u, v)$ of traversing the edge (u, v) from vertex u and thus producing the corresponding sound. The sum of the probabilities of the edges leaving any vertex equals 1. The probability of a path is defined to the product of the probabilities of its edges. We can view the probability of a path beginning at v_0 as the probability that a "random walk" beginning at v_0 will follow the specified path, where we randomly choose which edge to take leaving a vertex u according to the probabilities of the available edges leaving u.
>
> **b.** Extend your answer to part (a) so that if a path is returned, it is a *most probable path* starting at v_0 and having label s. Analyze the running time of your algorithm.

a. Our substructure will consist of trying to find suffixes of s of length one less starting at all the edges leaving v_0 with label σ_0. if any of them have a solution, then, there is a solution. If none do, then there is none. See the algorithm VITERBI for details.

```
VITERBI(G, s, v[0])
    if s.length = 0
        return v[0]
    for edges(v[0], v[1]) in V for some v[1]
        if sigma(v[0], v[1]) = sigma[1]
            res = VITERBI(G, (sigma[2], ..., sigma[k]), v[1])
            if res ≠ NO-SUCH-PATH
                return (v[0], res)
    return NO-SUCH-PATH
```

Since the subproblems are indexed by a suffix of s (of which there are only k) and a vertex in the graph, there are at most $O(k|V|)$ different possible arguments. Since each run may require testing a edge going to every other vertex, and each iteration of the **for** loop takes at most a constant amount of time other than the call to PROB-VITERBI, the final runtime is $O(k|V|^2)$.

b. For this modification, we will need to try all the possible edges leaving from v_0 instead of stopping as soon as we find one that works. The substructure is very similar. We'll make it so that instead of just returning the sequence, we'll have the algorithm also return the probability of that maximum probability sequence, calling the fields seq and prob respectively. See the algorithm PROB-VITERBI.

Since the runtime is indexed by the same things, we have that we will call it with at most $O(k|V|)$ different possible arguments. Since each run may require testing a edge going to every other vertex, and each iteration of the **for** loop takes at most a constant amount of time other than the call to PROB-VITERBI, the final runtime is $O(k|V|^2)$.

```
PROB-VITERBI(G, s, v[0])
    if s.length = 0
        return v[0]
    sols.seq = NO-SUCH-PATH
    sols.prob = 0
    for edges(v[0], v[1]) in V for some v[1]
        if sigma(v[0], v[1]) = sigma[1]
            res = PROB-VITERBI(G, (sigma[2], ..., sigma[k]), v[1])
            if p(v[0], v[1]) * res.prob ≥ sols.prob
                sols.prob = p(v[0], v[1]) * res.prob and sols.seq = v[0], res.seq
    return sols
```

Problem 15-8 Image compression by seam carving

> We are given a color picture consisting of an $m \times n$ array $A[1..m, 1..n]$ of pixels, where each pixel specifies a triple of red, green, and blue (RGB) intensities. Suppose that we wish to compress this picture slightly. Specifically, we wish to remove one pixel from each of the m rows, so that the whole picture becomes one pixel narrower. To avoid disturbing visual effects, however, we require that the pixels removed in two adjacent rows be in the same or adjacent columns; the pixels removed form a "seam" from the top row to the bottom row where successive pixels in the seam are adjacent vertically or diagonally.
>
> **a.** Show that the number of such possible seams grows at least exponentially in m, assuming that $n > 1$.
>
> **b.** Suppose now that along with each pixel $A[i,j]$, we have calculated a real-valued disruption measure $d[i,j]$, indicating how disruptive it would be to remove pixel $A[i,j]$. Intuitively, the lower a pixel's disruption measure, the more similar the pixel is to its neighbors. Suppose further that we define the disruption measure of a seam to be the sum of the disruption measures of its pixels.
>
> Give an algorithm to find a seam with the lowest disruption measure. How efficient is your algorithm?

a. If $n > 1$ then for every choice of pixel at a given row, we have at least 2 choices of pixel in the next row to add to the seam (3 if we're not in column 1 or n). Thus the total number of possibilities is bounded below by 2^m.

b. We create a table $D[1..m, 1..n]$ such that $D[i,j]$ stores the disruption of an optimal seam ending at position $[i,j]$, which started in row 1. We also create a table $S[i,j]$ which stores the list of ordered pairs indicating which pixels were used to create the optimal seam ending at position (i,j). To find the solution to the problem, we look for the minimum k entry in row m of table D, and use the list of pixels stored at $S[m,k]$ to determine the optimal seam. To simplify the algorithm $\text{Seam}(A)$, let $\text{MIN}(a,b,c)$ be the function which returns -1 if a is the minimum, 0 if b is the minimum, and 1 if c is the minimum value from among a, b, and c. The time complexity of the algorithm is $O(mn)$.

```
SEAM(A)
    let D[1..m, 1..n] be a table with zeros
    let S[1..m, 1..n] be a table with empty lists
    for i = 1 to n
        S[1, i] = (1, i)
        D[1, i] = d_{1i}
    for i = 2 to m
        for j = 1 to n
            if j == 1 // left-edge case
                if D[i - 1, j] < D[i - 1, j + 1]
```

```
                    D[i, j] = D[i - 1, j] + d_{ij}
                    S[i, j] = S[i - 1, j].insert(i, j)
                else
                    D[i, j] = D[i - 1, j + 1] + d_{ij}
                    S[i, j] = S[i - 1, j + 1].insert(i, j)
            else if j == n // right-edge case
                if D[i - 1, j - 1] < D[i - 1, j]
                    D[i, j] = D[i - 1, j - 1] + d_{ij}
                    S[i, j] = S[i - 1, j - 1].insert(i, j)
                else
                    D[i, j] = D[i - 1, j] + d_{ij}
                    S[i, j] = S[i - 1, j].insert(i, j)
            x = MIN(D[i - 1, j - 1], D[i - 1, j], D[i - 1, j + 1])
            D[i, j] = D[i - 1, j + x]
            S[i, j] = S[i - 1, j + x].insert(i, j)
    q = 1
    for j = 1 to n
        if D[m, j] < D[m, q]
            q = j
    print(S[m, q])
```

Problem 15-9 Breaking a string

> A certain string-processing language allows a programmer to break a string into two pieces. Because this operation copies the string, it costs n time units to break a string of n characters into two pieces. Suppose a programmer wants to break a string into many pieces. The order in which the breaks occur can affect the total amount of time used. For example, suppose that the programmer wants to break a 20-character string after characters 2, 8, and 10 (numbering the characters in ascending order from the left-hand end, starting from 1). If she programs the breaks to occur in left-to-right order, then the first break costs 20 time units, the second break costs 18 time units (breaking the string from characters 3 to 20 at character 8), and the third break costs 12 time units, totaling 50 time units. If she programs the breaks to occur in right-to-left order, however, then the first break costs 20 time units, the second break costs 10 time units, and the third break costs 8 time units, totaling 38 time units. In yet another order, she could break first at 8 (costing 20), then break the left piece at 2 (costing 8), and finally the right piece at 10 (costing 12), for a total cost of 40.
>
> Design an algorithm that, given the numbers of characters after which to break, determines a least-cost way to sequence those breaks. More formally, given a string S with n characters and an array $L[1..m]$ containing the break points, com- pute the lowest cost for a sequence of breaks, along with a sequence of breaks that achieves this cost.

The subproblems will be indexed by contiguous subarrays of the arrays of cuts needed to be made. We try making each possible cut, and take the one with cheapest cost. Since there are m to try, and there are at most m^2 possible things to index the subproblems with, we have that the m dependence is that the solution is $O(m^3)$. Also, since each of the additions is of a number that is $O(n)$, each of the iterations of the for loop may take time $O(\lg n + \lg m)$, so, the final runtime is $O(m^3 \lg n)$. The given algorithm will return $(\text{cost}, \text{seq})$ where cost is the cost of the cheapest sequence, and seq is the sequence of cuts to make.

```
CUT-STRING(L, i, j, l, r)
    if l == r
        return (0, [])
    minCost = ∞
    for k = i to j
        if l + r + CUT-STRING(L, i, k, l, L[k]).cost + CUT-STRING(L, k, j, L[k], j).cost
< minCost
            minCost = r - l + CUT-STRING(L, i, k, l, L[k]).cost + CUT-STRING(L, k + 1, j,
L[k], j).cost
            minSeq = L[k] + CUT-STRING(L, i, k, l, L[k]) + CUT-STRING(L, i, k + 1, l,
L[k])
    return (minCost, minSeq)
```

Sample call: ``cpp L = [3, 8, 10] S = 20 CUT-STRING(L, 0, len(L), 0, s)