Solutions to Chapters 15 & 16

15.1-1(5-1) Show how to modify the PRINT-STATIONS procedure given below to print out the situations in increasing order of station number.

```
PRINT-STATIONS(l, n)
1   i ← l*
2   print "line" i ", station" n
3   for j ←n downto 2
4       do i ←li[j]
5          print "line" i ", station" j – 1
```

Solution:

```
RECURSIVE-PRINT-STATIONS (l, i, j)
1   if j =0 then return
2   RECURSIVE-PRINT-STATIONS (l, li[j], j – 1)
3   print "line" i ", station" j
```

To print out all the stations, call PRINT-STATIONS (*l*, *l**, *n*)

15.2-1(5-5) Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is <5, 10, 3, 12, 5, 50, 6>.

Solution: Solve the matrix chain order for a specific problem. This can be done by computing MATRIX-CHAIN-ORDER(*p*) where *p* = <5, 10, 3, 12, 5, 50, 6>or simply using the equation:

$$m[i, j] = \begin{cases} 0 & \text{if } i = j, \\ \min_{i \le k < j}\{\min[i,k] + m[k+1, j] + p_{i-1} p_k p_j\} & \text{if } i < j. \end{cases}$$

The table is computed simply by the fact that *m*[*i*, *i*] = 0 for all *i*. This information is used to compute *m*[*i*, *i* + 1] for *i* = 1, …, *n* – 1 an so on.
The resulting table is the following:

The *m*-table:
m[1, 2] = 150, *m*[2, 3] = 360, *m*[3, 4] = 180, *m*[4, 5] = 3000, *m*[5, 6] = 1500
m[1, 3] = 330, *m*[2, 4] = 330, *m*[3, 5] = 930, *m*[4, 6] = 1860
m[1, 4] = 405, *m*[2, 5] = 2430, *m*[3, 6] = 1770
m[1, 5] = 1655, *m*[2, 6] = 1950
m[1, 6] = 2010

The *s*-table:
s[1, 2] = 1, *s*[2, 3] = 2, *s*[3, 4] = 3, *s*[4, 5] = 4, *s*[5, 6] = 5
s[1, 3] = 2, *s*[2, 4] = 2, *s*[3, 5] = 4, *s*[4, 6] = 4
s[1, 4] = 2, *s*[2, 5] = 2, *s*[3, 6 = 4

$s[1, 5] = 4$, $s[2, 6] = 2$
$s[1, 6] = 2$

Finally, we have $(A_1A_2)((A_3A_4)(A_5A_6))$.

15.4-1 (5-9) Determine an LCS of <1, 0, 0, 1, 0, 1, 0, 1> and <0, 1, 0, 1, 1, 0, 1, 1, 0>.

Solution:

	y_j	0	1	0	1	1	0	1	1	0
x_i	0	0	0	0	0	0	0	0	0	0
1	0	↑0	↖1	←1	↖1	↖1	←1	↖1	↖1	←1
0	0	↖1	↑1	↖2	←2	←2	↖2	←2	←2	↖2
0	0	↖1	↑1	↖2	↑2	↑2	↖3	←3	←3	↖3
1	0	↑1	↖2	↑2	↖3	↖3	↑3	↖4	↖4	←4
0	0	↖1	↑2	↖3	↑3	↑3	↖4	↑4	↑4	↖5
1	0	↑1	↖2	↑3	↖4	↖4	↑4	↖5	↖5	↑5
0	0	↖1	↑2	↖3	↑4	↑4	↖5	↑5	↑5	↖6
1	0	↑1	↖2	↑3	↖4	↖5	↑6	↖6	↖6	↑6

A longest common sequence could be <1, 0, 0, 1, 1, 0> as illustrated below (there could be some other LCS', if we choose ← instead of ↑ when $c[i – 1, j] = c[i, j – 1]$):

X: <–, 1, 0, –, –, 0, 1, 0, 1, 0, 1 >
Y: <0, 1, 0, 1, 1, 0, 1, –, 1, 0, –>

So the LCS is 100110.

15.5-3 (5-15) Suppose that instead of maintaining the table $w[i, j]$ we computed the value of $w[i, j]$ directly from equation (5-5) in line 8 of OPTIMAL-BST and use this computed value in line 10. How would this change affect the asymptotic running time of OPTIMAL-BST?

$$w(i, j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l \qquad (5\text{-}5)$$

```
OPTIMAL-BST(p, q, n)
1   for i ← 1 to n + 1
2       do e[i, i-1] ←q_{i-1}
3          w[i, i-1] ←q_{i-1}
4   for l ← 1 to n
5       do for i ← 1 to n – l + 1
6             do j ← i + l – 1
7                e[i, j] ← ∞
8                w[i, j] ← w[i, j – 1] + p_j + q_j
9                for r ← i to j
10                  do t ← e[i, r – 1] + e[r + 1, j] + w[i, j]
11                     if t < e[i, j]
12                        then e[i, j] ← t
```

13 $root[i, j] \leftarrow r$
14 **return** e and $root$

Solution: Actually, it would not change the asymptotic running time at all. To compute $w[i, j]$ manually on line 8 would require $\Theta(j - i)$ additions, instead of $\Theta(1)$ as in the book's version. However, line 9 does not a loop from i to j anyway, which takes $\Theta(j - i)$ time. Doing another $\Theta(j - i)$ work on line 8 would not affect the asymptotic running time of $\Theta(n^3)$.

15-4 (5-19) *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 tree rooted at the president. The personal 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 (CLRS). Each node of the tree holds in addition to 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 rating of the guests. Analyze the running time of your algorithm.

Solution: This problem may be solved in linear time using dynamic programming. Let $c[x]$ denote the conviviality of employee x. We wish to select a subset of employees S maximizing $\sum_{x \in S} c[x]$, such that if $x \in S$, then $parent[x] \notin S$.
In order to use dynamic programming, we must be able to compute the optimal solution of our problem in terms of optimal solutions to smaller subproblems of the same form. These optimal solutions to subproblems will be the following: let $M(x)$ denote the maximum possible conviviality sum if one were to invite only employees from the subtree rooted at employee x, such that x is invited. Similarly let $M'(x)$ denote the maximum conviviality sum for x's subtree if x is not invited. We can express $M(x)$ and $M'(x)$ recursively in terms of optimal solutions to "smaller" subproblems as follows:

$$M(x) = c[x] + \sum_{y:parent[y]=x} M'(y)$$
$$M'(x) = \sum_{y:parent[y]=x} \max\{M(y), M'(y)\}$$

The first equation states that the optimal way to select employees (including x) from x's subtree is to optimally select employees from the subtree of each child y of x such that y is not invited. The second equation expresses the fact that the optimal way to select employees (not including x) from x's subtree is to optimally select employees from subtrees of children y of x, where it is of no consequence whether or not y is selected. The following pseudocode show how we can compute $M(x)$ and $M'(x)$ for every employee x using a single recursive transversal of the company tree in $O(n)$ time:

```
SOLVE (x)
1   M(x) ← c[x]
2   M'(x) ← 0
3   y ← left-child[x]
4   while y ≠ NIL
5       do SOLVE (y)
6          M(x) ← M(x) + M'(y)
7          M'(x) ← M'(x) + max{ M(y), M'(y)}
8          y ← right-sibling[y]
```

The algorithm will be started by calling SOLVE(p), where p denotes the company president. Upon termination the optimal conviviality sum for the entire company will be given by max{$M(p)$, $M'(p)$}. How does one determine the set of employees to invite that achieves this maximum conviviality sum? This may be done in O(n) time with another tree transversal, by calling INVITE(p), illustrated by the following pseudocode:

```
INVITE (x)
1   if M(x) > M'(x)
2      then Invite x to the party
3           for all grandchildren y of x
4               INVITE (y)
5        else for all children y of x
6               INVITE (y)
```

The reasoning behind this reconstruction algorithm is the following: if $M(p) > M'(p)$, then the optimal solution must involve inviting the company president to the party. If this is the case, we then cannot invite any of the employees reporting immediately to the president, so we proceed to optimally invite employees, which are grandchildren of the president in the company tree. If $M(p) \leq M'(p)$, then we do not need to invite the president, and we then proceed to optimally invite employees from the subtrees of employees directly reporting to the president. Continuing this process recursively, we will be able to produce a set of employees to invite whose conviviality sum is equal to that of the optimal solution, $max\{M(p), M'(p)\}$.

15-6(5-20) *Moving on a checkerboard.*

Suppose that you are given an $n \times n$ checkerboard and a checker. You must move the checker from the bottom edge of the board to the top edge of the board according to the following rule. At each step you may move the checker to one of three squares:

1. the square immediately above,
2. the square that is one up and one to the left (but only if the checker is not already in the leftmost column),
3. the square that is one up and one to the right (but only if the checker is not already

in the rightmost column)

Each time you move from square x to square y, you receive $p(x, y)$ dollars. You are given $p(x, y)$ for all pairs (x, y) for which a move is legal. Do not assume that $p(x, y)$ is positive.

Give an algorithm that figures out the set of moves that will move the checker from somewhere along the bottom edge to somewhere along the top edge while gathering as many dollars as possible. Your algorithm is free to pick any square along the bottom edge as a starting point and any square along the top edge as a destination in order to maximize the number of dollars gathered along the way. What is the running time of your algorithm?

Solution: Here's a formal dynamic programming solution. We can construct a recursive function, which characterizes the structure of the problem, which would at the top level look something like:

$$max_j \{D[j, n]\} \ \forall j \in \{1, 2, \ldots, n\},$$

where $D[j, n]$ is a recursive call to find the best way to get to the board square (j, n), which here corresponds to column j, row n. Then define:

$$D[j, r] = max_i \{p((j + i, r - 1), (j, r))\} + D[j + 1, r - 1] \ \forall i \in \{-1, 0, +1\},$$

or that the best way to get to column j row r is the best from the column before it, the column itself, or the column after it — this is just making an equation for it. Then at the first level: $D[j, 1] = 0$,

Each entry can be computed in constant time, and we need to fill all the n^2 entries. Therefore, the running time is $\Theta(n^2)$.

15-7(5-21) *Scheduling to maximize profit.*

Suppose you have one machine and a set of n jobs $a_1, a_2, \ldots, a_n$ to process on that machine. Each job a_j has a processing time t_j, a profit p_j, and deadline d_j. The machine can process only one job at a time, and job a_j must run uninterruptedly for t_j consecutive time units. If jog a_j is completed by its deadline d_j, you receive a profit p_j, but if it is completed after its deadline, you receive a profit of 0. Give an algorithm to find the schedule that obtain the maximum amount of profit, assuming that all processing times are integers between 1 and n. What is the running time of your algorithm?

Solution: First we sort the jobs in the increasing order of d_j. Assume that n jobs $a_1, a_2, \ldots, a_n$, are sorted in this fashion. Consider the following recursive function. $S[j, d]$ denotes the subproblem of j jobs $a_0, \ldots, a_j$ and the deadline d by which we have to finish all our works. That is, the feasible solution must finish the last job by the time d.

If $t_i \le d$

$$S[j, d] = \max \begin{cases} S[j-1, d_j - t_j] + p_j & \text{if we select the job } a_j \\ S[j-1, d] & \text{otherwise} \end{cases}$$

If $t_i < d$

$$S[j, d] = S[j\text{-}1, d]$$

Since this recursive function satisfies the optimal substructure property, it generates the optimal solution. All we have to do is to build a table of size n by d_n and obtain the solution for $S[n, d_n]$ using above recursion. The running time is $max\{n \lg n, n\ max_{1\le j\le n}\ d_j\}$.

16.2-1(5-24) Prove that the fractional knapsack problem has the greedy-choice property.

Solution: Say that the greedy strategy uses the i most expensive items fully, and the rest is part of the i +1st most expensive item. Then suppose some non-greedy strategy worked better. It would then have to use less than all of one of the i most expensive items, or less of the i +1st most expensive item, in order to have space to use some other item. Say that it uses less of item j and that amount more of item k. You could replace that amount of item k with the missing amount of item j and have a mixture that is at least this valuable. You can repeat this process until all the amounts of items not included in the greedy mix but included in the "optimal" non-greedy solution have been replaced by amounts of items in the greedy solution; and your greedy mix is at least as valuable. This is a contradiction.

16.2-2(5-25) Give a dynamic programming solution to the 0-1 knapsack problem that runs in O(nW) time, where n is number of items and W is the maximum weight of items that the thief can put in his knapsack.

Solution: We first argue that the 0-1 knapsack problem shows an optimal substructure. If an optimal solution contains item n, the remaining choices must constitute an optimal solution to similar problem on items 1, 2, …, n – 1 with bound $W - w_n$. If an optimal solution does not contain item n, the solution must also be an optimal solution to similar problem on items 1, 2, …, n – 1 with bound W.

Let $m[i, j]$ represents the total value that can be taken from the first i items when the knapsack can hold j. Our problem is to get the maximum value for $m[n, W]$ where n is the number of given items and W is the maximum weight of items that the thief can put it in his knapsack. We can express this in the following formula:

$$m[i, j] = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0, \\ m[i-1, j] & \text{if } w_i > j, \\ \max\{v_i + m[i-1, j-w_i], m[i-1, j]\} & \text{if } i > 0 \text{ and } j \ge w_i. \end{cases}$$

The algorithm takes as input n, W, and the two sequences $v = \langle v_1, v_2, \ldots, v_n\rangle$ and $w = \langle w_1, w_2, \ldots, w_n\rangle$. It stores $m[i, j]$ values in a table, i.e. a 2-dimensional array, $m[0..n, 0..W]$, whose entries are computed in a row-major order. (That is, the first row of m is filled in from left to right, then the second row, and so on.) At the end of the computation, $m[n, W]$ contains the maximum value that can be packed into the knapsack.

We can use this recursion to create a straightforward dynamic programming

algorithm:

```
KNAPSACK(v, w, W)
1   for w ← 0 to W
2      do m[0, w] ←0
3   for i ←1 to n
4      do m[i, 0] ←0
5        for w ←1 to W
6           do if w_i ≤ w
7                then if v_i + m[i – 1, w - w_i] > m[i – 1, w]
8                        then m[i, w] ←v_i + m[i – 1, w – w_i]
9                        else m[i, w] ← m[i – 1, w]
10             else m[i, w] ← m[i – 1, w]
11  return m[n, W]
```

Since each $m[i, j]$ can be computed in constant time, the total running would be $O(nW)$.

16.2-4(5-27) Professor Midas drives an automobile from Neward to Reno along Interstate 80. His car's gas tank, when full, holds enough gas to travel n miles, and his map gives the distance between gas station on his route. The professor wishes to make as few gas stops as possible along the way. Give an efficient method by which Professor Midias can determine at which gas station he should stop, and prove that your strategy yields an optimal solution.

Solution: Professor Midas needs to select a gas station, whose distance from his current stop is closest to n but less than n. This should be the gas station where he will refuel. We will show that this greedy strategy gives an optimal solution.

First we show that the solution to the problem contains optimal solutions to subproblems. Let S be a solution of the problem, and G be a gas station in which the professor made a stop. If the professor did not made the least possible stops from his starting point to G, then by doing less stops he could have gotten a better solution from his starting point to G and thus from his starting point to his destination. But S is an optimal solution. This is why S contains optimal solutions to subproblems.

Now we will prove that the greedy choice results in an optimal solution. Let the stops according to our greedy algorithms be at gas stations G_1, G_2, ..., G_k . Let's assume that this solution is not optimal. Then the first stop of the optimal solution O_1 is either at G_1 or at a gas station before it. Otherwise the distance to that gas station will be more than the distance to G_1 and less than n and we would have selected it using the greedy strategy. The distance from O_1 to O_2 is less than n, and since O_1 is either G_1 or a gas station before it, the distance between G_1 and O_2 will be less than n. This means that we can replace O_1 with G_1 without violating the optimality of the solution (we use part 1).
Proceeding inductively in this manner we can replace all optimal solution gas stations

with greedy solution gas stations. This proves that the greedy approach gives an optimal solution.