1. Problem 15.2-1.
   I calculated these out by hand. It was tedious, but kind of fun, in a "a computer should really be doing this" sort of way.

					2010							
				1655			1950					
			405			2430			1770			
		330			330			930			1860	
	150			360			180			3000			1500	
0		0			0			0			0		0
						2						
					4		2					
				2		2		4				
			2		2		4		4			
		1		2		3		4		5		

   We're looking for an optimal parenthesization of $A_1A_2A_3A_4A_5A_6$. The result pyramid shows us that the first split is after A_2, so we have $(A_1A_2)(A_3A_4A_5A_6)$. Recursing on the list 3 through 6, the pyramid shows the optimal split is after A_4, so the final result is $(A_1A_2)((A_3A_4)(A_5A_6))$. This makes sense because to reduce the number of multiplications we want to eliminate the large matrix dimensions (50, between A_5 and A_6; 12, between A_3 and A_4; and 10, between A_1 and A_2) as soon as possible.

2. Problem 15.5-2.
   For this one I wrote a short Java program using the pseudocode in the book.

						3.12								
					2.44			2.61						
				1.83			1.96			2.13				
			1.34			1.41			1.48			1.55		
		1.02			0.93			1.04			1.01			1.20	
	0.62			0.68			0.57			0.57			0.72			0.78	
0.28			0.30			0.32			0.24			0.30			0.32			0.34
							5							
						3		5						
					3		5		5					
				2		3		5		6				
			2		3		4		5		6			
		2		3		3		5		6		7		
	1		2		3		4		5		6		7	

   The apex of the pyramid is 5, so the root is k_5. It two children are the trees defined by the subpyramids with bases 1 through 4 and 6 through 7. The apex of the left one

is 2 and the apex of the right one is 7, so k_2 and k_7 are the children of k_5. Likewise, k_1 and k_3 are the children of k_2, and k_6 is the left child of k_7. k_3 has right child k_4. The rest of the tree is finished with d_0 through d_7, from left to right, as leaves.

3. Problem 15-2.
The problem defines "neatness," but what we really want to minimize is the "sloppiness"—the sum of the cubes of the leftover space on each line. So we define $s(i)$, the minimum sloppiness of printing words i through n in a paragraph. To define it recursively, we put e words on the current line and then compute $s(i+e)$. We don't know what the optimal e is, so we try all possibilities that will fit on the line. Thus $s(i) = \min_e\{s(i+e) + (M - (e-1) - \sum_{k=i}^{i+e-1} l_k)^3\}$, where the min is over all possible values of e such that $(e-1) + \sum_{k=i}^{i+e-1} l_k \le M$. The base case is the last line, which is free: $s(i) = 0$ if $(n-i) + \sum_{k=i}^{n} l_k \le M$. The dynamic programming algorithm builds up from the base case:

Input: l, n, M
Output: s, the sloppiness values; w, the number of words to print on a line

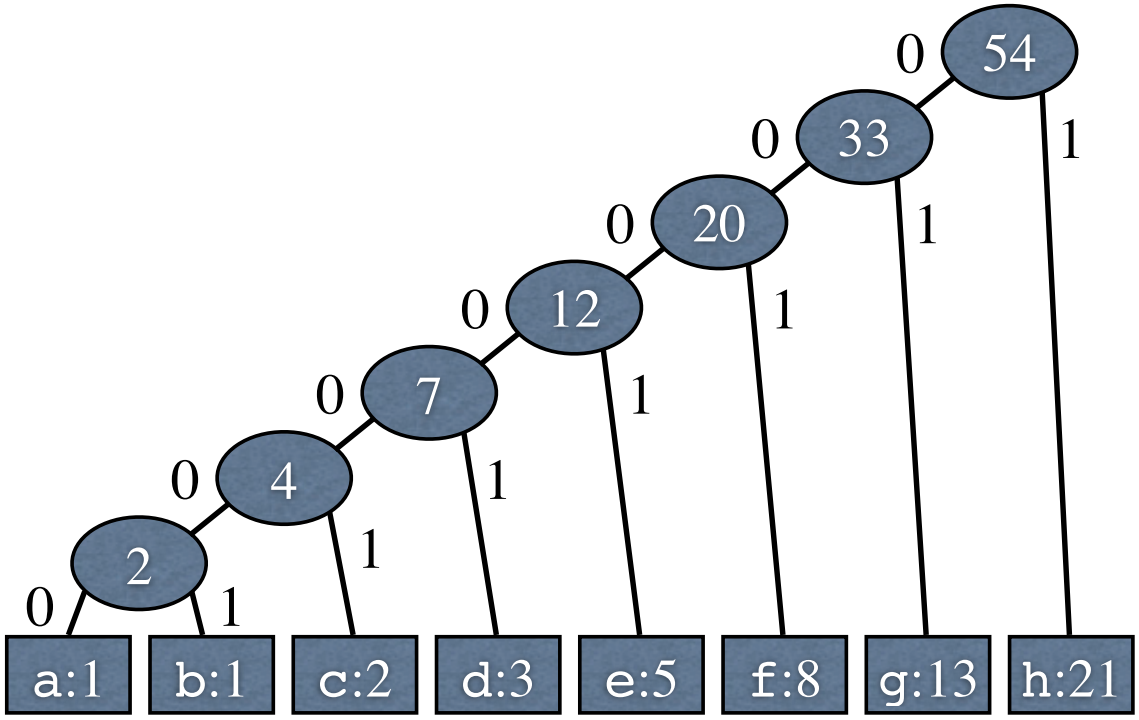
```
1: i ← n
2: while (n − i) + Σ_{k=i}^{n} l_k ≤ M do
3:     s[i] ← 0
4:     w[i] ← n − i + 1
5:     Decrement i
6: while i > 0 do
7:     e ← 1
8:     s[i] ← ∞
9:     while (e − 1) + Σ_{k=i}^{i+e−1} l_k ≤ M do
10:        s′ ← s[i + e] + (M − (e − 1) − Σ_{k=i}^{i+e−1} l_k)^3
11:        if s′ < s[i] then
12:            s[i] ← s′
13:            w[i] ← e
14:        Increment e
15:    Decrement i
16: return s and w
```

Input: sequence of words, w, n
Output: the neatly-printed paragraph

```
1: i ← 1
2: while i ≤ n do
3:     w′ = w[i]
4:     for e ← 1 to w′ do
5:         Print word i
6:         Increment i
7:     Start new line
```

Now we analyze the time and space complexity of this algorithm. We're storing an integer (the sloppiness value) and a number between 1 and $M/2$ (the number of words that will fit on a line) for each word. This is $\Theta(n)$ space. The running time is determined by the nested **while** loops. The outer loop runs n times in the worst case. The number of times the inner loop runs is bounded by the number of words that can fit on a line; this is bounded by $M/2$ and by n. So the total running time is $\Theta(n \min\{n, M/2\})$. Presumably $n > M/2$ for most inputs, so we can safely say the running time is $O(nM)$. However, the sums of word lengths $\sum l_k$ also take time to compute, up to the number of words possible on a line. We can compute this on the fly for an additional factor of $\min\{n, M/2\}$ time or precompute them for additional storage of $n \min\{n, M/2\}$. So the required resources are either $O(nM)$ space and $O(nM)$ time or $O(n)$ space and $O(nM^2)$ time.

4. Problem 16.3-2.



Let f_n be the nth Fibonacci number, with $f_1 = f_2 = 1$. I will show by induction that $\sum_{i=1}^{n} f_i = f_{n+2} - 1$. Base case: $f_1 + f_2 = 1 + 1 = 2 = 3 - 1 = f_4 - 1$. Now assume the hypothesis holds for all positive integers less than n. Then $\sum_{i=1}^{n} f_i = f_1 + f_2 + \sum_{i=3}^{n}(f_{i-1} + f_{i-2}) = 2 + \sum_{i=3}^{n} f_{i-1} + \sum_{i=3}^{n} f_{i-2} = 2 + \sum_{i=1}^{n-1} f_i - f_1 + \sum_{i=1}^{n-2} f_i = 1 + f_{n+1} - 1 + f_n - 1 = f_{n+2} - 1$. Therefore, when constructing the Huffman tree, the current sum of frequencies will always be less than all the remaining leaves except the smallest, and so the smallest leaf will always be combined with a running total. Thus the code will be a single 1 for the most frequent letter and $n - 1$ 0s for the least

frequent letter, and k 0s followed by a 1 for the letters in between, for $k = 1, \ldots, n-2$, from the most to the least frequent.

Problem 16.3-3.
The concept here is not really as complicated as I make it look below. For some reason I just felt like I had to write it up formally. The basic idea is that if you draw a path from a leaf to the root in the tree, the character for the leaf gets counted once for each of the nodes on that path (excluding the leaf itself). The definition of the cost on page 386 and the alternate computation described in this problem each has a different way of counting this same number. Now, the formal proof: The total cost of a tree T for a code is defined as $B(T) = \sum_{c \in C} f(c)d_T(c)$, where C is the alphabet. We must prove that the cost of the tree is also equal to $B'(T) = \sum_{n \in I} (f(left[n]) + f(right[n]))$, where I is the set of internal nodes of T. The question refers to the frequency of a node in the tree. This was not explicitly defined in the text, but they mean that if n is a leaf node corresponding to a character c, $f(n) = f(c)$, and if n is an internal node, $f(n) = f(left[n]) + f(right[n])$. This boils down to the fact that the frequency of a node n is the sum of the frequencies of all of n's decendant characters. Now we are ready to prove $B'(T) = B(T)$, which we do by induction. First, if T is a single node, then $B(T) = 0$, since $d_T(c) = 0$ for the only node in the tree. $B'(T) = 0$ in this case as well, since there are no internal nodes to sum over. Now let T be a code tree of height k. Let T_1 and T_2 be the left and right branches of T, respectively. Since the heights of these two are each strictly less than k, we may assume inductively that $B'(T_1) = B(T_1)$ and $B'(T_2) = B(T_2)$. Let r be the root node of T. Also let C_i and I_i be the characters in the tree T_i and the internal nodes of T_i, respectively. Thus, we have $B'(T) = \sum_{n \in I} (f(left[n]) + f(right[n])) = (f(left[r]) + f(right[r])) + \sum_{n \in I_1} (f(left[n]) + f(right[n])) + \sum_{n \in I_2} (f(left[n]) + f(right[n])) = f(left[r]) + f(right[r]) + B'(T_1) + B'(T_2) = f(left[r]) + f(right[r]) + B(T_1) + B(T_2) = f(left[r]) + f(right[r]) + \sum_{c \in C_1} f(c)d_{T_1}(c) + \sum_{c \in C_2} f(c)d_{T_2}(c) = f(left[r]) + f(right[r]) + \sum_{c \in C_1} f(c)(d_T(c) - 1) + \sum_{c \in C_2} f(c)(d_T(c) - 1) = \sum_{c \in C_1} f(c) + \sum_{c \in C_2} f(c) + \sum_{c \in C} f(c)d_T(c) - \sum_{c \in C_1} f(c) - \sum_{c \in C_2} f(c) = B(T)$. This completes the proof.

5. Problem 23.1-1.
We want to show that a minimum-weight edge e in a graph G is in some minimum-weight spanning tree of G. Let T be a MST of G. If T contains e, we are done. Otherwise, add e to T. This forms a cycle. Since e is a minumum-weight edge in G, there is an edge $f \neq e$ on the cycle that has $w(f) \geq w(e)$. Remove f to form a spanning tree T'. Then $w(T) \geq w(T')$ and T is minimum, so T' must also be minimum, and it contains e.

   Problem 23.1-8.
   I got some inspiration for this proof from Prof. Huang's solutions from USC. Suppose L is not the sorted list of edges of T'; let L' be the sorted list. Then L and L' differ somewhere; let k be the index of the first place they differ (starting from the minimum).

Suppose that $L[k] < L'[k]$ (for the other case the proof proceeds the same). Consider the $k-1$ weights the two trees have in common. The corresponding edges in T' form a forest F'. Any of the first k edges of T, when added to F', may or may not form a cycle, but a cycle can only be formed when both vertices of the edge belong to the same component of F'. Also, a single component of F' with m edges can only have at most m of the k edges of T, or else a cycle would exist in T. But F' has only $k-1$ edges total, so it can only account for at most $k-1$ edges of T. Therefore, there must be at least one of the first k edges of T that does not form a cycle when added to F'. Add this edge e to T'. This forms a cycle, and furthermore, some of the edges in this cycle were not contained in F' and therefore have weight greater than or equal to $L'[k]$, by the construction of F'. But e has weight strictly less than $L'[k]$, by the choice of k. Therefore, we can remove an edge of weight greater than e and decrease the total weight of T'. This contradicts the assumption that T' was minimum. Thus two minimum spanning trees must have the same weight sequences.