Chapter 15

Michelle Bodnar, Andrew Lohr

April 12, 2016

Exercise 15.1-1

Proceed by induction. The base case of $T(0) = 2^0 = 1$. Then, we apply the inductive hypothesis and recall equation (A.5) to get that

$$T(n) = 1 + \sum_{j=0}^{n-1} T(j) = 1 + \sum_{j=0}^{n-1} 2^j = 1 + \frac{2^n - 1}{2 - 1} = 1 + 2^n - 1 = 2^n$$

Exercise 15.1-2

Let $p_1 = 0$, $p_2 = 4$, $p_3 = 7$ and $n = 4$. The greedy strategy would first cut off a piece of length 3 since it has highest density. The remaining rod has length 1, so the total price would be 7. On the other hand, two rods of length 2 yield a price of 8.

Exercise 15.1-3

Now, instead of equation (15.1), we have that

$$r_n = \max\{p_n, r_1 + r_{n-1} - c, r_2 + r_{n-2} - c, \ldots, r_{n-1} + r_1 - c\}$$

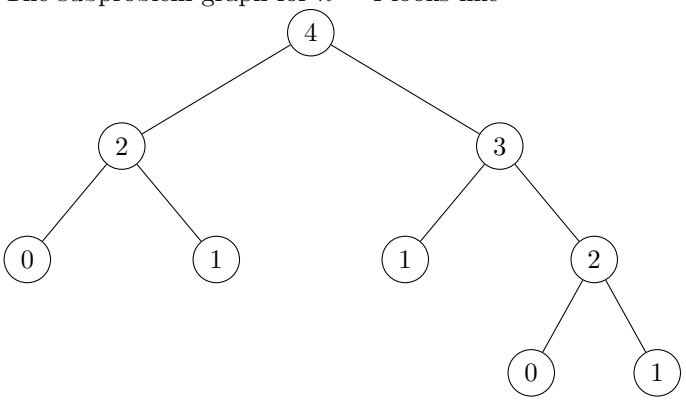
And so, to change the top down solution to this problem, we would change MEMOIZED-CUT-ROD-AUX(p,n,r) as follows. The upper bound for i on line 6 should be $n - 1$ instead of n.Also, after the for loop, but before line 8, set $q = \max\{q - c, p[i]\}$.

Exercise 15.1-4

Create a new array called s. Initialize it to all zeros in MEMOIZED-CUT-ROD(p,n) and pass it as an additional argument to MEMOIZED-CUT-ROD-AUX(p,n,r,s). Replace line 7 in MEMOIZED-CUT-ROD-AUX by the following: $t = p[i] + MEMOIZED - CUT - ROD - AUX(p, n - i, r, s)$. Following this, if $t > q$, set $q = t$ and $s[n] = i$. Upon termination, $s[i]$ will contain the size of the first cut for a rod of size i.

Exercise 15.1-5

The subproblem graph for $n = 4$ looks like



The number of vertices in the tree to compute the nth Fibonacci will follow the recurrence

$$V(n) = 1 + V(n-2) + V(n-1)$$

And has initial condition $V(1) = V(0) = 1$. This has solution $V(n) = 2 * Fib(n) - 1$ which we will check by direct substitution. For the base cases, this is simple to check. Now, by induction, we have

$$V(n) = 1 + 2 * Fib(n-2) - 1 + 2 * Fib(n-1) - 1 = 2 * Fib(n) - 1$$

The number of edegs will satisfy the recurrence

$$E(n) = 2 + E(n-1) + E(n-2)$$

and having base cases $E(1) = E(0) = 0$. So, we show by induction that we have $E(n) = 2 * Fib(n) - 2$. For the base cases it clearly holds, and by induction, we have

$$E(n) = 2 + 2 * Fib(n-1) - 2 + 2 * Fib(n-2) - 2 = 2 * Fib(n) - 2$$

We will present a $O(n)$ bottom up solution that only keeps track of the the two largest subproblems so far, since a subproblem can only depend on the solution to subproblems at most two less for Fibonacci.

Exercise 15.2-1

An optimal parenthesization of that sequence would be $(A_1A_2)((A_3A_4)(A_5A_6))$ which will require $5 * 50 * 6 + 3 * 12 * 5 + 5 * 10 * 3 + 3 * 5 * 6 + 5 * 3 * 6 = 1500 + 180 + 150 + 90 + 90 = 2010$.

Exercise 15.2-2

Algorithm 1 DYN-FIB(n)

```
prev = 1
prevprev = 1
if n ≤ 1 then
    return 1
end if
for i=2 upto n do
    tmp = prev + prevprev
    prevprev = prev
    prev = tmp
end for
return prev
```

Algorithm 2 MATRIX-CHAIN-MULTIPLY(A,s,i,j)

```
if i == j then
    Return A_i
end if
Return MATRIX-CHAIN-MULTIPLY(A,s,i,s[i,j]) · MATRIX-CHAIN-
MULTIPLY(A,s,s[i,j]+1,j)
```

The following algorithm actually performs the optimal multiplication, and is recursive in nature:

Exercise 15.1-3

By indution we will show that $P(n)$ from eq (15.6) is $\geq 2^n - 1 \in \Omega(2^n)$. The base case of n=1 is trivial. Then, for $n \geq 2$, by induction and eq (15.6), we have

$$P(n) = \sum_{k=1}^{n-1} P(k)P(n-k) \geq \sum_{k=1}^{n-1} 2^k 2^{n-k} = (n-1)(2^n - 1) \geq 2^n - 1$$

So, the conclusion holds.

Exercise 15.2-4

The subproblem graph for matrix chain multiplication has a vertex for each pair (i, j) such that $1 \leq i \leq j \leq n$, corresponding to the subproblem of finding the optimal way to multiply $A_i A_{i+1} \cdots A_j$. There are $n(n-1)/2 + n$ vertices. Vertex (i, j) is connected by an edge directed to vertex (k, l) if $k = i$ and $k \leq l < j$ or $l = j$ and $i < k \leq j$. A vertex (i, j) has outdegree $2(j - i)$. There are $n - k$ vertices such that $j - i = k$, so the total number of edges is

$$\sum_{k=0}^{n-1} 2k(n-k).$$

Exercise 15.1-5

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.

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

Exercise 15.2-6

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.

Exercise 15.3-1

The runtime of of enumerating is just $n * P(n)$, while is we were running RECURSIVE-MATRIX-CHAIN, it would also have to run on all of the internal nodes of the subproblem tree. Also, the enumeration approach wouldn't have as much overhead.

Exercise 15.3-2

Let $[i..j]$ denote the call to Merge Sort to sort the elements in positions i through j of the original array. The recursion tree will have $[1..n]$ as its root, and at any node $[i..j]$ will have $[i..(j-i)/2]$ and $[(j-i)/2+1..j]$ as its left

and right children, respectively. If $j - i = 1$, there will be no children. The memoization approach fails to speed up Merge Sort because the subproblems aren't overlapping. Sorting one list of size n isn't the same as sorting another list of size n, so there is no savings in storing solutions to subproblems since each solution is used at most once.

Exercise 15.3-3

This modification of the matrix-chain-multiplication problem does still exhibit the optimal substructure property. Suppose we split a maximal multiplication of $A_1, \ldots, A_n$ between A_k and A_{k+1} then, we must have a maximal cost multiplication on either side, otherwise we could substitute in for that side a more expensive multiplication of $A_1, \ldots, A_n$.

Exercise 15.3-4

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 $1000 \cdot 100 \cdot 20 + 1000 \cdot 20 \cdot 10 + 1000 \cdot 10 \cdot 1000 = 12,200,000$ scalar multiplications. On the other hand, suppose we had fully parenthesized the matrices to multiply as $((A_1(A_2A_3))A_4)$. Then we would only require $100 \cdot 20 \cdot 10 + 1000 \cdot 100 \cdot 10 + 1000 \cdot 10 \cdot 1000 = 11,020,000$ scalar multiplications, which is fewer than Professor Capulet's method. Therefore her greedy approach yields a suboptimal solution.

Exercise 15.3-5

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.

Exercise 15.3-6

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.

Exercise 15.4-1

An LCS is $\langle 1, 0, 1, 0, 1, 0 \rangle$. A concise way of seeing this is by noticing that the first list contains a "00" while the second contains none, Also, the second list contains two copies of "11" while the first contains none. Inorder to reconcile this, any LCS will have to skip at least three elements. Since we managed to do this, we know that our common subsequence was maximal.

Exercise 15.4-2

The algorithm PRINT-LCS(c,X,Y) prints the LCS of X and Y from the completed table by computing only the necessary entries of B on the fly. It runs in $O(m + n)$ time because each iteration of the while loop decrements either i or j or both by 1, and halts when either reaches 0. The final for loop iterates at most $\min(m, n)$ times.

Exercise 15.4-3

Exercise 15.4-4

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.

Exercise 15.4-5

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

Algorithm 3 PRINT-LCS(c,X,Y)

$n = c[X.length, Y.length]$
Initialize an array s of length n
$i = X.length$ and $j = Y.length$
while $i > 0$ and $j > 0$ **do**
    if $x_i == y_j$ **then**
        $s[n] = x_i$
        $n = n - 1$
        $i = i - 1$
        $j = j - 1$
    else if $c[i-1, j] \geq c[i, j-1]$ **then**
        $i = i - 1$
    else
        $j = j - 1$
    end if
end while
for $k = 1$ to $s.length$ **do**
    Print $s[k]$
end for

Algorithm 4 MEMO-LCS-LENGTH-AUX(X,Y,c,b)

m = $|X|$
n = $|Y|$
if $c[m, n]! = 0$ or $m == 0$ or $n == 0$ **then**
    return
end if
if $x_m == y_n$ **then**
    $b[m, n] = \nwarrow$
    c[m,n] =MEMO-LCS-LENGTH-AUX(X[1,. . . , m-1],Y[1,. . . ,n-1],c,b) +1
else if $MEMO - LCS - LENGTH - AUX(X[1, \ldots, m-1], Y, c, b) \geq MEMO - LCS - LENGTH - AUX(X, Y[1, \ldots, n-1], c, b)$ **then**
    $b[m, n] = \uparrow$
    c[m,n] =MEMO-LCS-LENGTH-AUX(X[1,. . . , m-1],Y,c,b)
else
    $b[m, n] = \leftarrow$
    c[m,n] =MEMO-LCS-LENGTH-AUX(X,Y[1,. . . ,n-1],c,b)
end if

Algorithm 5 MEMO-LCS-LENGTH(X,Y)

let c be a (passed by reference) $|X|$ by $|Y|$ array initiallized to 0
let b be a (passed by reference) $|X|$ by $|Y|$ array
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 $O(|L||L'|) = O(n^2)$.

Exercise 15.4-6

The algorithm LONG-MONOTONIC(S) returns the longest monotonically increasing subsequence of S, where S 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 $O(\log(n))$ time. Since every other line in the for-loop takes constant time, the total run-time is $O(n \log n)$.

Algorithm 6 LONG-MONOTONIC(S)

```
1:  Initialize an array B of integers length of n, where every value is set equal
    to ∞.
2:  Initialize an array C of empty lists length n.
3:  L = 1
4:  for i = 1 to n do
5:      if A[i] < B[1] then
6:          B[1] = A[i]
7:          C[1].head.key = A[i]
8:      else
9:          Let j be the largest index of B such that B[j] < A[i]
10:         B[j + 1] = A[i]
11:         C[j + 1] = C[j]
12:         C[j + 1].insert(A[i])
13:         if j + 1 > L then
14:             L = L + 1
15:         end if
16:     end if
17: end for
18: Print C[L]
```

Exercise 15.5-1

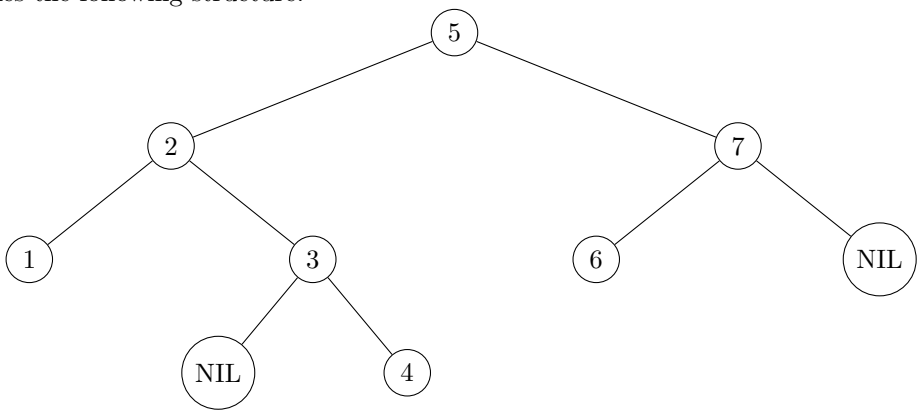
Run the given algorithm with the initial argument of $i = 1$ and $j = m[1].length$.

Exercise 15.5-2

Algorithm 7 CONSTRUCT-OPTIMAL-BST(root,i,j)

```
if i > j then
    return nil
end if
if i == j then
    return a node with key k_i and whose children are nil
end if
let n be a node with key k_{root[i,j]}
n.left = CONSTRUCT-OPTIMAL-BST(root,i,root[i,j]-1)
n.right = CONSTRUCT-OPTIMAL-BST(root,root[i,j]+1,j)
return n
```

After painstakingly working through the algorithm and building up the tables, we find that the cost of the optimal binary search tree is 3.12. The tree takes the following structure:



Exercise 15.5-3

Each of the $\Theta(n^2)$ values of $w[i, j]$ would require computing those two sums, both of which can be of size $O(n)$, so, the asymptotic runtime would increase to $O(n^3)$.

Exercise 15.5-4

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

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,

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

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 can actually will have to consider this many possible subproblems by taking $|G|$ to be the complete graph on $|V|$ vertices.

Problem 15-2

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

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,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

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 Printing-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 $l_1, l_2, \ldots, l_n$. 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.

Algorithm 8 Printing-Neatly(n)

```
1: Let P[1..n] and C[1..n] be a new tables.
2: for k = n downto 1 do
3:     if Σ_{i=k}^{n} l_i + n − k < M then
4:         C[k] = 0
5:     end if
6:     q = ∞
7:     for j = 1 downto n − k do
8:         if Σ_{m=1}^{j} l_{k+j} + j − 1 < M and (M − Σ_{m=1}^{j} l_{k+j} + j − 1) + C[k+j+1] < q
   then
9:             q = (M − Σ_{m=1}^{j} l_{k+j} + j − 1) + C[k + j + 1]
10:            P[k] = k + j
11:        end if
12:    end for
13:    C[k] = q
14: end for
```

Problem 15-5

a. We will index our subproblems by two integers, $1 \leq i \leq m$ and $1 \leq j \leq n$. 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 $EDIT(x, y, i, j)$

b. We will set $cost(delete) = cost(insert) = 2$, $cost(copy) = -1$, $cost(replace) = 1$, and $cost(twiddle) = cost(kill) = \infty$. Then a minimum cost translation of the first string into the second corresponds to an alignment. where 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. A

Algorithm 9 EDIT(x,y,i,j)

let $m = x.length$ and $n = y.length$
if $i = m$ **then**
  return (n-j)cost(insert)
end if
if $j = n$ **then**
  return $\min\{(m-i)cost(delete), cost(kill)\}$
end if
$o_1, \ldots, o_5$ initialized to ∞
if x[i] = y[j] **then**
  $o_1 = cost(copy) + EDIT(x, y, i+1, j+1)$
end if
$o_2 = cost(replace) + EDIT(x, y, i+1, j+1)$
$o_3 = cost(delete) + EDIT(x, y, i+1, j)$
$o_4 = cost(insert) + EDIT(x, y, i, j+1)$
if $i < m-1$ and $j < n-1$ **then**
  if $x[i] = y[j+1]$ and $x[i+1] = y[j]$ **then**
    $o_5 = cost(twiddle) + EDIT(x, y, i+2, j+2)$
  end if
end if
return $\min_{i \in [5]}\{o_i\}$

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.

Problem 15-6

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,

$$C[x] = \min \left(\sum_{y \text{ is a child of } x} C[y], \sum_{y \text{ is a grandchild of } x} C[y] \right).$$

The runtime of the algorithm is $O(n^2)$ where n is the number of vertices, because we solve n subproblems, each in constant time, but the tree traversals required to find the appropriate next node to solve could take linear time.

Problem 15-7

a. Our substructure will consist of trying to find suffixes of s of length one less starting at all the edges leaving ν_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.

Algorithm 10 $VITERBI(G, s, \nu_0)$

```
if s.length = 0 then
    return ν_0
end if
for edges (ν_0, ν_1) ∈ V for some ν_1 do
    if σ(ν_0, ν_1) = σ_1 then
        res = VITERBI(G, (σ_2, ..., σ_k), ν_1)
        if res != NO-SUCH-PATH then
            return ν_0, res
        end if
    end if
end for
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 ν_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

Algorithm 11 $PROB-VITERBI(G, s, \nu_0)$

```
if s.length = 0 then
    return ν_0
end if
let sols.seq = NO − SUCH − PATH, and sols.prob = 0
for edges (ν_0, ν_1) ∈ V for some ν_1 do
    if σ(ν_0, ν_1) = σ_1 then
        res = PROB − VITERBI(G, (σ_2, ..., σ_k), ν_1)
        if p(ν_0, ν_1) · res.prob >= sols.prob then
            sols.prob = p(ν_0, ν_1) · res.prob and sols.seq = ν_0, res.seq
        end if
    end if
end for
return sols
```

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 $O(k|V|^2)$

Problem 15-8

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 Seam(A), let $MIN(a, b, c)$ 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 $O(mn)$.

Problem 15-9

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

Algorithm 12 Seam(A)

```
Initialize tables D[1..m, 1..n] of zeros and S[1..m, 1..n] of empty lists
for i = 1 to n do
    S[1, i] = (1, i)
    D[1, i] = d_1i
end for
for i = 2 to m do
    for j = 1 to n do
        if j == 1 then //Handles the left-edge case
            if D[i − 1, j] < D[i − 1, j + 1] then
                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)
            end if
        else if j == n then //Handles the right-edge case
            if D[i − 1, j − 1] < D[i − 1, j] then
                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)
            end if
        end if
        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)
    end for
end for
q = 1
for j = 1 to n do
    if D[m, j] < D[m, q] then q = j
    end if
end for
Print the list stored at S[m, q].
```

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 (cost,seq) where cost is the cost of the cheapest sequence, and seq is the sequence of cuts to make

Algorithm 13 CUT-STRING(L,i,j,l,r)

if $l = r$ **then**
  return (0,[])
end if
$mincost = \infty$
for k from i to j **do**
  if $l + r + CUT - STRING(L, i, k, l, L[k]).cost + CUT - STRING(L, k, j, L[k], j).cost < mincost$ **then**
    $mincost = l + r + CUT - STRING(L, i, k, l, L[k]).cost + CUT - STRING(L, k, j, L[k], j).cost$
    $minseq = L[k]$ concatenated with the sequence returned from$CUT - STRING(L, i, k, l, L[k])$ and from $CUT - STRING(L, i, k, l, L[k])$
  end if
end for
return (mincost,minseq)

Problem 15-10

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 money-moving 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.

Algorithm 14 Invest(d,n)

Initialize tables I and R of size 11, all filled with zeros
for $k = 10$ downto 1 **do**
  $q = 1$
  for $i = 1$ to n **do**
    if $r_{ik} > r_{qk}$ **then** // i now holds the investment which looks best for a given year
      $q = i$
    end if
  end for
  if $R[k+1] + dr_{I[k+1]k} - f_1 > R[k+1] + dr_{qk} - f_2$ **then** //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
    $R[k] = R[k+1] + dr_{qk} - f_2$
    $I[k] = q$
  end if
end for
Return I as an optimal strategy 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

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

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

Algorithm 15 Baseball(N,X,P)

```
Initialize an N+1 by X+1 table B
Initialize an array P of length N
for i = 0 to N do
    B[i,0] = 0
end for
for j = 1 to X do
    B[0,j] = 0
end for
for i = 1 to N do
    for j = 1 to X do
        if j < i.cost then
            B[i,j] = B[i-1,j]
        end if
        q = B[i-1,j]
        p = 0
        for k = 1 to p do
            if B[i-1,j-i.cost] + i.value > q then
                q = B[i-1,j-i.cost] + i.value
                p = k
            end if
        end for
        B[i,j] = q
        P[i] = p
    end for
end for
Print: The total VORP is B[N,X] and the players are:
i = N
j = X
C = 0
for k = 1 to N do //Prints the players from the table
    if B[i,j] ≠ B[i-1,j] then
        Print P[i]
        j = j - i.cost
        C = C + i.cost
    end if
    i = i - 1
end for
Print: The total cost is C
```